Creatio Academy

Front-end development

Version 8.0

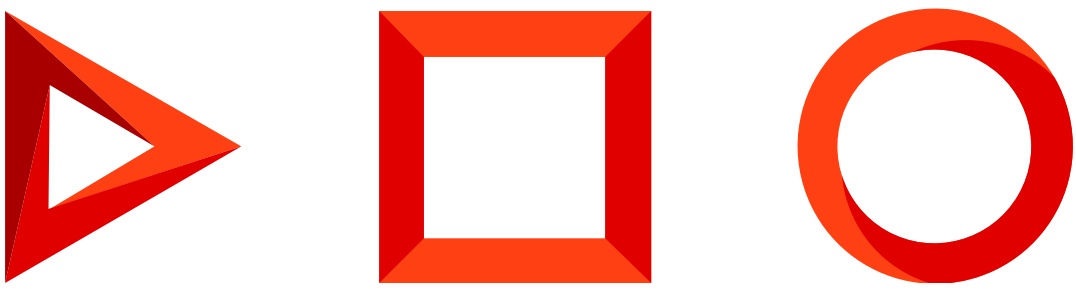


Table of Contents

Module basics

Advanced

AMD concept

Creatio front-end is a set of function blocks implemented in separate **modules**. In accordance with the **Asynchronous Module Definition (AMD) concept**, Creatio loads modules and their dependencies asynchronously at runtime. Therefore, the AMD concept lets you load only data with which you need to work currently. Learn more about the AMD concept in Wikipedia.

Different JavaScript frameworks support the AMD concept. Creatio uses the **RequireJS loader** to work with modules. Learn more on the official RequireJS website.

Modular development in Creatio

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

The **Module programming pattern** declares the module creation in JavaScript. Learn more in a separate article: JavaScript Module Pattern: In-Depth. The classic **pattern implementation** uses anonymous functions that return a specific value, for example, object, function, etc., associated with a module. In this case, the module value is exported to a global object.

Example that exports the module value to a global object

```
/* Immediately invoked function expression (IIFE). The anonymous function that initializes the m
(function () {
    /* Access a module on which the current module depends.
    Load the module into the SomeModuleDependency global variable before accessing it.
    The "this" context is a global object. */
    var moduleDependency = this.SomeModuleDependency;
    /* In the global object property, declare a function that returns the module value. */
    this.myGlobalModule = function () { return someModuleValue; };
}());
```

When the interpreter finds a functional expression like this in the code, the interpreter immediately resolves it. As a result, a function that returns the module value will be placed to the `myGlobalModule` global object's property.

The **concept specifics** are as follows:

- Declaration and use of dependency modules are complex.
- Load all module dependencies before Creatio starts executing the anonymous function.
- Dependency modules must be loaded via the `<script>` HTML element in the page header. Use global variable names to access the dependency module. In this case, the developer needs to implement the load order of module dependencies.

- As a result of the previous item, Creatio loads the modules before the browser starts rendering the page, so the modules cannot access page controls to implement custom logic.

The **special features** of using the concept in Creatio are as follows:

- You cannot load modules dynamically.
- You might need to apply additional logic when loading modules.
- Managing a large number of modules with multiple dependencies that can overlap adds complexity.

RequireJS loader

The **RequireJS loader** provides a mechanism that declares and loads modules based on the AMD concept and lets you take into account the specifics listed above.

The **operating principles** of the RequireJS loader are as follows:

- The module is declared in the `define()` function that registers a factory function to instantiate the module. At the same time, the `define()` function does not load the module immediately when this function is called. Learn more about the `define()` function on the GitHub website.
- Module dependencies are passed as an array of string values, not via the global object properties.
- The loader loads the module dependencies that are passed as arguments of the `define()` function. Modules are loaded asynchronously. The loader sets the load order arbitrarily.
- After the specified module dependencies are loaded, the loader calls a factory function that returns the module value. The loaded module dependencies are passed to the function as arguments.

Example that declares a module


Beginner

Example that uses the `define()` function to declare the `SumModule` module

```
/* The SumModule module implements the functionality that adds two numbers.
The module has no dependencies. Therefore, the function passes an empty array as the second argu
define("SumModule", [], function () {
    /* The body of the anonymous function contains the internal implementation of the module's f
    var calculate = function (a, b) { return a + b; };
    /* The function returns a value that is an object. In this case, the object is the module. *
    return {
        /* Description of the object. In this case, the module is an object that has the summ pr
        summ: calculate
    };
});
```

define() function JS

Beginner

The **purpose** of the `define()` function is to declare an asynchronous module in the source code. The loader works with this module.

Declare the module

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

Parameters

`ModuleName`

The string that contains the module name. Optional.

If you do not specify the parameter, the loader assigns a module name automatically based on the module location in the Creatio script tree. The name is required to access the module from other Creatio parts, including asynchronous loading as a dependency of another module.

`dependencies`

An array of module names on which the current module depends. Optional.

The RequireJS loader loads the module dependencies that are passed as arguments to the `define()` function. The loader calls the factory function after loading the dependencies listed in the `dependencies` array.

`function(dependencies)`

An anonymous factory function that instantiates the module. Required.

Pass the objects that the loader associates with module dependencies as function parameters. List dependencies in the `dependencies` array. The array arguments are required to access the properties and methods of the dependency modules in the current module. The order of dependencies in the `dependencies` array corresponds to the order in which the parameters are listed in the factory function.

The factory function returns the value that the loader handles similarly to the exported value of the current module.

The **types** of values that the factory function returns are as follows:

- Object. In this case, the object is the module. The browser caches the module after the initial download. If the module declaration changes after the client downloads the module, for example, as part of implementing the configuration logic, clear cache and re-download the module.

- Module function factory. The constructor receives the scope object as an argument. As a result, downloading a module creates a module instance (**instantiated module**) in the client. Re-downloading the module to the client via the `require()` function creates another module instance. Creatio treats these instances of the same module as individual modules. View the example that declares an instantiated module in the `CardModule` schema of the `NUI` package.

Module types

Advanced

Base modules

Creatio implements the following **base modules**:

- `ext-base`. Implements the `ExtJs` framework functionality.
- `terrasoft` in the `Terrasoft` namespace. Implements access to system operations, core variables, etc.
- `sandbox`. Implements a mechanism that exchanges messages among modules.

Example that accesses the `ext-base`, `terrasoft`, and `sandbox` modules

```
/* Define a module and retrieve links to dependency modules. */
define("ExampleModule", ["ext-base", "terrasoft", "sandbox"],
    /* "Ext" is the link to the object that provides access to the ExtJS framework.
    "Terrasoft" is the link to the object that provides access to system variables, core variabl
    "sandbox" is the link to the object required to exchange messages among modules. */
    function (Ext, Terrasoft, sandbox) {
});
```

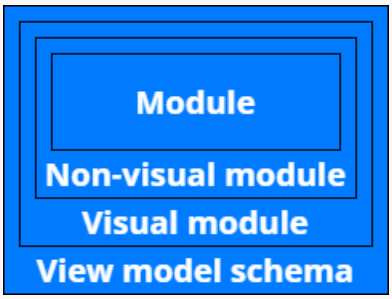
Most client modules use base modules. You do not have to specify base modules in dependencies. After you create a module object, the `Ext`, `Terrasoft`, and `sandbox` objects become available as the `this.Ext`, `this.Terrasoft`, and `this.sandrenderbox` object properties.

Client modules

Client module types

- non-visual module
- visual module
- replacing module

View the client module hierarchy below.



Non-visual module

The **purpose** of a non-visual module is to implement Creatio functionality that is usually not related to binding data and displaying data in the UI. For example, business rule modules (the `BusinessRuleModule` schema in the `NUI` package) and utility modules that implement service functions are non-visual modules.

To **implement a non-visual module**, follow the instructions in a separate article: Client module.

Visual module

Visual modules are modules that implement view models (`ViewModel`) in Creatio based on the MVVM pattern. Learn more about the MVVM pattern in Wikipedia.

The **purpose** of a visual module is to encapsulate data visualized in the UI controls and the methods of working with data. For example, section, detail, and page modules are visual modules.

To **implement a visual module**, follow the instructions in a separate article: Client module.

Replacing module

The **purpose** of a replacing module is to extend the base module's functionality. Modules that replace base functionality modules do not support inheritance in its traditional sense. You cannot use the resources of the replaced module in the replacing module. Instead, recreate resources in the replacing schema.

To **implement a replacing module**, follow the instructions in a separate article: Client module.

Client module methods

Creatio lets you implement the following **methods** in client modules:

- `init()`. Implements the logic executed as part of loading the module. The client core calls this method first automatically when loading the module. The `init()` method usually subscribes to events of other modules and initializes the module values.
- `render(renderTo)`. Implements the module visualization logic. The client core calls this method automatically when loading the module. To ensure data is displayed correctly, the mechanism that binds the view (`View`) and view model (`ViewModel`) must be triggered before data visualization. As such, this mechanism is usually initiated in the `render()` method. by calling the `bind()` method in the view object. If the module is loaded into a container, Creatio passes the link to that container to the `render()` method as an argument. The `render()` method is required for visual modules.

Utility modules

Although module is an isolated software unit, it can use the functionality of other modules. To do this, import the needed module as a dependency. Use a factory function argument to access an instance of a dependency module.

During development, you can group auxiliary and service general-purpose methods into separate **utility modules</141>. Import the utility module into a module to use the functionality.**

Work with resources

Resources are additional schema properties. Add resources to the client module schema in the properties area of the Module Designer (the + button).

Creatio lets you use the following **resources**:

- localizable strings (the [*Localizable strings*] property)
- images (the [*Images*] property)

The `[ClientModuleName]Resources` module contains the resources that the Creatio core generates for each client module automatically.

To **access the resource module from the client module**, import the resource module into the client module as a dependency.

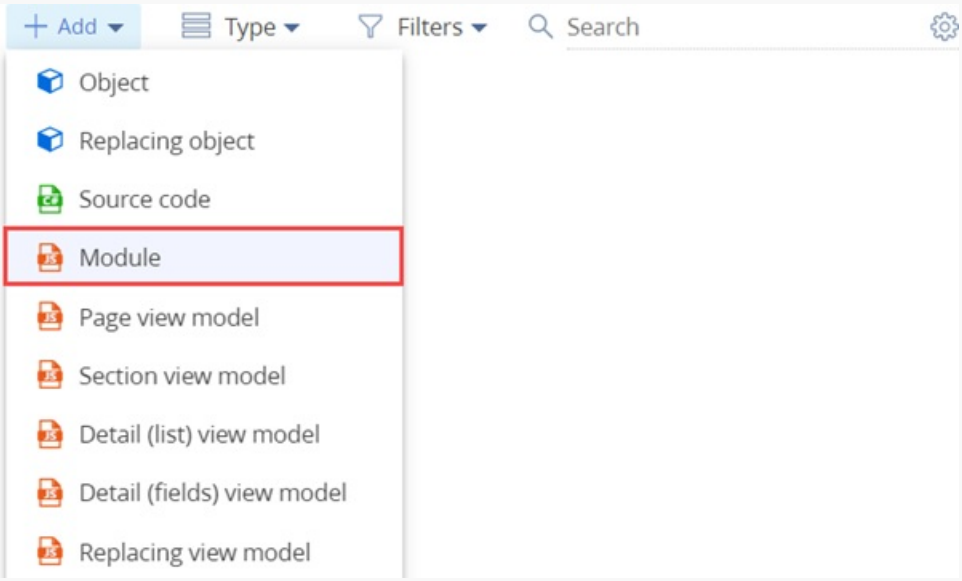
Create a standard module

 Medium

Example. Create a standard module that contains the `init()` and `render()` methods. When loading the module, the client core must call the `init()` method, then the `render()` method. Each method must call a message box.

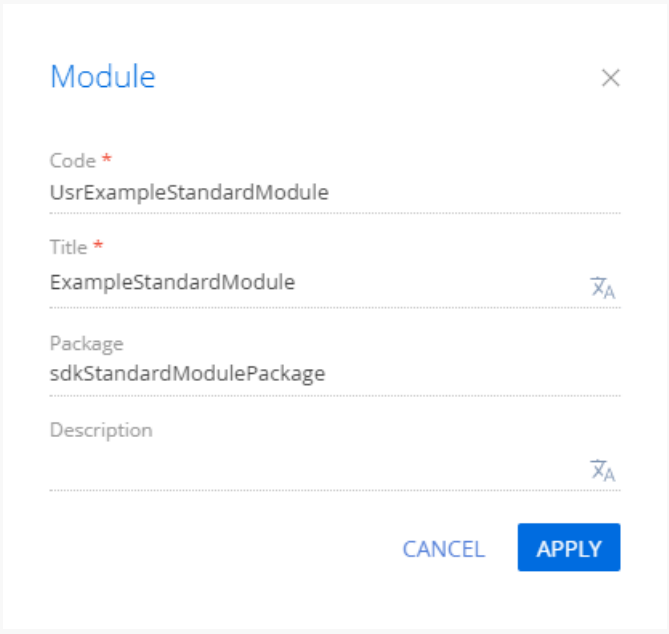
Create a visual module

1. Open the [*Configuration*] section and select a custom package to add the schema.
2. Click [*Add*] → [*Module*] on the section list toolbar.



3. Fill out the schema properties in the Module Designer.

   - Set [*Code*] to "`UsrExampleStandardModule`."
   - Set [*Title*] to "`ExampleStandardModule`."



   Click [*Apply*] to apply the changes.

4. Add the source code in the Module Designer.

```
UsrExampleStandardModule
```

```
/* Declare a module called UsrExampleStandartModule. The module has no dependencies. Therefor
define("UsrExampleStandardModule", [], function () {
    return {
```

```
        /* The client core calls this method first automatically when loading the module. */
        init: function () {
            alert("Calling the init() method of the UsrExampleStandardModule module");
        },
        /* The client core calls this method automatically when loading the module into a con
        render: function (renderTo) {
            alert("Calling the render() method of the UsrExampleStandardModule module. The mo
        }
    };
});
```

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

Outcome of the example

To **view the outcome of the example**, create the following request string.

Template URL

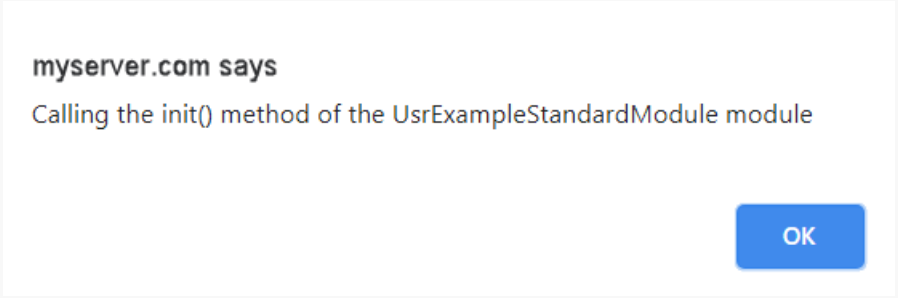
```
[Creatio URL]/0/NUI/ViewModule.aspx#[SomeModuleName]
```

Example URL

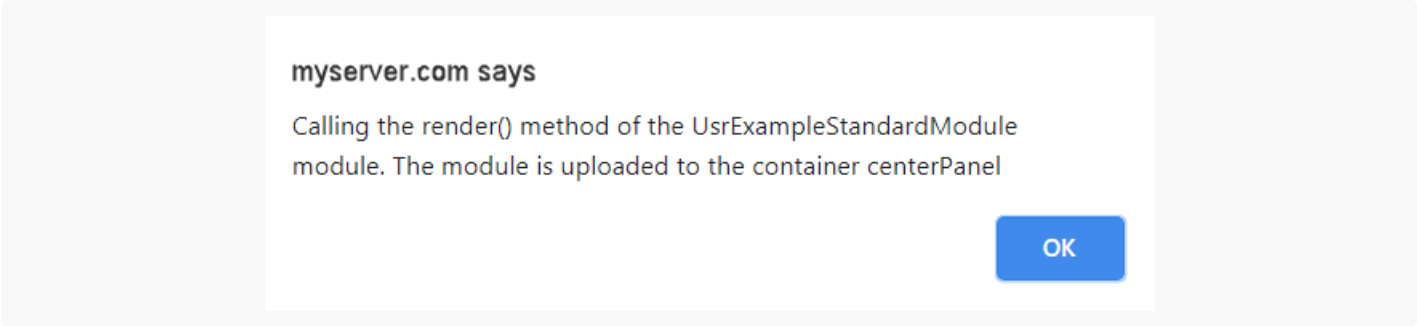
```
http://myserver.com/0/NUI/ViewModule.aspx#UsrExampleStandardModule
```

When loading the module, the client core must call the `init()` method, then the `render()` method. Each method calls a message box.

Call the `init()` method of the `UsrExampleStandardModule` module



Call the `render()` method of the `UsrExampleStandardModule` module



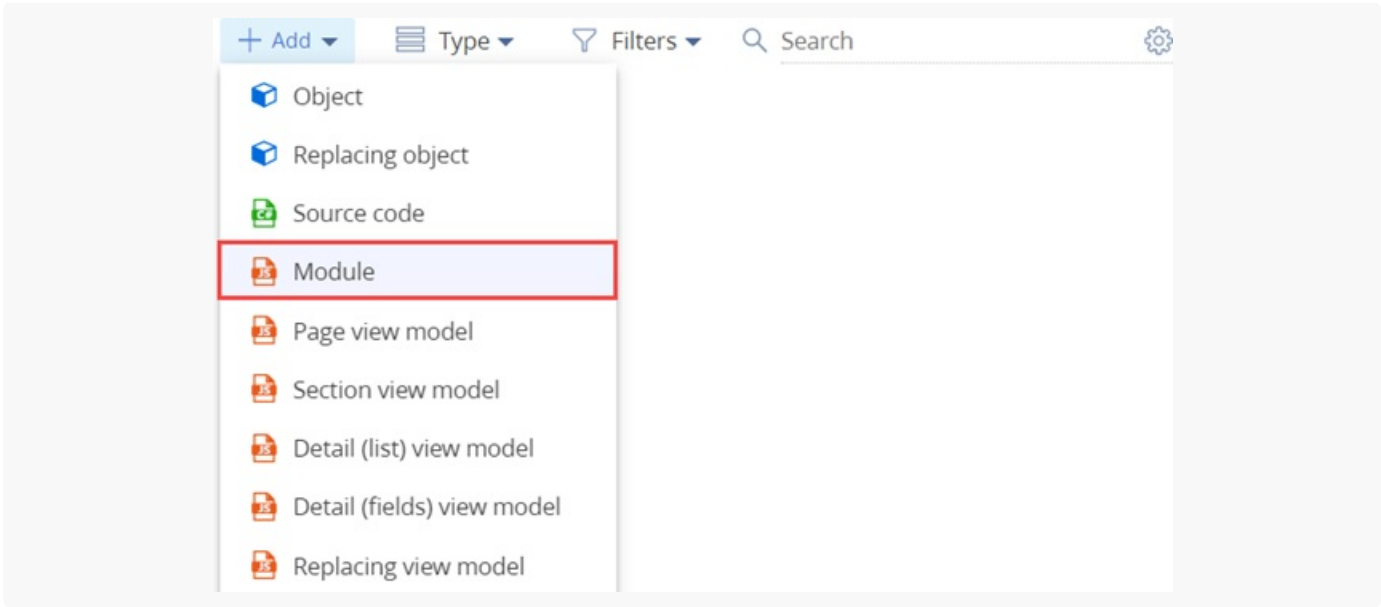
Create a utility module

Medium

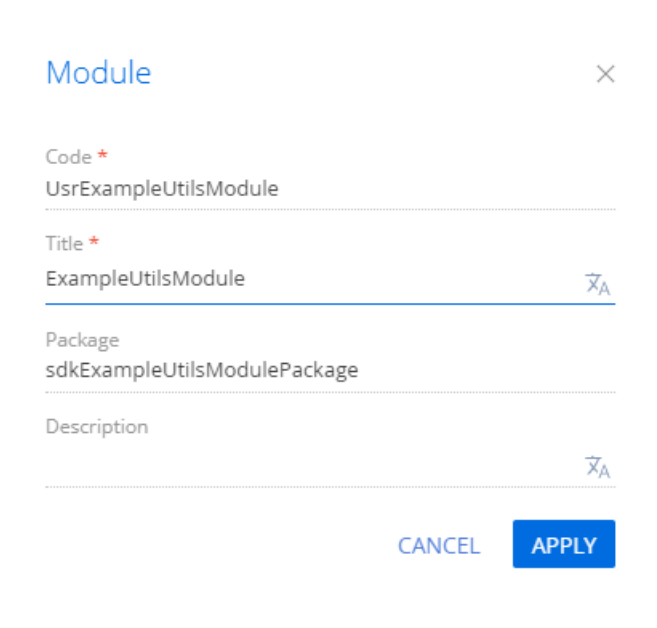
Example. Create a standard module that contains the `init()` and `render()` methods. When loading the module, the client core must call the `init()` method, then the `render()` method. Each method must call a message box. Implement the method that displays the message box using a utility module.

1. Create a utility module

1. Open the [*Configuration*] section and select a custom package to add the schema.
2. Click [*Add*] → [*Module*] on the section list toolbar.



3. Fill out the schema properties in the Module Designer.
   - Set [*Code*] to "`UsrExampleUtilsModule`."
   - Set [*Title*] to "`ExampleUtilsModule`."



Click [*Apply*] to apply the changes.

4. Add the source code in the Module Designer.

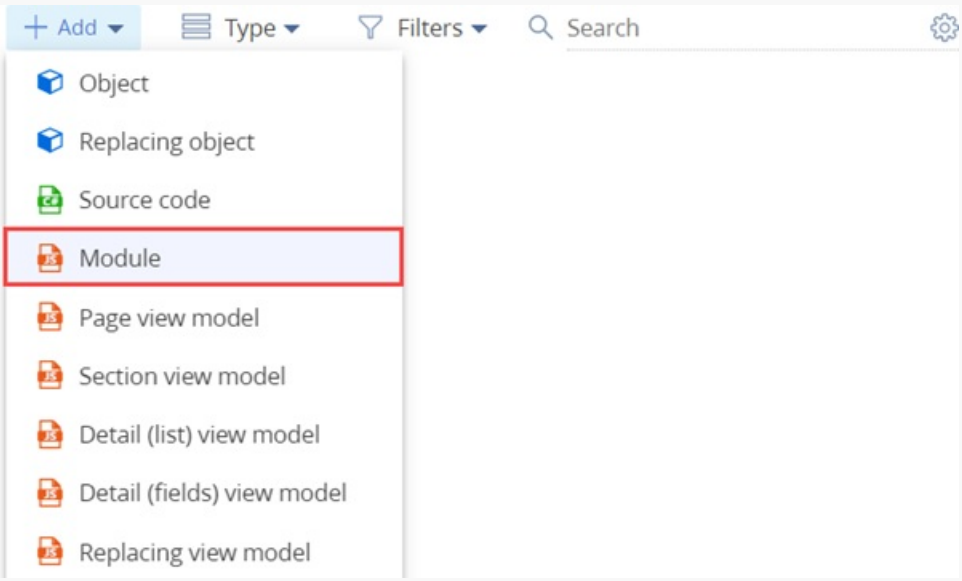
UsrExampleUtilsModule

```
/* Declare a module called UsrExampleUtilsModule. The module has no dependencies. Therefore,
The module contains the method that displays the message box. */
define("UsrExampleUtilsModule", [], function () {
    return {
        /* The method that displays the message box. The "information" method argument contai
        showInformation: function (information) {
            alert(information);
        }
    };
});
```

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

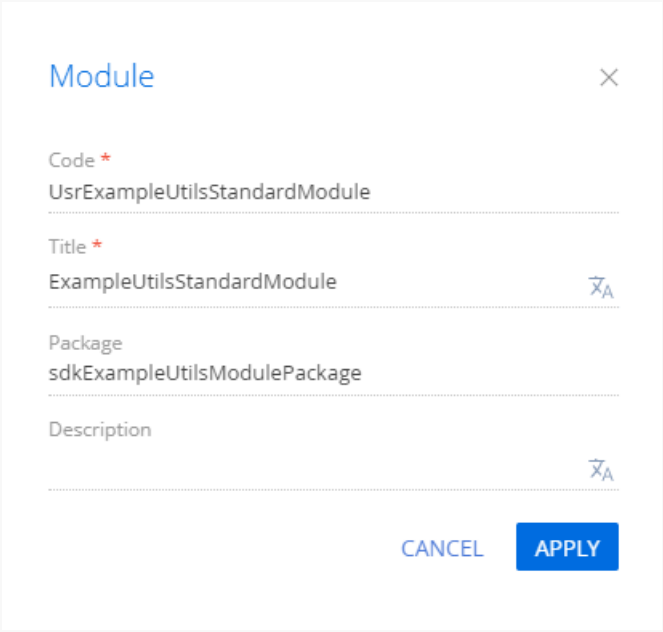
2. Create a visual module

1. Open the [*Configuration*] section and select a custom package to add the schema.
2. Click [*Add*] → [*Module*] on the section list toolbar.



3. Fill out the schema properties in the Module Designer.

   - Set [*Code*] to "`UsrExampleUtilsStandardModule`."
   - Set [*Title*] to "`ExampleUtilsStandardModule`."



   Click [*Apply*] to apply the changes.

4. Add the source code in the Module Designer.

```
UsrExampleUtilsStandardModule
```

```
/* Declare a module called UsrExampleUtilsStandardModule. The module imports the UsrExampleUt
define("UsrExampleUtilsStandardModule", ["UsrExampleUtilsModule"], function (UsrExampleUtilsM
    return {
```

```
        /* The init() and render() methods call a utility method to display the message that
        init: function () {
            UsrExampleUtilsModule.showInformation("Calling the init() method of the UsrExampl
        },
        render: function (renderTo) {
            UsrExampleUtilsModule.showInformation("Calling the render() method of the UsrExam
        }
    };
});
```

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

Outcome of the example

To **view the outcome of the example**, create the following request string.

Template URL

```
[Creatio URL]/0/NUI/ViewModule.aspx#[SomeModuleName]
```

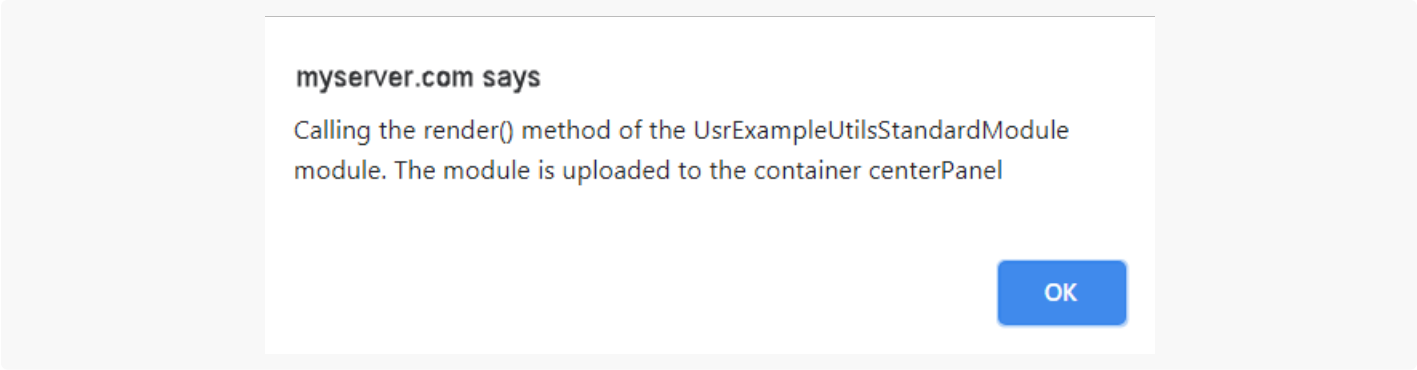
Example URL

```
http://myserver.com/0/NUI/ViewModule.aspx#UsrExampleUtilsStandardModule
```

When loading the module, the client core must call the `init()` method, then the `render()` method. Each method calls a message box.

Call the `init()` method of the `UsrExampleUtilsStandardModule` module

Call the `render()` method of the `UsrExampleUtilsStandardModule` module



Client schema

Easy

A **client view model schema** is a visual module schema that implements the front-end part of Creatio. A client view model schema is a configuration object for generating views and view models by `ViewGenerator` and `ViewModelGenerator`. Learn more about module types and their specificities in a separate article: Client module types.

Develop a client schema

Ways to develop client view model schemas:

- The [*Configuration*] section. Learn more about development in the [*Configuration*] section in a separate article: View model schema.
- Section Wizard. Learn more about section development in the Section Wizard in user documentation: Create a new section
- Detail Wizard. Learn more about detail development in the Detail Wizard in user documentation: Create a detail.

Structure elements of the client schema:

- Auto-generated code. Contains the description of the schema, its dependencies, localized resources, and messages.
- Rendering styles. Only available in some types of client schemas.
- The schema source code. A syntactically correct JavaScript code that defines the module.

Use **marker comments** in the schema source code for the `diff`, `modules`, `details`, and `businessRules` properties.

The **purpose** of marker comments is to uniquely identify the client schema properties. When you open the Wizard, Creatio validates the presence of marker comments as shown in the table below.

Validation rules for marker comments in client schemas

Schema type	Required marker comments

`EditViewModelSchema` **model view schema of a record page**	`details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,`
	`modules: /**SCHEMA_MODULES*/{}/**SCHEMA_MODULES*/,`
	`diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/,`
	`businessRules: /**SCHEMA_BUSINESS_RULES*/{}/**SCHEMA_BUSINESS_RULES*/`

`ModuleViewModelSchema` **model view schema of a section**	`modules: /**SCHEMA_MODULES*/{}/**SCHEMA_MODULES*/,` `diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/`
`EditControlsDetailViewModelSchema` **model view schema of a detail with fields**	
`DetailViewModelSchema` **model view schema of a detail**	
`GridDetailViewModelSchema` **model view schema of a detail that has a list**	

Client schema properties

The source code of client schemas has a generic structure available below.

Source code of a client schema

```
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*/
    };
});
```

After the module is loaded, Creatio calls the anonymous factory function, which returns the schema configuration object. **Properties** of the configuration object schema:

- `entitySchemaName`. The name of the entity schema used by the current client schema.
- `mixins`. A configuration object that contains a mixin declaration.
- `attributes`. A configuration object that contains schema attributes.
- `messages`. A configuration object that contains schema messages.
- `methods`. A configuration object that contains schema methods.
- `rules`. A configuration object that contains schema business rules.
- `businessRules`. A configuration object that contains schema business rules created or modified by the Section Wizard or Detail Wizard. The `/**SCHEMA_BUSINESS_RULES*/` marker comments are required since they are necessary for the operation of the Wizards.
- `modules`. A configuration object that contains schema modules. The `/ ** SCHEMA_MODULES * /` marker comments are required since they are necessary for the operation of the Wizards.

  > **Note.** The `details` property loads a detail to a page. Since a detail is also a module, we recommend using the `modules` property instead.

- `diff`. A configuration object array that contains the schema view description. The `/**SCHEMA_DIFF*/` marker comments are required since they are necessary for the operation of the Wizards.
- `properties`. A configuration object that contains view model properties.
- `$-properties`. Automatically generated properties for the attributes of the view model schema.

Schema name (`entitySchemaName`)

To implement the entity schema name, use the required `entitySchemaName` property. Simply specify it in one of the inheritance hierarchy schemas.

Example that declares the `entitySchemaName` property

```
define("ClientSchemaName", [], function () {
```

```
    return {
        /* Object schema (model). */
        entitySchemaName: "EntityName",
        /* ... */
    };
});
```

Mixins (`mixins`)

A mixin is a class that extends the functions of other classes. JavaScript does not support multiple inheritances. However, mixins let you extend the schema functionality without duplicating the logic used in the schema methods. You can use the same set of actions in different client schemas of Creatio. Create a mixin to avoid duplicating the code in each schema. Mixins are **different** from other modules added to the dependency list in the way of calling their methods from the module schema. You can call to their methods directly, much like those of a schema. Use the `mixins` property to implement mixins.

Mixin management **procedure**:

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 client schema.

Create a mixin

Create a mixin similarly to an object schema.

Assign a name to the mixin

When naming mixins, use an `-able` suffix in the schema name. For example, name a mixin that enables serializing in the components `Serializable`. If a mixin name cannot end "-able," end the schema name in `Mixin`.

Attention. Do not use words like `Utilities`, `Extension`, `Tools`, or similar in the names. They make the purpose of the mixin impossible to discern based on the mixin name.

Connect the namespace

Enable a corresponding name array in the mixin (`Terrasoft.configuration.mixins` for the configuration, `Terrasoft.core.mixins` for the core).

Implement the mixin functionality

Mixins cannot depend on the internal implementation of the schema to which to apply them. Mixins must be independent mechanisms that receive a set of parameters, process them, and, if needed, return a result. Design

mixins as modules that must be connected to the schema dependency list when the `define()` function declares the schema.

View the mixin structure below.

Mixin structure

```
define("MixinName", [], function() {
    Ext.define("Terrasoft.configuration.mixins.MixinName", {
        alternateClassName: "Terrasoft.MixinName",
        /* Mixin functionality. */
    });
    return Ext.create(Terrasoft.MixinName);
})
```

Use the mixin

The mixin implements the functionality needed in the client schema. To receive the set of mixin actions, specify the mixin in the `mixins` block of the client schema.

Use a mixin in the client schema

```
/* MixinName is a module where the mixin class is implemented. */
define("ClientSchemaName", ["MixinName"], function () {
    return {
        /* SchemaName is the name of the entity. */
        entitySchemaName: "SchemaName",
        mixins: {
            /* Connect the mixin. */
            MixinName: "Terrasoft.NameSpace.Mixin"
        },
        attributes: {},
        messages: {},
        methods: {},
        rules: {},
        modules: /**SCHEMA_MODULES*/{}/**SCHEMA_MODULES*/,
        diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/
    };
});
```

Once you connect the mixin, you can use its methods, attributes, and fields in the client schema as if they were part of the client schema. That way, method calls are more concise than when using a separate schema. For example, `getDefaultImageResource` is a mixin function. To call the `getDefaultImageResource` mixin function in the custom schema to which the mixin is connected, use `this.getDefaultImageResource();`.

Note. To overload a mixin function, create a function with the same name in the client schema. As a result, Creatio will use the function of the schema, and not that of the mixin, when calling.

Attributes (`attributes`)

Use the `attributes` property to implement attributes.

Messages (`messages`)

The **purpose** of messages is to organize data exchange between modules. Use the `messages` property to implement messages. Use the `Terrasoft.MessageMode` enumeration to set the message **mode**.

Message mode types

Message mode	Description	Connection
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` .

Aside from modes, you can also specify the message **direction**.

Message direction types

Message direction	Description	Connection
Publishing	The message can only be published, i. e., it is an **outbound** message.	To set the message direction to publishing, set the `direction` property to `this.Terrasoft.MessageDirectionType.PUBLISH`.
Subscription	The message can only be subscribed to, i. e., it is an **inbound** message.	To set the message direction to subscription, set the `direction` property to `this.Terrasoft.MessageDirectionType.SUBSCRIBE`.
Bidirectional	The bidirectional mode enables publishing of and subscription to the same message in different instances of a single class or within a single schema inheritance hierarchy. The same message cannot be announced with different directions in a single schema inheritance hierarchy. Learn more about using bidirectional messages in cases where that is a requirement in a separate article: Sandbox.	Corresponds to `the Terrasoft.MessageDirectionType.BIDIRECTIONAL` enumeration value.

Message publication

Declare a message with the "`publishing`" direction in the schema where you want to publish the message.

Example that declares a message with the "`publishing`" direction

```
messages: {
    /* Message name. */
    "GetColumnsValues": {
        /* Set the message mode to address. */
        mode: this.Terrasoft.MessageMode.PTP,
        /* Set the message direction to "publishing." */
        direction: this.Terrasoft.MessageDirectionType.PUBLISH
```

```
    }
}
```

Publishing is done by calling the `publish` method from the `sandbox` class instance.

Message publishing example

```
// GetColumnsValues method that gets the message publishing result.
getColumnsValues: function(argument) {
/* Message publishing.
GetColumnsValues is the message name.
argument is the argument passed to the handler function of the subscriber. An argument is an obj
key is an array of message filtering tags. */
return this.sandbox.publish("GetColumnsValues", argument, ["key"]);
}
```

Attention. Message publishing can return the handler function results only in the **address** mode.

Message subscription

Declare a message with the "`subscription`" direction in the subscription schema.

Example that declares a message with the "`subscription`" direction

```
messages: {
    /* Message name. */
    "GetColumnsValues": {
        /* Set the message mode to address. */
        mode: this.Terrasoft.MessageMode.PTP,
        /* Set the message direction to "subscription." */
        direction: this.Terrasoft.MessageDirectionType.SUBSCRIBE
    }
}
```

The subscription is made by calling the `subscribe` method in the `sandbox` class instance.

Message subscription example

```
/* GetColumnsValues is the message name.
messageHandler is the message handler function.
context is the execution scope of the handler function.
key is an array of message filtering tags. */
```

```
this.sandbox.subscribe("GetColumnsValues", messageHandler, context, ["key"]);
```

In the **address** mode, the `messageHandler` method returns the object, which is processed as the result of message publishing. In **broadcasting** mode, the `messageHandler` method does not return a value.

messageHandler method (address mode)

```
methods: {
    messageHandler: function(args) {
        /* Return an object to process as a message publishing result. */
        return { };
    }
}
```

messageHandler method (broadcast mode)

```
methods: {
    messageHandler: function(args) {
    }
}
```

Methods(`methods`)

To implement a method, use the `methods` property.The property contains a collection of methods that form the schema business logic and affect the view model. By default, the scope of methods is the scope of view model.

The **purpose** of the `methods` property.

1. Create new methods.
2. Extend the basic methods of parent schemas.

Business rules (`rules` and `businessRules`)

Business rules are Creatio mechanisms that let you customize the behavior of fields on a page or detail. To implement business rules, use the `rules` and `businessRules` properties. Use the `businessRules` property for business rules created or modified in the Section Wizard or the Detail Wizard.

The **purposes** of business rules:

- Hide or show fields.
- Lock or unlock fields for editing.
- Make fields required or optional.
- Filter lookup fields based on values in other fields.

Creatio implements the business rule functionality in the `BusinessRuleModule` client module. To use the business rule functionality, add `BusinessRuleModule` to the list of schema dependencies.

Example that adds `BusinessRuleModule` to the dependency list

```
define("CustomPageModule", ["BusinessRuleModule"],
    function(BusinessRuleModule) {
        return {
            /* Implement the client module. */
        };
    });
```

The `RuleType` enumeration of the `BusinessRuleModule` module defines business rule types.

Specifics of business rules

Specifics of business rule declaration:

- Describe business rules in the `rules` schema property.
- Apply business rules to view model columns and not to controls.
- Name each business rule.
- Set business rule parameters in the configuration object.

Business rules defined in the `businessRules` property have the following **features**:

- They are generated by the Section or Detail Wizards.
- When you create a new business rule, the corresponding Wizard generates the name and adds the rule to the client schema of the record page view model.
- Creatio does not use the `BusinessRuleModule` enumerations when describing generated business rules.
- The `/**SCHEMA_BUSINESS_RULES*/` marker comments are required since they are necessary for the operation of the Wizards.
- They have a higher priority during runtime.
- When a business rule is disabled, Creatio sets the `enabled` property of the configuration object to `false`.
- When a business rule is removed, the configuration object remains in the client schema of the record page view model, but Creatio sets the `removed` property to `true`.

Attention. We do not recommend editing the `businessRules` property of the client schema.

Edit an existing business rule

After a Wizard edits a manually created business rule, the business rule's configuration object in the `rules` property of the record page view model remains unchanged. At the same time, a new version of the business

rule configuration object with the same name is created in the `businessRules` property.

When Creatio processes a business rule during runtime, the business rule defined in the `businessRules` property takes precedence. Subsequent changes to this business rule in the `rules` property will not affect Creatio in any way.

Note. Changes made to the configuration object of the `businessRules` property take precedence when you delete or disable a business rule.

Modules (`modules`)

To implement modules, use the `modules` property. Its configuration object declares and configures modules and details loaded on the page. The `/ ** SCHEMA_MODULES * /` marker comments are required since they are necessary for the operation of the Wizards.

Note. The `details` property loads a detail to a page. Since a detail is also a module, we recommend using the `modules` property instead.

Example that uses the `modules` property

```
modules: /**SCHEMA_MODULES*/{
    /* Load the module.
    Module title. Must be the same as the name property in the diff array. */
    "TestModule": {
        /* Optional. The ID of the module to load. If not specified, Creatio will generate it au
        "moduleId": "myModuleId",.
        /* If no parameter is specified, Creatio will use BaseSchemaModuleV2 for loading. */
        "moduleName": "MyTestModule",
        /* Configuration object. When the module is loaded, the object is passed as instanceConf
        "config": {
            "isSchemaConfigInitialized": true,
            "schemaName": "MyTestSchema",
            "useHistoryState": false,
            /* Additional module parameters. */
            "parameters": {
                /* Parameters passed to the schema during the initialization. */
                "viewModelConfig": {
                    masterColumnName: "PrimaryContact"
                }
            }
        }
    },

    /* Load the detail.
    Detail name. */
```

```
    "Project": {
            /* Detail schema name. */
            "schemaName": "ProjectDetailV2",
            "filter": {
            /* The column of the section object schema. */
            "masterColumn": "Id",
            /* The column of a detail object schema. */
            "detailColumn": "Opportunity"
        }
    }
}/**SCHEMA_MODULES*/
```

Array of modifications (`diff`)

To implement an array of modifications, use the `diff` property, which contains an array of configuration objects. The **purpose** of the array of modifications is to build a representation of the module in the Creatio interface. Each element in the array represents metadata from which Creatio generates various interface controls. The `/**SCHEMA_DIFF*/` marker comments are required since they are necessary for the operation of the Wizards.

The alias mechanism

When developing new versions, you sometimes need to move page elements to new zones. In situations where users have customized the record page, such changes can have unpredictable consequences. The `alias` **mechanism** interacts with the `diff` builder to provide partial backward compatibility when changing the UI in new product versions. The builder is the `json-applier` class that merges base schema and client extension schema parameters.

The `alias` property contains data about the previous name of the element. Creatio creates the `diff` array of modifications based on that data, considering not only elements with a new name but also with the name specified in `alias`. In essence, `alias` is a configuration object that links the new and old elements. When creating a `diff` array of modifications, the `alias` configuration object can disallow application of some properties and operations to the element where it is declared. You can add the `alias` object to any element in the `diff` array of modifications.

Relationship between a view and model

The **purpose** of the `bindTo` property is to indicate the relationship between a view model attribute and a view object property.

Declare the property in the `values` property of the configuration objects in the `diff` array of modfications.

View an example that uses the `bindTo` property below.

Example that uses the `bindTo` property

```
diff: [
    {
        "operation": "insert",
```

```
        "parentName": "CombinedModeActionButtonsCardLeftContainer",
        "propertyName": "items",
        "name": "MainContactButton",
        /* Properties passed to the component's constructor. */
        "values": {
            /* Set the type of the added element to button. */
            "itemType": Terrasoft.ViewItemType.BUTTON,
            /* Bind the button title to the localizable schema string. */
            "caption": {bindTo: "Resources.Strings.OpenPrimaryContactButtonCaption"},
            /* Bind the button click handler method. */
            "click": {bindTo: "onOpenPrimaryContactClick"},
            /* The display style of the button. */
            "style": Terrasoft.controls.ButtonEnums.style.GREEN,
            /* Bind the button availability property. */
            "enabled": {bindTo: "ButtonEnabled"}
        }
    }
]
```

Tabs are objects that contain the `tabs` value in their `propertyName` property.

Creatio implements an alternative way to use the `bindTo` property for **tab titles**.

Alternative way to use the `bindTo` property

```
...
{
    "operation": "insert",
    "name": "GeneralInfoTab",
    "parentName": "Tabs",
    /* Imply that the object is a tab. */
    "propertyName": "tabs",
    "index": 0,
    "values": {
        /* $ replaces usage of bindTo: {...}. */
        "caption": "$Resources.Strings.GeneralInfoTabCaption",
        "items": []
    }
},
...
```

`diff` property declaration rules

- **Best use of converters**.

  A **converter** is a function executed in the `viewModel` environment. A converter accepts the values of the

`viewModel` property and returns a result of the corresponding type. To ensure that Wizards operate correctly, format the `diff` property value in JSON. Therefore, the value of the converter must be the name of the view model method rather than an inline function.

Correct use of the converter

```
methods: {
    someFunction: function(val) {
        /* ... */
    }
},

diff: /**SCHEMA_DIFF*/[
  {
      /* ... */
      "bindConfig": {
          "converter": "someFunction"
      }
      /* ... */
  }
]/**SCHEMA_DIFF*/
```

Incorrect use of the converter

```
diff: /**SCHEMA_DIFF*/[
    {
        /* ... */
        "bindConfig": {
            "converter": function(val) {
            /* ... */
            }
        }
    }
]/**SCHEMA_DIFF*/
```

Correct use of the generator

```
methods: {
    someFunction: function(val) {
        /* ... */
    }
},
```

```
diff: /**SCHEMA_DIFF*/[
  {
      /* ... */
      "values": {
          "generator": "someFunction"
      }
      /* ... */
  }
]/**SCHEMA_DIFF*/
```

Incorrect use of the generator

```
diff: /**SCHEMA_DIFF*/[
    {
        /* ... */
        "values": {
            "generator": function(val) {
                /* ... */
            }
        }
    }
]/**SCHEMA_DIFF*/
```

- **Parent element**.

  The **parent element (container)** is the DOM element where the module renders its view. To ensure that the Wizards operate as intended, place a single child element in the parent container.

  Example of the correct placement of a view in the parent element

  ```
  <div id="OpportunityPageV2Container" class="schema-wrap one-el" data-item-marker="Opportunity
      <div id="CardContentWrapper" class="card-content-container page-with-left-el" data-item-m
  </div>
  ```

  Example of incorrect placement of a view in the parent element

  ```
  <div id="OpportunityPageV2Container" class="schema-wrap one-el" data-item-marker="Opportunity
      <div id="CardContentWrapper" class="card-content-container page-with-left-el" data-item-m
      <div id="DuplicateContainer" class="DuplicateContainer"></div>
  </div>
  ```

  When you add, change, move an element (`insert`, `merge`, `move` operations), specify the `parentName`

property (the parent element's name) in the `diff` property.

Example of the correct definition of the view element in the diff property

```
{
  "operation": "insert",
  "name": "SomeName",
  "propertyName": "items",
  "parentName": "SomeContainer",
  "values": {}
}
```

Example of incorrect definition of the view element in the diff property

```
{
  "operation": "insert",
  "name": "SomeName",
  "propertyName": "items",
  "values": {}
}
```

If the `parentName` property is missing, the Wizard will be unable to configure the page. Creatio will display a corresponding error message.

The value of the `parentName` property must match the name of the parent element in the corresponding base page schema. For example, this is `CardContentContainer` for record pages.

If you specify the name of a non-existent container element as the parent element in the `parentName` property, a "`Schema cannot have more than one root object`" error will occur, since the added element will be placed in the root container.

- **Unique names.**

  Each element in the `diff` array must have a unique name.

Example of the correct addition of elements to a diff array

```
{
  "operation": "insert",
  "name": "SomeName",
  "values": { }
},
{
  "operation": "insert",
  "name": "SomeSecondName",
  "values": { }
```

```
}
```

Example of incorrect addition of elements to a diff array

```
{
  "operation": "insert",
  "name": "SomeName",
  "values": { }
},
{
  "operation": "insert",
  "name": "SomeName",
  "values": { }
}
```

- **Placement of view elements**.

  To ensure the view elements are customizable and changeable, place them on the **layout grid**. In Creatio, each row of the layout grid has 24 cells (columns). Use the `layout` property to place elements on the grid.

  Grid element **properties**:

  - `column`. The index of the left column.
  - `row`. The index of the top row.
  - `colSpan`. The number of spanned columns.
  - `rowSpan`. The number of spanned rows.

  Example that places elements

```
{
  "operation": "insert",
  "parentName": "ParentContainerName",
  "propertyName": "items",
  "name": "ItemName",
  "values": {
    /* Element placement. */
    "layout": {
        /* Start at column zero. */
        "column": 0,
        /* Place in the fifth row of the grid. */
        "row": 5,
        /* Span 12 columns wide. */
        "colSpan": 12,
        /* Occupy one row. */
        "rowSpan": 1
```

```
      },
      "contentType": Terrasoft.ContentType.ENUM
    }
  }
```

- **Number of operations**.

  If you change the client schema without a Wizard, we recommend adding no more than one operation per schema element to ensure that the Wizard operates as intended.

Properties (properties)

To implement properties, use the `properties` property, which contains a JavaScript object.

View an example that uses the `properties` property in the `SectionTabsSchema` schema of the `NUI` package below.

Example that uses the `properties` property

```
define("SectionTabsSchema", [],
    function() {
        return {
            ...
            /* Declare the properties property. */
            properties: {
                /* The parameters property. Array. */
                parameters: [],
                /* modulesContainer property. Entity. */
                modulesContainer: {}
            },
            methods: {
                ...
                /* Initialization method. Always executed first. */
                init: function(callback, scope) {
                    ...
                    /* Call the method uses the view model properties. */
                    this.initContainers();
                    ...
                },
                ...
                /* The method where to use the properties. */
                initContainers: function() {
                    /* Use the modulesContainer property. */
                    this.modulesContainer.items = [];
                    ...
                    /* Use 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)[
                        this.modulesContainer.items.push(container);
                    }, this);
                },
                ...
            },
        ...
        }
    });
```

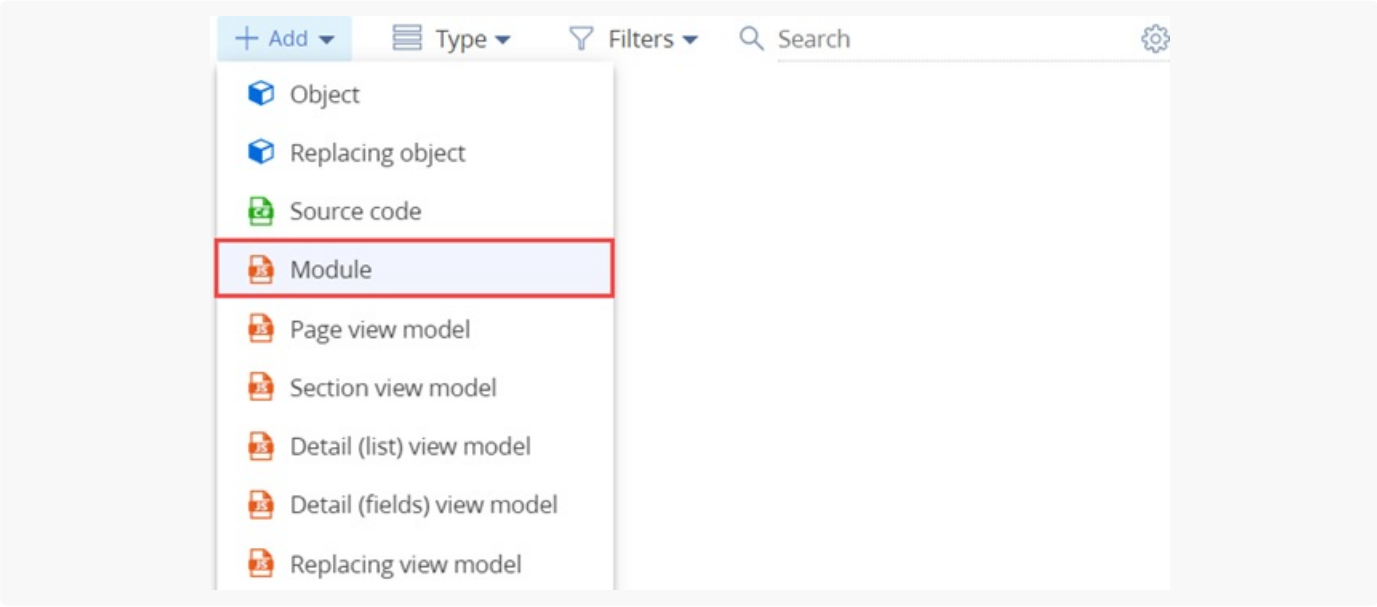
Overload a mixin method

Beginner

Example. Connect the `ContentImageMixin` mixin to the custom schema, override the mixin method.

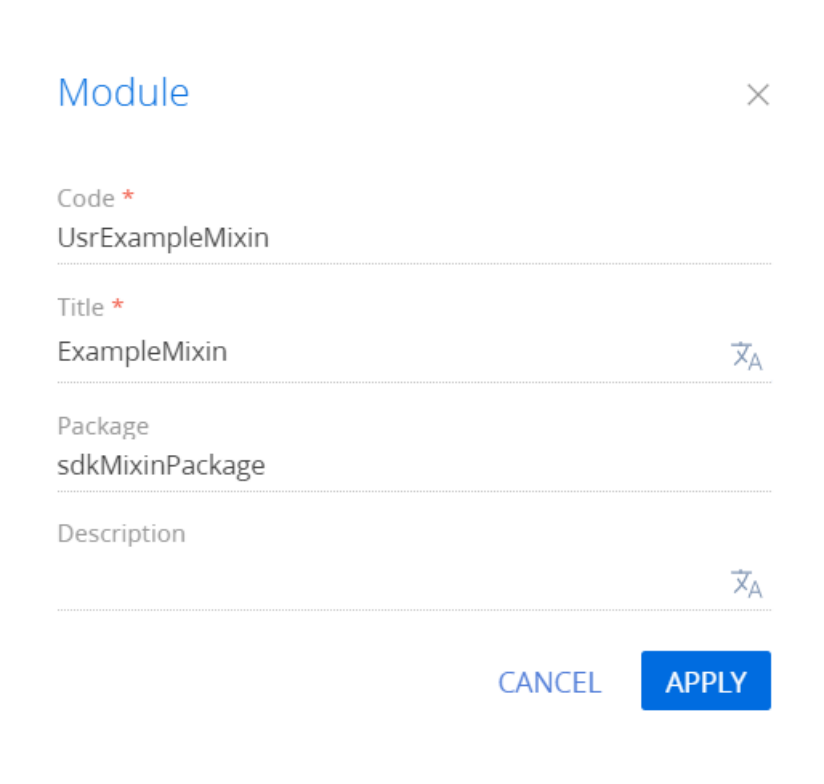
1. Create a mixin

1. Go to the [*Configuration*] section and select a custom package to add the schema.
2. Click [*Add*] → [*Module*] on the section list toolbar.



3. Fill out the schema properties in the Schema Designer.

   - Set [*Code*] to "`UsrExampleMixin`."
   - Set [*Title*] to "`ExampleMixin`."



Click [*Apply*] to apply the properties.

4. Add the source code in the Schema Designer.

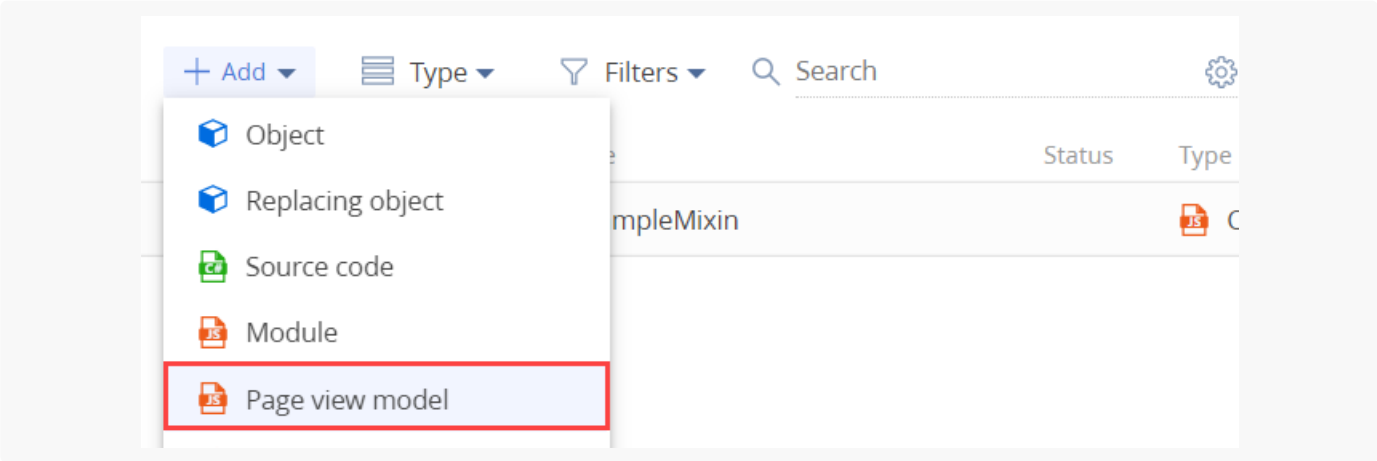
Module source code

```
/* Define the module. */
define("ContentImageMixin", [ContentImageMixinV2Resources], function() {
    /* Define the ContentImageMixin class. */
    Ext.define("Terrasoft.configuration.mixins.ContentImageMixin", {
        /* Alias (shorthand for the class name). */
        alternateClassName: "Terrasoft.ContentImageMixin",
        /* Mixin functionality. */
        getImageUrl: function() {
            var primaryImageColumnValue = this.get(this.primaryImageColumnName);
            if (primaryImageColumnValue) {
                return this.getSchemaImageUrl(primaryImageColumnValue);
            } else {
                var defImageResource = this.getDefaultImageResource();
                return this.Terrasoft.ImageUrlBuilder.getUrl(defImageResource);
            }
        }
    });
    return Ext.create(Terrasoft.ContentImageMixin);
});
```

5. Click [*Save*] on the Designer toolbar.

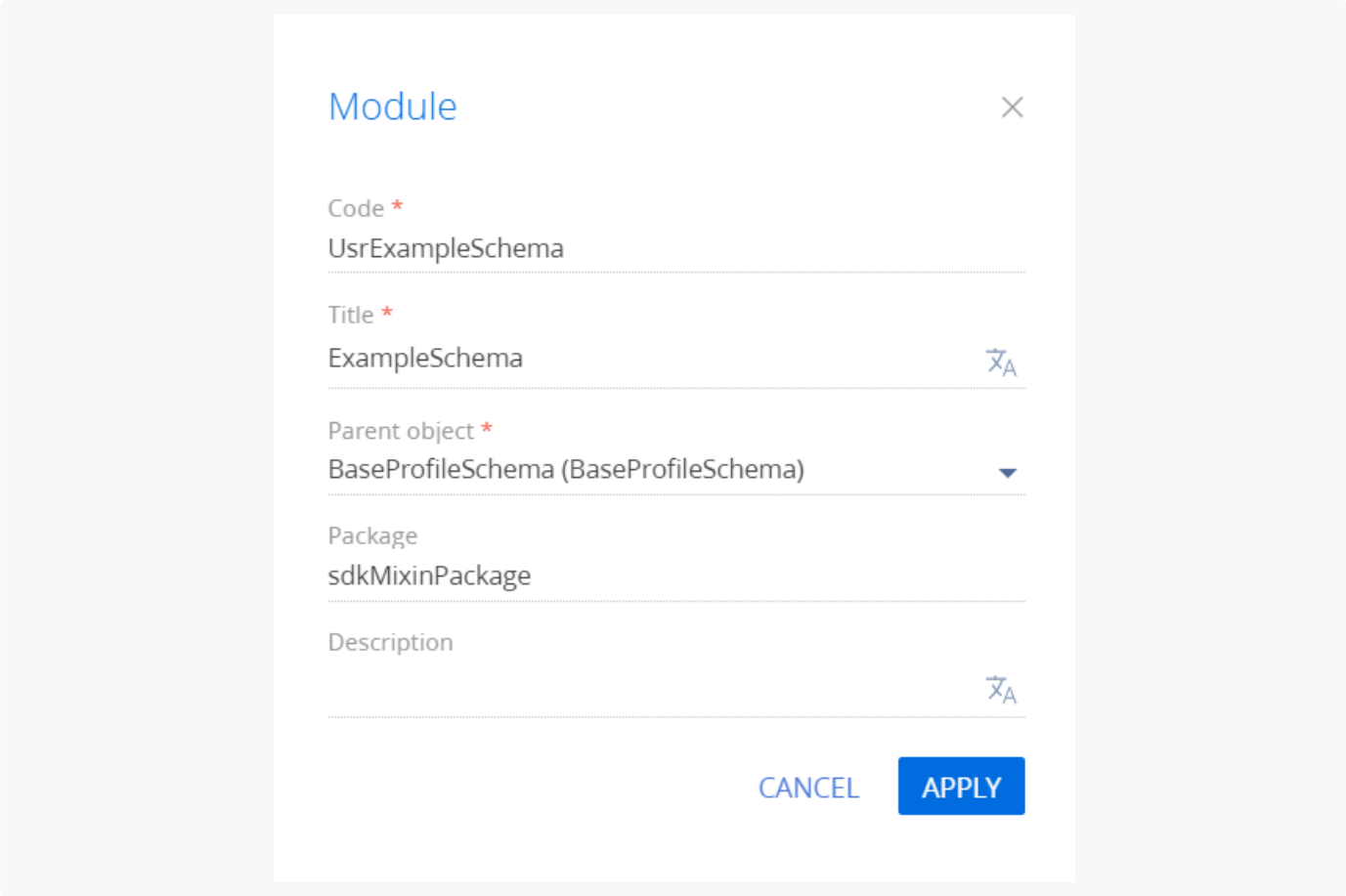
2. Connect the mixin

1. Go to the [*Configuration*] section and select a custom package to add the schema.
2. Click [*Add*] → [*Page view model*] on the section list toolbar.



3. Fill out the schema properties in the Schema Designer.

   - Set [*Code*] to "`UsrExampleSchema`."
   - Set [*Title*] to "`ExampleSchema`."
   - Set [*Parent object*] to "`BaseProfileSchema`."



Click [*Apply*] to apply the properties.

To use the mixin, enable it in the `mixins` block of the `ExampleSchema` custom schema.

3. Overload the mixin method

Add the source code in the Schema Designer. In the `method` block, override the `getReadImageURL()` mixin method. Use the overridden function in the `diff` block.

Module source code

```
/* Declare the module. Include as a dependency the ContentImageMixin module, in which the mixin
define("UsrExampleSchema", ["ContentImageMixin"], function() {
    return {
        entitySchemaName: "ExampleEntity",
        mixins: {
            /* Connect the mixin to the schema. */
            ContentImageMixin: "Terrasoft.ContentImageMixin"
        },
        details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
        diff: /**SCHEMA_DIFF*/[
            {
                "operation": "insert",
```

```
                "parentName": "AddRightsItemsHeaderImagesContainer",
                "propertyName": "items",
                "name": "AddRightsReadImage",
                "values": {
                    "classes": {
                        "wrapClass": "rights-header-image"]
                    },
                    "getSrcMethod": "getReadImageUrl",
                    "imageTitle": resources.localizableStrings.ReadImageTitle,
                    "generator": "ImageCustomGeneratorV2.generateSimpleCustomImage"
                }
            }]/**SCHEMA_DIFF*/,
        methods: {
            getReadImageUrl: function() {
                /* Custom implementation. */
                console.log("Contains custom logic");
                /* Call the mixin method. */
                this.mixins.ContentImageMixin.getImageUrl.apply(this, arguments);
            }
        },
        rules: {}
    };
});
```

Click [*Save*] on the Designer toolbar.

Method declaration example

Medium

Example. Add the [*Email*] column validation logic to the logic of the `setValidationConfig` method located in the `Terrasoft.configuration.BaseSchemaViewModel` class.

Protected method example

```
methods: {
    /* Method name. */
    setValidationConfig: function() {
        /* Call the setValidationConfig method of the parent schema. */
        this.callParent(arguments);
        /* Set up validation for the [Email] column. */
        this.addColumnValidator("Email", EmailHelper.getEmailValidator);
    }
}
```

New method example

```
methods: {
    /* Method name. */
    getBlankSlateHeaderCaption: function() {
        /* Get the value of the MasterColumnInfo column. */
        var masterColumnInfo = this.get("MasterColumnInfo");
        /* Return the result value of the method. */
        return masterColumnInfo ? masterColumnInfo.caption : "";
    },
    /* Method name. */
    getBlankSlateIcon: function() {
        /* Return the result value of the method. */
        return this.Terrasoft.ImageUrlBuilder.getUrl(this.get("Resources.Images.BlankSlateIcon")
    }
}
```

Array of modifications usage example

Medium

```
diff: /**SCHEMA_DIFF*/[
  {
      "operation": "insert",
      "name": "CardContentWrapper",
      "values": {
          "id": "CardContentWrapper",
          "itemType": Terrasoft.ViewItemType.CONTAINER,
          "wrapClass": "card-content-container"],
          "items": []
      }
  },
  {
      "operation": "insert",
      "name": "CardContentContainer",
      "parentName": "CardContentWrapper",
      "propertyName": "items",
      "values": {
          "itemType": Terrasoft.ViewItemType.CONTAINER,
          "items": []
      }
  },
```

```
  {
      "operation": "insert",
      "name": "HeaderContainer",
      "parentName": "CardContentContainer",
      "propertyName": "items",
      "values": {
          "itemType": Terrasoft.ViewItemType.CONTAINER,
          "wrapClass": ["header-container-margin-bottom"],
          "items": []
      }
  },
  {
      "operation": "insert",
      "name": "Header",
      "parentName": "HeaderContainer",
      "propertyName": "items",
      "values": {
          "itemType": Terrasoft.ViewItemType.GRID_LAYOUT,
          "items": [],
          "collapseEmptyRow": true
      }
  }
]/**SCHEMA_DIFF*/
```

Example of using the alias mechanism for repeated schema replacement

Medium

The `diff` array of modifications has an initial "`Name`" element with a set of properties. The element is located in the `Header` container. This schema is replaced several times. The "`Name`" element is modified and moved freely.

diff property of the base schema

```
diff: /**SCHEMA_DIFF*/ [
    {
        /* Insert operation. */
        "operation": "insert",
        /* The name of the parent element into which to insert the element. */
        "parentName": "Header",
        /* The name of the parent element property on which to operate. */
        "propertyName": "items",
        /* The name of the element. */
        "name": "Name",
        /* The object of element property values. */
```

```
        "values": {
            /* Layout. */
            "layout": {
                /* Column number. */
                "column": 0,
                /* The row number. */
                "row": 1,
                /* The number of combined columns. */
                "colSpan": 24
            }
        }
    }
] /**SCHEMA_DIFF*/
```

diff property after the first replacement of the base schema

```
diff: /**SCHEMA_DIFF*/ [
    {
        /* The operation that combines the properties of two elements. */
        "operation": "merge",
        "name": "Name",
        "values": {
            "layout": {
                "column": 0,
                /* The row number. The element has been moved. */
                "row": 8,
                "colSpan": 24
            }
        }
    }
] /**SCHEMA_DIFF*/
```

diff property after the second replacement of the base schema

```
diff: /**SCHEMA_DIFF*/ [
    {
        /* Move operation. */
        "operation": "move",
        "name": "Name",
        /* The name of the parent element where the move operation is done. */
        "parentName": "SomeContainer"
    }
] /**SCHEMA_DIFF*/
```

In the new version, the element named "Name" has been moved from the `SomeContainer` element to the `ProfileContainer` element and must remain there despite the client customization. To enforce this, the element gets a new name "NewName" and the `alias` configuration object is added to it.

```
diff: /**SCHEMA_DIFF*/ [
    {
        /* Insert operation. */
        "operation": "insert",
        /* The name of the parent element into which to insert the element. */
        "parentName": "ProfileContainer",
        /* The name of the parent element property on which to operate. */
        "propertyName": "items",
        /* New name. */
        "name": "NewName",
        /* The object of element property values . */
        "values": {
            /* Bind to the property or function value. */
            "bindTo": "Name",
            /* Layout. */
            "layout": {
                /* Column number. */
                "column": 0,
                /* The row number. */
                "row": 0,
                /* The number of combined columns. */
                "colSpan": 12
            }
        },

        /* alias configuration object. */
        "alias": {
            /* The old name of the element. */
            "name": "Name",
            /* An array of custom replacing properties to ignore. */
            "excludeProperties": "layout" ],
            /* An array of custom replacing operations to ignore. */
            "excludeOperations": [ "remove", "move" ]
        }
    }
] /**SCHEMA_DIFF*/
```

The new element now has `alias`. The parent element changed, as is its location on the record page. The `excludeProperties` property stores a set of properties that will be ignored when the delta is applied, while `excludeOperations` stores a set of operations that will not be applied to this element from the replacements.

In this example, the `layout` properties of all "Name" descendants are excluded, and the `remove` and `move` operations are also prohibited. This means that the "NewName" element will contain only the root `layout` property

and all properties of the "Name" element from the replacements except `Layout`. The same applies to operations.

Result for the builder of the `diff` array of modifications

```
diff: /**SCHEMA_DIFF*/ [
    {
        /* Insert operation. */
        "operation": "insert",
        /* The name of the parent element into which to insert the element. */
        "parentName": "ProfileContainer",
        /* The name of parent element property on which to operate. */
        "propertyName": "items",
        /* New name. */
        "name": "NewName",
        /* The object of element property values. */
        "values": {
            /* Bind to the property or function value. */
            "bindTo": "Name",
            /* Layout. */
            "layout": {
                /* Column number. */
                "column": 0,
                /* The row number. */
                "row": 0,
                /*/ The number of combined columns. */
                "colSpan": 12
            },
        }
    },
] /**SCHEMA_DIFF*/
```

attributes property JS

Beginner

The `attributes` property of the client schema contains a configuration object with its properties.

Primary properties

`dataValueType`

The attribute data type. Creatio will use it when generating the view. The `Terrasoft.DataValueType` enumeration represents the available data types.

Available values (`DataValueType`)

GUID	0
TEXT	1
INTEGER	4
FLOAT	5
MONEY	6
DATE_TIME	7
DATE	8
TIME	9
LOOKUP	10
ENUM	11
BOOLEAN	12
BLOB	13
IMAGE	14
CUSTOM_OBJECT	15
IMAGELOOKUP	16
COLLECTION	17
COLOR	18
LOCALIZABLE_STRING	19
ENTITY	20
ENTITY_COLLECTION	21
ENTITY_COLUMN_MAPPING_COLLECTION	22
HASH_TEXT	23
SECURE_TEXT	24
FILE	25

FILE	25
MAPPING	26
SHORT_TEXT	27
MEDIUM_TEXT	28
MAXSIZE_TEXT	29
LONG_TEXT	30
FLOAT1	31
FLOAT2	32
FLOAT3	33
FLOAT4	34
LOCALIZABLE_PARAMETER_VALUES_LIST	35
METADATA_TEXT	36
STAGE_INDICATOR	37

`type`

Column type. An optional parameter `BaseViewModel` uses internally. The `Terrasoft.ViewModelColumnType` enumeration represents the available column types.

Available values (`ViewModelColumnType`)

ENTITY_COLUMN	0
CALCULATED_COLUMN	1
VIRTUAL_COLUMN	2
RESOURCE_COLUMN	3

`value`

Attribute value. Creatio sets the view model value to this parameter when the view model is created. The `value` attribute accepts numeric, string, and boolean values. If the attribute type implies the use of a lookup

type value (array, object, collection, etc.), initialize its initial value using a method.

Use example

Example that uses basic attribute properties

```
attributes: {
    /* Attribute name. */
    "NameAttribute": {
        /* Data type. */
        "dataValueType": this.Terrasoft.DataValueType.TEXT,
        /* Column type. */
        "type": this.Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN,
        /* The default value. */
        "value": "NameValue"
    }
}
```

Additional properties

`caption`

Attribute title.

`isRequired`

The flag that marks the attribute as required.

`dependencies`

Dependency on another attribute of the model. For example, set an attribute based on the value of another attribute. Use the property to create calculated fields.

`lookupListConfig`

The property that manages the lookup field properties. Learn more about using this parameter in a separate article: Filter the lookup field. This is a configuration object that can contain optional properties.

Optional properties

`columns`	An array of column names to add to a request in addition to the [*Id*] column and the primary display column.
`orders`	An array of configuration objects that determine the data sorting.
`filter`	The method that returns an instance of the `Terrasoft.BaseFilter` class or its descendant that will be applied to the request. Cannot be used combined with the `filters` property.
`filters`	An array of filters (methods that return collections of the `Terrasoft.FilterGroup` class). Cannot be used combined with the `filter` property.

Use example

Example that uses additional attribute properties

```
attributes: {
    /* Attribute name. */
    "Client": {
        /* Attribute title. */
        "caption": { "bindTo": "Resources.Strings.Client" },
        /* The attribute is required. */
        "isRequired": true
    },

    /* Attribute name. */
    "ResponsibleDepartment": {
        lookupListConfig: {
            /* Additional columns. */
            columns: "SalesDirector",
            /* Sorting column. */
            orders: [ { columnPath: "FromBaseCurrency" } ],
            /* Filter definition function. */
            filter: function()
            {
                /* Return a filter by the [Type] column, which equals the Competitor consta
                return this.Terrasoft.createColumnFilterWithParameter(
                this.Terrasoft.ComparisonType.EQUAL,
                "Type",
                ConfigurationConstants.AccountType.Competitor);
            }
        }
    },
    /* Attribute name. */
    "Probability": {
        /* Define the column dependency. */
```

```
        "dependencies": [
        {
            /* Depends on the Stage] column. */
            "columns": [ "Stage" ],
            /* The name of the [Stage] column's change handler method.
            The setProbabilityByStage() method is defined in the methods property of the s
            "methodName": "setProbabilityByStage"
        }
        ]
    }
},
methods: {
    /* [Stage] column's change handler method. */
    setProbabilityByStage: function()
    {
        /* Get the value of the [Stage] column. */
        var stage = this.get("Stage");
        /* Condition for changing the [Probability] column. */
        if (stage.value && stage.value ===
        ConfigurationConstants.Opportunity.Stage.RejectedByUs)
        {
            /* Set the value of the [Probability] column. */
            this.set("Probability", 0);
        }
    }
}
```

messages property

Medium

The `messages` property of the client schema contains a configuration object with its properties.

Properties

`mode`

Message mode. The `Terrasoft.MessageMode` enumeration represents the available modes.

Available values (`MessageMode`)

`PTP`	Address.
`BROADCAST`	Broadcasting.

`direction`

Message direction. The `Terrasoft.MessageDirectionType` enumeration represents the available modes.

Available values (`MessageDirectionType`)

`PUBLISH`	Publishing.
`SUBSCRIBE`	Subscription.
`BIDIRECTIONAL`	Bidirectional.

rules and businessRules properties JS

Beginner

The `rules` and `businessRules` properties of the client schema contain a configuration object with its own properties.

Primary properties

`ruleType`

Rule type. Defined by the `BusinessRuleModule.enums.RuleType` enumeration value.

Available values (`BusinessRuleModule.enums.RuleType`)

`BINDPARAMETER`	Business rule type. Use this rule type to link properties of a column to values of different parameters. For example, set up the visibility of a column or enable a column depending on the value of another column.
`FILTRATION`	Business rule type. Use the FILTRATION rule to set up filtering of values in view model columns. For example, filter a `LOOKUP` column depending on the current status of a page.

`property`

Use for the `BINDPARAMETER` business rule type. Control property. Set by the `BusinessRuleModule.enums.Property` enumeration value.

Available values (`BusinessRuleModule.enums.Property`)

`VISIBLE`	Whether visible.
`ENABLED`	Whether available.
`REQUIRED`	Whether required.
`READONLY`	Whether read-only.

`conditions`

Use for the `BINDPARAMETER` business rule type. Condition array for rule application. Each condition is a configuration object.

Properties of the configuration object

<table>
<tr><td><code>leftExpression</code></td><td>Expression of the left side of the condition. Represented by a configuration object.
Properties of the configuration object

<code>type</code>

The expression type. Set by the <code>BusinessRuleModule.enums.ValueType</code> enumeration value.

Available values (<code>BusinessRuleModule.enums.ValueType</code>)

<table><tr><td><code>CONSTANT</code></td><td>A constant.</td></tr><tr><td><code>ATTRIBUTE</code></td><td>The value of the view model column.</td></tr><tr><td><code>SYSSETTING</code></td><td>System setting.</td></tr><tr><td><code>SYSVALUE</code></td><td>A system value. The list element of the <code>Terrasoft.core.enums.SystemValueType</code> system values.</td></tr></table>
<code>attribute</code>

Name of the model column.

<code>attributePath</code>

Meta-path to the lookup schema column

<code>value</code>

Comparison value.</td></tr>
<tr><td><code>comparisonType</code></td><td>Type of comparison. Set by the <code>Terrasoft.core.enums.ComparisonType</code> enumeration value.</td></tr>
<tr><td><code>rightExpression</code></td><td>Expression of the right side of the condition. Similar to <code>leftExpression</code>.</td></tr>
</table>

`logical`

Use for the `BINDPARAMETER` business rule type. The logical operation that combines the conditions from the conditions property. Set by the `Terrasoft.LogicalOperatorType` enumeration value.

`autocomplete`

Use for the `FILTRATION` business rule type. Reverse filtering flag. Can be `true` or `false`.

`autoClean`

Use for the `FILTRATION` business rule type. The flag that enables automated value cleanup when the column by which to filter changes. Can be `true` or `false`.

`baseAttributePatch`

Use for the `FILTRATION` business rule type. Meta-path to the lookup schema column that will be used for filtering. Apply the feedback principle when building the column path, similar to `EntitySchemaQuery`. Generate the path relative to the schema to which the model column links.

`comparisonType`

Use for the `FILTRATION` business rule type. Type of comparison operation. Set by the `Terrasoft.ComparisonType` enumeration value.

`type`

Use for the `FILTRATION` business rule type. The value type for comparison `baseAttributePatch`. Set by the `BusinessRuleModule.enums.ValueType` enumeration value.

`attribute`

Use for the `FILTRATION` business rule type. The name of the view model column. Describe this property if the `ATTRIBUTE` value `type` is indicated.

`attributePath`

Use for the `FILTRATION` business rule type. Meta-path to the object schema column. Apply the feedback principle when building the column path, similar to `EntitySchemaQuery`. Generate the path relative to the schema to which the model column link.

`value`

Use for the `FILTRATION` business rule type. Filtration value. Describe this property if the `ATTRIBUTE` value `type` is indicated.

Additional properties

Use additional properties only for the `businessRules` property.

`uId`.

Unique rule ID. The "`GUID`" type value.

`enabled`

Enabling flag. Can be `true` or `false`.

`removed`

The flag that indicates whether the rule is removed. Can be `true` or `false`.

`invalid`

The flag that indicates whether the rule is valid. Can be `true` or `false`.

Use examples

Example of a BINDPARAMETER business rule created by the Wizard

```
define("SomePage", [], function() {
    return {
        /* ... */
        businessRules: /**SCHEMA_BUSINESS_RULES*/{
        /* A set of rules for the Type column of the view model. */
        "Type": {
            /* The rule code the Wizard generates. */
            "ca246daa-6634-4416-ae8b-2c24ea61d1f0": {
                /* Unique rule ID. */
                "uId": "ca246daa-6634-4416-ae8b-2c24ea61d1f0",
                /* Enabling flag. */
                "enabled": true,
                /* The flag that indicates whether the rule is removed. */
                "removed": false,
                /* The checkbox that indicates whether the rule is valid. */
                "invalid": false,
                /* Rule type. */
                "ruleType": 0,
                /* The code for the property that controls the rule. */
                "property": 0,
                /* A logical relationship between several rule conditions. */
                "logical": 0,
                /* An array of conditions that trigger the rule.
```

```
                Compare the Account.PrimaryContact.Type value to the Type column value. */
                 "conditions": [
                     {
                     /* Type of comparison operation. */
                     "comparisonType": 3,
                     /* Expression of the left side of the condition. */
                     "leftExpression": {
                         /* The expression type is a column (attribute) of a view model. */
                         "type": 1,
                         /* The name of the view model column. */
                         "attribute": "Account",
                         /* Path to the column in the Account lookup schema, whose value to
                         "attributePath": "PrimaryContact.Type"
                     },
                     /* Expression of the right side of the condition. */
                     "rightExpression": {
                         /* The expression type is a column (attribute) of a view model. */
                         "type": 1,
                         /* The name of the view model column. */
                         "attribute": "Type"
                     }
                     }
                 ]
             }
        }
        }/**SCHEMA_BUSINESS_RULES*/
        /* ... */
    };
});
```

Example of a FILTRATION business rule created by the Wizard

```
define("SomePage", [], function() {
    return {
        /* ... */
        businessRules: /**SCHEMA_BUSINESS_RULES*/{
        /* A set of rules for the Type column of the view model. */
        "Account": {
            /* The rule code the Wizard generates. */
            "a78b898c-c999-437f-9102-34c85779340d": {
                /* Unique rule ID. */
                "uId": "a78b898c-c999-437f-9102-34c85779340d",
                /* Enabling flag. */
                "enabled": true,
                /* The flag that indicates whether the rule is removed. */
                "removed": false,
                /* The flag that indicates whether the rule is valid. */
```

```
                "invalid": false,
                /* Rule type. */
                "ruleType": 1,
                /* Path to the column for filtering in the Account lookup schema referenced
                "baseAttributePatch": "PrimaryContact.Type",
                /* The comparison type in the filter. */
                "comparisonType": 3,
                /* Expression type is a column (attribute) of a view model. */
                "type": 1,
                /* Name of the view model column, by which to filter the records. */
                "attribute": "Type"
            }
        }
        }/**SCHEMA_BUSINESS_RULES*/
        /* ... */
    };
});
```

diff property JS

Medium

The `diff` property of the client schema contains an array of configuration objects with their properties.

Properties

`operation`

Operation on elements.

Available values

`set`	Setsthe schema element to the value determined by the `values` parameter.
`merge`	Merges the values from the parent, replaced, and replacing schemas. The properties from the `values` parameter of the last descendant override the other values.
`remove`	Deletes the element from the schema.
`move`	Moves the element to a different parent element.
`insert`	Adds the element to the schema.

`name`

The name of the schema element on which the operation is run.

`parentName`

The name of the parent schema element where to place the element during the `insert` operation, or to which to move the element during the `move` operation.

`propertyName`

The name of the parent element parameter for the `insert` operation. Also used in the `remove` operation when only certain element parameters must be removed rather than the entire element.

`index`

The index to which to move or add the parameter. Use the parameter with the `insert` and `move` operations. If the parameter is not specified, then insert is the last element of the array.

`values`

An object whose properties to set or combine with the properties of the schema element. Use with `set`, `merge` and `insert` operations.

The `Terrasoft.ViewItemType` enumeration represents the set of basic elements that can be displayed on the page

Available values (`ViewItemType`)

`GRID_LAYOUT`	0	A grid element that includes the placement of other controls.
`TAB_PANEL`	1	A set of tabs.
`DETAIL`	2	Detail
`MODEL_ITEM`	3	View model element.
`MODULE`	4	Module.
`BUTTON`	5	Button.
`LABEL`	6	Caption.
`CONTAINER`	7	Containers.
`MENU`	8	Drop-down list.

MENU_ITEM	9	Drop-down list element.
MENU_SEPARATOR	10	Drop-down list separator.
SECTION_VIEWS	11	Section views.
SECTION_VIEW	12	Section view.
GRID	13	List.
SCHEDULE_EDIT	14	Scheduler.
CONTROL_GROUP	15	Element group.
RADIO_GROUP	16	Switcher group.
DESIGN_VIEW	17	Configurable view.
COLOR_BUTTON	18	Color.
IMAGE_TAB_PANEL	19	A set of tabs with icons.
HYPERLINK	20	Hyperlink.
INFORMATION_BUTTON	21	Info button that has a tooltip.
TIP	22	Tooltip.
COMPONENT	23	Component.
PROGRESS_BAR	30	Indicator.

`alias`

Configuration object.

`alias` object properties

`name`	The name of the element to which the new element is connected. Creatio will use this name to locate the elements in the replaced schemas and to connect the elements with the new element. The `name` property of the `diff` revision array element cannot equal the `alias.name` property.
`excludeProperties`	An property array of the `values` object of the element from the `diff` modification array. The properties will not be applied when building `diff`.
`excludeOperations`	An array of operations that must not be applied to this element when building the `diff` array of modifications.

Use example

Example that uses the `alias` object

```
/* diff array. */
diff: /**SCHEMA_DIFF*/ [
  {
    /* The operation to perform on the element. */
    "operation": "insert",
    /* New name. */
    "name": "NewElementName",
    /* Element value. */
    "values": {
        /* ... */
    },
    /* alias configuration object. */
    "alias": {
        /* The previous name of the element. */
        "name": "OldElementName",
        /* An array of excluded properties. */
        "excludeProperties": "layout", "visible", "bindTo" ],
        /* An array of ignored operations. */
        "excludeOperations": [ "remove", "move", "merge" ]
    }
  },
  /* ... */
]
```

Controls

Medium

Controls are objects that organize interaction between the user and Creatio. For example, buttons, details, fields, etc. Usually, navigation bars, dialog boxes, and toolbars display controls.

`Terrasoft.controls.Component` is the parent class for controls. `Terrasoft.BaseObject` class is the parent class for the `Component` class. The `Bindable` mixin of the `Component` class lets you bind control properties to the needed view model properties, methods, or attributes.

Declare events in the control to ensure it operates as intended. Control inherits the following **events** from the `Component` class:

- `added`. Triggered after the control is added to the container.
- `afterrender`. Triggered after the control is added and the HTML view is added to the DOM.
- `afterrerender`. Triggered after the control is rendered and the HTML view is updated in the DOM.
- `beforererender`. Triggered before the control is rendered and the HTML view is added to the DOM.
- `destroy`. Triggered before the control is deleted.
- `destroyed` Triggered after the control is deleted.
- `init`. Triggers after the control is initialized.

Learn more about the `Component` class events in the JS classes reference.

Control can subscribe to browser events and define its own events.

Implement the control in the `diff` array of modifications.

Array of modifications (`diff`)

```
/* diff array of modifications. */
diff: [{
    /* Run the operation that inserts the control into the page. */
    "operation": “insert”,
    /* The name of the parent element to insert the control. */
    "parentName": "CardContentContainer",
    /* The property of the parent element with which to execute the operation. */
    "propertyName": "items",
    /* The meta name of the button to add. */
    "name": "ExampleButton",
    /* Control values.
    Properties to pass to the element constructor. */
    "values": {
        /* Set the type of the added element to button. */
        "itemType": "Terrasoft.ViewItemType.BUTTON",
        /* The button title. */
        "caption": "ExampleButton",
        /* Bind the handler method of the button click. */
        "click": {"bindTo": "onExampleButtonClick"},
        /* The display style of the button. */
        "style": Terrasoft.controls.ButtonEnums.style.GREEN
    }
}]
```

The template (`template <tpl>`) defines the control appearance. Creatio generates the control view in the page view while rendering the control to the page view. Generation is based on the specified template.

The control element has no business logic. The module where you add the control implements the business logic.

The control has `styles` and `selectors` attributes that are defined in the `Component` parent class. These attributes let you customize styles flexibly.

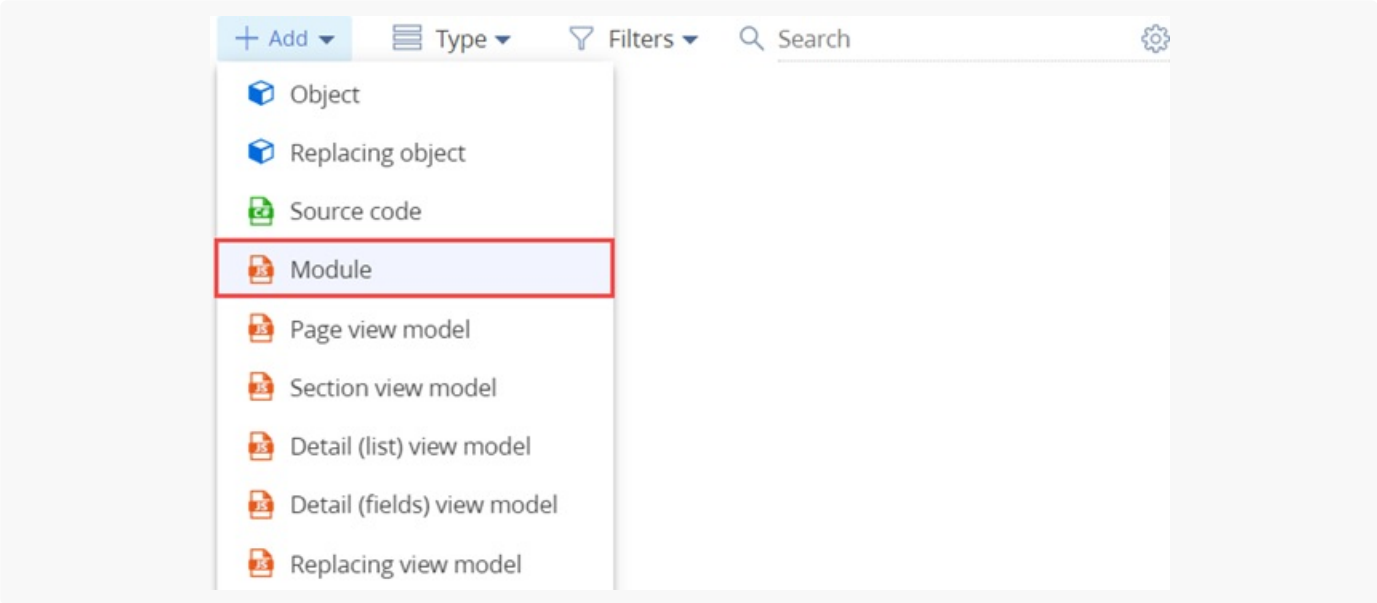
Add a custom control to a record page

Advanced

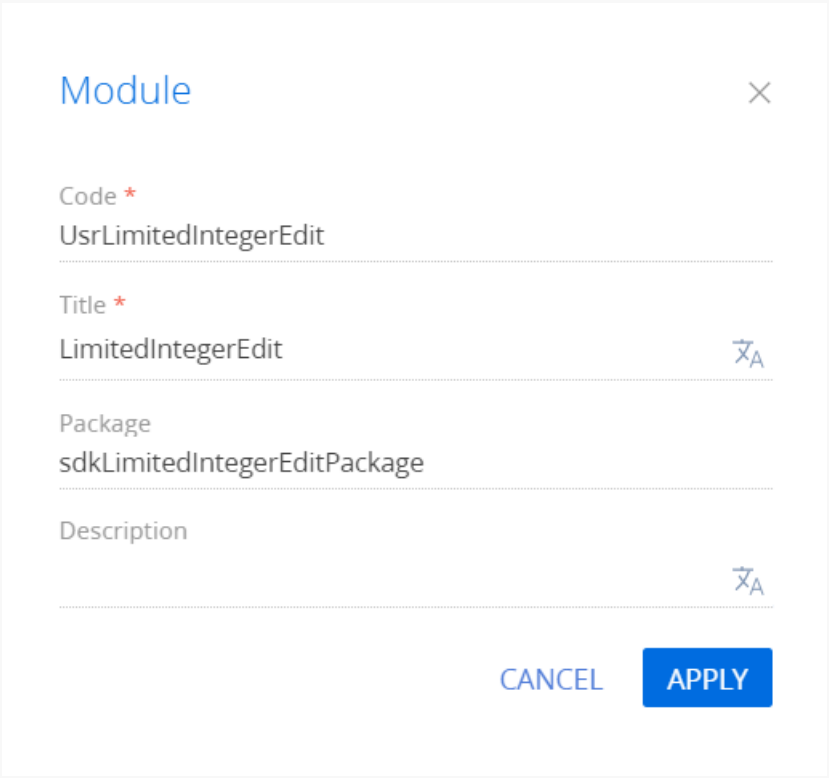
Example. Add a custom control to a contact page. The control must take integers. Upon pressing `Enter`, check the entered value and display a message if the integer is out of the `[-300; 300]` range. Use `Terrasoft.controls.IntegerEdit` as the parent control.

1. Create a module

1. Open the [*Configuration*] section and select a custom package to add the schema.
2. Click [*Add*] → [*Module*] on the section list toolbar.



3. Fill out the schema properties in the Module Designer.
   - Set [*Code*] to "`UsrLimitedIntegerEdit`."
   - Set [*Title*] to "`LimitedIntegerEdit`."



Click [*Apply*] to apply the changes.

4. Add the source code in the Module Designer.

   - Besides the standard `extend` and `alternateClassName` properties, set the range of allowed values (`minLimit` and `maxLimit` properties). Use default values for the properties.
   - Implement the business logic of the control. To do this, overload the `onEnterKeyPressed()` method. Call base logic to generate value change events and implement the `isOutOfLimits()` method after the call. The method checks if the entered value is within the range of allowed values. If the number is out of range, display a warning message in the input field.

   Regardless of displaying the warning message, Creatio saves the entered value and transfers it to the schema view model that uses the control.

`UsrLimitedIntegerEdit`

```
/* Declare a module called UsrLimitedIntegerEdit. The module has no dependencies. Therefore,
define("UsrLimitedIntegerEdit", [], function () {
    /* Declare a class of the control. */
    Ext.define("Terrasoft.controls.UsrLimitedIntegerEdit", {
        /* Base class. */
        extend: "Terrasoft.controls.IntegerEdit",
        /* Class alias. */
        alternateClassName: "Terrasoft.UsrLimitedIntegerEdit",
        /* The minimum allowed value. */
        minLimit: -1000,
        /* The maximum allowed value. */
```

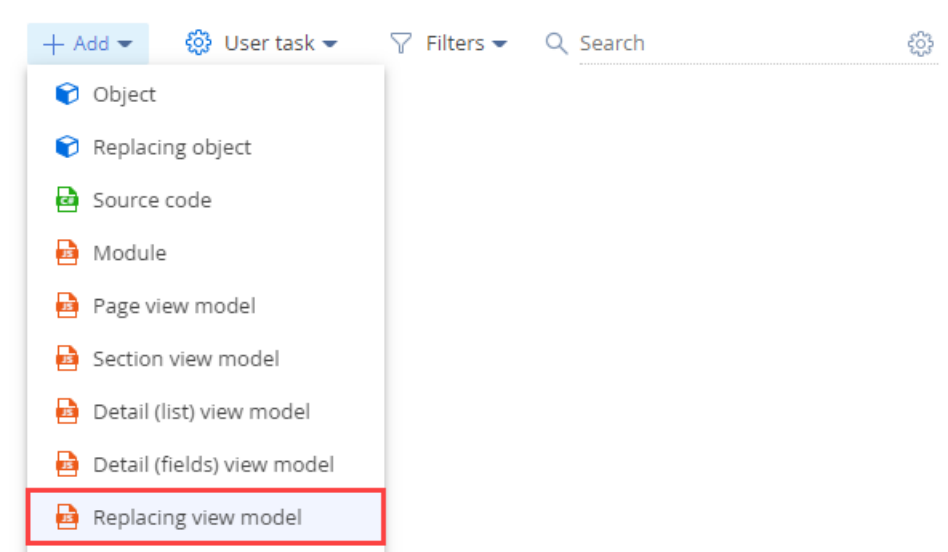
```
        maxLimit: 1000,
        /* Check if the entered value is within the range of allowed values. */
        isOutOfLimits: function (numericValue) {
            if (numericValue < this.minLimit || numericValue > this.maxLimit) {
                return true;
            }
            return false;
        },
        /* Overload the method that handles Enter press. */
        onEnterKeyPressed: function () {
            /* Call base functionality. */
            this.callParent(arguments);
            /* Receive the entered value. */
            var value = this.getTypedValue();
            /* Convert the value to a numeric type. */
            var numericValue = this.parseNumber(value);
            /*  Check if the entered value is within the range of allowed values. */
            var outOfLimits = this.isOutOfLimits(numericValue);
            if (outOfLimits) {
                /* Generate a warning message. */
                var msg = "Value " + numericValue + " is out of limits [" + this.minLimit + "
                /* Modify the configuration object to display the 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 that displays the warning message. */
            this.setMarkOut();
        },
    });
});
```

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

Note. You can use the business logic of the `onEnterKeyPressed()` method when implementing the business logic of the `onBlur()` method that handles focus loss event.

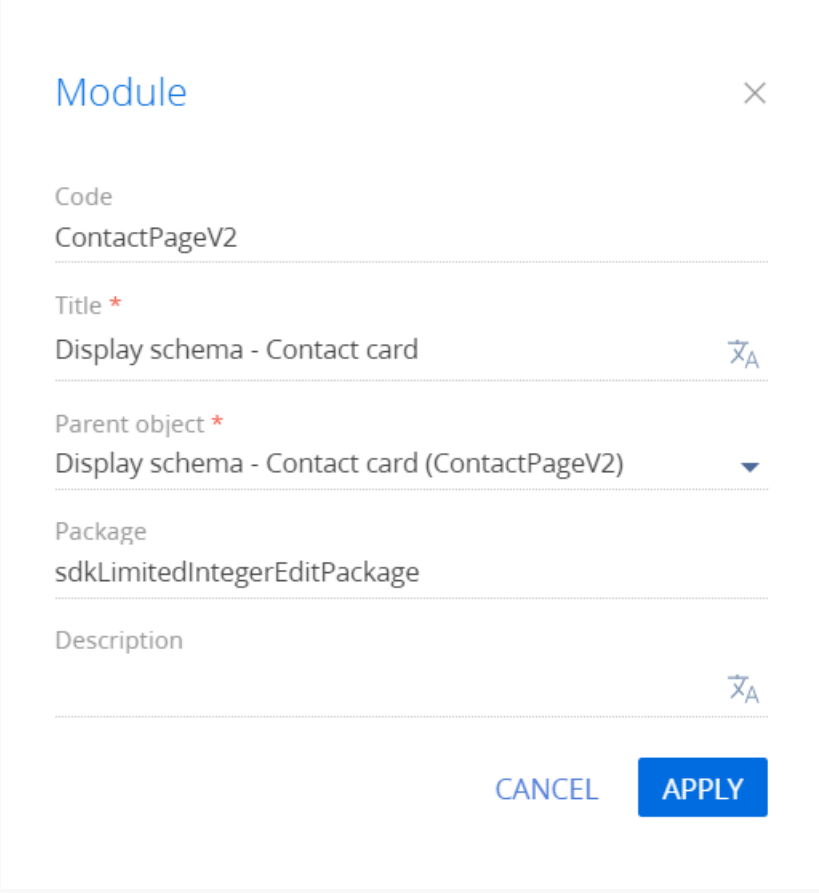
2. Add the control to a contact page

1. Open the [*Configuration*] section and select a custom package to add the schema.
2. Click [*Add*] → [*Replacing view model*] on the section list toolbar.



3. Fill out the **schema properties**.

   - Set [*Code*] to "`ContactPageV2`."
   - Set [*Title*] to "`Display schema - Contact card`."
   - Select "`ContactPageV2`" in the [*Parent object*] property.



4. Add the source code in the Module Designer.

- Add the `ScoresAttribute` attribute to the `attributes` property. The attribute is bound to the value of the control's input field. Instead of an attribute, you can use an integer column of an object that is linked to a record page schema.
- Add a configuration object that defines the property values of a control instance to the `diff` array of modifications.
  - Bind the value of the `value` property to the `ScoresAttribute` attribute.
  - Set the range of allowed values (`minLimit` and `maxLimit` properties). If you do not enter the `minLimit` and `maxLimit` properties in the configuration object explicitly, the default range of valid values is `[-1000; 1000]`.

View the source code of the replacing view model schema of the contact page below.

ContactPageV2

```
/* Declare a module. Specify the module that declares the control class as a dependency. */
define("ContactPageV2", ["UsrLimitedIntegerEdit"], function () {
    return {
        attributes: {
            /* The attribute that is bound to the control value. */
            "ScoresAttribute": {
                /* The attribute data type is integer. */
                "dataValueType": this.Terrasoft.DataValueType.INTEGER,
                /* The attribute type is virtual column. */
                "type": this.Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN,
                /* The default value. */
                "value": 0
            }
        },
        diff: /**SCHEMA_DIFF*/[
            {
                "operation": "insert",
                "parentName": "ProfileContainer",
                "propertyName": "items",
                "name": "Scores",
                "values": {
                    "contentType": Terrasoft.ContentType.LABEL,
                    "caption": {"bindTo": "Resources.Strings.ScoresCaption"},
                    "layout": {
                        "column": 0,
                        "row": 6,
                        "colSpan": 24
                    }
                }
            },
            {
                /* Add the control to the page. */
```

```
                "operation": "insert",
                /* The meta name of the parent container to add the field. */
                "parentName": "ProfileContainer",
                /* Add the field to the parent element's collection of elements. */
                "propertyName": "items",
                /* The meta name of the schema column to which the component is bound. */
                "name": "ScoresValue",
                /* The properties to pass to the element's constructor. */
                "values": {
                    /* Set the type of the control to "component." */
                    "itemType": Terrasoft.ViewItemType.COMPONENT,
                    /* The class name. */
                    "className": "Terrasoft.UsrLimitedIntegerEdit",
                    /* The value of the "value" component property is linked to the ScoresAtt
                    "value": { "bindTo": "ScoresAttribute" },
                    /* The minLimit property value. */
                    "minLimit": -300,
                    /* The maxLimit property value. */
                    "maxLimit": 300,
                    /* Set up the component layout in the container. */
                    "layout": {
                        /* The column number. */
                        "column": 0,
                        /* The row number. */
                        "row": 6,
                        /* The column span. */
                        "colSpan": 24
                    }
                }
            }
        ]/**SCHEMA_DIFF*/
    };
});
```

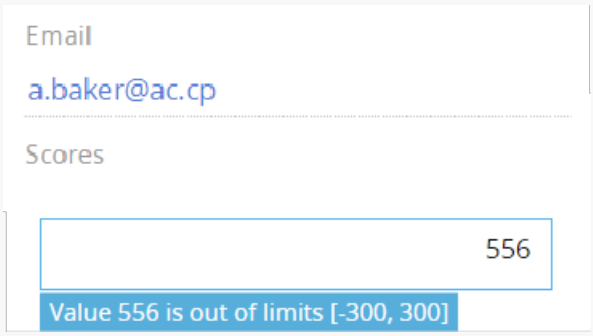
5. Click [*Save*] on the Designer's toolbar.

Outcome of the example

To **view the outcome of the example**:

1. Open the contact page.
2. Enter a value that is out of the range of valid values in the [*Scores*] field.

As a result, Creatio will display the corresponding warning message on the contact page.



Module class

Medium

Declare a module class

Class declaration is a function of the `ExtJS` JavaScript framework. To **declare a class**, use the `define()` method of the global `Ext` object. This is the standard library mechanism.

Example that declares a class using the `define()` method

```
/* Class name that uses a namespace. */
Ext.define("Terrasoft.configuration.ExampleClass", {
    /* The short class name. */
    alternateClassName: "Terrasoft.ExampleClass",
    /* Name of the class from which the current class inherits. */
    extend: "Terrasoft.BaseClass",
    /* The configuration object that contains declarations of static properties and methods. */
    static: {
        /* Example of a static property. */
        myStaticProperty: true,

        /* Example of a static method. */
        getMyStaticProperty: function () {
            /* Example that accesses a static property. */
            return Terrasoft.ExampleClass.myStaticProperty;
        }
    },
    /* Example of a dynamic property. */
    myProperty: 12,
    /* Example of a dynamic class method. */
    getMyProperty: function () {
        return this.myProperty;
    }
});
```

View the examples that create class instances below.

Example that creates a class instance using the full name

```
/* Create a class instance using the full name. */
var exampleObject = Ext.create("Terrasoft.configuration.ExampleClass");
```

Example that creates a class instance using a short name

```
/* Create a class instance using the alias (the short name). */
var exampleObject = Ext.create("Terrasoft.ExampleClass");
```

Inherit from a module class

In most cases, you need to inherit the module class from the `BaseModule` or `BaseSchemaModule` classes of the `Terrasoft.configuration` namespace.

The `BaseModule` and `BaseSchemaModule` classes implement the following **methods**:

- `init()`. Implements the logic that is executed when loading the module. The client core calls this method first automatically when loading the module. The `init()` method usually subscribes to events of other modules and initializes the module values.
- `render(renderTo)`. Implements the module visualization logic. The client core calls this method automatically when loading the module. To ensure data is displayed correctly, trigger the mechanism that binds the view (`View`) and view model (`ViewModel`) before data visualization. Usually, this mechanism is initiated in the `render()` method by calling the `bind()` method in the view object. If you load the module into a container, pass the link to the container to the `render()` method as an argument. The `render()` method is required for visual modules.
- `destroy()`. Responsible for deleting the module view, deleting the view model, unsubscribing from previously subscribed messages, and deleting the module class object.

View the example of a module class that inherits from the `Terrasoft.BaseModule` class below. The module adds a button to DOM. When you click the button, Creatio displays a message and deletes the button from DOM.

Example of a module class that inherits from the `Terrasoft.BaseModule` class

```
define("ModuleExample", [], function () {
    Ext.define("Terrasoft.configuration.ModuleExample", {
        /* The short class name. */
        alternateClassName: "Terrasoft.ModuleExample",
        /* Name of the class from which the current class inherits. */
        extend: "Terrasoft.BaseModule",
        /* Required. If omitted, Creatio generates an error on the Terrasoft.core.BaseObject lev
```

```
        Ext: null,
        /* Required. If omitted, Creatio generates an error on the Terrasoft.core.BaseObject lev
        sandbox: null,
        /* Required. If omitted, Creatio generates an error on the Terrasoft.core.BaseObject lev
        Terrasoft: null,
        /* View model. */
        viewModel: null,
        /* View. The example uses a button. */
        view: null,
        /* If you do not implement the init() method in the current class, Creatio calls the ini
        init: function () {
            /* Execute the logic of the init() method of the parent class. */
            this.callParent(arguments);
            this.initViewModel();
        },
        /* Initialize the view model. */
        initViewModel: function () {
            /* Save the scope of a module class to provide access to the module from the view mo
            var self = this;
            /* Create 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");
                        /* Call the method that unloads the view and view model to delete the bu
                        self.destroy();
                    }
                }
            });
        },
        /* Create a view (button), link it to a view model, and add it to DOM. */
        render: function (renderTo) {
            /* Create a button as a view. */
            this.view = this.Ext.create("Terrasoft.Button", {
                /* Container to add the button. */
                renderTo: renderTo,
                /* The id HTML attribute. */
                id: "example-btn",
                /* Class name. */
                className: "Terrasoft.Button",
                /* Caption. */
                caption: {
```

```
                    /* Link the button caption to the captionBtn property of the view model. */
                    bindTo: "captionBtn"
                },
                /* Handler method for the button click event. */
                click: {
                    /* Link the handler method for the button click event to the onClickBtn() me
                    bindTo: "onClickBtn"
                },
                /* Button style. */
                style: this.Terrasoft.controls.ButtonEnums.style.GREEN
            });
            /* Bind the view and view model. */
            this.view.bind(this.viewModel);
            /* Return the view that is added to DOM. */
            return this.view;
        },
        /* Delete unused objects. */
        destroy: function () {
            /* Delete the view to delete the button from DOM. */
            this.view.destroy();
            /* Delete the unused view model. */
            this.viewModel.destroy();
        }
    });
    /* Return the module object. */
    return Terrasoft.ModuleExample;
});
```

Overload module class members

When you inherit from a module class, you can overload public and private properties and methods of a base module in an inheritor class.

Private class properties or methods are properties or methods whose names start with an underscore character, for example, `_privateMemberName`.

The **purpose** of tracking is to check if overloads of private properties or methods declared in parent classes are executed when declaring a custom class. The browser console displays an overload warning in debug mode. Learn more in a separate article: Front-end debugging.

To **track overloads of private members of a module class**, use the `Terrasoft.PrivateMemberWatcher` class.

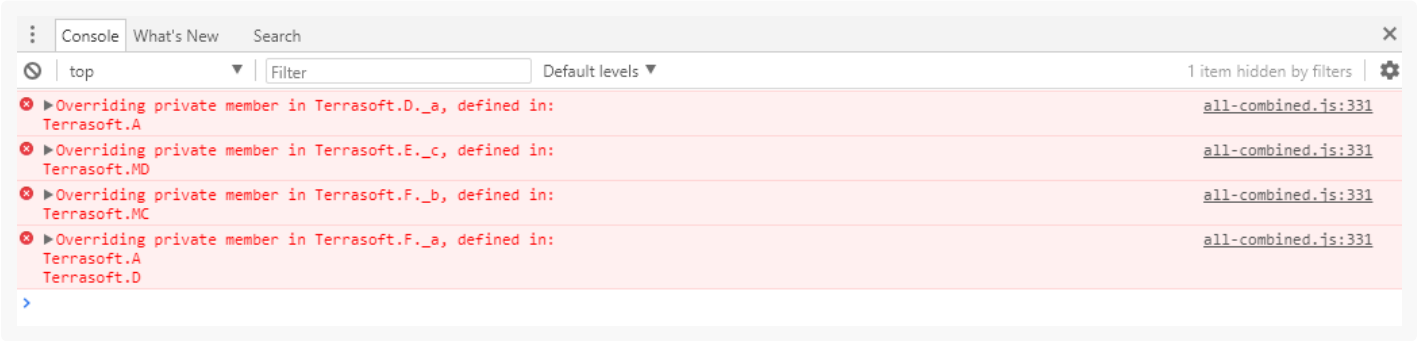
For example, a custom package includes the `UsrPrivateMemberWatcher` module schema.

`UsrPrivateMemberWatcher`

```
define("UsrPrivateMemberWatcher", [], function() {
    Ext.define("Terrasoft.A", {_a: 1});
    Ext.define("Terrasoft.B", {extend: "Terrasoft.A"});
```

```
    Ext.define("Terrasoft.MC", {_b: 1});
    Ext.define("Terrasoft.C", {extend: "Terrasoft.B", mixins: {ma: "Terrasoft.MC"}});
    Ext.define("Terrasoft.MD", {_c: 1});
    /* Override the _a property. */
    Ext.define("Terrasoft.D", {extend: "Terrasoft.C", _a: 3, mixins: {mb: "Terrasoft.MD"}});
    /* Override the _c property. */
    Ext.define("Terrasoft.E", {extend: "Terrasoft.D", _c: 3});
    /* Override the _a and _b properties. */
    Ext.define("Terrasoft.F", {extend: "Terrasoft.E", _b: 3, _a: 0});
});
```

After Creatio loads the `UsrPrivateMemberWatcher` module, the browser console will display a warning about overloading the private members of the base classes.



Initialize a module class instance

You can initialize a module class instance in the following **ways**:

- synchronous initialization
- asynchronous initialization

Initialize a module class instance synchronously

The 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 explicitly on load. For example, the module's class methods are loaded synchronously when the code below is executed.

```
this.sandbox.loadModule([moduleName])
```

Creatio calls the `init()` method first, then the `render()` method.

Example that implements a synchronously initialized module

```
define("ModuleExample", [], function () {
```

```
    Ext.define("Terrasoft.configuration.ModuleExample", {
        alternateClassName: "Terrasoft.ModuleExample",
        Ext: null,
        sandbox: null,
        Terrasoft: null,
        init: function () {
            /* Execute first when initializing the module. */
        },
        render: function (renderTo) {
            /* Execute after init() method when initializing the module. */
        }
     });
});
```

Initialize a module class instance asynchronously

The 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 explicitly on load. For example, the module class methods are loaded asynchronously when the code below is executed.

```
this.sandbox.loadModule([moduleName], {
    isAsync: true
})
```

Creatio calls the `init()` method first. A callback function that has the scope of the current module is passed as a parameter of the `init()` method. When the callback function is called, Creatio executes the `render()` method. The view is added to DOM only after the `render()` method is executed.

Example that implements an asynchronously initialized module

```
define("ModuleExample", [], function () {
    Ext.define("Terrasoft.configuration.ModuleExample", {
        alternateClassName: "Terrasoft.ModuleExample",
        Ext: null,
        sandbox: null,
        Terrasoft: null,
        /* Execute first when initializing the module. */
        init: function (callback) {
            setTimeout(callback, 2000);
        },
        render: function (renderTo) {
            /* Execute with a 2-second delay specified in the parameter in the setTimeout() func
        }
    });
});
```

Module chain

A **module chain** is a mechanism that lets you display a view of a model instead of a view of a different model. For example, to set the field value on the current page, display the `SelectData` page that enables users to select a lookup value. I. e., display the module view of the lookup selection page in place of the module container of the current page.

To **create a chain**, add the `keepAlive` property to the configuration object of the module to load.

View an example that calls the `selectDataModule` module from the `CardModule` current page module below. The `CardModule` module enables users to select a lookup value.

Example that calls a module from a different module

```
sandbox.loadModule("selectDataModule", {
    /* The view ID of the module to load. */
    id: "selectDataModule_id",
    /* Add the view to the current page container. */
    renderTo: "cardModuleContainer",
    /* Specify not to unload the module. */
    keepAlive: true
});
```

After the code is executed, Creatio generates a chain from the current page module and page module that lets you select a lookup value. If you add another element to the chain, users will be able to click the [*Add new record*] button to open a new page from the `selectData` current page module. You can add as many modules to a chain as needed.

An **active module** is the last chain element that is displayed on the page. If you use a module from the middle of a chain as the active module, Creatio deletes all elements after the active module from the chain. To **activate a module in the chain**, pass the module ID to the `loadModule()` function as a parameter.

Example that calls the `loadModule()` function

```
sandbox.loadModule("someModule", {
    id: "someModuleId"
});
```

The core will delete the elements of the chain, then call the `init()` and `render()` methods. The container that includes the previous active module is passed to the `render()` method. The presence of a module in a chain does not affect the module operability.

If you omit the `keepAlive` property or add it as `keepAlive: false` to a configuration object when calling the `loadModule()` method, Creatio deletes the module chain.

Sandbox

Medium

A **module** is a code fragment encapsulated in a separate block that is loaded and executed independently. A module has no information about other Creatio modules besides the names of modules on which it depends. Modules can interact with each other via messages. To **organize the module interaction**, use a `sandbox` object.

The `sandbox` object lets you execute the following **actions**:

- Organize the message exchange among the modules.
- Load and unload modules on request.

Attention. To enable module interaction with other Creatio modules, specify the `sandbox` module as a dependency.

Organize the message exchange among the modules

To exchange messages among the modules, Creatio must execute the following **actions**:

- Register a message.
- Publish a message.
- Subscribe to a message.

A module that needs to inform other Creatio modules about changes to its status publishes a message. A module that needs to receive messages about changes to statuses of other modules subscribes to these messages.

Note. If the module exports a class constructor, you do not have to add `ext-base`, `terrasoft`, `sandbox` base modules as dependencies. The `Ext`, `Terrasoft`, and `sandbox` objects are available as the `this.Ext`, `this.Terrasoft`, `this.sandbox` object properties.

Register a message

You can register a message in the following **ways**:

- using the `sandbox.registerMessages(messageConfig)` method
- using a module schema

Register a message using the `sandbox.registerMessages(messageConfig)` method

The `messageConfig` parameter is a configuration object that contains module messages. The configuration object is a key-value collection.

Template of the message configuration object

```
"MessageName": {
    mode: [Message mode],
    direction: [Message direction]
}
```

- `MessageName` is the key of the collection item that contains the message name.
- `mode` is the message operation mode. Contains the value of the `Terrasoft.core.enums.MessageMode` enumeration. Learn more about the `MessageMode` enumeration in the JS class reference.
  - **Broadcast**. The number of subscribers to the message is unknown in advance. Corresponds to the `Terrasoft.MessageMode.BROADCAST` enumeration value.
  - **Address**. One subscriber handles a message. Corresponds to the `Terrasoft.MessageMode.PTP` enumeration value. You can specify multiple subscribers, but only one handles the message, usually the last registered subscriber.
- `direction`. Message direction. Contains the value of the `Terrasoft.core.enums.MessageDirectionType` enumeration. Learn more about the `MessageDirectionType` enumeration in the JS class reference.
  - **Publish**. The module publishes the message to `sandbox`. Corresponds to the `Terrasoft.MessageDirectionType.PUBLISH` enumeration value.
  - **Subscribe**. The module subscribes to a message published from another module. Corresponds to the `Terrasoft.MessageDirectionType.SUBSCRIBE` enumeration value.
  - **Bidirectional**. The module publishes and subscribes to the same message in different instances of the same class or the same schema inheritance hierarchy. Corresponds to the `Terrasoft.MessageDirectionType.BIDIRECTIONAL` enumeration value.

To **cancel the message registration in the module**, use the `sandbox.unRegisterMessages(messages)` method. The `messages` parameter is the message name or array of message names.

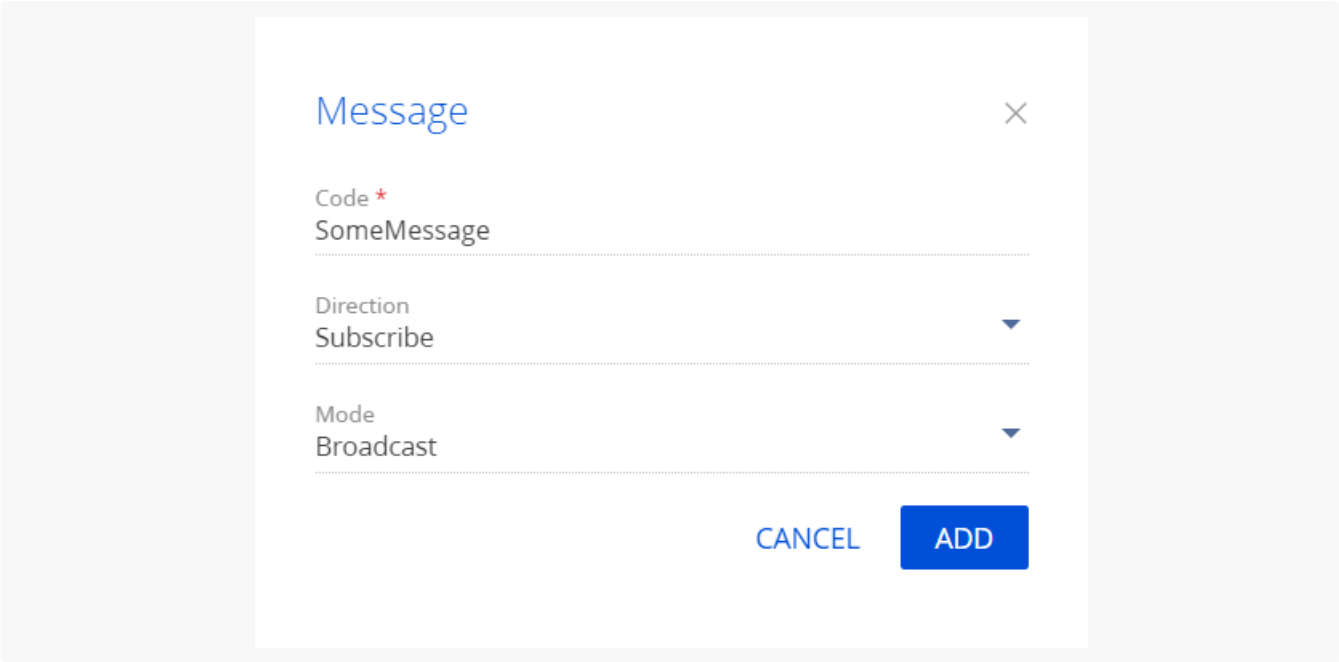
To **register a message in a view model**, declare a message configuration object in the `messages` schema property.

Register a message using the module schema

1. Open the [*Configuration*] section and open a module schema.
2. Add a message to the module schema.
   a. Click the + button in the context menu of the [*Messages*] node.
   b. Fill out the message properties.
      - Enter the message name in the [*Name*] property. The name must match the key in the module configuration object.
      - Select the message direction in the [*Direction*] property. Available values:

- [*Subscribe*]: subscription to the message
- [*Publish*]: message publication

- Select the message operation mode in the [*Mode*] property. Available values:
  - [*Broadcast*]: broadcast message
  - [*Address*]: address message



j. Click [*Add*] to add a message.

You do not need to register messages in a view model schema.

Publish a message

To **publish a message**, use the `sandbox.publish(messageName, messageArgs, tags)` method.

If the published message contains a tag array, Creatio calls handlers for which one or more tags match. If the published message does not contain a tag array, Creatio calls untagged handlers.

Subscribe to a message

To **subscribe to a message**, use the `sandbox.subscribe(messageName, messageHandler, scope, tags)` method.

Load and unload modules on request

Creatio lets you load and unload modules not specified as dependencies when working with UI.

Load a module on request

To **load a module on request**, use the `sandbox.loadModule(moduleName, config)` method. Method **parameters**:

- `moduleName` is a module name.
- `config` is a configuration object that contains the module messages. Required for visual modules.

View the examples that call the `sandbox.loadModule()` method below.

Example that loads a module without parameters

```
this.sandbox.loadModule("ProcessListenerV2");
```

Example that loads a module with parameters

```
this.sandbox.loadModule("CardModuleV2", {
    renderTo: "centerPanel",
    keepAlive: true,
    id: moduleId
});
```

Unload a module on request

To **unload a module on request**, use the `sandbox.unloadModule(id, renderTo, keepAlive)` method. Method **parameters**:

- `id` is a module ID.
- `renderTo` is the container name from which to remove the visual module view. Required for visual modules.
- `keepAlive` indicates whether to save module model. The core can save the model when unloading the module. The saved model lets you use properties, methods, and messages. Not recommended.

View the examples that call the `sandbox.unloadModule()` method below.

Example that unloads a non-visual module

```
/* Retrieve the ID of the module to unload. */
getModuleId: function() {
    return this.sandbox.id + "_ModuleName";
},

/* Unload a non-visual module. */
this.sandbox.unloadModule(this.getModuleId());
```

Example that unloads a visual module

```
/* Retrieve the ID of the module to unload. */
getModuleId: function() {
    return this.sandbox.id + "_ModuleName";
},

/* Unload a visual module loaded into the "ModuleContainer" container. */
this.sandbox.unloadModule(this.getModuleId(), "ModuleContainer");
```

Create a module chain

If you want to display a model view in place of another model view, use a **module chain**. For example, you can use a module chain to populate a field using a lookup value. To do this, display the module view of the lookup selection page in place of the module container of the current page.

To **create a chain**, add the `keepAlive` property to the configuration object of the module to load.

Implement message exchange between modules

 Medium

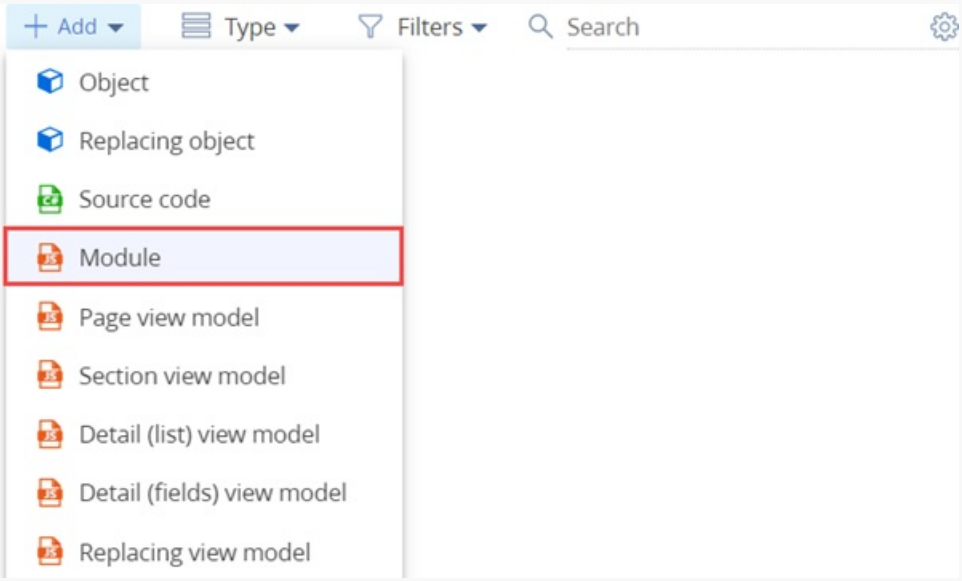
Example. Create the `UsrSomeModule` module. Implement the following **messages** in the module:

- `MessageToSubscribe` address message that has [*Subscribe*] direction
- `MessageToPublish` broadcast message that has [*Publish*] direction

Subscribe to the `MessageToSubscribe` message sent by another module. Cancel the message registration.

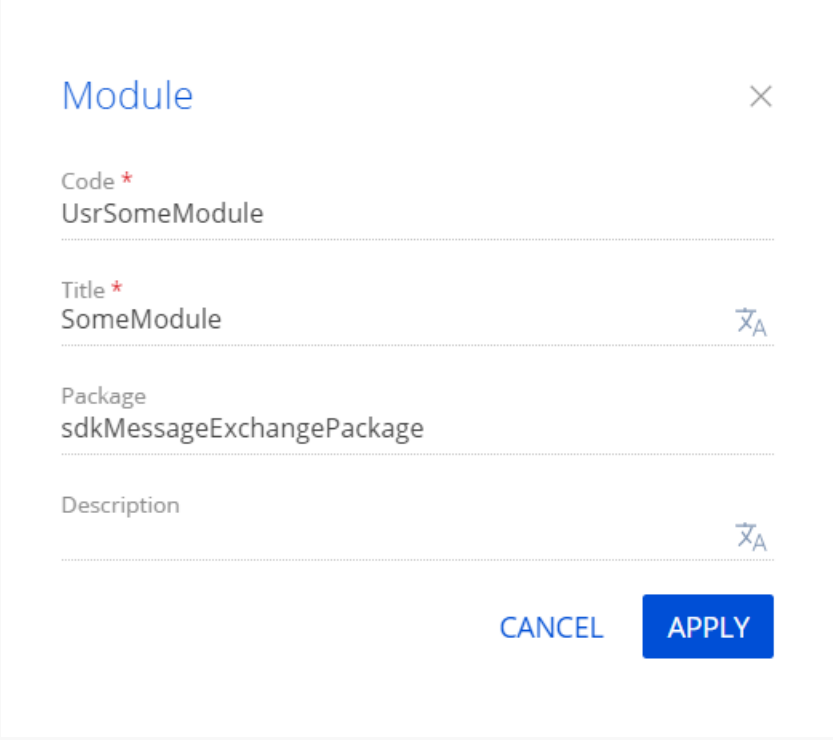
1. Create a module

1. Open the [*Configuration*] section and select a custom package to add the schema.
2. Click [*Add*] → [*Module*] on the section list toolbar.



3. Fill out the schema properties in the Module Designer.

   - Set [*Code*] to "`UsrSomeModule`."
   - Set [*Title*] to "`SomeModule`."



   Click [*Apply*] to apply the changes.

4. Add the source code in the Module Designer.

```
UsrSomeModule
```

```
/* Declare a module called UsrSomeModule. The module has no dependencies.
Therefore, an empty array is passed as the second module parameter. */
define("UsrSomeModule", [], function() {
    Ext.define("Terrasoft.configuration.UsrSomeModule", {
        alternateClassName: "Terrasoft.UsrSomeModule",
        extend: "Terrasoft.BaseModule",
        Ext: null,
        sandbox: null,
        Terrasoft: null,

        init: function() {
            this.callParent(arguments);
        },
        destroy: function() {
            this.callParent(arguments);
        }
    });
    return Terrasoft.UsrSomeModule;
});
```

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

2. Register a message

1. Declare message configuration objects in the `messages` schema property.
2. Add to the `init()` method the `sandbox.registerMessages()` method call that registers messages.

Register a module message

```
...
/* Collection of the configuration message objects. */
messages: {
    "MessageToSubscribe": {
        mode: Terrasoft.MessageMode.PTP,
        direction: Terrasoft.MessageDirectionType.SUBSCRIBE
    },
    "MessageToPublish": {
        mode: Terrasoft.MessageMode.BROADCAST,
        direction: Terrasoft.MessageDirectionType.PUBLISH
    }
},
...
init: function() {
    this.callParent(arguments);
    /* Register a message collection. */
```

```
    this.sandbox.registerMessages(this.messages);
},
...
```

3. Publish a message

1. Implement the `processMessages()` method in the module schema.
2. In the `processMessages()` method, call the `sandbox.publish()` method that publishes the `MessageToPublish` message.
3. Add the `processMessages()` method call to the `init()` method.

Publish a module message

```
...
init: function() {
    ...
    this.processMessages();
},
...
processMessages: function() {
    this.sandbox.publish("MessageToPublish", null, [this.sandbox.id]);
},
...
```

4. Subscribe to a message

1. Add the `sandbox.subscribe()` method call to the `processMessages()` method. The `sandbox.subscribe()` method subscribes to the `MessageToSubscribe` message sent by another module.
2. Specify the `onMessageSubscribe()` handler method in the method parameters and add it to the module source code.

Subscribe to a message from another module

```
...
processMessages: function() {
    this.sandbox.subscribe("MessageToSubscribe", this.onMessageSubscribe, this, ["resultTag"]);
    this.sandbox.publish("MessageToPublish", null, [this.sandbox.id]);
},
onMessageSubscribe: function(args) {
    console.log("'MessageToSubscribe' received");
    /* Modify the parameter. */
    args.arg1 = 15;
    args.arg2 = "new arg2";
```

```
    /* Return the result. */
    return args;
},
...
```

5. Cancel the message registration

Cancel the message registration

```
...
destroy: function() {
    if (this.messages) {
        var messages = this.Terrasoft.keys(this.messages);
        /* Cancel the message array registration. */
        this.sandbox.unRegisterMessages(messages);
    }
    this.callParent(arguments);
}
...
```

Complete source code of the page schema

Accept the result from a subscriber module (address message)

 Medium

Example. Implement the `MessageWithResult` address message in the module. The message must have the [*Publish*] direction and accept the result from a subscriber module.

Example that implements the `MessageWithResult` address message

```
...
/* Collection of the configuration message objects. */
messages: {
    ...
    "MessageWithResult": {
        mode: Terrasoft.MessageMode.PTP,
        direction: Terrasoft.MessageDirectionType.PUBLISH
    }
}
```

```
...
processMessages: function() {
    ...
    /* Publish messages and receive the result of their handling by the subscriber module. */
    var result = this.sandbox.publish("MessageWithResult", {arg1:5, arg2:"arg2"}, ["resultTag"])
    /* Display the result at the browser console. */
    console.log(result);
}
...
```

Accept the result from a subscriber module (broadcast message)

Medium

Example. Implement the `MessageWithResultBroadcast` broadcast message in the module. The message must have the [*Publish*] direction and accept the result from a subscriber module.

Example that implements the `MessageWithResultBroadcast` broadcast message

```
...
/* Collection of the configuration message objects. */
messages: {
    ...
    "MessageWithResult": {
        mode: Terrasoft.MessageMode.PTP,
        direction: Terrasoft.MessageDirectionType.PUBLISH
    }
}
...
processMessages: function() {
    ...
    var arg = {};
    /* Publish messages and receive the result of their handling by the subscriber module. The r
    this.sandbox.publish("MessageWithResultBroadcast", arg, ["resultTag"]);
    /* Display the result at the browser console. */
    console.log(arg.result);
}
...
```

Implement asynchronous message

exchange

Medium

Example. Implement asynchronous exchange among the modules.

1. Set the configuration object as a parameter of the handler function in the module that publishes the message.
2. Add a callback function to the configuration object.

Example that publishes messages and retrieves the result

```
...
this.sandbox.publish("AsyncMessageResult",
/* Configuration object specified as a handler function parameter. */
{
    /* Callback function. */
    callback: function(result) {
        this.Terrasoft.showInformation(result);
    },
    /* Scope of the callback function execution. */
    scope: this
});
...
```

3. Return asynchronous result in the handler method of the subscriber module the module subscribes to a message. Use the callback function parameter of the published message.

Example that subscribes to a message

```
...
this.sandbox.subscribe("AsyncMessageResult",
/* Message handler function. */
function(config) {
    /* Handle the incoming parameter. */
    var config = config || {};
    var callback = config.callback;
    var scope = config.scope || this;
    /* Prepare the resulting message. */
    var result = "Message from callback function";
    /* Execute the callback function. */
    if (callback) {
        callback.call(scope, result);
    }
},
/* Execution scope of the message handler function. */
```

```
    this);
  ...
```

Example that uses bidirectional messages

Medium

The `BaseEntityPage` schema of the `CrtNUI` package registers the `CardModuleResponse` message. The `BaseEntityPage` schema is the base schema of the record page's view model.

BaseEntityPage

```
define("BaseEntityPage", [...], function(...) {
    return {
        messages: {
            ...
            "CardModuleResponse": {
                "mode": this.Terrasoft.MessageMode.PTP,
                "direction": this.Terrasoft.MessageDirectionType.BIDIRECTIONAL
            },
            ...
        },
        ...
    };
});
```

For example, Creatio publishes a message after saving the modified record. The `BasePageV2` child schema of the `CrtNUI` package implements this functionality.

BasePageV2

```
define("BasePageV2", [..., "LookupQuickAddMixin", ...], function(...) {
    return {
        ...
        methods: {
            ...
            onSaved: function(response, config) {
                ...
                    this.sendSaveCardModuleResponse(response.success);
                ...
            },
            ...
            sendSaveCardModuleResponse: function(success) {
                var primaryColumnValue = this.getPrimaryColumnValue();
```

```
                var infoObject = {
                    action: this.get("Operation"),
                    success: success,
                    primaryColumnValue: primaryColumnValue,
                    uId: primaryColumnValue,
                    primaryDisplayColumnValue: this.get(this.primaryDisplayColumnName),
                    primaryDisplayColumnName: this.primaryDisplayColumnName,
                    isInChain: this.get("IsInChain")
                };
                return this.sandbox.publish("CardModuleResponse", infoObject, [this.sandbox.id])
            },
            ...
        },
        ...
    };
});
```

The `LookupQuickAddMixin` mixin is listed in the `BasePageV2` schema as a dependency. The mixin implements the subscription to the `CardModuleResponse` message. Learn more in a separate article: Client schema.

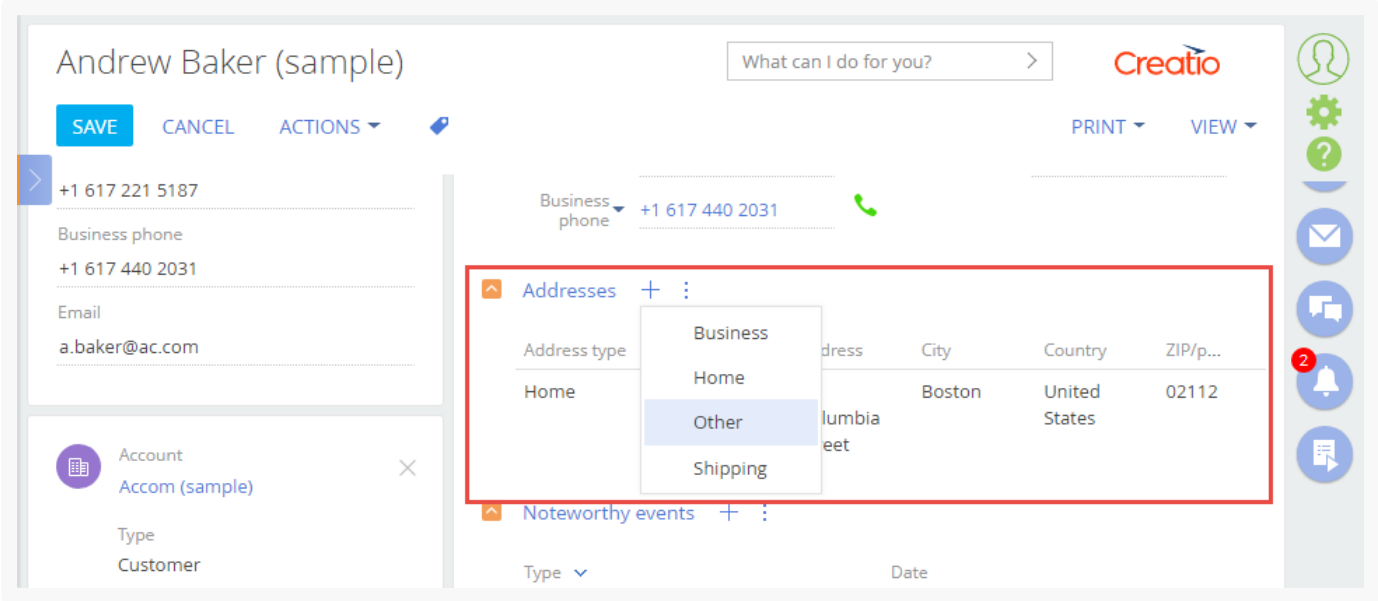
LookupQuickAddMixin

```
define("LookupQuickAddMixin", [...], function(...) {
    Ext.define("Terrasoft.configuration.mixins.LookupQuickAddMixin", {
        alternateClassName: "Terrasoft.LookupQuickAddMixin",
        ...
        /* Declare the message. */
        _defaultMessages: {
            "CardModuleResponse": {
                "mode": this.Terrasoft.MessageMode.PTP,
                "direction": this.Terrasoft.MessageDirectionType.BIDIRECTIONAL
            }
        },
        ...
        /* Register the message. */
        _registerMessages: function() {
            this.sandbox.registerMessages(this._defaultMessages);
        },
        ...
        /* Initialize a class instance. */
        init: function(callback, scope) {
            ...
            this._registerMessages();
            ...
        },
        ...
        /* Execute after adding a new record to a lookup. */
```

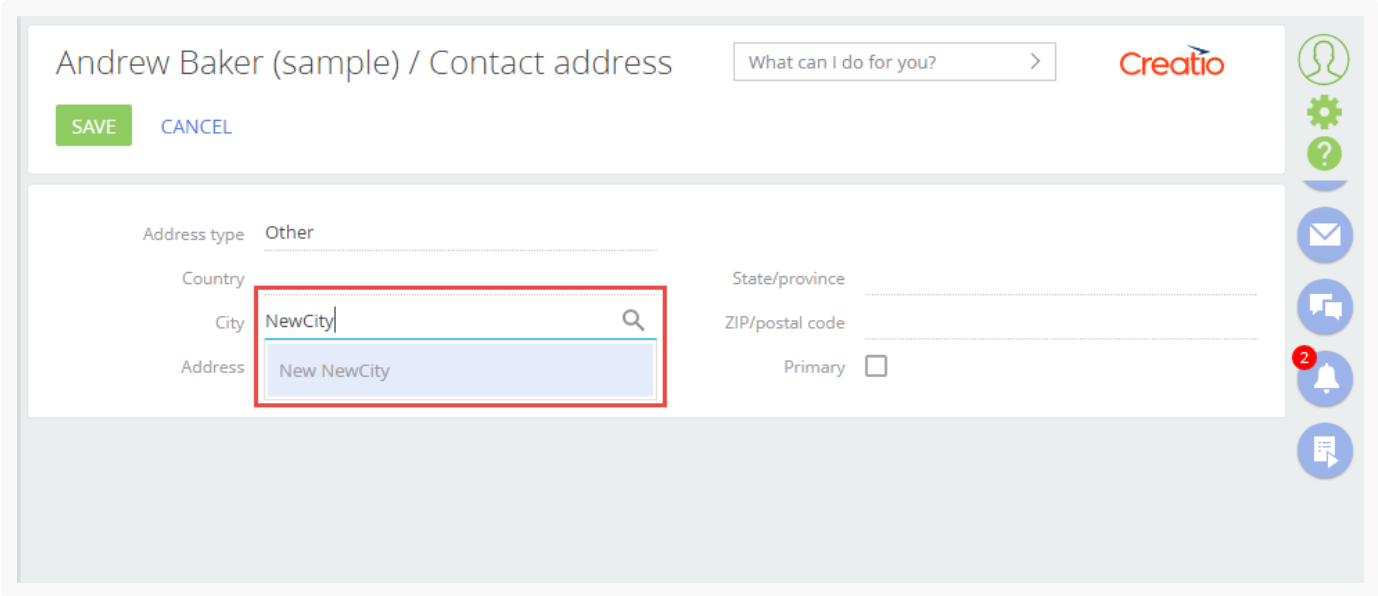
```
        onLookupChange: function(newValue, columnName) {
            ...
            /* Execute a chain of method calls.
            As a result, the _subscribeNewEntityCardModuleResponse() method is called. */
            ...
        },
        ...
        /* The method that subscribes to the "CardModuleResponse" message.
        The callback function sets the lookup field to the value sent when the message was publi
        _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;
});
```

The procedure that handles bidirectional messages when adding a new address to a contact page is as follows:

1. Creatio loads the `ContactAddressPageV2` module into the module chain on the [*Addresses*] detail.

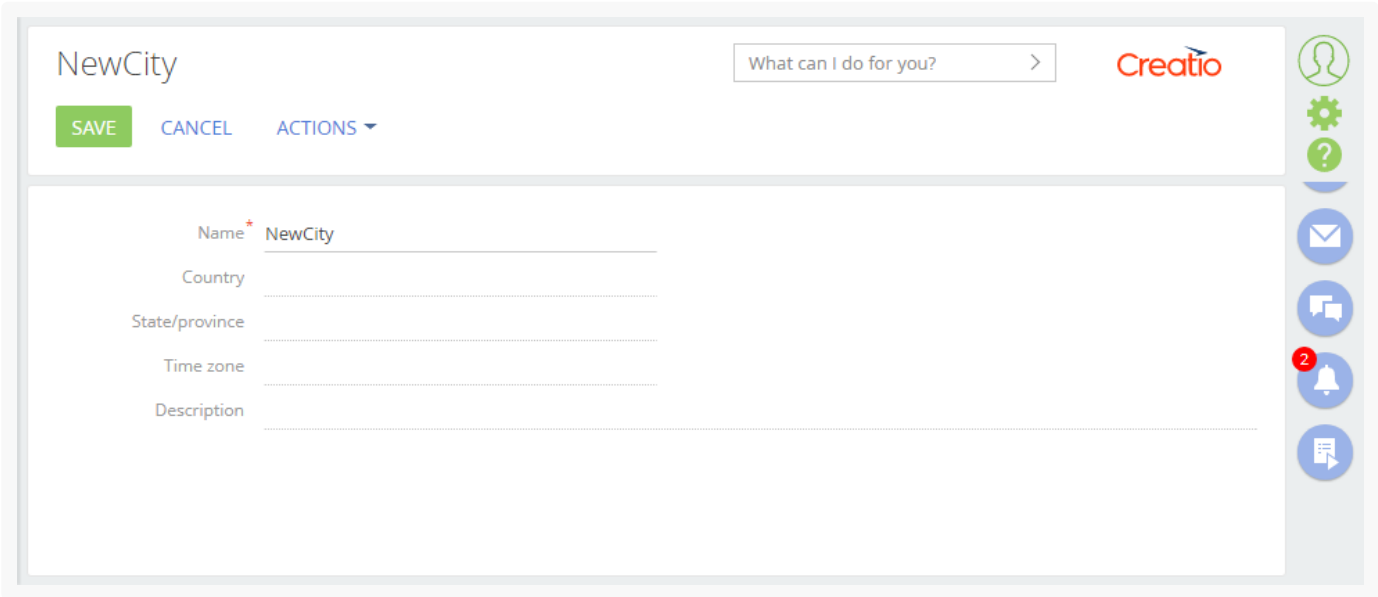


2. The contact address page is opened.



Since the `ContactAddressPageV2` schema inherits the `BaseEntityPage` and `BasePageV2` schemas, the `ContactAddressPageV2` schema already has the `CardModuleResponse` message registered. This message is also registered in the `_registerMessages()` method of the `LookupQuickAddMixin` mixin when the mixin is initialized in the `BasePageV2` schema as a dependency.

3. The `onLookupChange()` method of the `LookupQuickAddMixin` mixin is called when adding a new value, for example, a city, to the lookup fields of the `ContactAddressPageV2` page.
4. The `CityPageV2` module is loaded into the module chain.
5. The `onLookupChange()` method calls the `_subscribeNewEntityCardModuleResponse()` method that subscribes to the `CardModuleResponse` message.
6. The city page (`CityPageV2` schema in the `CrtUIv2` package) is opened.



7. Since the `CityPageV2` schema inherits the `BasePageV2` schema, the `onSaved()` method of the base schema is executed after the user saves the record ([*Save*] button).
8. The `onSaved()` method calls the `sendSaveCardModuleResponse()` method that publishes the `CardModuleResponse`

message. At the same time, the object that contains the necessary results of saving is passed.

9. After the message is published, the callback function (`_subscribeNewEntityCardModuleResponse()` method in the `LookupQuickAddMixin` mixin) of the subscriber is executed. The method processes the results of saving the new city to the lookup.

Thus, publishing and subscribing to a bidirectional message are executed as part of a single schema inheritance hierarchy. In this hierarchy, the `BasePageV2` base schema contains all required functionality.

Set up module loading

Medium

Example. Load the `UsrCardModule` custom visual module into the `UsrModule` custom module.

1. Create a class of a visual module

Create a `UsrCardModule` class of module that inherits from the `BaseSchemaModule` base class. The class must be instantiated, i. e., return a constructor function. In this case, you can pass the required parameters to a constructor when loading the module externally.

UsrCardModule

```
/* Module that returns an instance of a class. */
define("UsrCardModule", [...], function(...) {
    Ext.define("Terrasoft.configuration.UsrCardModule", {
        /* Class alias. */
        alternateClassName: "Terrasoft.UsrCardModule",
        /* Parent class. */
        extend: "Terrasoft.BaseSchemaModule",
        /* The flag that indicates that the schema parameters are set externally. */
        isSchemaConfigInitialized: false,
        /* The flag that indicates that the history status is used when loading the module. */
        useHistoryState: true,
        /* The schema name of the displayed entity. */
        schemaName: "",
        /* The flag that indicates that the section list is displayed in combined mode.
        If set to false, the page displays SectionModule. */
        isSeparateMode: true,
        /* Object schema name. */
        entitySchemaName: "",
        /* Primary column value. */
        primaryColumnValue: Terrasoft.GUID_EMPTY,
        /* Record page mode. */
        operation: ""
    });
```

```
    /* Return a class instance. */
    return Terrasoft.UsrCardModule;
}
```

2. Create a module class to load the visual module

Create a `UsrModule` module class that inherits from the `BaseModel` base class.

UsrModule

```
define("UsrModule", [...], function(...) {
    Ext.define("Terrasoft.configuration.UsrModule", {
        alternateClassName: "Terrasoft.UsrModule",
        extend: "Terrasoft.BaseModel",
        Ext: null,
        sandbox: null,
        Terrasoft: null,
    });
}
```

3. Load the module

You can pass parameters to the constructor of the instantiated module class when loading the module. To do this:

1. Create a configuration object in the `UsrModule` class module.
2. Specify the required values as the properties of the configuration object.
3. Load the `UsrCardModule` visual module using the `sandbox.loadModule()` method.
4. Add the `instanceConfig` property to the `sandbox.loadModule()` method.
5. Pass the configuration object that contains the required values as the value of the `instanceConfig` property.

UsrModule

```
...
init: function() {
    this.callParent(arguments);
    /* The configuration object. Specify object properties as parameters of the constructor. */
    var configObj = {
        isSchemaConfigInitialized: true,
        useHistoryState: false,
        isSeparateMode: true,
        schemaName: "QueueItemEditPage",
        entitySchemaName: "QueueItem",
```

```
        operation: ConfigurationEnums.CardStateV2.EDIT,
        primaryColumnValue: "{3B58C589-28C1-4937-B681-2D40B312FBB6}"
    };

    /* Load module. */
    this.sandbox.loadModule("UsrCardModule", {
        renderTo: "DelayExecutionModuleContainer",
        id: this.getQueueItemEditModuleId(),
        keepAlive: true,
        /* Specify the configuration object in the module constructor as a parameter. */
        instanceConfig: configObj
    }
});
...
```

To pass additional parameters when loading the module, use the `parameters` property of the configuration object. Pre-implement the same property in the module class or one of the parent classes. The `parameters` property is defined in the `BaseModule` base class. When a module instance is created, the `parameters` property of the module is initialized using the values passed in the `parameters` property of the configuration object.

sandbox object

Medium

A `sandbox` object is a core component required to organize the module interaction.

The `sandbox` object lets you execute the following **actions**:

- Organize the message exchange among the modules.
- Load and unload modules on request.

Methods

`registerMessages(messageConfig)`

Registers module messages.

Parameters

`{Object} messageConfig`

Configuration object of module messages. Configuration object is a key-value collection where every item is as follows.

Item of configuration object

```
/* Key of the collection item. The key is the message name. */
"MessageName": {
    /* Value of the collection item. */
    mode: [Режим работы сообщения],
    direction: [Направление сообщения]
}
```

Properties of the collection item values

`{Terrasoft.MessageMode} mode`	Message operation mode. Contains the value of the `Terrasoft.MessageMode (Terrasoft.core.enums.MessageMode)` enumeration.
	
	Available values (`Terrasoft.MessageMode`)
	
	`BROADCAST`
	Broadcast message mode where the number of message subscribers is unknown in advance.
	
	`PTP`
	Address message mode where only one subscriber can handle a message.

`{Terrasoft.MessageDirectionType} direction`	Message direction. Contains the value of the `Terrasoft.MessageDirectionType (Terrasoft.core.enums.MessageDirectionType)` enumeration.
	
	Available values (`Terrasoft.MessageDirectionType`)
	
	`PUBLISH`
	The message direction is publishing. The module only publishes the message to `sandbox`.
	
	`SUBSCRIBE`
	The message direction is subscription. The

	module only subscribes to a message that is published by another module. `BIDIRECTIONAL` The message is bidirectional. The module publishes and subscribes to the same message in different instances of the same class or the same schema inheritance hierarchy.

`unRegisterMessages(messages)`

Cancels message registration.

Parameters

`{String|Array} messages`	Name or array of message names.

`publish(messageName, messageArgs, tags)`

Publishes a message to `sandbox`.

Parameters

`{String} messageName`	A string that contains the message name. For example, `"MessageToSubscribe."`
`{Object} messageArgs`	An object passed as a parameter to the message handler method in the subscriber module. If incoming parameters are omitted from the handler method, set the `messageArgs` parameter to `null`.
`{Array} tags`	A tag array that lets you uniquely identify the module that sends the message. Usually, the `[this.sandbox.id]` value is used. `sandbox` identifies subscribers and publishers of a message based on the array of tags.

Use examples

Examples that use the `publish()` method

```
/* Publish the message without parameters and tags. */
this.sandbox.publish("MessageWithoutArgsAndTags");

/* Publish the message without using parameters for the handler method. */
this.sandbox.publish("MessageWithoutArgs", null, [this.sandbox.id]);

/* Publish the message using the parameter for the handler method. */
this.sandbox.publish("MessageWithArgs", {arg1: 5, arg2: "arg2"}, ["moduleName"]);

/* Publish the message using an arbitrary array of tags. */
this.sandbox.publish("MessageWithCustomIds", null, ["moduleName","otherTag"]);
```

`subscribe(messageName, messageHandler, scope, tags)`

Subscribes to a message.

Parameters

`{String} messageName`	A string that contains the message name. For example, `"MessageToSubscribe."`
`{Function} messageHandler`	A method handler is called when module receives a message. It can be either an anonymous function or module method. You can specify a parameter in the method definition. Pass the parameter value when publishing a message using the `sandbox.publish()` method.
`{Object} scope`	The execution scope of the `messageHandler` handler method.
`{Array} tags`	An array of tags that lets you uniquely identify the module that sends the message. `sandbox` identifies subscribers and publishers of a message based on the array of tags.

Use examples

Example that uses the `subscribe()` method

```
/* Subscribe to a message without parameters for the handler method.
Method handler is an anonymous function. Execution scope is the current module.
The getSandboxId() method returns a tag that matches the tag of the published message. */
this.sandbox.subscribe("MessageWithoutArgs", function(){console.log("Message without argume
```

```
/* Subscribe to a message using the parameter for the handler method. */
this.sandbox.subscribe("MessageWithArgs", function(args){console.log(args)}, this, ["module

/* Subscribe to a message using an arbitrary tag.
Use a tag from the array of tags of the published message.
Implement the myMsgHandler() handler method separately. */
this.sandbox.subscribe("MessageWithCustomIds", this.myMsgHandler, this, ["otherTag"]);
```

`loadModule(moduleName, config)`

Loads the module.

Parameters

`{String} moduleName`	Module name.

`{Object} config`	Configuration object that contains the module parameters. Required for visual modules.
	
	Properties of the configuration object
	
	`{String} id`
	Module ID. If the ID is missing, the module generates it automatically.
	
	`{String} renderTo`
	The name of the container that displays the view of the visual module. Passed as the `render()` method parameter of the loaded module. Required for visual modules.
	
	`{Boolean} keepAlive`
	The flag that indicates whether to add the module to the module chain. Required for navigation between the module views in the browser.
	
	`{Boolean} isAsync`
	The flag that indicates whether to initialize the module asynchronously.

`{Object} instanceConfig`

Lets you pass parameters to the class constructor of an instantiated module when the module is loaded. To do this, specify a configuration object as the value of the `instanceConfig` property. An **instantiated module** is a module that returns a constructor function.

You can pass the following **property types** to a module instance:

- `string`
- `boolean`
- `number`
- `date` (the value will be copied)
- `object` (literal objects only)

Do not pass class instances, HTMLElement descendants, etc., as property values. When passing parameters to the constructor of the `BaseObject` descendant module, consider the following restriction: Creatio cannot pass a parameter that is not described in a module class or one of the parent classes.

`{Object} parameters`

Passes additional parameters to the module when loading a module. Pre-implement the same property in a module class or one of the parent classes. The `parameters` property is implemented in the `BaseModule` base class. When a module instance is created, the `parameters` property of the module is initialized using the values passed in the `parameters` property of the configuration object.

Use examples

Example that uses the `loadModule()` method

```
/* Load the module without using additional parameters. */
this.sandbox.loadModule("ProcessListenerV2");

/* Load the module using additional parameters. */
this.sandbox.loadModule("CardModuleV2", {
    renderTo: "centerPanel",
    keepAlive: true,
    id: moduleId
});
```

`unloadModule(id, renderTo, keepAlive)`

Unloads the module.

Parameters

`{String} id`	Module identifier.
`{String} renderTo`	The name of the container to remove the view of the visual module. Required for visual modules.
`{Boolean} keepAlive`	The flag that indicates whether to save the module model. The core can save a module model when unloading the module to use the properties, methods, and messages of the model.

Use examples

Example that uses the `unloadModule()` method

```
/* Retrieve the ID of an unloaded module. */
getModuleId: function() {
    return this.sandbox.id + "_ModuleName";
},

...
/* Unload a non-visual module. */
this.sandbox.unloadModule(this.getModuleId());

...
/* Unload a visual module previously loaded into the "ModuleContainer" container. */
this.sandbox.unloadModule(this.getModuleId(), "ModuleContainer");
```

Send messages via WebSocket

Advanced

WebSocket sends messages. Creatio broadcasts messages received via WebSocket to subscribers using the `ClientMessageBridge` client module schema. Within Creatio, messages are sent using a `sandbox` object. This is a broadcast message named `SocketMessageReceived`. You can subscribe to the message and handle the received data.

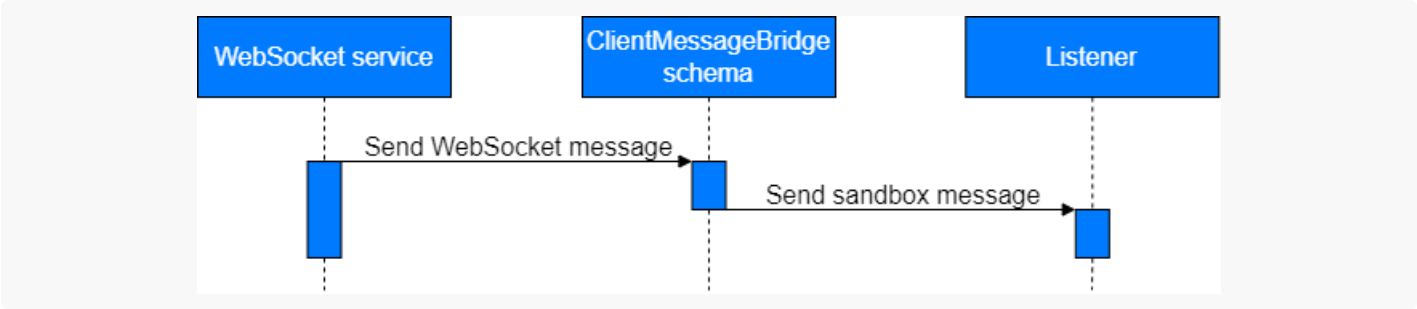
Implement custom logic that sends a message

To **implement custom logic that sends a message received via WebSocket**:

1. Create a replacing schema of the `ClientMessageBridge` client module schema. Learn more in a separate article: Client module.
2. Add the message to the `messages` property of a client schema. Learn more in a separate article: messages property.
3. Add the message received via WebSocket to the configuration object of schema messages. To do this, overload the `init()` parent method.
4. Trace the message sending. To do this, overload the `afterPublishMessage()` base method.

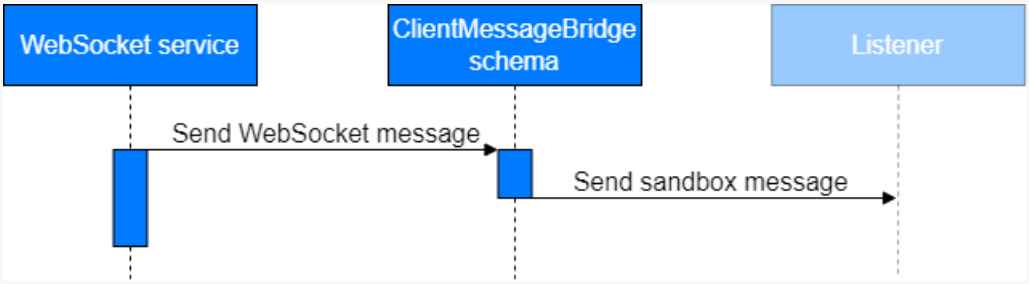
Save the message to history

Message history workflow is based on the `Listener` handler that is a part of message publishing process in Creatio.



If the `Listener` handler is not already loaded, Creatio executes the following **actions**:

1. Save unhandled messages to history.
2. Check availability of the `Listener` handler before publishing a message.
3. Publish all saved messages in their order of reception after the handler is loaded.
4. Clear history after publishing messages from history.



The following **abstract methods** of the `BaseMessageBridge` class implement saving messages to history and working with them via `localStorage` browser repository:

- `saveMessageToHistory()`. Saves a new message to the message collection.
- `getMessagesFromHistory()`. Receives an array of messages by name.
- `deleteSavedMessages()`. Deletes saved messages by name.

The `ClientMessageBridge` schema implements the abstract methods of the `BaseMessageBridge` parent class.

To **implement saving messages to history**, set the `isSaveHistory` property to `true` when adding a configuration object.

Example that saves messages to history

```
init: function() {
   /* Call the parent init() method. */
   this.callParent(arguments);
   /* Add a new configuration object to the collection of configuration objects. */
   this.addMessageConfig({
       /* The name of the message received via WebSocket. */
       sender: "OrderStepCalculated",
       /* The name of the WebSocket message sent in Creatio via sandbox. */
       messageName: "OrderStepCalculated",
       /* Whether to save messages to history. */
       isSaveHistory: true
    });
},
```

To **implement working with messages via another repository**:

1. Specify the `BaseMessageBridge` class as a parent class.
2. Implement custom `saveMessageToHistory()`, `getMessagesFromHistory()`, and `deleteSavedMessages()` methods in the class that inherits from the `BaseMessageBridge` class.

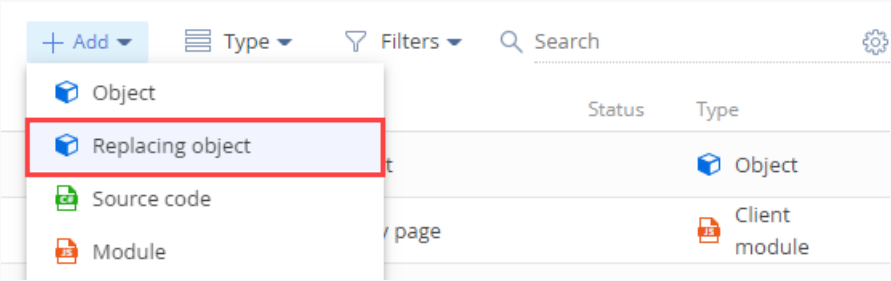
Implement a subscriber to a WebSocket message

Advanced

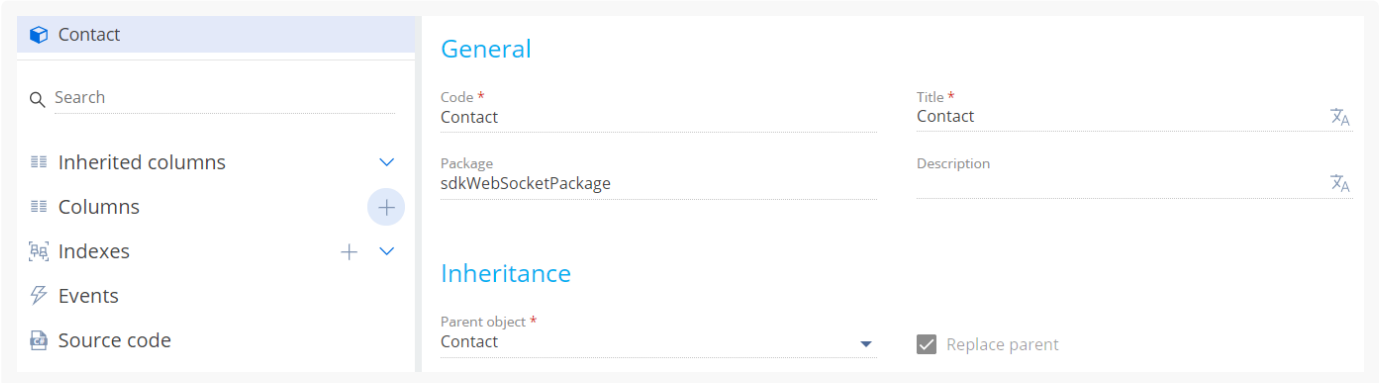
Example. Publish the `NewMessage` message in the back-end when you save a contact. Send the message via WebSocket. The message must include the contact birthday and name. Implement the `NewMessage` message sending in the front-end. Display messages in the browser console.

1. Create a replacing object schema

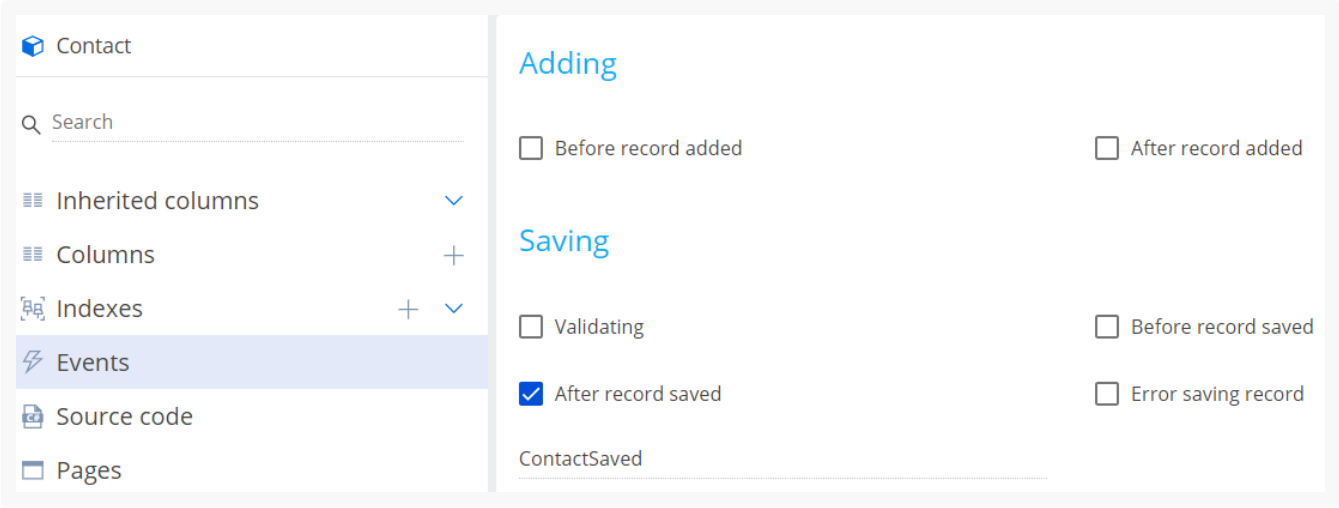
1. Open the [*Configuration*] section and select a custom package to add the schema.
2. Click [*Add*] → [*Replacing object*] on the section list toolbar.



3. Fill out the **schema properties**.
   - Set [*Code*] to "`Contact`."
   - Set [*Title*] to "`Contact`."
   - Select "`Contact`" in the [*Parent object*] property.



4. Add an **event** to the schema.
   a. Open the [*Events*] node of the object structure.
   b. Go to the [*Saving*] block → select the [*After record saved*] checkbox. Creatio names the event `ContactSaved`.

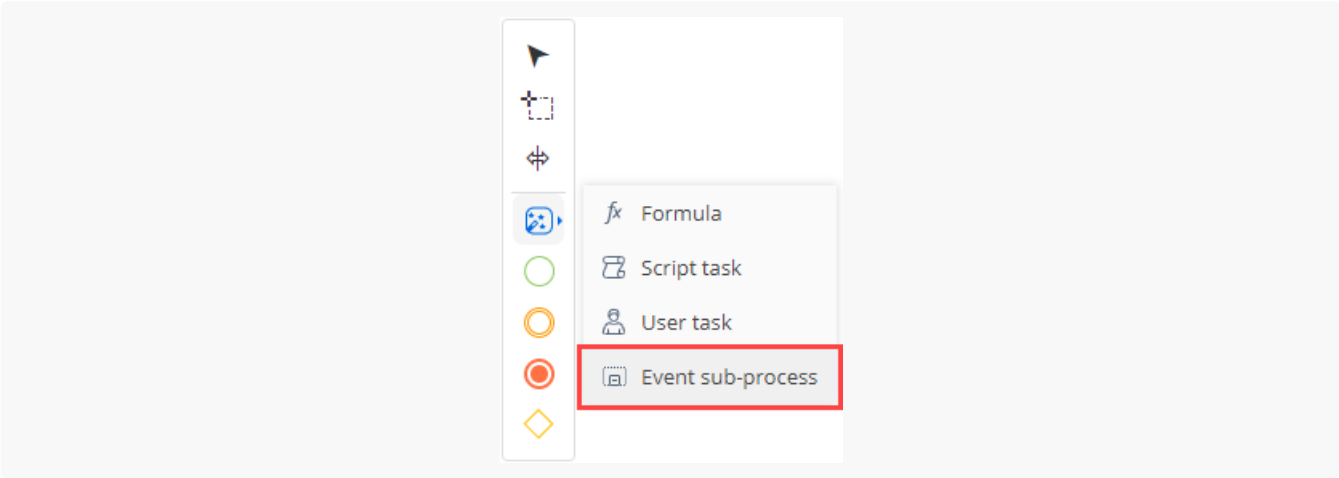


c. Click [*Save*] on the Object Designer's toolbar.

5. Implement an **event sub-process**.

   a. Click [*Open process*] on the Object Designer's toolbar.

   b. Click [*System actions*] in the element area of the Designer and drag the[*Event sub-process*] element to the working area of the Process Designer.

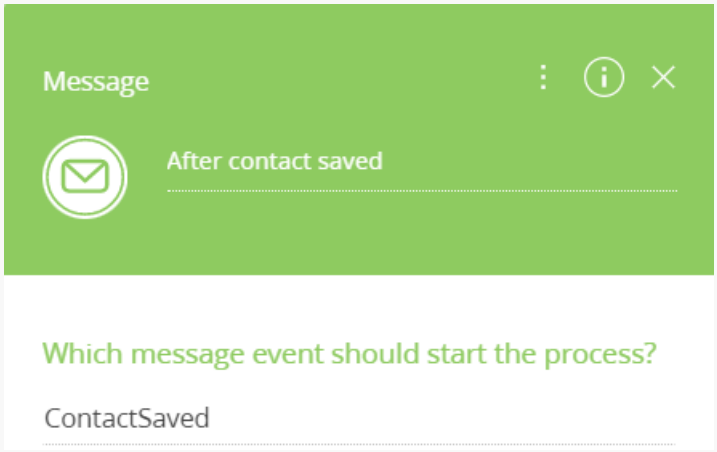


   c. Set the [*Title*] property in the element setup area to "`Contact Saved Sub-process`."

   d. Set up the **event sub-process elements**.

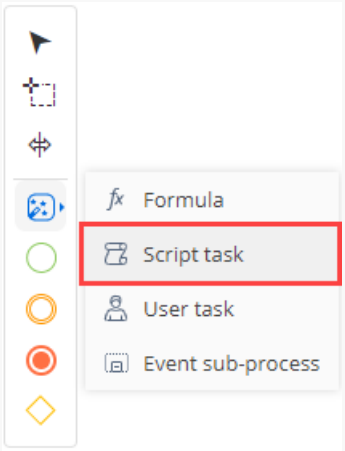
      a. Set up the [*Message*] **start event**.

         - Set [*Title*] to "`After contact saved`."
         - Set [*Which message event should start the process?*] to "`ContactSaved`."



d. Add the [*Script task*] **system action**.

   a. Click [*System actions*] in the element area of the Designer and drag the [*Script task*] system action to the working area of the sub-process.



   b. Name the [*Script task*] system action "`Publish a message via WebSocket.`"

   c. Add the code of the [*Script task*] system action.

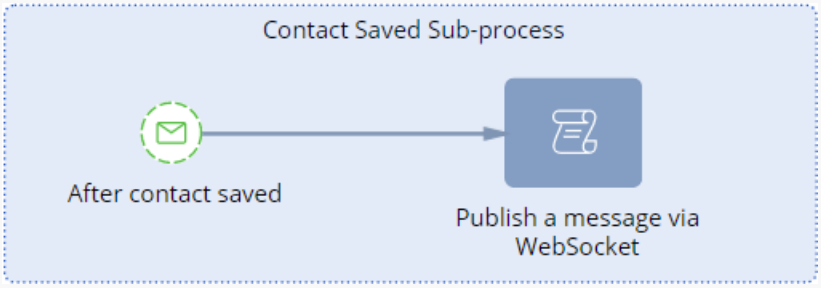
Code of the [*Script task*] system action

```
/* Receive the contact name. */
string userName = Entity.GetTypedColumnValue<string>("Name");
/* Receive the date of the contact birthday. */
DateTime birthDate = Entity.GetTypedColumnValue<DateTime>("BirthDate");
/* Generate the message text. */
string messageText = "{\"birthday\": \"" + birthDate.ToString("s") + "\", \"name\": \
/* Set the message name. */
string sender = "NewMessage";
/* Publish the message via WebSocket. */
MsgChannelUtilities.PostMessageToAll(sender, messageText);
return true;
```

d. Click [*Save*] on the Process Designer's toolbar.

e. Set up the **sequence flow**. Click in the menu of the [*Message*] start event and connect the [*Message*] start event to the [*Publish a message via WebSocket*] system action.

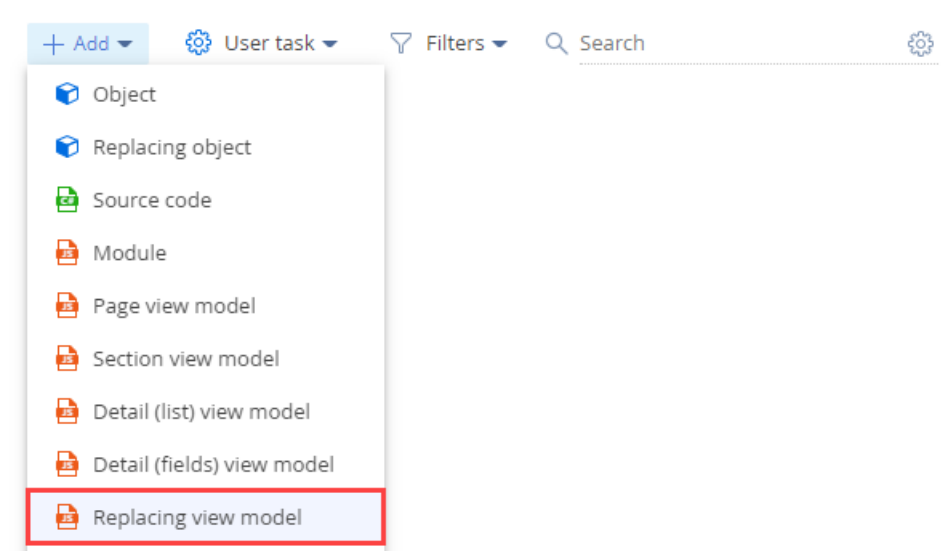
View the event sub-process in the figure below.



6. Click [*Save*] then [*Publish*] on the Process Designer's toolbar.

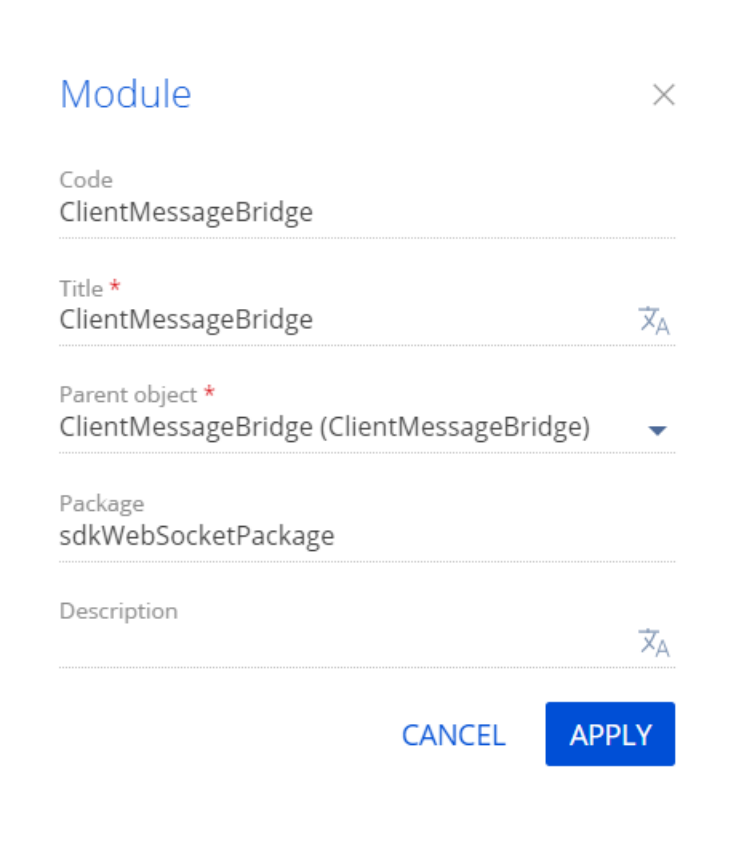
2. Implement message sending in Creatio

1. Open the [*Configuration*] section and select a custom package to add the schema.
2. Click [*Add*] → [*Replacing view model*] on the section list toolbar.



3. Fill out the **schema properties**.

- Set [*Code*] to "`ClientMessageBridge`."
- Set [*Title*] to "`ClientMessageBridge`."
- Select "`ClientMessageBridge`" in the [*Parent object*] property.



4. Implement sending of the `NewMessage` broadcast message.

   - In the `messages` property, bind the `NewMessage` broadcast message that can be published in Creatio.
   - Overload the following parent methods in the `methods` property:
     - `init()`. Adds a message received via WebSocket to the schema's message configuration object.
     - `afterPublishMessage`. Monitors the message sending.

   View the source code of the replacing view model schema below.

`ClientMessageBridge`

```
define("ClientMessageBridge", ["ConfigurationConstants"], function(ConfigurationConstants) {
    return {
        /* Messages. */
        messages: {
            /* Message name. */
            "NewMessage": {
                /* Broadcast message. */
                "mode": Terrasoft.MessageMode.BROADCAST,
                /* The message direction is publishing. */
                "direction": Terrasoft.MessageDirectionType.PUBLISH
            }
        },
        /* Methods. */
```

```
        methods: {
            /* Initialize the schema. */
            init: function() {
                /* Call the parent method. */
                this.callParent(arguments);
                /* Add a new configuration object to the collection of configuration objects.
                this.addMessageConfig({
                    /* The name of the message received via WebSocket. */
                    sender: "NewMessage",
                    /* The name of the WebSocket message sent in Creatio via sandbox. */
                    messageName: "NewMessage"
                });
            },
            /* Method executed after the message is published. */
            afterPublishMessage: function(
                /* The name of the message used to send the message. */
                sandboxMessageName,
                /* Message body. */
                webSocketBody,
                /* Result of sending the message. */
                result,
                /* Configuration object that sends the message. */
                publishConfig) {
                /* Verify that the message matches the message added to the configuration obj
                if (sandboxMessageName === "NewMessage") {
                    /* Save the body to local variables. */
                    var birthday = webSocketBody.birthday;
                    var name = webSocketBody.name;
                    /* Display the body in the browser console. */
                    window.console.info("Published message: " + sandboxMessageName +
                        ". Name: " + name +
                        "; birthday: " + birthday);
                }
            }
        }
    };
});
```

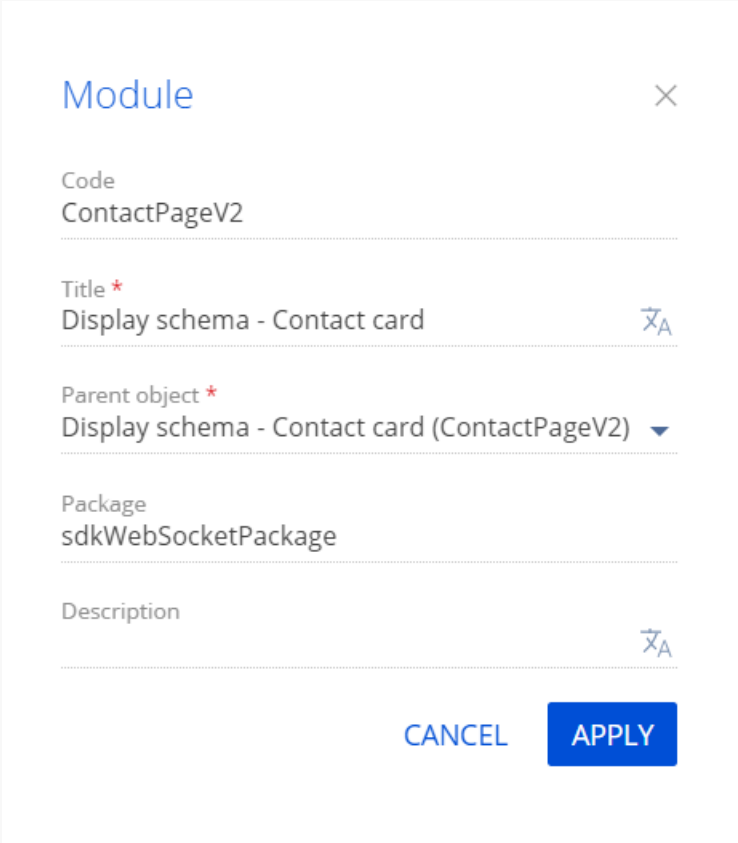
5. Click [*Save*] on the Designer's toolbar.

3. Implement subscription to the message

1. Open the [*Configuration*] section and select a custom package to add the schema.
2. Click [*Add*] → [*Replacing view model*] on the section list toolbar.
3. Fill out the **schema properties**.

   - Set [*Code*] to "`ContactPageV2`."

- Set [*Title*] to "`Display schema - Contact card`."
- Select "`ContactPageV2`" in the [*Parent object*] property.



4. Implement subscription to the `NewMessage` broadcast message.

   - In the `messages` property, bind the `NewMessage` broadcast message to subscribe.
   - Overload the `init()` parent method in the `methods` property. The method subscribes to the `NewMessage` message. Implement the `onNewMessage()` handler method that handles the object received in the message and returns the result to the browser console.

   View the source code of the replacing view model schema below.

`ContactPageV2`

```
define("ContactPageV2", [], function(BusinessRuleModule, ConfigurationConstants) {
    return {
        /* Object schema name. */
        entitySchemaName: "Contact",
        messages: {
            /* Message name. */
            "NewMessage": {
                /* Broadcast message. */
                "mode": Terrasoft.MessageMode.BROADCAST,
                /* The message direction is subscription. */
```

```
                "direction": Terrasoft.MessageDirectionType.SUBSCRIBE
            }
        },
        /* Methods. */
        methods: {
            /* Initialize the schema. */
            init: function() {
                /* Call the parent method. */
                this.callParent(arguments);
                /* Subscribe to the NewMessage message. */
                this.sandbox.subscribe("NewMessage", this.onNewMessage, this);
            },
            /* Handle the reception event of the NewMessage message. */
            onNewMessage: function(args) {
                /* Save the message body to local variables. */
                var birthday = args.birthday;
                var name = args.name;
                /* Display the body in the browser console. */
                window.console.info("Received message: NewMessage. Name: " +
                    name + "; birthday: " + birthday);
            }
        }
    };
});
```

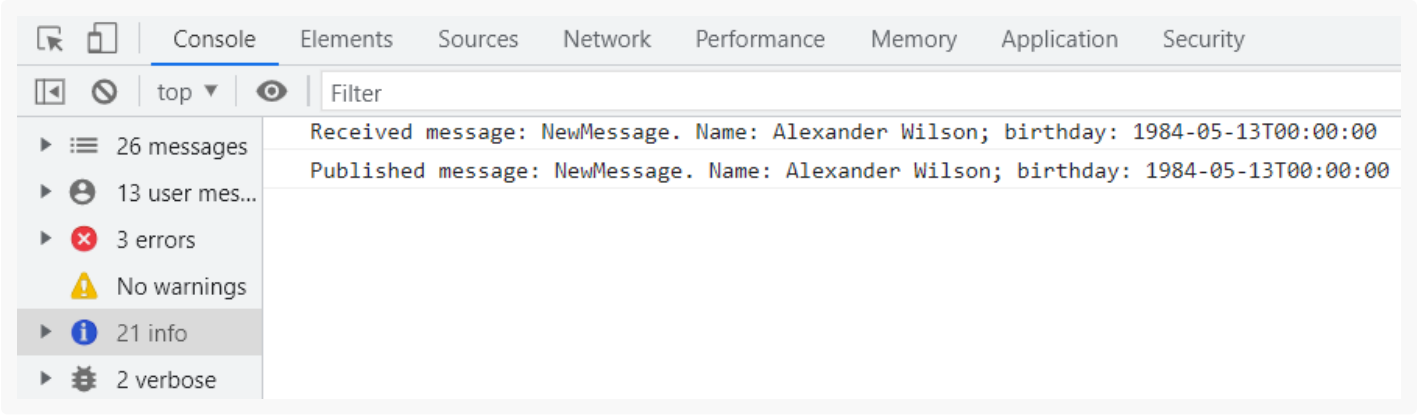
5. Click [*Save*] on the Designer's toolbar.

Outcome of the example

To **view the outcome of the example**:

1. Clear the browser cache.
2. Refresh the [*Contacts*] section page.
3. Open the contact page. For example, Alexander Wilson.
4. Open the [*Console*] tab in the browser console.
5. Modify an arbitrary field.
6. Save the contact.

As a result, the browser console will display the received and sent `NewMessage` messages.



ClientMessageBridge class JS

Advanced

Creatio broadcasts messages received via WebSocket to subscribers using the `ClientMessageBridge` client module schema.

Properties

`LocalStoreName`

The name of the repository that stores the message history.

`LocalStore` `Terrasoft.LocalStore`

A class instance that implements access to the local storage.

Methods

`init()`

Initializes the default value.

`saveMessageToHistory(sandboxMessageName, webSocketBody)`

Saves the message to the storage if the message has no subscribers and the configuration object indicates that saving is required.

Parameters

`sandboxMessageName: String`	The message name that Creatio uses while sending the message.
`webSocketBody: Object`	A message received via WebSocket.

`getMessagesFromHistory(sandboxMessageName)`

Returns an array of saved messages from repository.

Parameters

`sandboxMessageName: String`	The message name that Creatio uses while sending the message.

`deleteSavedMessages(sandboxMessageName)`

Deletes a saved message from repository.

Parameters

`sandboxMessageName: String`	The message name that Creatio uses while sending the message.

Data-operations (front-end)

 Advanced

Front-end Creatio modules implement data management using the high-level `EntitySchemaQuery` class that builds database selection queries.

The **major features** of the `EntitySchemaQuery` class:

- Data selection queries created using `EntitySchemaQuery` apply the access permissions of the current user.
- The caching mechanism lets you optimize operations by accessing cached query results without direct access to the database.

The data management **procedure** for front-end Creatio modules is as follows:

1. Create an instance of the `EntitySchemaQuery` class.
2. Specify the root schema.
3. Configure a path to the root schema column to add the column to the query.
4. Create filter instances.
5. Add the filters to the query.

6. Filter the query results.

Configure the column paths relative to the root schema

Base an `EntitySchemaQuery` query on the root schema. The **root schema** is the database table relative to which to build paths to the query columns, including the columns of the joined tables. To use a table column in a query, set the path to the column correctly.

Specify the column path using direct connections

The template for configuring the column path using the **direct connections** of `LookupColumnName.LookupSchema'sColumnSchemaName`.

For example, a `[City]` root schema contains a `[Country]` lookup column. The column is connected to the `[Country]` lookup schema via the `Id` column.

The path to the column that contains the name of the country connected to the city, built using the direct connections `Country.Name`. Where:

- `Country` is the name of the lookup column in the `[City]` root schema. Links to the `[Country]` schema.
- `Name` is the name of the column in the `[Country]` lookup schema.

Specify the column path using the reverse connections

The template for configuring the column path using the **reverse connections**

```
[JoinableSchemaName:NameOfTheColumnToLinkTheJoinableSchema:NameOfTheColumnToLinkTheCurrentSchema].
JoinableSchema'sColumnName
```

.

The path to the column that contains the name of the contact who added the city, built using the reverse connections `[Contact:Id:CreatedBy].Name`. Where:

- `Contact` is the name of the joinable schema.
- `Id` is the name of the `[Contact]` schema's column to connect the joinable schema.
- `CreatedBy` is the name of the `[City]` schema's lookup column to connect the current schema.
- `Name` is the value of the `[City]` schema's lookup column.

If the schema's lookup column to connect the current schema is `[Id]`, you do not have to specify it: `[JoinableSchemaName:NameOfTheColumnToConnectTheJoinableSchema].RootSchema'sColumnName`. For example, `[Contact:City].Name`.

Add the columns to the query

The `EntitySchemaQuery` query column is a `Terrasoft.EntityQueryColumn` class instance. Specify the main **features** in the column instance properties:

- header
- value for display
- usage flags
- sorting order and position

Add columns to the query using the `addColumn()` method that returns the query column instance. The `addColumn()` methods configure the name of the column relative to the root schema according to the rules described above: Configure the column paths relative to the root schema. The variations of the `addColumn()` method let you add query columns that contain different parameters. See the variations in the table below.

Variations of the `addColumn()` method

Column type	Method
The column by the specified path relative to the root schema	`addColumn(column, columnAlias)`
The instance of the query column class	
The parameter column	`addParameterColumn(paramValue, paramDataType, columnAlias)`
The function column	`addFunctionColumn(columnPath, functionType, columnAlias)`
The aggregative function column	`addAggregationSchemaColumnFunctionColumn(columnPath, aggregationType, columnAlias)`

Retrieve the query results

The **result** of the `EntitySchemaQuery` query is a collection of Creatio entities. Each collection instance is a dataset string returned by the query.

The **ways to retrieve the query results** are as follows:

- Call the `getEntity()` method to retrieve a particular dataset string by the specified primary key.
- Call the `getEntityCollection()` method to retrieve the entire resulting dataset.

Manage the query filters

Filters are sets of conditions applied when displaying the query data. In SQL terms, a filter is a separate predicate (condition) of the `WHERE` operator.

To create a simple filter in `EntitySchemaQuery`, use the `createFilter()` method that returns the created object of the `Terrasoft.CompareFilter` filter. `EntitySchemaQuery` implements methods that create special kinds of filters.

The `EntitySchemaQuery` instance contains the `filters` property that represents the filter collection of the query (the `Terrasoft.FilterGroup` class instance). The `Terrasoft.FilterGroup` class instance is a collection of `Terrasoft.BaseFilter` elements.

Follow this procedure to add a filter to the query:

- Create a filter instance for the query (the `createFilter()` method, methods that create special kinds of filters).
- Add the filter instance to the query filter collection (the `add()` collection method).

By default, Creatio uses the logical `AND` operation to combine the filters added to the `filters` collection. The `logicalOperation` property of the `filters` collection lets you specify the logical operation for combining filters. The `logicalOperation` accepts the values of the `Terrasoft.core.enums.LogicalOperatorType` enumeration (`AND`, `OR`).

`EntitySchemaQuery` queries let you manage the filters involved in the creation of the resulting dataset. Each element of the `filters` collection includes the `isEnabled` property that determines whether the element takes part in building the resulting query (`true` / `false`). Creatio defines the similar `isEnabled` property for the `filters` collection. If you set this property to `false`, the query filtering will be disabled, yet the query filters collection will remain unchanged. Thus, you can create a query filters collection and use various combinations without modifying the collection directly.

Configure the column paths in the `EntitySchemaQuery` filters according to the general rules for configuring the paths to columns relative to the root schema.

Examples that configure the column paths

Medium

Column path relative to the root schema

- The root schema: `[Contact]`.
- The column that contains the contact address: `Address`.

Example that creates an `EntitySchemaQuery` query to return the values of this column

```
/* Create an instance of the EntitySchemaQuery class with the Contact root schema. */
var esq = this.Ext.create("Terrasoft.EntitySchemaQuery", {
    rootSchemaName: "Contact"
});
/* Add an Address column, set its alias to Address. */
esq.addColumn("Address", "Address");
```

Column path that uses the direct connections

- The root schema: `[Contact]`.

- The column that contains the account name: `Account.Name`.
- The column that contains the name of the account's primary contact: `Account.PrimaryContact.Name`.

Example that creates an `EntitySchemaQuery` query to return the values of these columns

```
/* Create an instance of the EntitySchemaQuery class with the Contact root schema. */
var esq = this.Ext.create("Terrasoft.EntitySchemaQuery", {
    rootSchemaName: "Contact"
});
/* Add an Account lookup column. Then add the Name column from the Account schema linked to the
esq.addColumn("Account.Name", "AccountName");
/* Add an Account lookup column. Then add the PrimaryContact lookup column from the Account sche
esq.addColumn("Account.PrimaryContact.Name", "PrimaryContactName");
```

Column path that uses the reverse connections

- The root schema: `[Contact]`.
- The column that contains the name of the contact who added the city: `[Contact:Id:CreatedBy].Name`.

Example that creates an `EntitySchemaQuery` query to return the values of this column

```
/* Create an EntitySchemaQuery class instance with the Contact root schema. */
var esq = this.Ext.create("Terrasoft.EntitySchemaQuery", {
    rootSchemaName: "Contact"
});
/* Join another Contact schema to the root schema by the Owner column and select the Name columr
esq.addColumn("[Contact:Id:Owner].Name", "OwnerName");
/* Join the Contact schema to the Account lookup column by the PrimaryContact column and select
esq.addColumn("Account.[Contact:Id:PrimaryContact].Name", "PrimaryContactName");
```

Examples that add columns to the query

Medium

Column from the root schema

Example. Add the column from the root schema to the query column collection.

Example that adds the column from the root schema to the query column collection

```
var esq = this.Ext.create(Terrasoft.EntitySchemaQuery, {
    rootSchemaName: "Activity"
});
esq.addColumn("DurationInMinutes", "ActivityDuration");
```

Aggregate column

Example 1. Add the aggregate column to the query column collection. The column must have the `SUM` aggregation type that applies to all table records.

Example that adds the aggregate column to the query column collection

```
var esq = this.Ext.create(Terrasoft.EntitySchemaQuery, {
    rootSchemaName: "Activity"
});
esq.addAggregationSchemaColumn("DurationInMinutes", Terrasoft.AggregationType.SUM, "ActivitiesDu
```

Example 2. Add the aggregate column to the query column collection. The column must have the `COUNT` aggregation type that applies to unique table records.

Example that adds the aggregate column to the query column collection

```
var esq = this.Ext.create(Terrasoft.EntitySchemaQuery, {
    rootSchemaName: "Activity"
});
esq.addAggregationSchemaColumn("DurationInMinutes", Terrasoft.AggregationType.COUNT, "UniqueActi
```

Parameter column

Example. Add the parameter column with `TEXT` data type to the query column collection.

Example that adds the parameter column to the query column collection

```
var esq = this.Ext.create(Terrasoft.EntitySchemaQuery, {
    rootSchemaName: "Activity"
});
```

```
esq.addParameterColumn("DurationInMinutes", Terrasoft.DataValueType.TEXT, "DurationColumnName");
```

Function column

Example 1. Add the function column with `LENGTH` (value size, in bytes) data type to the query column collection.

Example that adds the function column to the query column collection

```
var esq = this.Ext.create(Terrasoft.EntitySchemaQuery, {
    rootSchemaName: "Activity"
});
esq.addFunctionColumn("Photo.Data", Terrasoft.FunctionType.LENGTH, "PhotoLength");
```

Example 2. Add the function column with `DATE_PART` (date part) data type to the query column collection. Use the day of the week as the value.

Example that adds the function column to the query column collection.

```
var esq = this.Ext.create(Terrasoft.EntitySchemaQuery, {
    rootSchemaName: "Activity"
});
esq.addDatePartFunctionColumn("StartDate", Terrasoft.DatePartType.WEEK_DAY, "StartDay");
```

Example 3. Add the function column to the query column collection. The column must have the `MACROS` type that does not need to be parameterized: `PRIMARY_DISPLAY_COLUMN` (primary column for display).

Example that adds the function column to the query column collection

```
var esq = this.Ext.create(Terrasoft.EntitySchemaQuery, {
    rootSchemaName: "Activity"
});
esq.addMacrosColumn(Terrasoft.QueryMacrosType.PRIMARY_DISPLAY_COLUMN, "PrimaryDisplayColumnValue
```

Examples that retrieve the query results

Medium

Dataset string by the specified primary key

Example. Retrieve a particular dataset string by the specified primary key

Example that retrieves a particular dataset string

```
/* Retrieve the ID of the mini page object. */
var recordId = this.get("Id");
/* Create an instance of the Terrasoft.EntitySchemaQuery class with the Contact root schema. */
var esq = this.Ext.create("Terrasoft.EntitySchemaQuery", {
    rootSchemaName: "Contact"
});
/* Add a column that contains the name of the account’s primary contact. */
esq.addColumn("Account.PrimaryContact.Name", "PrimaryContactName");
/* Retrieve one record from the selection by the ID of the mini page object. Display the record
esq.getEntity(recordId, function(result) {
    if (!result.success) {
        // For example, error processing/logging.
        this.showInformationDialog("Data query error");
        return;
    }
    this.showInformationDialog(result.entity.get("PrimaryContactName"));
}, this);
```

Note. If you retrieve lookup columns, the `this.get()` returns the object, not ID of the record in the database. To retrieve the ID, use the `value` property. For example, `this.get('Account').value`.

Resulting dataset

Example. Retrieve the entire resulting dataset.

Example that retrieves the entire dataset

```
var message = "";
/* Create an instance of the Terrasoft.EntitySchemaQuery class with the Contact root schema. */
var esq = Ext.create("Terrasoft.EntitySchemaQuery", {
    rootSchemaName: "Contact"
});
```

```
/* Add a column that contains the name of the account connected to the contact. */
esq.addColumn("Account.Name", "AccountName");
/* Add a column that contains the name of the account's primary contact. */
esq.addColumn("Account.PrimaryContact.Name", "PrimaryContactName");
/* Retrieve the entire record collection and display it in the information box. */
esq.getEntityCollection(function (result) {
    if (!result.success) {
        /* For example, error processing/logging. */
        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);
```

Examples that manage the query filters

 Medium

Example that manages the query filters

```
/* Create a query instance with the Contact root schema. */
var esq = Ext.create("Terrasoft.EntitySchemaQuery", {
    rootSchemaName: "Contact"
});
esq.addColumn("Name");
esq.addColumn("Country.Name", "CountryName");

/* Create a first filter instance. */
var esqFirstFilter = esq.createColumnFilterWithParameter(Terrasoft.ComparisonType.EQUAL, "Countr

/* Create a second filter instance. */
var esqSecondFilter = esq.createColumnFilterWithParameter(Terrasoft.ComparisonType.EQUAL, "Count

/* Combine the filters in the query filters collection using the OR logical operator. */
esq.filters.logicalOperation = Terrasoft.LogicalOperatorType.OR;

/* Add the filters to the query collection. */
esq.filters.add("esqFirstFilter", esqFirstFilter);
esq.filters.add("esqSecondFilter", esqSecondFilter);

/* Add the objects (query results) filtered by the two filters to the collection. */
esq.getEntityCollection(function (result) {
```

```
    if (result.success) {
        result.collection.each(function (item) {
            /* Process the collection elements. */
        });
    }
}, this);

/* Specify that the second filter is not used to build the resulting query. At the same time, do
esqSecondFilter.isEnabled = false;

/* Add the objects (query results) filtered only by the first filter to the collection. */
esq.getEntityCollection(function (result) {
    if (result.success) {
        result.collection.each(function (item) {
            /* Process the collection elements. */
        });
    }
}, this);
```

Example that uses other filter creation methods

```
/* Create a query instance with the Contact root schema. */
var esq = Ext.create("Terrasoft.EntitySchemaQuery", {
    rootSchemaName: "Contact"
});
esq.addColumn("Name");
esq.addColumn("Country.Name", "CountryName");

/* Select all contacts that do not have a country specified. */
var esqFirstFilter = esq.createColumnIsNullFilter("Country");

/* Select all contacts that have birth dates between 01.1.1970 and 01.1.1980. */
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 the created filters to the query collection. */
esq.filters.add("esqFirstFilter", esqFirstFilter);
esq.filters.add("esqSecondFilter", esqSecondFilter);

/* Add the objects (query results) filtered by the two filters to the collection. */
esq.getEntityCollection(function (result) {
    if (result.success) {
        result.collection.each(function (item) {
            /* Process the collection elements. */
        });
    }
```

```
}, this);
```

EntitySchemaQuery class JS

Medium

The `EntitySchemaQuery` class builds queries to select records in a database.

Methods

abortQuery()

Aborts the query.

addAggregationSchemaColumn(columnPath, aggregationType, [columnAlias], aggregationEvalType)

Creates an instance of the `Terrasoft.FunctionQueryColumn` function column with the specified `AGGREGATION` type. Adds the instance to the query column collection.

Parameters

`{String} columnPath`	The path to the column to add, relative to `rootSchema`.

`{Terrasoft.AggregationType} aggregationType`	The aggregate function type.
	
	Available values (`Terrasoft.AggregationType`)
	
	`AVG`
	The average of all elements.
	
	`COUNT`
	The number of elements.
	
	`MAX`
	The maximal element.
	
	`MIN`
	The minimal element.

	`NONE`
	The aggregate function type is undefined.
	
	`SUM`
	The sum of all elements.

`{String} columnAlias`	The column alias. Optional.

`{Terrasoft.AggregationEvalType} aggregationEvalType`	The aggregate function scope.
	
	Available values (`Terrasoft.AggregationEvalType`)
	
	`NONE`
	The aggregate function scope is undefined.
	
	`ALL`
	Apply to all elements.
	
	`DISTINCT`
	Apply to unique values.

`addColumn(column, [columnAlias], [config])`

Creates an instance of the `Terrasoft.EntityQueryColumn` column. Adds the instance to the query column collection.

Parameters

`{String/Terrasoft.BaseQueryColumn} column`	The path to the column to add (relative to `rootSchema`) or an instance of the `Terrasoft.BaseQueryColumn` query column.
`{String} columnAlias`	The column alias. Optional.
`{Object} config`	The configuration object of the query column.

`addDatePartFunctionColumn(columnPath, datePartType, [columnAlias])`

Creates an instance of the `Terrasoft.FunctionQueryColumn` function column with the `DATE_PART` type. Adds the

instance to the query column collection.

Parameters

`{String} columnPath`	The path to the column to add, relative to `rootSchema`.

`{Terrasoft.DatePartType} datePartType`	The date part to use as the value.
	
	Available values (`Terrasoft.DatePartType`)
	
	`NONE`
	Empty value.
	
	`DAY`
	Day.
	
	`WEEK`
	Week.
	
	`MONTH`
	Month.
	
	`YEAR`
	Year.
	
	`WEEK_DAY`
	Day of the week.
	
	`HOUR`
	Hour.
	
	`HOUR_MINUTE`
	Minute.

`{String} columnAlias`	The column alias. Optional.

`addDatePeriodMacrosColumn(macrosType, [macrosValue], [columnAlias])`

Creates an instance of the `Terrasoft.FunctionQueryColumn` function column with the `MACROS` type that must be parameterized. Adds the instance to the query column collection. For example, the following N days, the third quarter of the year, etc.

Parameters

`{Terrasoft.QueryMacrosType} macrosType`	The column macro type.
`{Number/Date} macrosValue`	The auxiliary variable for the macro. Optional.
`{String} columnAlias`	The column alias. Optional.

`addFunctionColumn(columnPath, functionType, [columnAlias])`

Creates an instance of the `Terrasoft.FunctionQueryColumn` function column. Adds the instance to the query column collection.

Parameters

`{String} columnPath`	The path to the column to add, relative to `rootSchema`.

`{Terrasoft.FunctionType} functionType`	The function type.
	Available values (`Terrasoft.FunctionType`)
	`NONE`
	The functional expression type is undefined.
	`MACROS`
	The insertion that uses a macro.
	`AGGREGATION`
	The aggregate function.
	`DATE_PART`
	The date part.
	`LENGTH`
	The value size, in bytes. Use with binary data.

`{String} columnAlias`	The column alias. Optional.

`addMacrosColumn(macrosType, [columnAlias])`

Creates an instance of the `Terrasoft.FunctionQueryColumn` function column with the `MACROS` type that does not need to be parameterized. For example, current month, current user, primary column, etc. Adds the instance to the query column collection.

Parameters

`{Terrasoft.QueryMacrosType} macrosType`	The column macro type. Available values (`Terrasoft.QueryMacrosType`) `NONE` The macro type is undefined.

`CURRENT_USER`

The current user.

`CURRENT_USER_CONTACT`

The current user contact.

`YESTERDAY`

Yesterday.

`TODAY`

Today.

`TOMORROW`

Tomorrow.

`PREVIOUS_WEEK`

Previous week.

`CURRENT_WEEK`

Current week.

`NEXT_WEEK`

Next week.

`PREVIOUS_MONTH`

Previous month.

`CURRENT_MONTH`

Current month.

`NEXT_MONTH`

Next month.

`PREVIOUS_QUARTER`

Previous quarter.

`CURRENT_QUARTER`

Current quarter.

`NEXT_QUARTER`

Next quarter.

`PREVIOUS_HALF_YEAR`

Previous 6 months.

`CURRENT_HALF_YEAR`

Current 6 months.

`NEXT_HALF_YEAR`

Next 6 months.

`PREVIOUS_YEAR`

Previous year.

`CURRENT_YEAR`

Current year.

`PREVIOUS_HOUR`

Previous hour.

`CURRENT_HOUR`

Current hour.

`NEXT_HOUR`

	NEXT_HOUR Next hour. NEXT_YEAR Next year. NEXT_N_DAYS Next N days. Must be parameterized. PREVIOUS_N_DAYS Previous N days. Must be parameterized. NEXT_N_HOURS Next N hours. Must be parameterized. PREVIOUS_N_HOURS Previous N hours. Must be parameterized. PRIMARY_COLUMN The primary column. PRIMARY_DISPLAY_COLUMN The primary column for display. PRIMARY_IMAGE_COLUMN The primary column for image display.
`{String} columnAlias`	The column alias. Optional.

`addParameterColumn(paramValue, paramDataType, [columnAlias])`

Creates an instance of the `Terrasoft.ParameterQueryColumn` parameter column. Adds the instance to the query column collection.

Parameters

{Mixed} paramValue	The parameter value. Must correspond to the data type.
{Terrasoft.DataValueType} paramDataType	The parameter data type.
{String} columnAlias	The column alias. Optional.

---

createBetweenFilter(leftExpression, rightLessExpression, rightGreaterExpression)

Creates a `Between` filter instance.

Parameters

{Terrasoft.BaseExpression} leftExpression	The expression to check in the filter.
{Terrasoft.BaseExpression} rightLessExpression	The initial expression of the filtering scope.
{Terrasoft.BaseExpression} rightGreaterExpression	The final expression of the filtering scope.

---

createColumnBetweenFilterWithParameters(columnPath, lessParamValue, greaterParamValue, paramData

Creates a `Between` filter instance to check if the column is within the specified scope.

Parameters

{String} columnPath	The path to the check column, relative to the `rootSchema` root schema.
{Mixed} lessParamValue	The initial value of the filter.
{Mixed} greaterParamValue	The final value of the filter.
{Terrasoft.DataValueType} paramDataType	The parameter data type.

`createColumnFilterWithParameter(comparisonType, columnPath, paramValue, paramDataType)`

Creates a `Compare` filter to instance to compare the column to a specified value.

Parameters

`{Terrasoft.ComparisonType} comparisonType`	The comparison operation type.
`{String} columnPath`	The path to the check column, relative to the `rootSchema` root schema.
`{Mixed} paramValue`	The parameter value.
`{Terrasoft.DataValueType} paramDataType`	The parameter data type.

`createColumnInFilterWithParameters(columnPath, paramValues, paramDataType)`

Creates an `In` filter instance to compare the value of a specified column to a parameter.

Parameters

`{String} columnPath`	The path to the check column, relative to the `rootSchema` root schema.
`{Array} paramValues`	The parameter value array.
`{Terrasoft.DataValueType} paramDataType`	The parameter data type.

`createColumnIsNotNullFilter(columnPath)`

Creates an `IsNull` filter instance to check the specified column.

Parameters

`{String} columnPath`	The path to the check column, relative to the `rootSchema` root schema.

`createColumnIsNullFilter(columnPath)`

Creates an `IsNull` filter instance to check the specified column.

Parameters

`{String} columnPath`	The path to the check column, relative to the `rootSchema` root schema.

`createCompareFilter(comparisonType, leftExpression, rightExpression)`

Creates a `Compare` filter instance.

Parameters

`{Terrasoft.ComparisonType} comparisonType`	The comparison operation type.
`{Terrasoft.BaseExpression} leftExpression`	The expression to check in the filter.
`{Terrasoft.BaseExpression} rightExpression`	The filtering expression.

`createExistsFilter(columnPath)`

Creates an `Exists` filter instance for the `[Exists by the specified condition]` type comparison. Sets the expression of the column at the specified path as the check value.

Parameters

`{String} columnPath`	The path to the column for whose expression to configure the filter.

`createFilter(comparisonType, leftColumnPath, rightColumnPath)`

Creates an instance of the `Terrasoft.CompareFilter` class filter to compare the values of two columns.

Parameters

`{Terrasoft.ComparisonType} comparisonType`	The comparison operation type.
`{String} leftColumnPath`	The path to the check column, relative to the `rootSchema` root schema.
`{String} rightColumnPath`	The path to the filter column, relative to the `rootSchema` root schema.

`createFilterGroup()`

Creates a filter group instance.

`createInFilter(leftExpression, rightExpressions)`

Creates an `In` filter instance.

Parameters

`{Terrasoft.Base Expression} left Expression`	The expression to check in the filter.
`{Terrasoft.Base Expression[]} right Expressions`	The array of expressions to compare to `leftExpression`.

`createIsNotNullFilter(leftExpression)`

Creates an `IsNull` filter instance.

Parameters

`{Terrasoft.Base Expression} left Expression`	The expression to check by the `IS NOT NULL` condition.

`createIsNullFilter(leftExpression)`

Creates an `IsNull` filter instance.

Parameters

`{Terrasoft.Base Expression} left Expression`	The expression to check by the `IS NULL` condition.

`createNotExistsFilter(columnPath)`

Creates an `Exists` filter instance for the `[Does not exist by the specified condition]` type comparison. Sets the expression of the column located at the specified path as the check value.

Parameters

`{String} columnPath`	The path to the column for whose expression to configure the filter.

`createPrimaryDisplayColumnFilterWithParameter(comparisonType, paramValue, paramDataType)`

Creates a filter object to compare the primary column to a parameter.

Parameters

`{Terrasoft.ComparisonType} comparisonType`	The comparison type.
`{Mixed} paramValue`	The parameter value.
`{Terrasoft.DataValueType} paramDataType`	The parameter data type.

`destroy()`

Deletes the object instance. If the object has already been deleted, writes an error message to the console. Calls the `onDestroy` virtual method to redefine in subclasses.

`enablePrimaryColumnFilter(primaryColumnValue)`

Enables filters by the primary key.

Parameters

`{String/Number} primaryColumnValue`	The value of the primary key.

`error(message)`

Writes an error message to the message log.

Parameters

`{String} message`	The error message to write to the message log.

`execute(callback, scope)`

The request to execute the query on the server.

Parameters

`{Function} callback`	The function to call after receiving the server response.
`{Object} scope`	The scope within which to call the `callback` function.

`{Object} getDefSerializationInfo()`

Returns the object that contains additional information for serialization.

`getEntity(primaryColumnValue, callback, scope)`

Returns the entity instance by the specified `primaryColumnValue` primary key. Calls the `callback` function within the `scope` scope after retrieving the data.

Parameters

`{String/Number} primaryColumnValue`	The value of the primary key.
`{Function} callback`	The function to call after receiving the server response.
`{Object} scope`	The scope within which to call the `callback` function.

`getEntityCollection(callback, scope)`

Returns the collection of entity instances that represent the query outcome. Calls the `callback` function within the `scope` scope after retrieving the data.

Parameters

`{Function} callback`	The function to call after receiving the server response.
`{Object} scope`	The scope within which to call the `callback` function.

`{Object} getTypeInfo()`

Returns the information about the element type.

`log(message, [type])`

Writes a message to the message log.

Parameters

`{String|Object} message`	The message to write to the message log.
`{Terrasoft.LogMessageType} type`	The type of the `callback` message log. Optional. By default, `console.log`. The `Terrasoft.core.enums.LogMessageType` enumeration specifies the available values..

`onDestroy()`

Deletes the event subscriptions and destroys the object.

`serialize(serializationInfo)`

Serializes the object in JSON.

Parameters

`{String} serializationInfo`	The results in JSON.

`setSerializableProperty(serializableObject, propertyName)`

Assigns the property name to the object if the object is not empty or not a function.

Parameters

`{Object} serializableObject`	The serializable object.
`{String} propertyName`	The property name.

`warning(message)`

Writes a warning message to the message log.

Parameters

`{String} message`	The message to write to the message log.

DataManager class JS

Medium

DataManager class JS

`DataManager` is a singleton class available via the `Terrasoft` global object. The class provides the `dataStore` repository. You can upload the contents of one or more database tables to the repository.

```
dataStore: {
    /* The DataManagerItem type data collection of the SysModule schema. */
    SysModule: sysModuleCollection,
 /* The DataManagerItem type data collection of the SysModuleEntity schema. */
    SysModuleEntity: sysModuleEntityCollection
}
```

Each record of the collection represents the record of the corresponding database table.

Properties

`{Object} dataStore`

The data collection repository.

`{String} itemClassName`

The record class name. Has the `Terrasoft.DataManagerItem` value.

Methods

`{Terrasoft.Collection} select(config, callback, scope)`

If `dataStore` does not contain a data collection that has the `config.entitySchemaName` name, the method builds and executes a database query, then returns the retrieved data. Otherwise, the method returns the data collection from `dataStore`.

Parameters

`{Object} config`	The configuration object.
	
	Configuration object properties
	
	`{String} entitySchemaName`
	The schema name.
	
	`{Terrasoft.FilterGroup} filters`
	The conditions.

`{Function} callback`	The callback function.
`{Object} scope`	The scope of the callback function.

`{Terrasoft.DataManagerItem} createItem(config, callback, scope)`

Creates a new record of the `config.entitySchemaName` type. The record columns have the `config.columnValues` values.

Parameters

`{Object} config`	The configuration object.
	
	Configuration object properties
	
	`{String} entitySchemaName`
	The schema name.
	
	`{Object} columnValues`
	The record column values.

`{Function} callback`	The callback function.
`{Object} scope`	The scope of the callback function.

`{Terrasoft.DataManagerItem} addItem(item)`

Adds the `item` record to the schema data collection.

Parameters

`{Terrasoft.DataManagerItem} item`	The record to add.

`{Terrasoft.DataManagerItem} findItem(entitySchemaName, id)`

Returns the record from the data collection of the schema that has the `entitySchemaName` name and `id` ID.

Parameters

`{String} entitySchemaName`	The data collection name.
`{String} id`	The record ID.

`{Terrasoft.DataManagerItem} remove(item)`

Selects the `isDeleted` flag for the `item` record. Once the changes are recorded, the record will be deleted from the database.

Parameters

`{Terrasoft.DataManagerItem} item`	The record to delete.

`removeItem(item)`

Deletes the record from the schema data collection.

Parameters

`{Terrasoft.DataManagerItem} item`	The record to delete.

`{Terrasoft.DataManagerItem} update(config, callback, scope)`

Updates the record that has the `config.primaryColumnValue` primary column value with the values from `config.columnValues`.

Parameters

`{Object} config`	The configuration object.
	
	Configuration object properties
	
	`{String} entitySchemaName`
	The schema name.
	
	`{String} primaryColumnValue`
	The primary column value.
	
	`{Mixed} columnValues`
	The column values.

`{Function} callback`	The callback function.
`{Object} scope`	The scope of the callback function.

`{Terrasoft.DataManagerItem} discardItem(item)`

Discards the changes to the `item` record made as part of the current `DataManager` session.

Parameters

`{Terrasoft.DataManagerItem} item`	The record, changes to which to discard.

`{Object} save(config, callback, scope)`

Saves the data collections of the schemas specified in `config.entitySchemaNames` to the database.

Parameters

`{Object} config`	The configuration object. Configuration object properties `{String[]} entitySchemaName` The name of the schema to save. Leave empty to save the data collections of all schemas.
`{Function} callback`	The callback function.
`{Object} scope`	The scope of the callback function.

DataManagerItem class JS

Properties

`{Terrasoft.BaseViewMode} viewModel`

The object projection of the database record.

Methods

`setColumnValue(columnName, columnValue)`

Assigns the new `columnValue` value to the column that has the `columnName` name.

Parameters

`{String} columnName`	The column name.
`{String} columnValue`	The column value.

`{Mixed} getColumnValue(columnName)`

Returns the value of the column that has the `columnName` name.

Parameters

`{String} columnName`	The column name.

`{Object} getValues()`

Returns the values of all record columns.

`remove()`

Selects the `isDeleted` flag for the record.

`discard()`

Discards the changes to the record made as part of the current `DataManager` session.

`{Object} save(callback, scope)`

Records the changes in the database.

Parameters

`{Function} callback`	The callback function.
`{Object} scope`	The scope of the callback function.

`{Boolean} getIsNew()`

Returns the flag that marks the record as new.

`{Boolean} getIsChanged()`

Returns the flag that marks the record as changed.

Use examples

Retrieve the records from the [*Contact*] table

```
/* Define the configuration object. */
var config = {
    /* The entity schema name. */
    entitySchemaName: "Contact",
    /* Exclude duplicates from the resulting dataset. */
    isDistinct: true
};
/* Retrieve the data. */
Terrasoft.DataManager.select(config, function (collection) {
    /* Save the retrieved records to the local repository. */
    collection.each(function (item) {
```

```
        Terrasoft.DataManager.addItem(item);
    });
}, this);
```

Add a new record to the `DataManager` object

```
/* Define the configuration object. */
var config = {
    /* The entity schema name. */
    entitySchemaName: "Contact",
    /* The column values. */
    columnValues: {
        Id: "00000000-0000-0000-0000-000000000001",
        Name: "Name1"
    }
};
/* Create a new record. */
Terrasoft.DataManager.createItem(config, function (item) {
    Terrasoft.DataManager.addItem(item);
}, this);
```

Retrieve the record and change the column value

```
/* Retrieve the record. */
var item = Terrasoft.DataManager.findItem("Contact",
     "00000000-0000-0000-0000-000000000001");
/* Assign the new "Name2" value to the Name column. */
item.setColumnValue("Name", "Name2");
```

Delete the record from `DataManager`

```
/* Define the configuration object. */
var config = {
    /* The entity schema name. */
    entitySchemaName: "Contact",
    /* The primary column value. */
    primaryColumnValue: "00000000-0000-0000-0000-000000000001"
};
/* Select the isDeleted flag for the item record. */
Terrasoft.DataManager.remove(config, function () {
}, this);
```

Discard the changes made as part of the current `DataManager` session.

```
/* Retrieve the record. */
var item = Terrasoft.DataManager.findItem("Contact",
     "00000000-0000-0000-0000-000000000001");
/* Discard the changes to the record. */
Terrasoft.DataManager.discardItem(item);
```

Record the changes in the database

```
/* Define the configuration object. */
var config = {
    /* The entity schema name. */
    entitySchemaNames: ["Contact"]
};
/* Save the changes in the database. */
Terrasoft.DataManager.save(config, function () {
}, this);
```

JS classes reference

 Advanced

Program interface documentation (JavaScript API) for platform core (JavaScript core) is available on a separate web resource https://academy.creatio.com/api/jscoreapi/7.15.0/index.html.