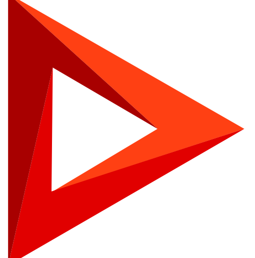
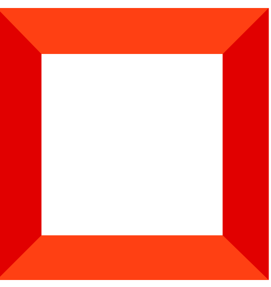
Modules

Version 8.0





This documentation is provided under restrictions on use and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this documentation, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

© 2023 Creatio. All rights reserved.

Table of Contents

Module basics **4**
AMD concept 4
Modular development in Creatio 4
RequireJS loader 5
Example that declares a module **5**
define() function **5**
Parameters 6
Module types **7**
Base modules 7
Client modules 7
Create a standard module **9**
Create a visual module 9
Outcome of the example 11
Create a utility module **12**
1. Create a utility module 12
2. Create a visual module 13
Outcome of the example 15
Module class **16**
Declare a module class 16
Inherit from a module class 17
Overload module class members 19
Initialize a module class instance 20
Module chain 22

© 2023 Creatio. All rights reserved.

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.

© 2023 Creatio. All rights reserved.

- 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

© 2023 Creatio. All rights reserved.

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.

© 2023 Creatio. All rights reserved.

- 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.

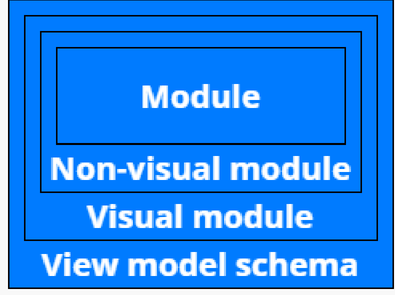
Client modules

Client module types

- non-visual module
- visual module
- replacing module

View the client module hierarchy below.

© 2023 Creatio. All rights reserved.



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.

© 2023 Creatio. All rights reserved.

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

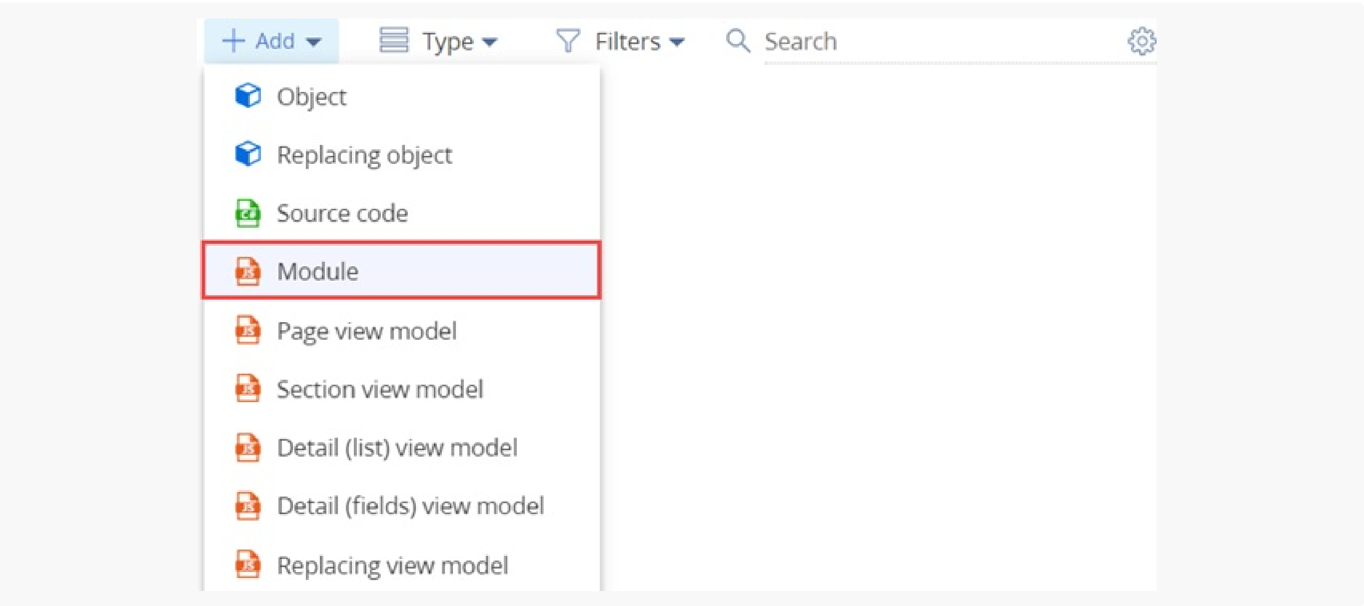
Medium

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

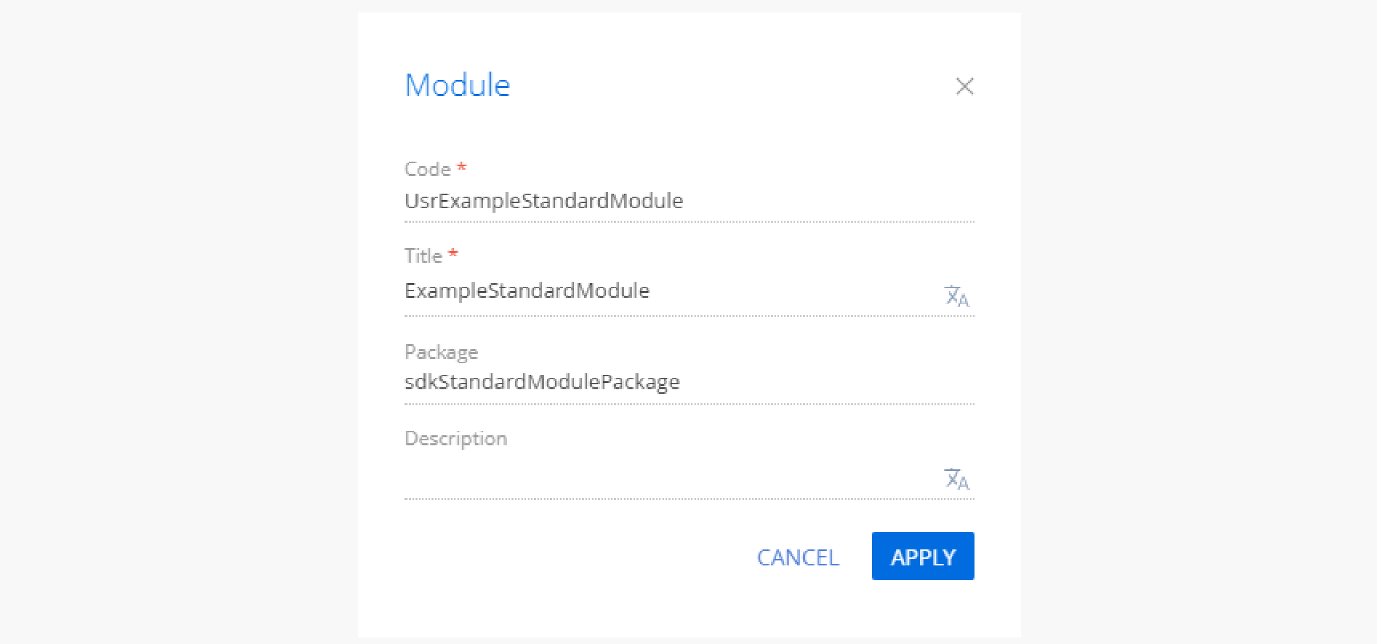
1. Open the [*Configuration*] section and select a custom package to add the schema.
2. Click [*Add*] → [*Module*] on the section list toolbar.

© 2023 Creatio. All rights reserved.



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 {
```

© 2023 Creatio. All rights reserved.

```
        /* 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

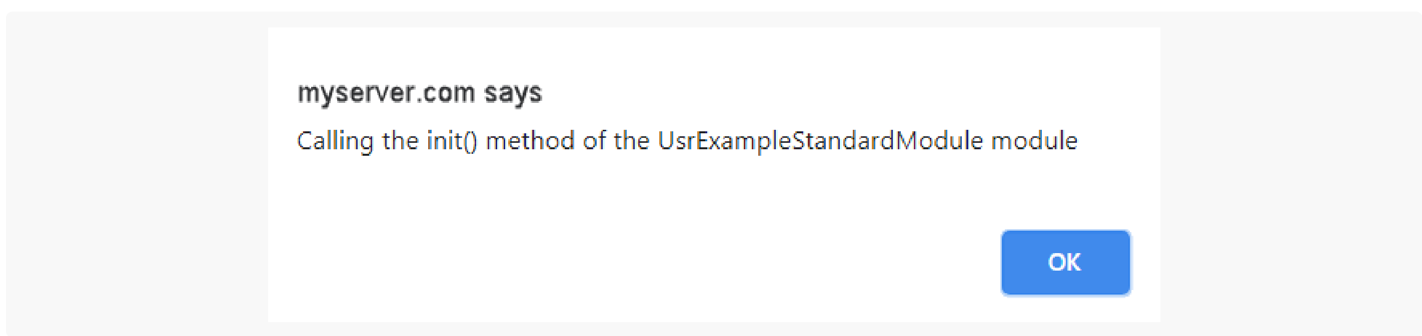
```
[Creatio URL]/0/NUI/ViewModule.aspx#[SomeModuleName]
```

Example URL

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

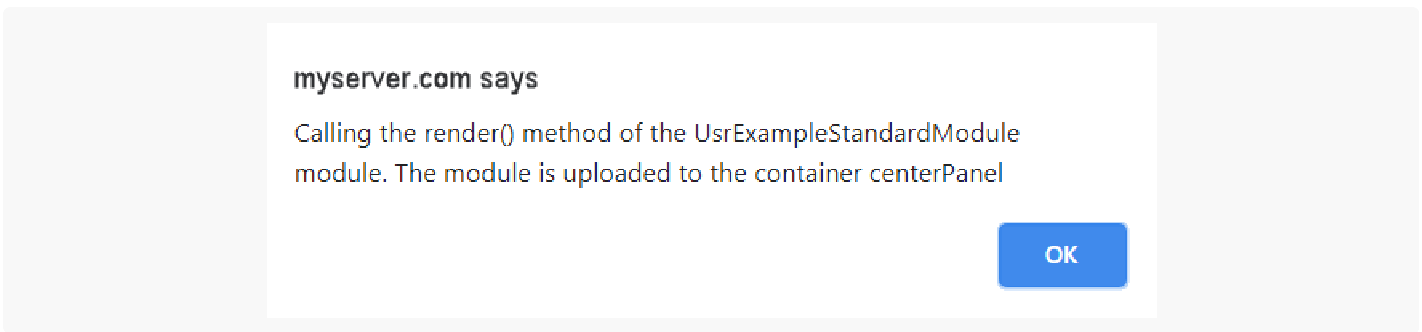
When loading the module, the client core must call the `init()` method, then the `render()` method. Each method calls a message box.

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



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

© 2023 Creatio. All rights reserved.



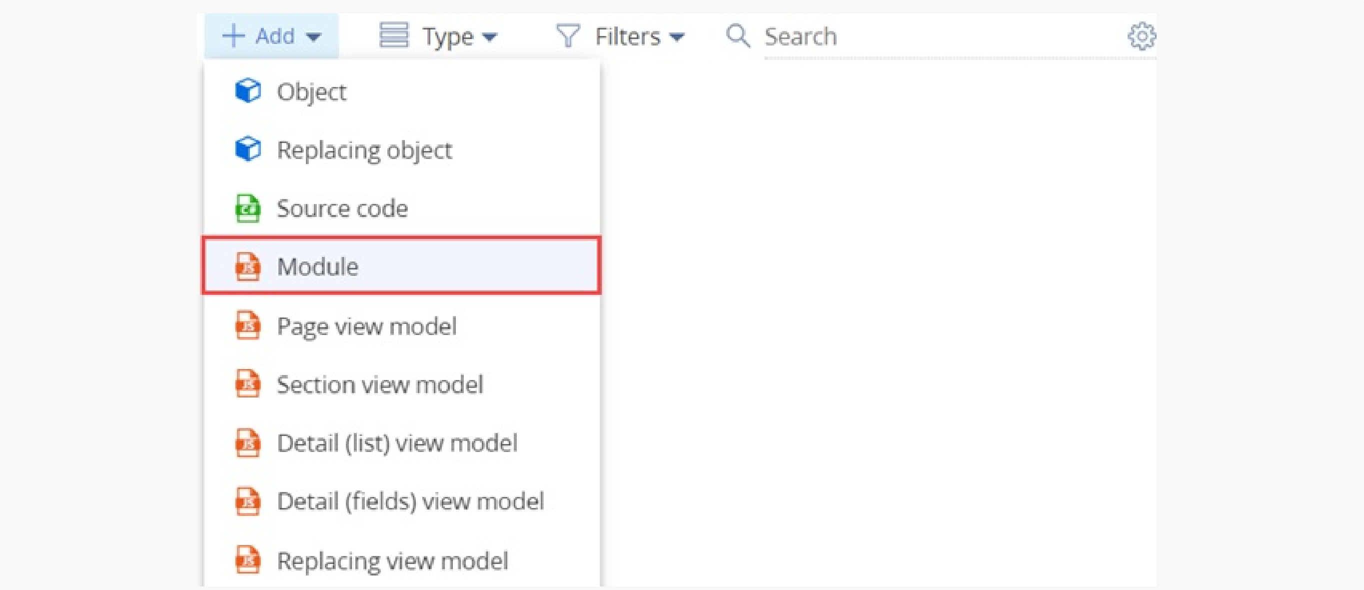
Create a utility module

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.

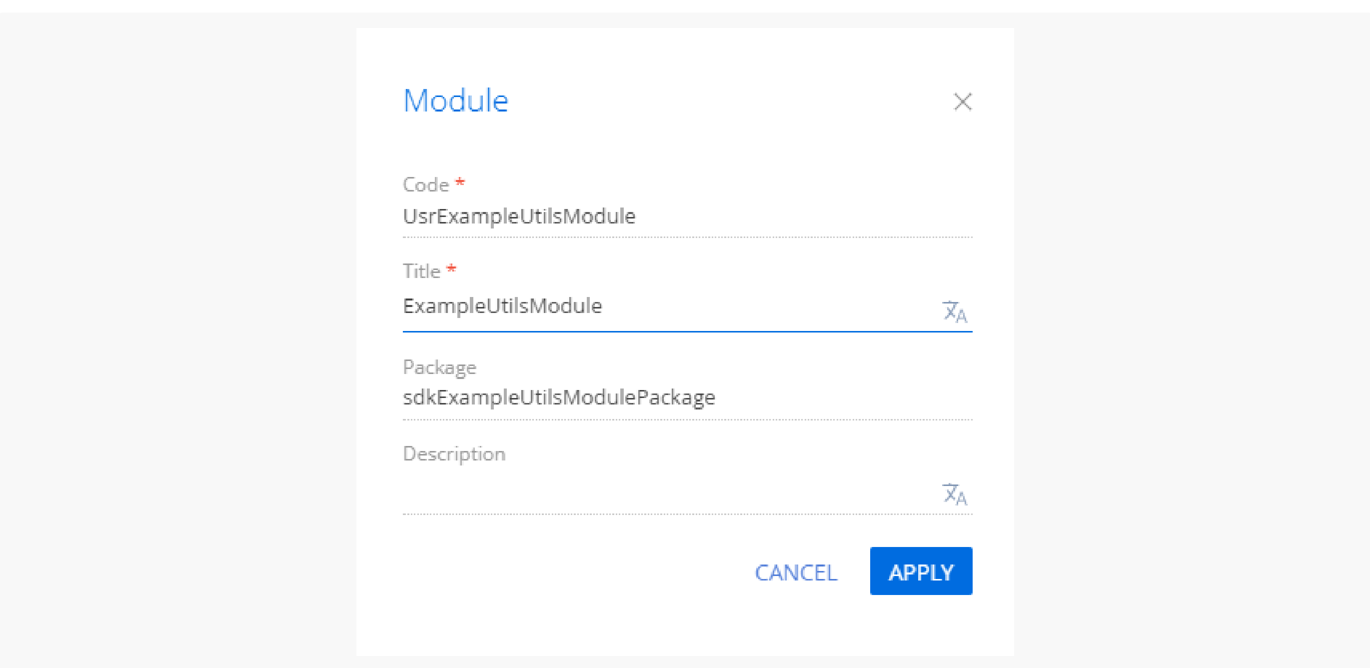
1. Create a utility module

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`."

© 2023 Creatio. All rights reserved.



Click [*Apply*] to apply the changes.

4. Add the source code in the Module Designer.

UsrExampleUtilsModule

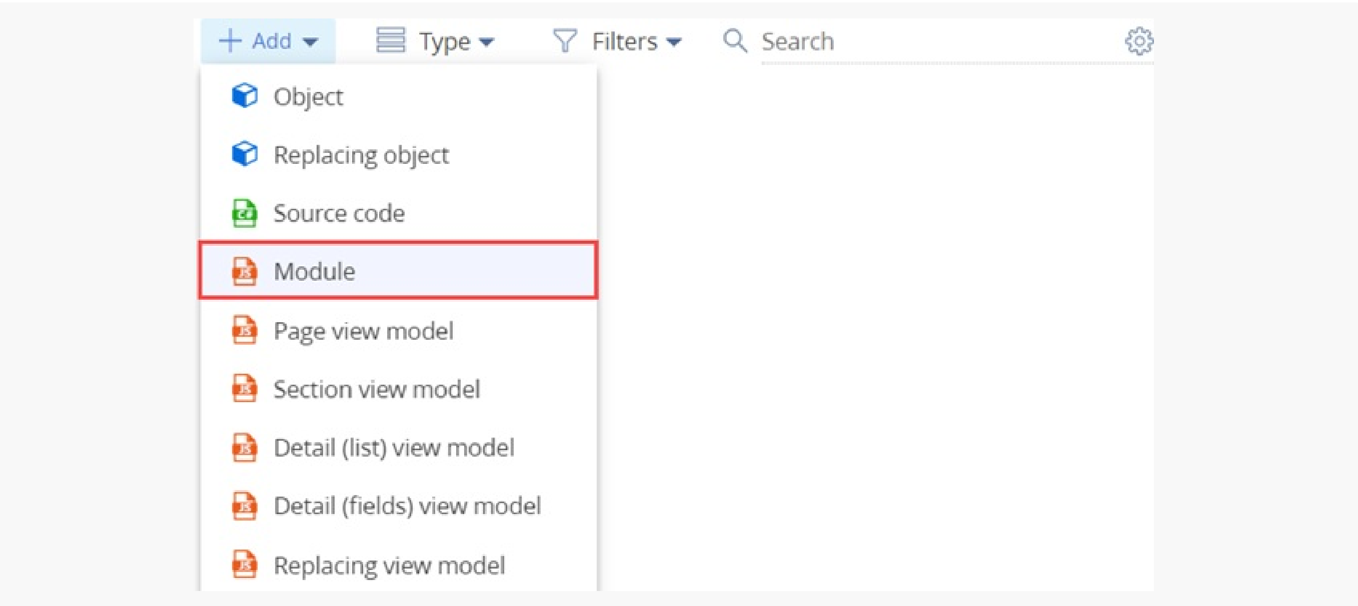
```
/* Declare a module called UsrExampleUtilsModule. The module has no dependencies. Therefore,
The module contains the method that displays the message box. */
define("UsrExampleUtilsModule", [], function () {
    return {
        /* The method that displays the message box. The "information" method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

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

2. Create a visual module

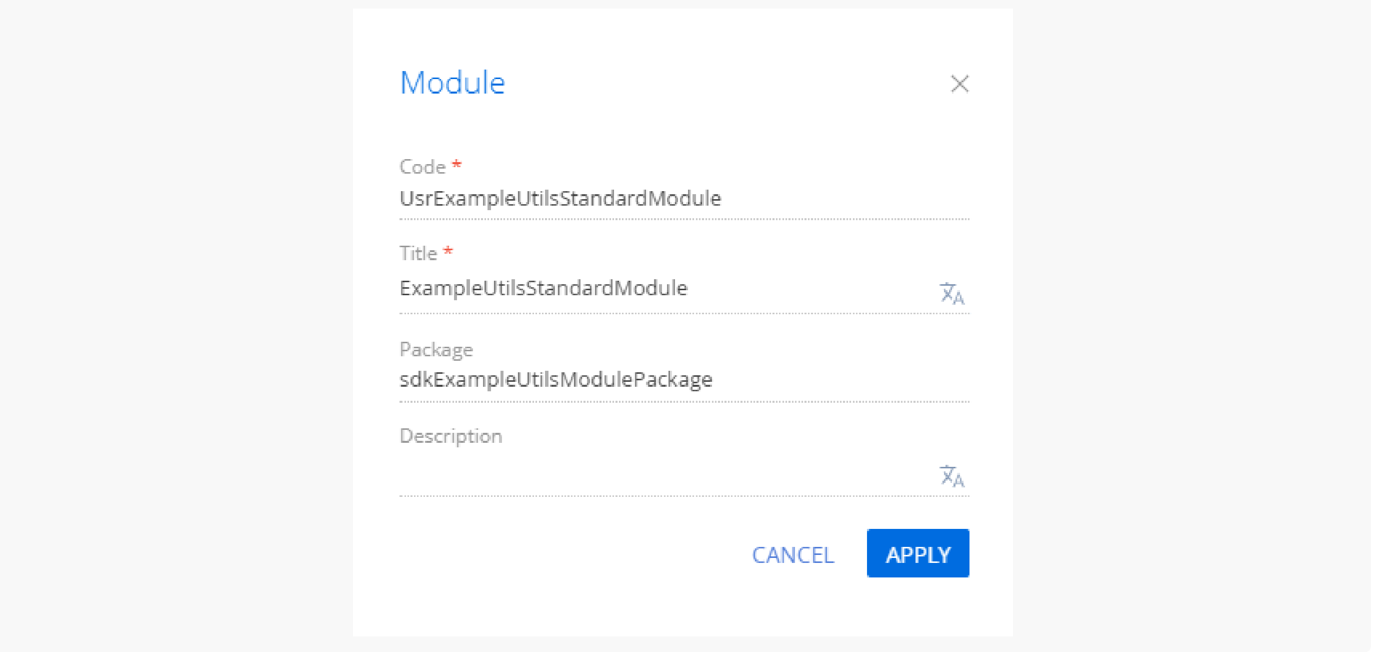
1. Open the [*Configuration*] section and select a custom package to add the schema.
2. Click [*Add*] → [*Module*] on the section list toolbar.

© 2023 Creatio. All rights reserved.



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 {
```

© 2023 Creatio. All rights reserved.

```
        /* 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

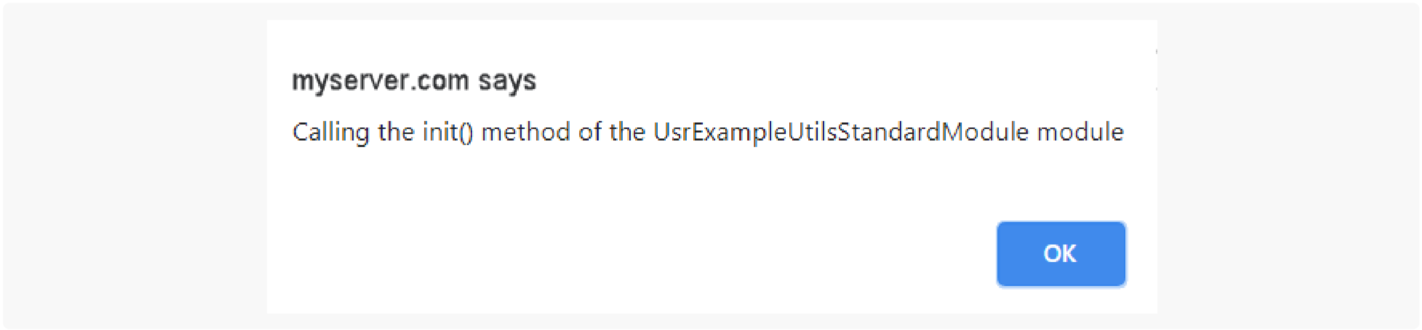
```
[Creatio URL]/0/NUI/ViewModule.aspx#[SomeModuleName]
```

Example URL

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

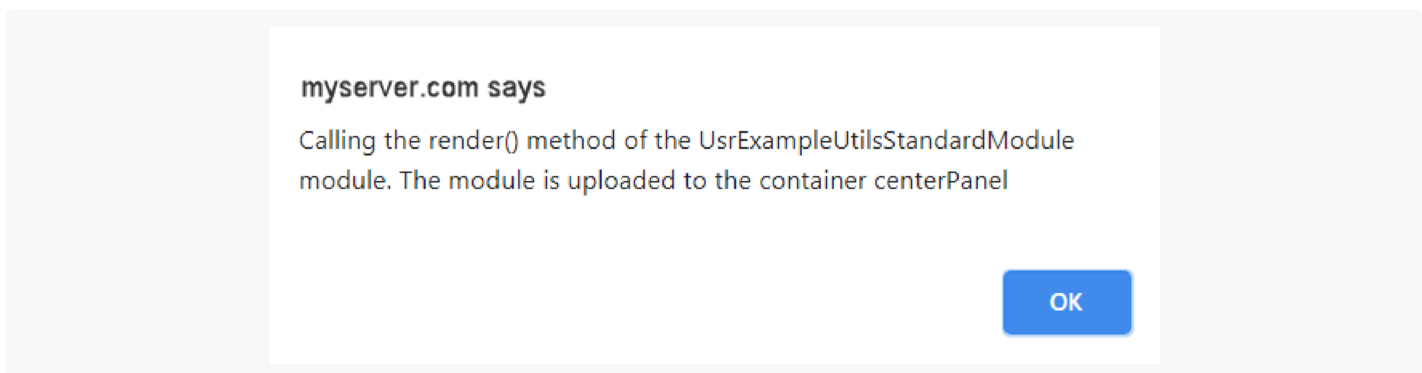
When loading the module, the client core must call the `init()` method, then the `render()` method. Each method calls a message box.

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



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

© 2023 Creatio. All rights reserved.



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;
    }
});
```

© 2023 Creatio. All rights reserved.

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
```

© 2023 Creatio. All rights reserved.

```
        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: {
```

© 2023 Creatio. All rights reserved.

```
                    /* 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.

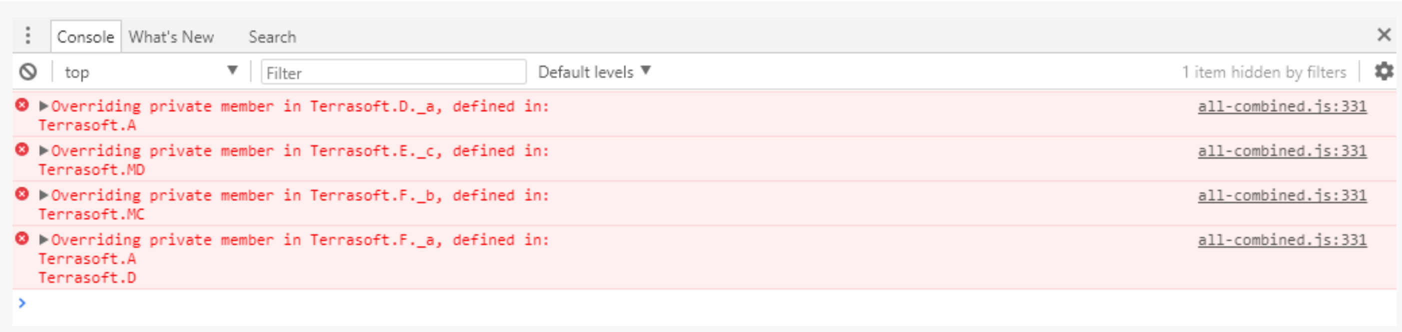
UsrPrivateMemberWatcher

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

© 2023 Creatio. All rights reserved.

```
    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 () {
```

© 2023 Creatio. All rights reserved.

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

© 2023 Creatio. All rights reserved.

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.

© 2023 Creatio. All rights reserved.