Creatio Academy

WebSocket messages

Send messages via WebSocket

Version 8.0

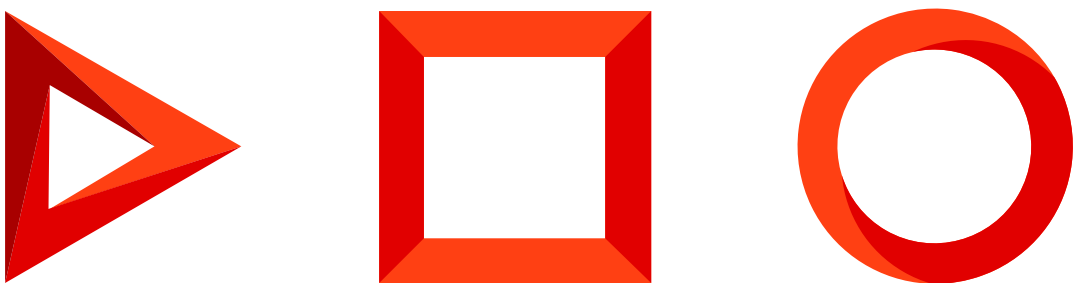


Table of Contents

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.

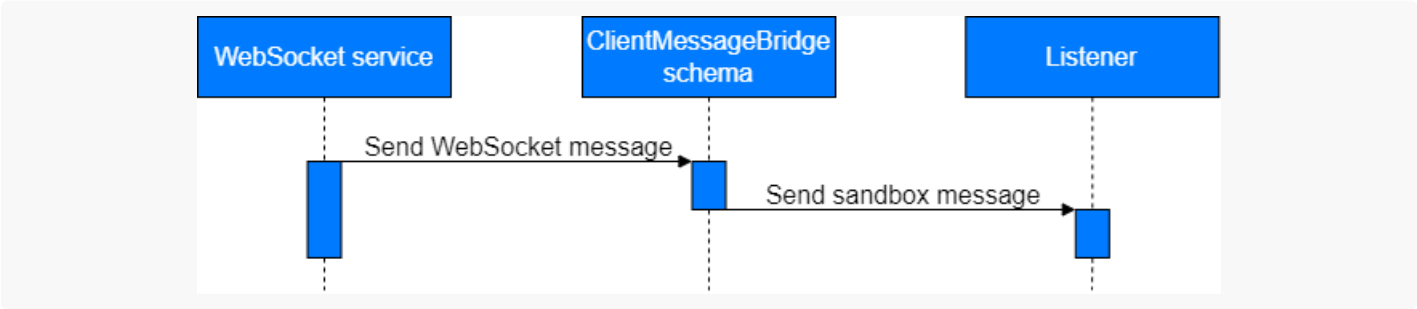
Implement custom logic that sends a message

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.

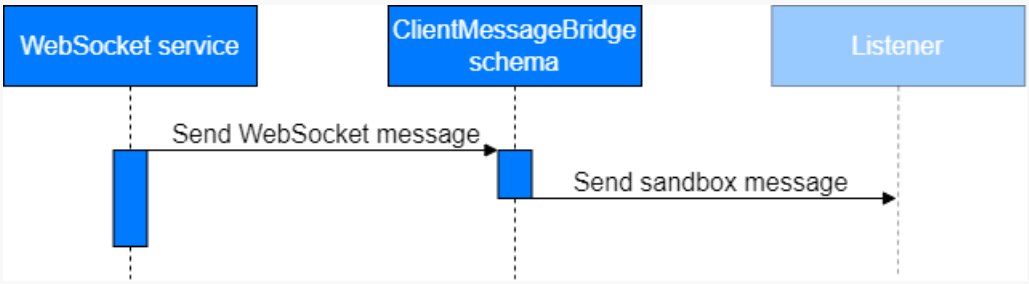
Save the message to history

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.

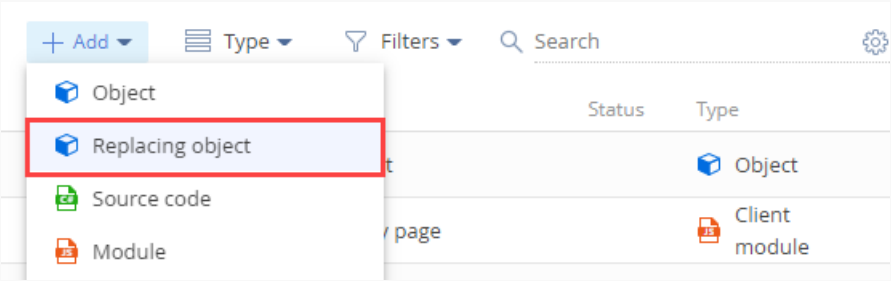
Implement a subscriber to a WebSocket message

Advanced

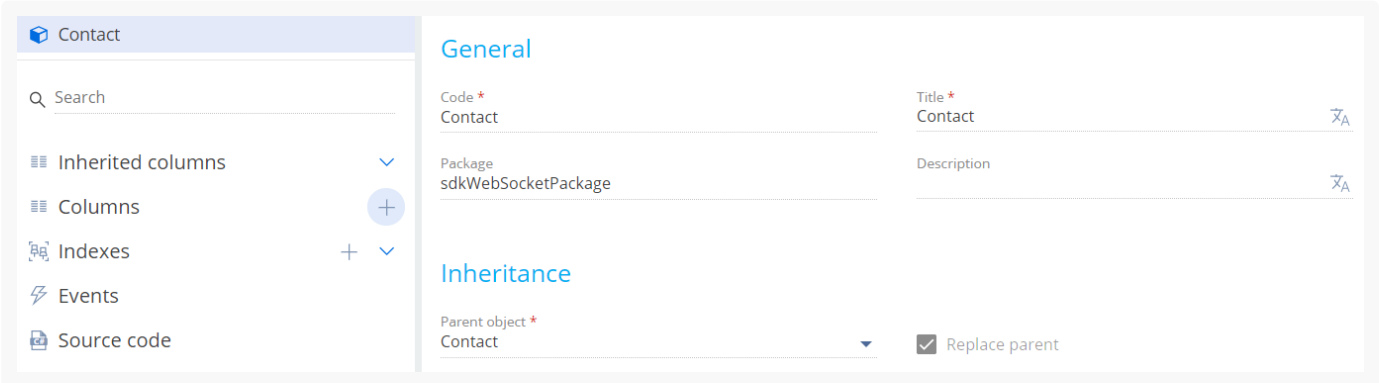
Example. Publish the `NewMessage` message in the back-end when you save a contact. Send the message via WebSocket. The message must include the contact birthday and name. Implement the `NewMessage` message sending in the front-end. Display messages in the browser console.

1. Create a replacing object schema

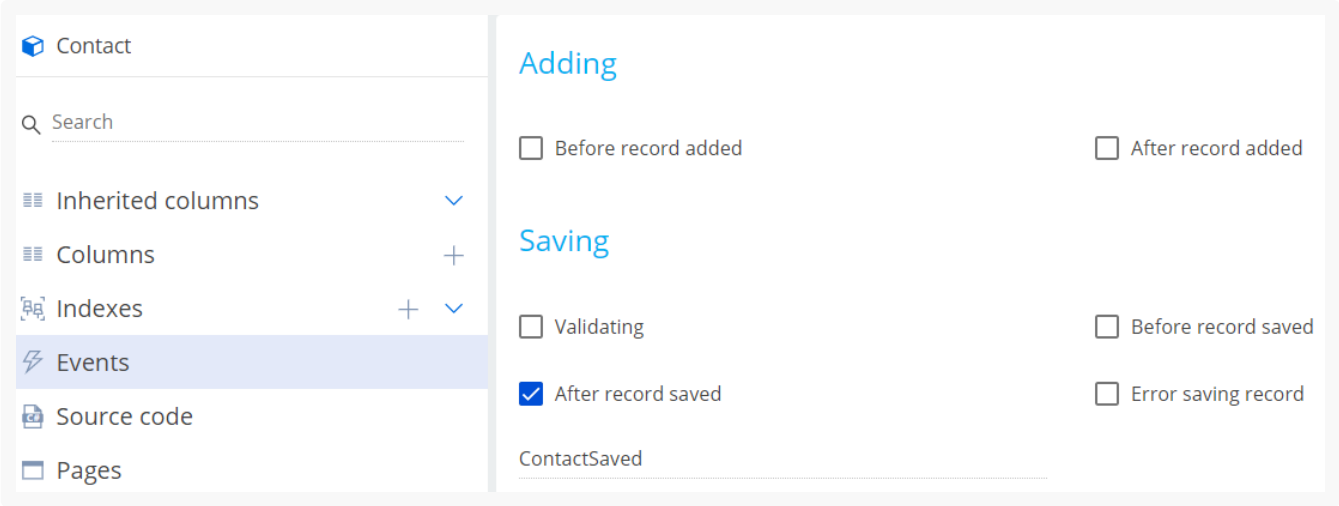
1. Open the [*Configuration*] section and select a custom package to add the schema.
2. Click [*Add*] → [*Replacing object*] on the section list toolbar.



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`.

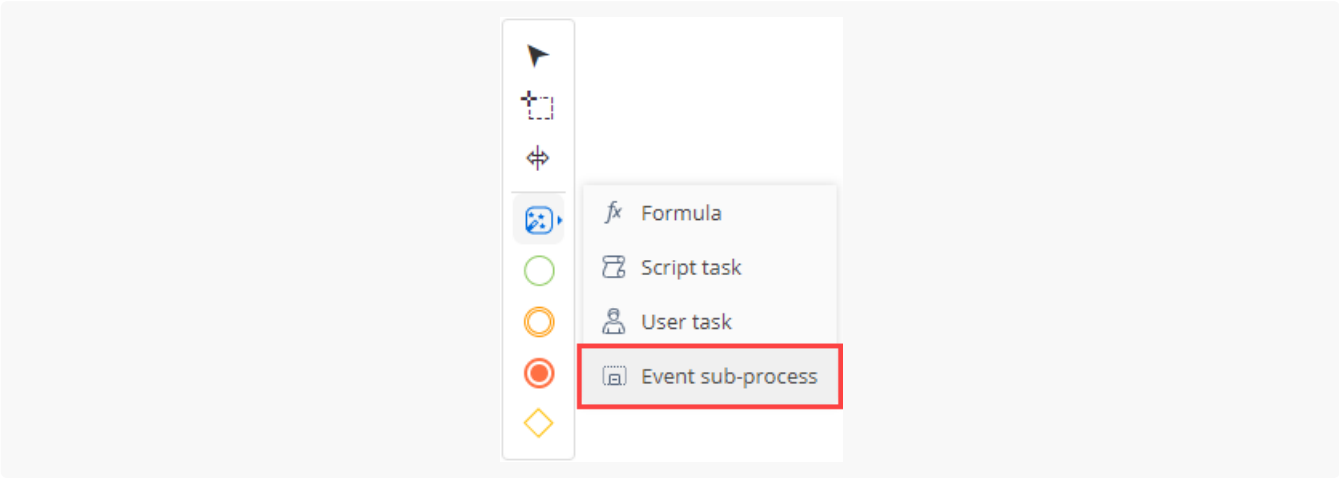


c. Click [*Save*] on the Object Designer's toolbar.

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.

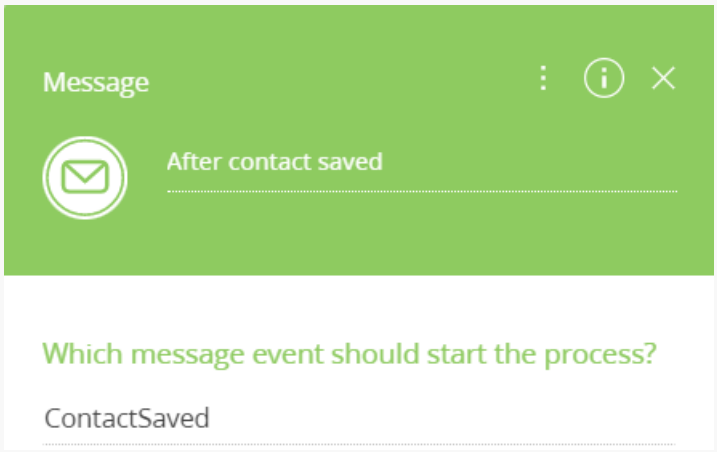


   c. Set the [*Title*] property in the element setup area to "`Contact Saved Sub-process`."

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

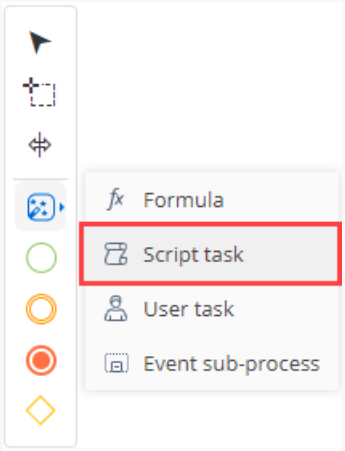
      a. Set up the [*Message*] **start event**.

      - 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.

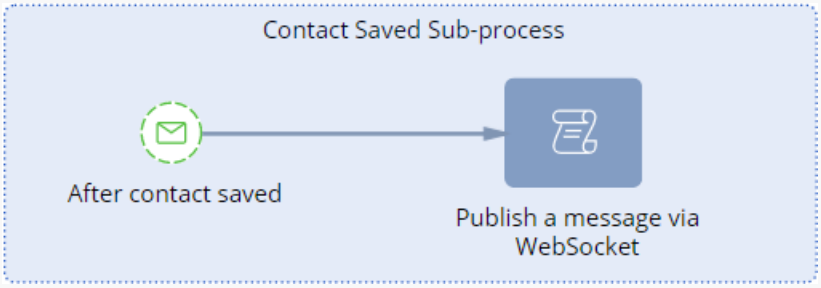
Code of the [*Script task*] system action

```
/* 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.

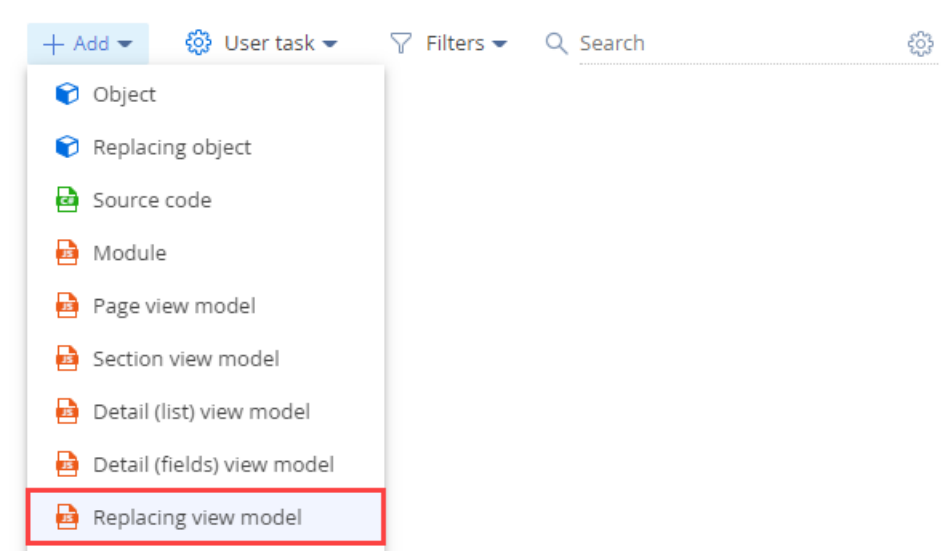
View the event sub-process in the figure below.



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

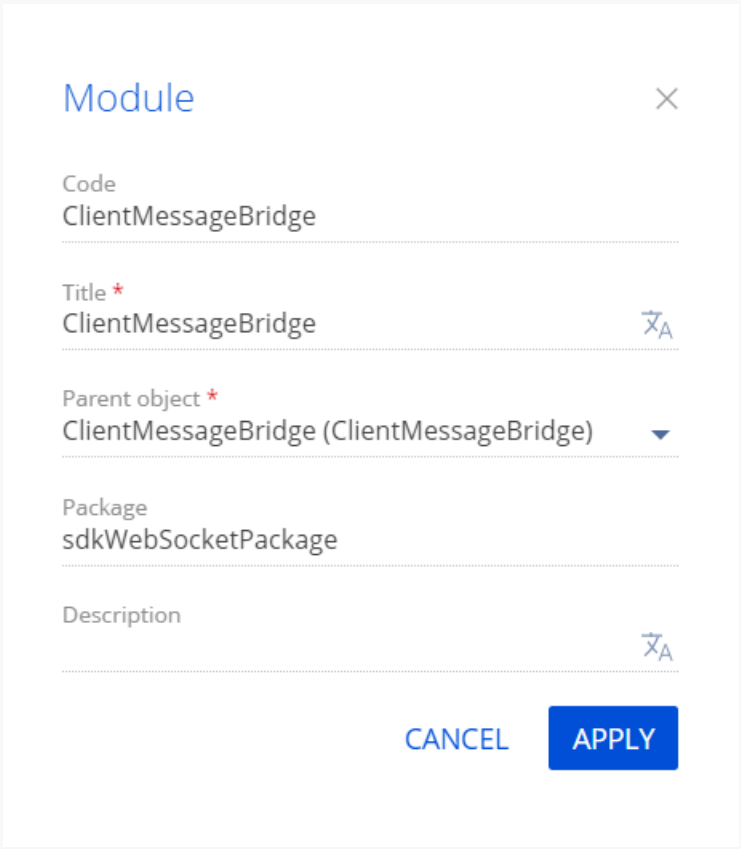
2. Implement message sending in Creatio

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

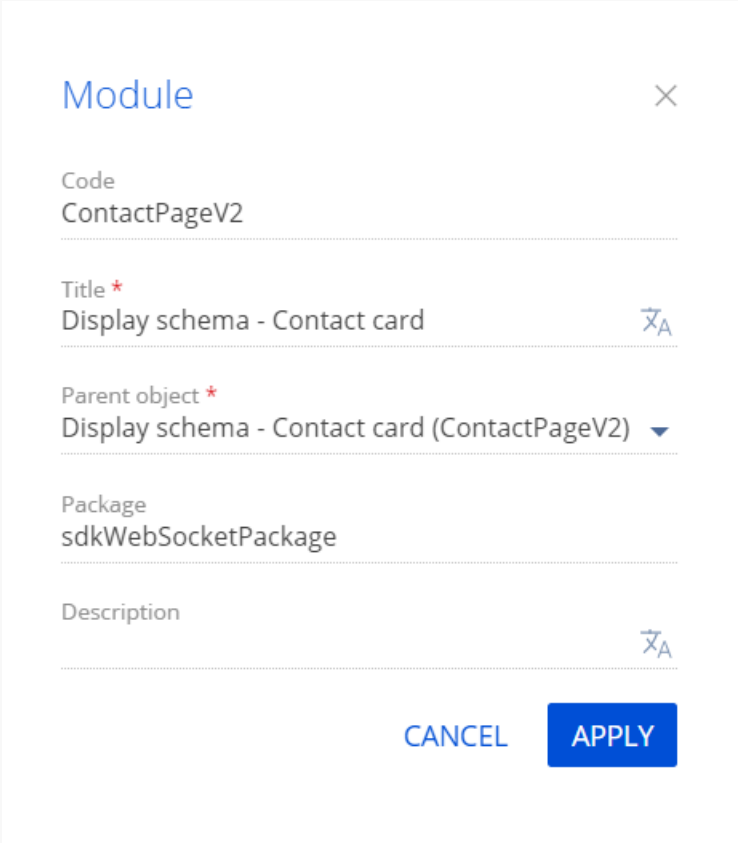
5. Click [*Save*] on the Designer's toolbar.

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

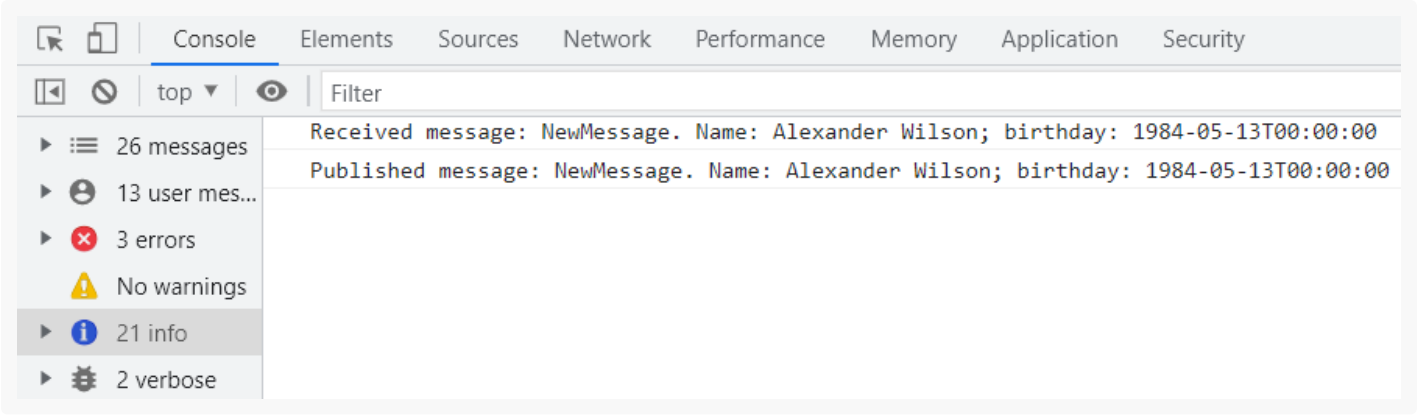
5. Click [*Save*] on the Designer's toolbar.

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.