Creatio Academy

Module message exchange

Sandbox

Version 8.0

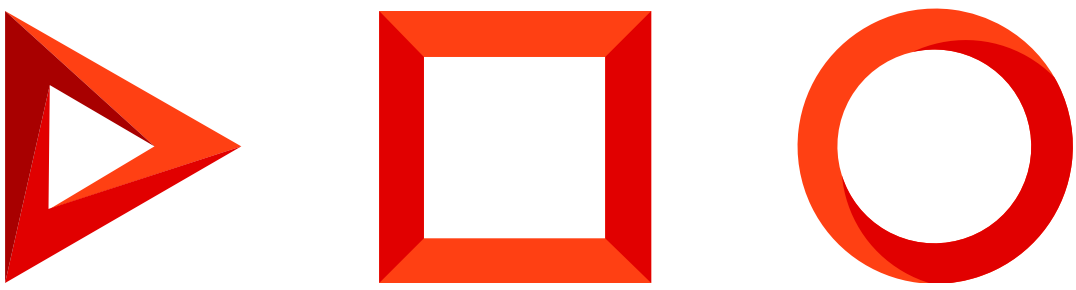


Table of Contents

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.

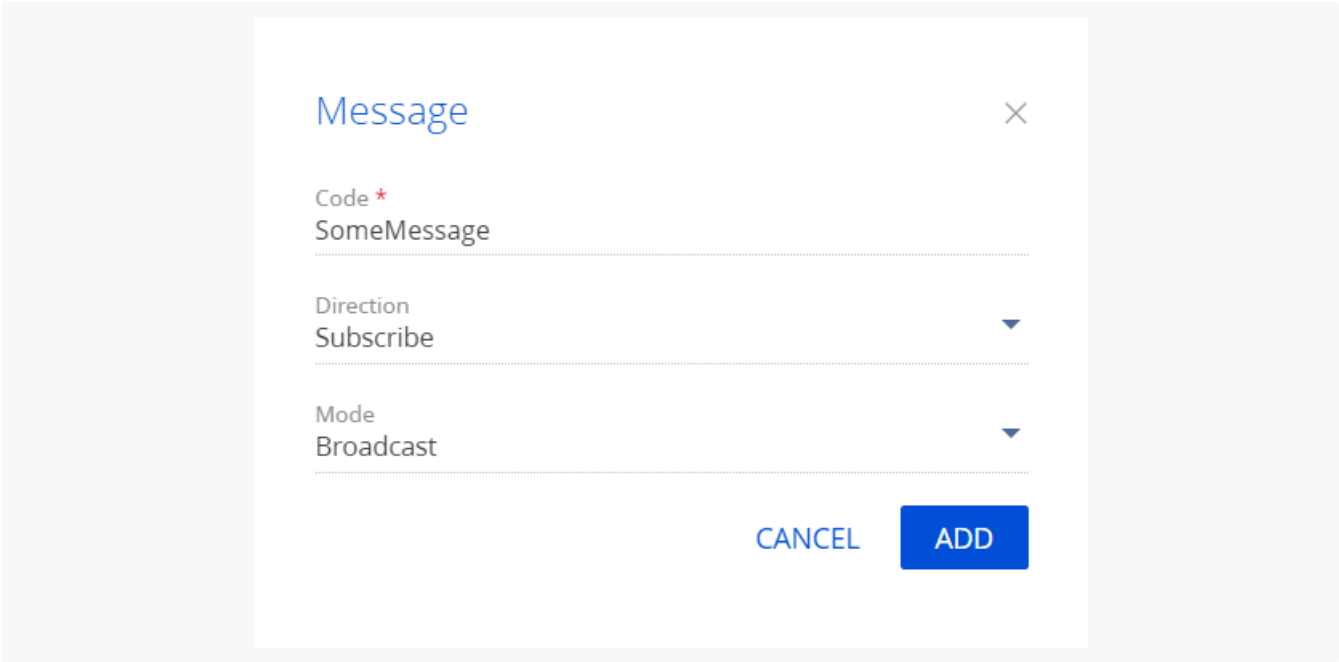
To **register a message in a view model**, declare a message configuration object in the `messages` schema property.

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

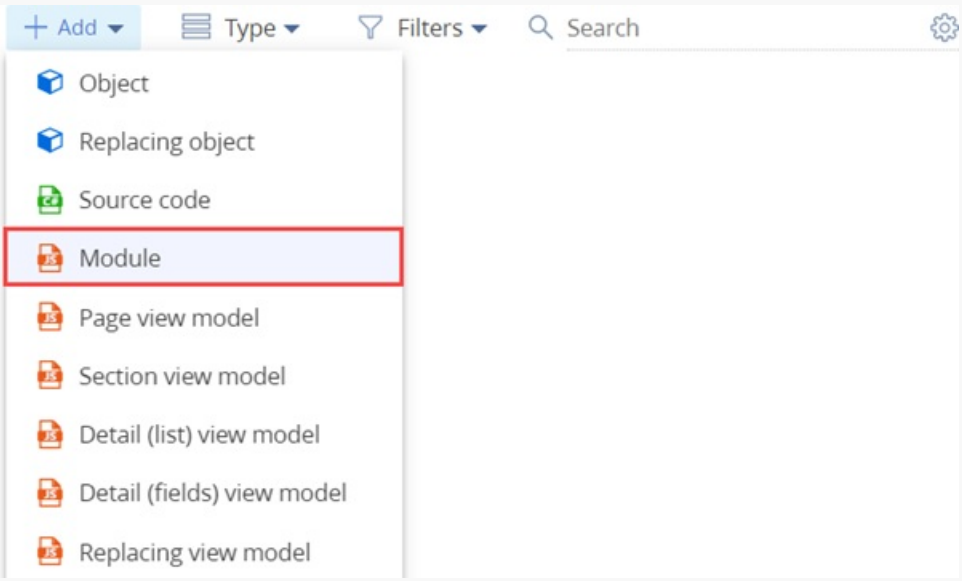
Example. Create the `UsrSomeModule` module. Implement the following **messages** in the module:

- `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.

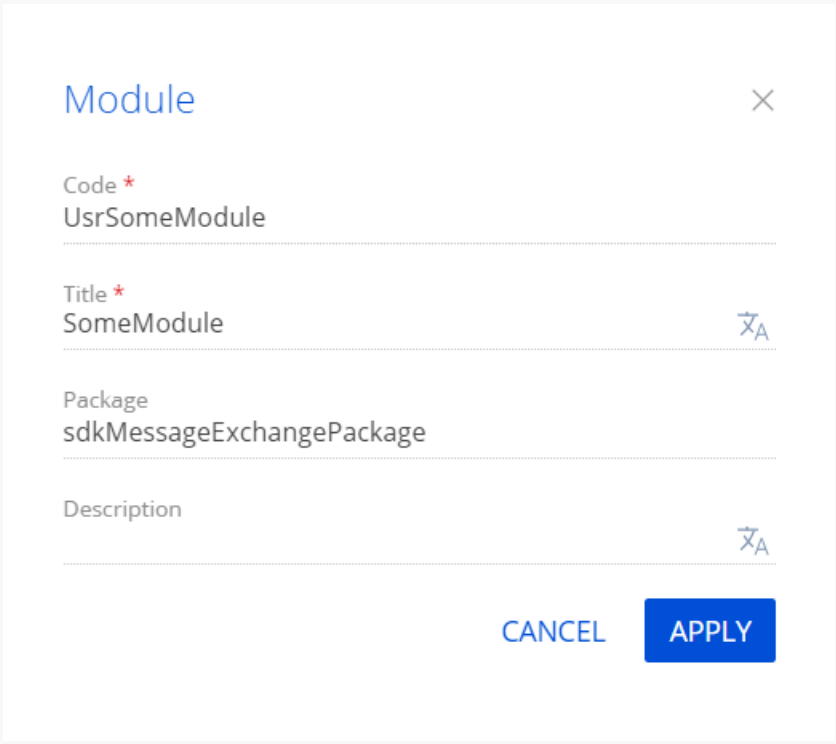
1. Create a module

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);
...
```