Creatio Academy

Front-end development

Sandbox

Version 8.0

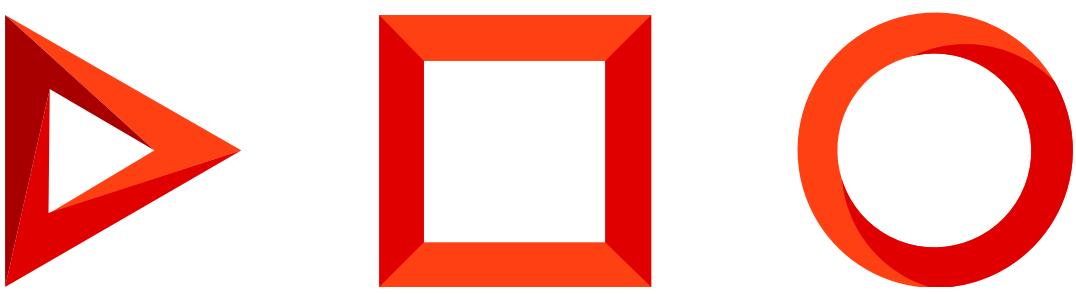


Table of Contents

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.

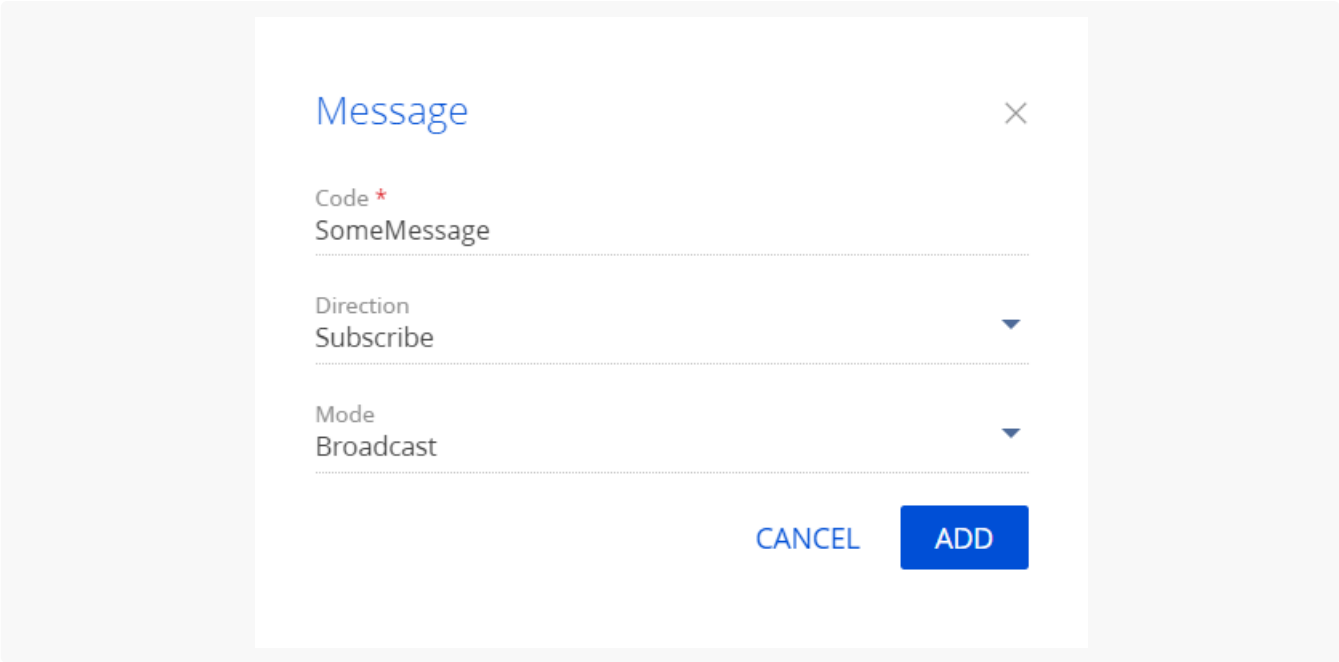
To **register a message in a view model**, declare a message configuration object in the `messages` schema property.

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

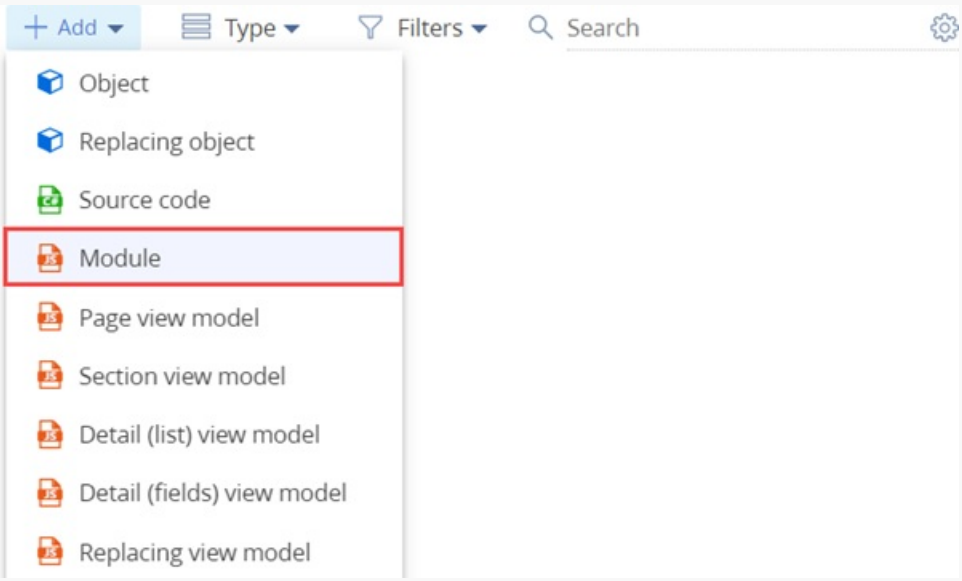
Example. Create the `UsrSomeModule` module. Implement the following **messages** in the module:

- `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.

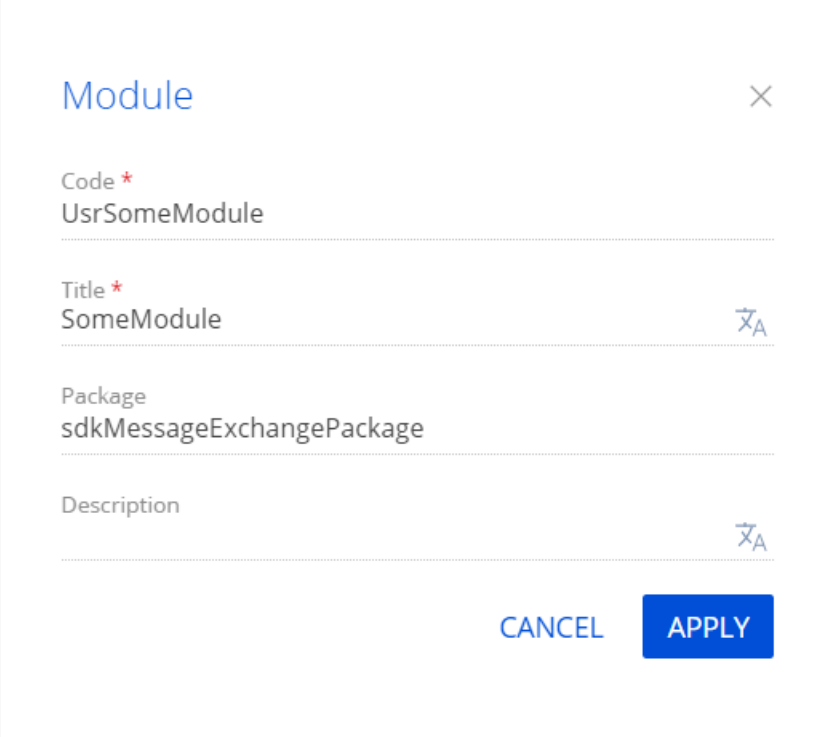
1. Create a module

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.

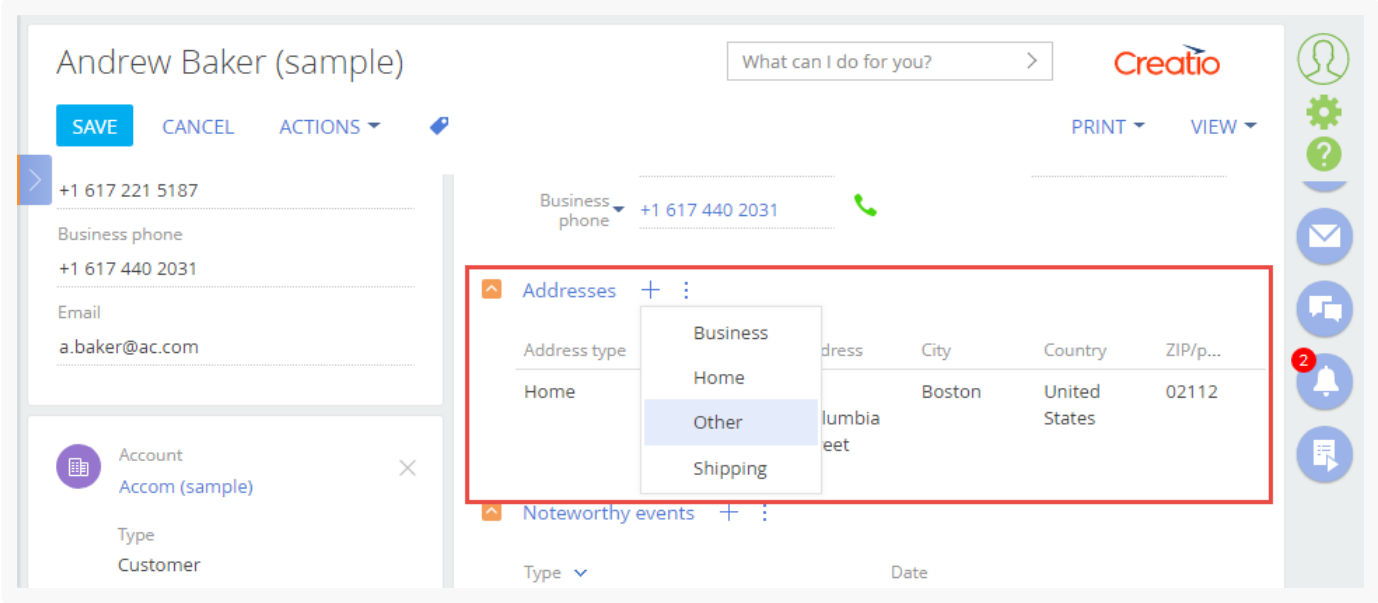
LookupQuickAddMixin

```
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. */
```

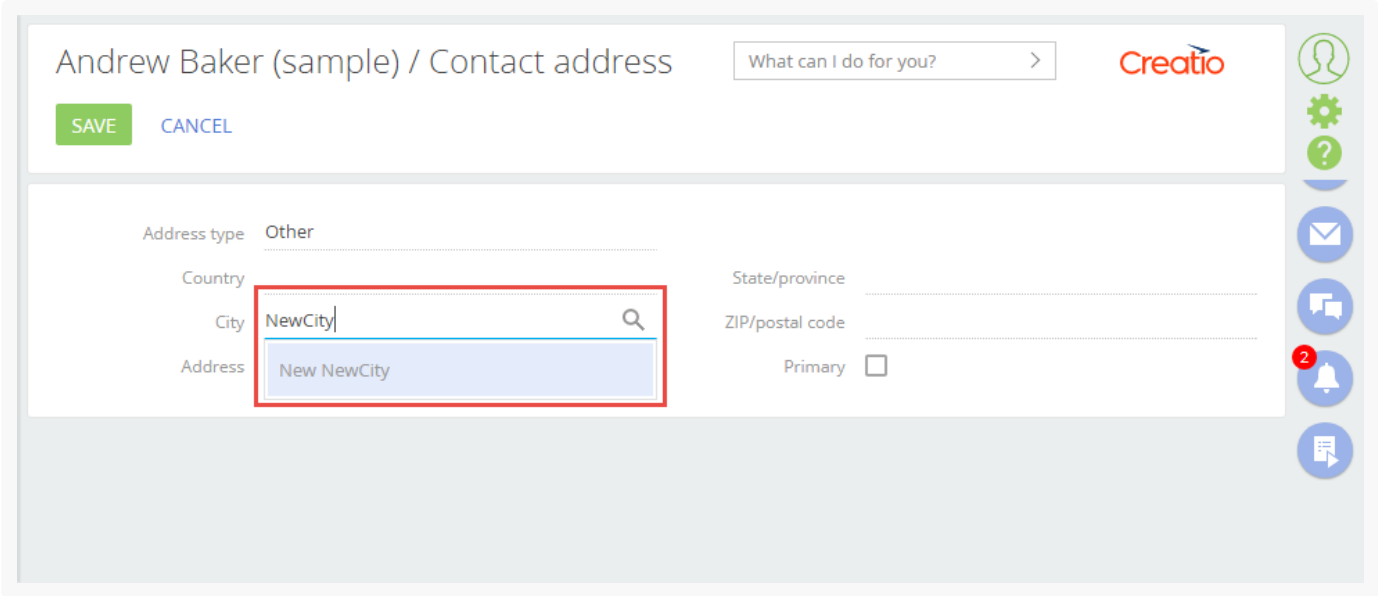
```
        onLookupChange: function(newValue, columnName) {
            ...
            /* Execute a chain of method calls.
            As a result, the _subscribeNewEntityCardModuleResponse() method is called. */
            ...
        },
        ...
        /* The method that subscribes to the "CardModuleResponse" 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.

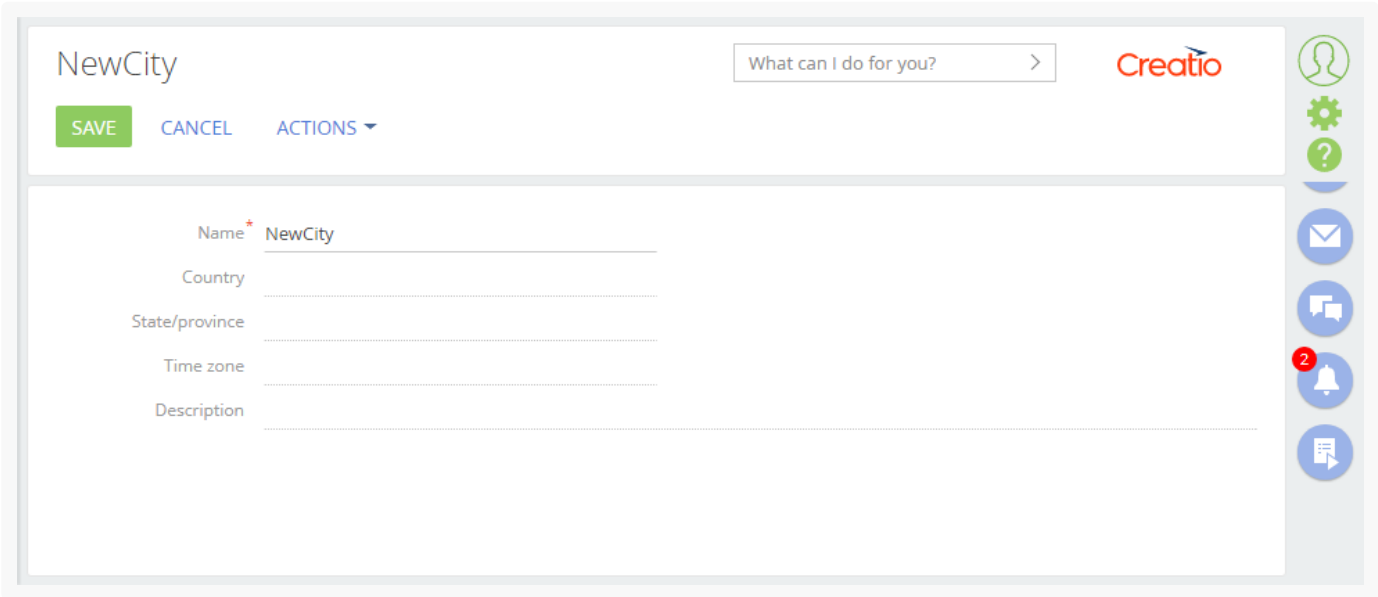


2. The contact address page is opened.



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");
```