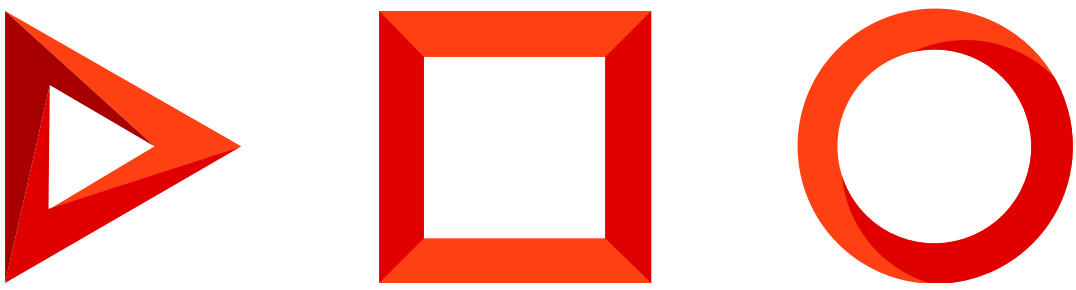
Creatio Academy

WebSocket

Версия 8.0



Эта документация предоставляется с ограничениями на использование и защищена законами об интеллектуальной собственности. За исключением случаев, прямо разрешенных в вашем лицензионном соглашении или разрешенных законом, вы не можете использовать, копировать, воспроизводить, переводить, транслировать, изменять, лицензировать, передавать, распространять, демонстрировать, выполнять, публиковать или отображать любую часть в любой форме или посредством любые значения. Обратный инжиниринг, дизассемблирование или декомпиляция этой документации, если это не требуется по закону для взаимодействия, запрещены.

Информация, содержащаяся в данном документе, может быть изменена без предварительного уведомления и не может гарантировать отсутствие ошибок. Если вы обнаружите какие-либо ошибки, сообщите нам о них в письменной форме.

Содержание

Передача сообщений по WebSocket

Сложный

Для транслирования сообщений, полученных по WebSocket, подписчикам внутри системы в Creatio используется схема `ClientMessageBridge`.

Механизм передачи сообщений

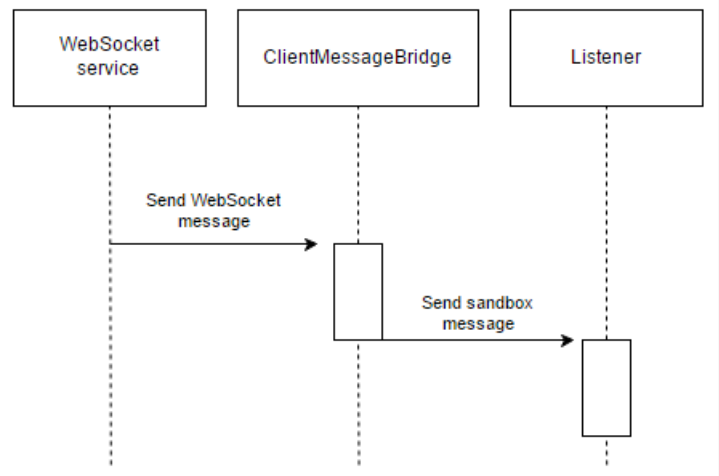
Если для сообщения, полученного по WebSocket, в расширяющих схемах `ClientMessageBridge` не определена дополнительная логика, то используется широковещательная отправка сообщения внутри системы через `sandbox` с именем "SocketMessageReceived". Подписавшись на это сообщение, можно обработать данные, полученные по WebSocket.

Для дополнительных настроек публикуемого сообщения, то необходимо выполнить следующе действия:

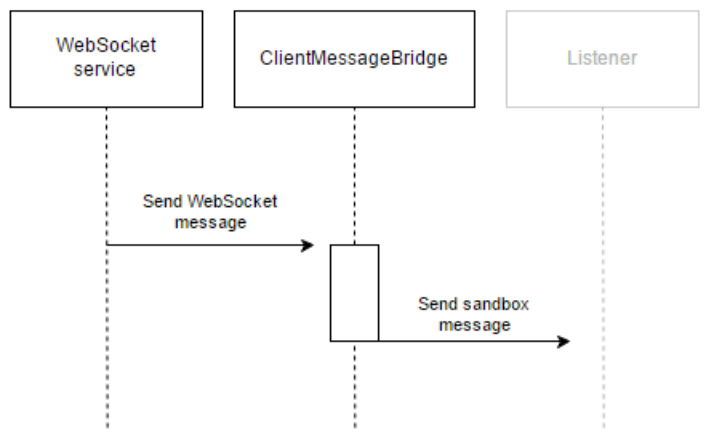
1. Создать замещающую клиентскую схему, в которой родительской схемой будет выступать "ClientMessageBridge".
2. В свойстве `messages` привязать к схеме сообщение с необходимыми настройками.
3. Выполнить перегрузку родительского метода `init()` для добавления сообщения, полученного по WebSocket, в конфигурационный объект сообщений схемы.
4. Выполнить перегрузку базового метода `afterPublishMessage()` для отслеживания момента рассылки сообщения.

Механизм сохранения истории сообщений

В идеальном случае на момент публикации сообщения внутри системы присутствует обработчик "Listener".



Но возможна ситуация, когда обработчик еще не загружен.



В этом случае, чтобы не потерять необработанные сообщения, Creatio выполняет следующие действия:

1. Сообщения сохраняются в истории.
2. Перед публикацией каждого сообщения проверяется наличие обработчика.
3. Когда обработчик загружен, ему публикуются все сохраненные сообщения в порядке их получения.
4. После публикации сообщений из истории вся история очищается.

Для реализации описанной возможности необходимо при добавлении нового конфигурационного объекта, указать в нем признак `isSaveHistory` равным `true`.

Пример конфигурирования обработки сообщения с сохранением в историю

```
init: function() {
   // Вызов родительского метода init().
   this.callParent(arguments);
   // Добавление нового конфигурационного объекта в коллекцию конфигурационных объектов.
   this.addMessageConfig({
       // sender – имя сообщения, получаемого по WebSocket.
       sender: "OrderStepCalculated",
       // messageName – имя сообщения, с которым оно будет разослано внутри системы.
       messageName: "OrderStepCalculated",
       // isSaveHistory – признак, указывающий на необходимость сохранения истории.
       isSaveHistory: true
    });
},
```

В классе `ClientMessageBridge` реализованы абстрактные методы родительского класса `BaseMessageBridge`, которые обеспечивают сохранение истории сообщений и работу с ними с использованием `localStorage` браузера:

- `saveMessageToHistory()` — обеспечивает сохранение нового сообщения в коллекции сообщений;
- `getMessagesFromHistory()` — обеспечивает получение массива сообщений по передаваемому имени;

- `deleteSavedMessages()` — удаляет все сохраненные сообщения по передаваемому имени.

Для реализации работы с другим типом хранилища требуется создать наследник класса `BaseMessageBridge` и добавить в него необходимую реализацию абстрактных методов `saveMessageToHistory()`, `getMessagesFromHistory()` и `deleteSavedMessages()`.

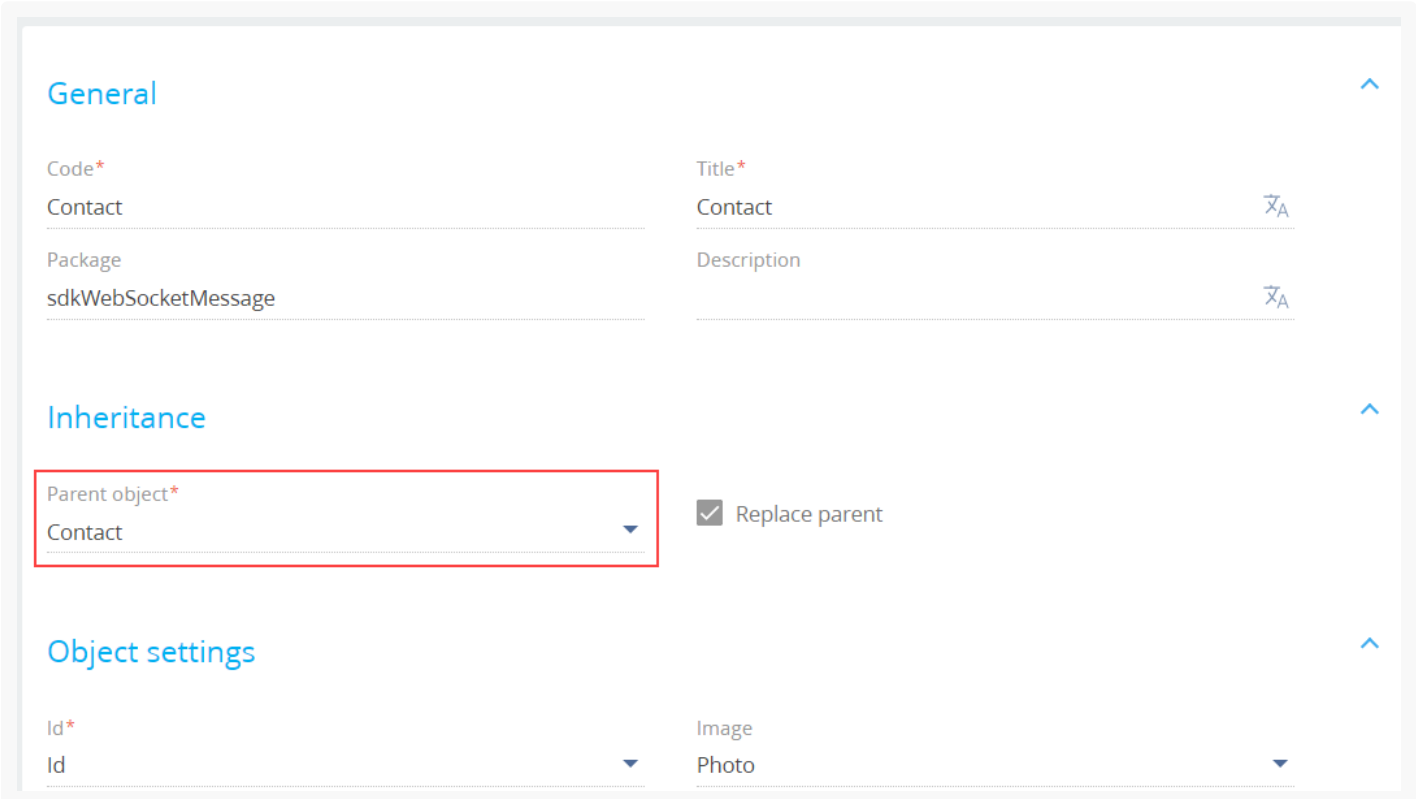
Настроить нового подписчика

Сложный

Пример. При сохранении контакта на стороне сервера необходимо по WebSocket опубликовать сообщение с именем `NewUserSet`, содержащее информацию о дне рождения и ФИО контакта. На стороне клиента необходимо реализовать рассылку сообщений `NewUserSet` внутри системы. Дополнительно перед рассылкой необходимо обработать свойство `birthday` сообщения, полученного по WebSocket, а после рассылки вызвать метод `afterPublishUserBirthday` утилитного класса. В завершение необходимо реализовать подписку на сообщение, разосланное на стороне клиента, например, в схеме страницы контакта.

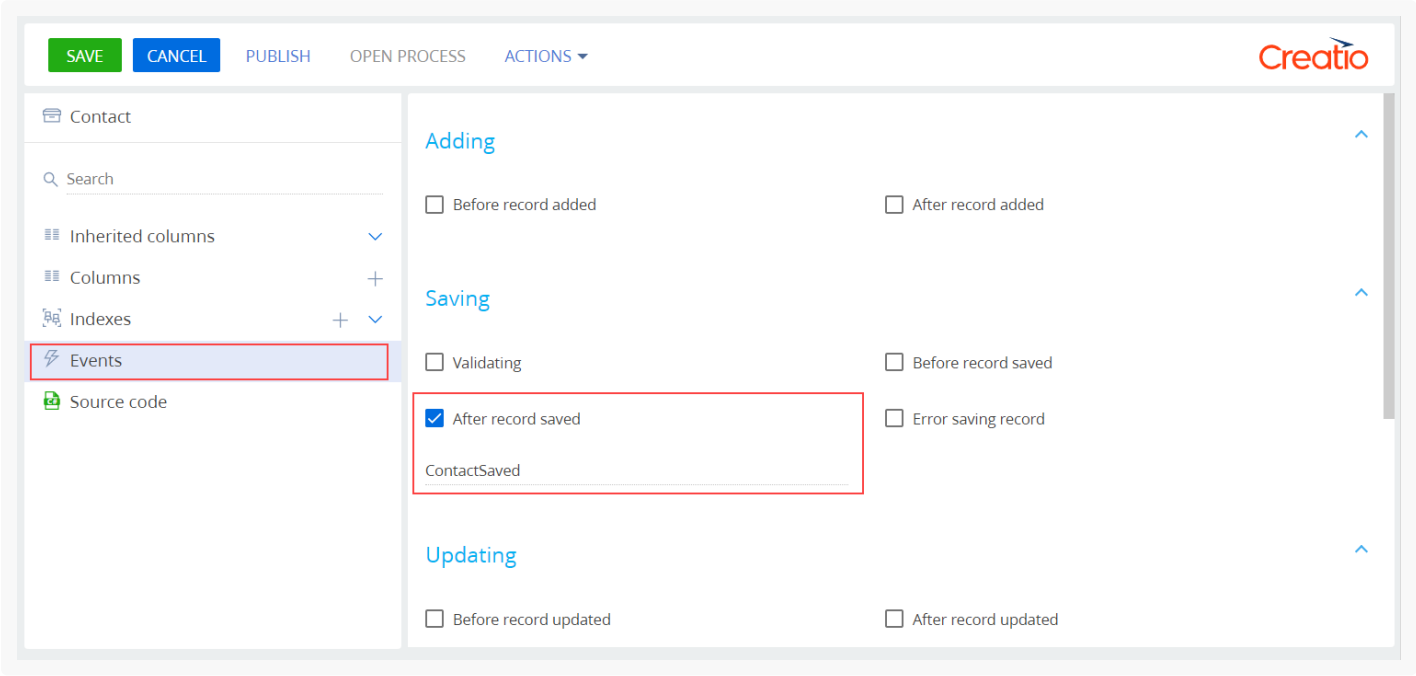
1. Создать замещающий объект [*Контакт*]

Прежде чем добавлять функциональность публикации сообщения по WebSocket, создайте замещающий объект, указав родителем [*Контакт*] ([*Contact*]).



2. Создать событие "После сохранения записи"

Добавьте функциональность публикации сообщения по WebSocket после сохранения записи контакта. Для этого в списке событий объекта выберите событие "`После сохранения записи`" ("`After record saved`").

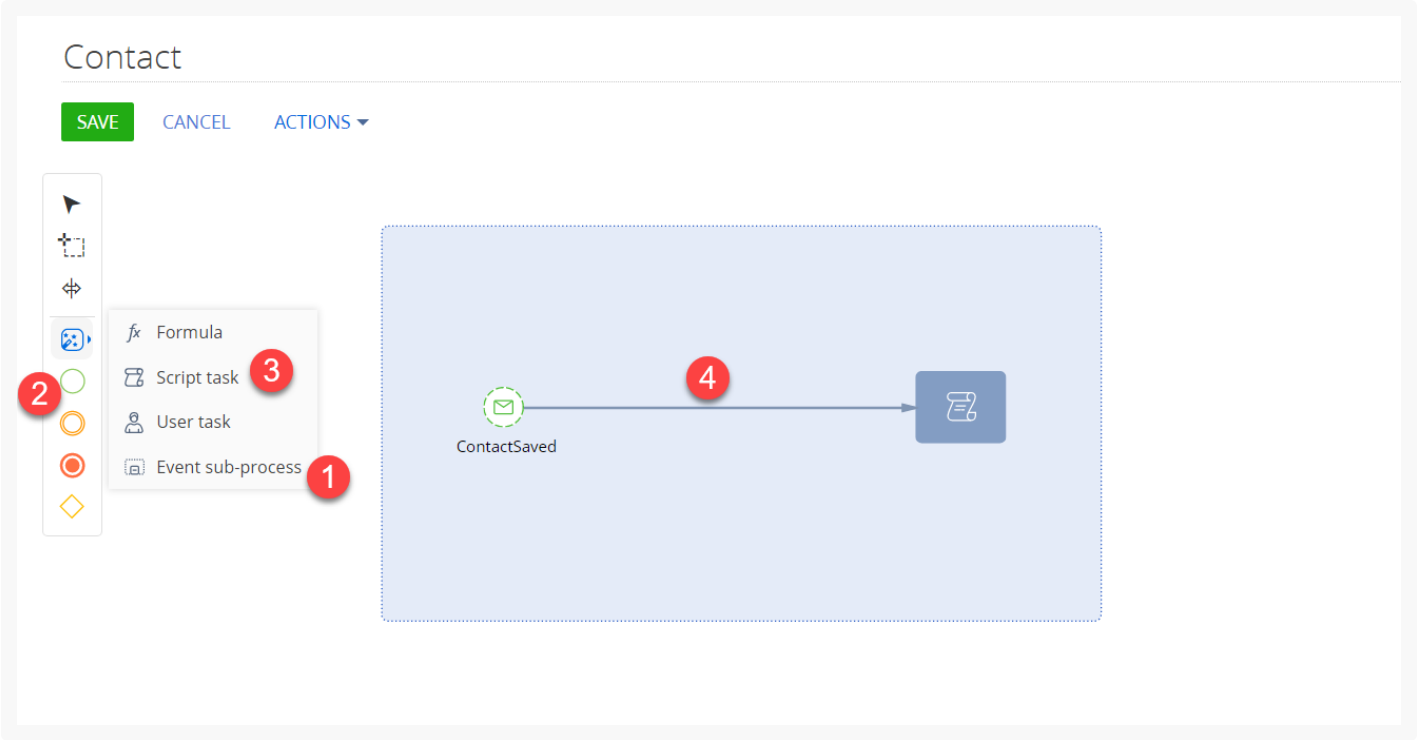


3. Реализовать событийный подпроцесс

В обработчике события "`После сохранения записи`" реализуйте событийный подпроцесс, который запускается сообщением `ContactSaved`.

1. Добавьте элемент событийного подпроцесса (1).
2. Добавьте элемент сообщения (2), указав имя сообщения `ContactSaved`.
3. Добавьте элемент сценария (3).
4. Соедините объект сообщения и сценария связью (4).

Создание подпроцесса для обработки сообщения



4. Добавить логику публикации сообщения по WebSocket

Для добавления логики публикации сообщения по WebSocket откройте элемент [*Задание-сценарий*] событийного подпроцесса и добавьте следующий исходный код.

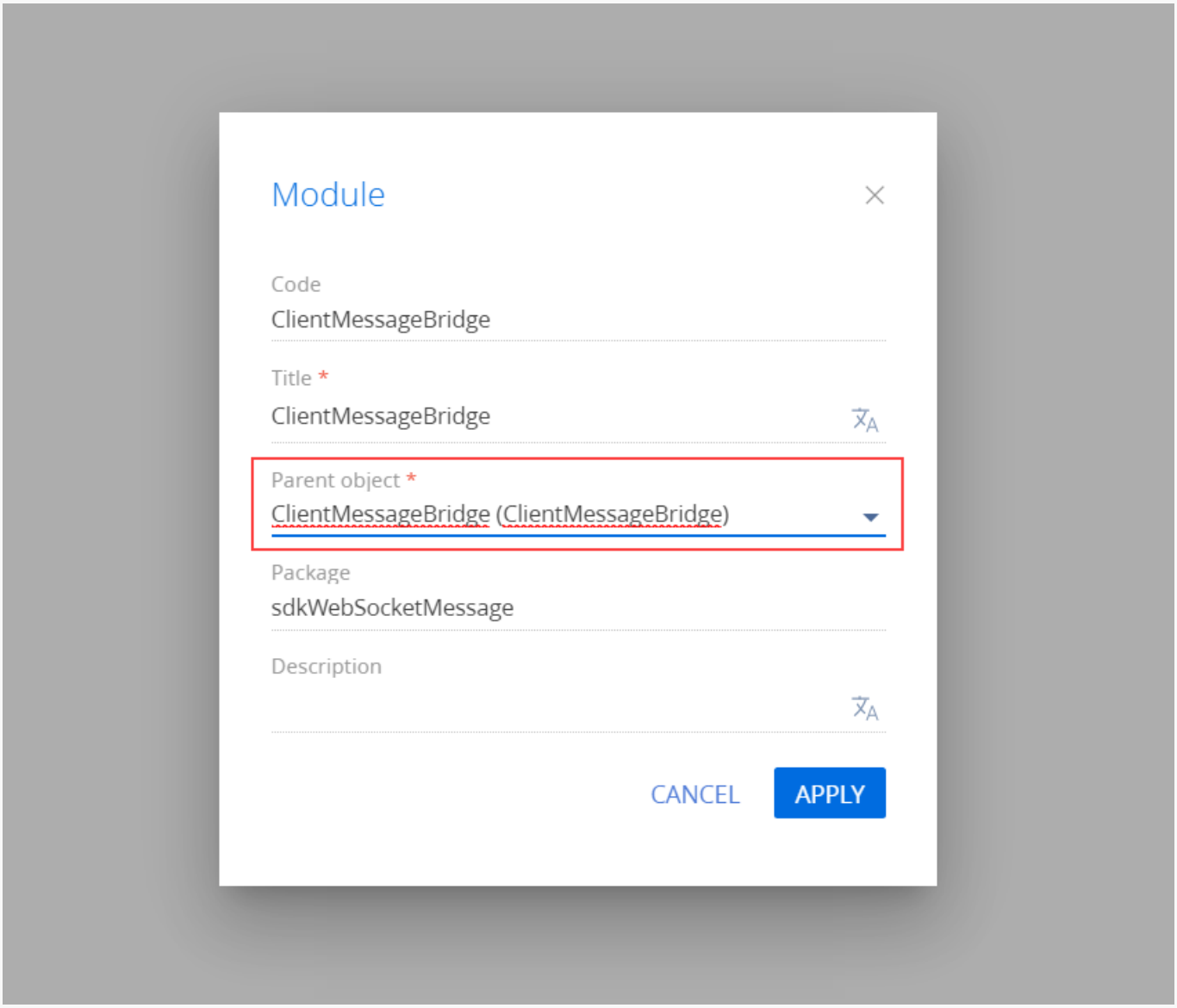
Пример добавления логики публикации сообщения по WebSocket

```
// Получение имени контакта.
string userName = Entity.GetTypedColumnValue<string>("Name");
// Получение даты дня рождения контакта.
DateTime birthDate = Entity.GetTypedColumnValue<DateTime>("BirthDate");
// Формирование текста сообщения.
string messageText = "{\"birthday\": \"" + birthDate.ToString("s") + "\", \"name\": \"" + userNa
// Присвоение сообщению имени.
string sender = "NewUserSet";
// Публикация сообщения по WebSocket.
MsgChannelUtilities.PostMessageToAll(sender, messageText);
return true;
```

После этого сохраните и опубликуйте событийный подпроцесс.

5. Реализовать рассылку сообщения внутри приложения

Для этого в пользовательском пакете создайте замещающую модель представления, указав в качестве родительского объекта схему `ClientMessageBridge` пакета `NUI`.



Для реализации рассылки широковещательного сообщения `NewUserSet` добавьте в схему следующий исходный код.

ClientMessageBridge.js

```
define("ClientMessageBridge", ["ConfigurationConstants"],
    function(ConfigurationConstants) {
        return {
            // Сообщения.
            messages: {
                // Имя сообщения.
                "NewUserSet": {
                    // Тип сообщения – широковещательное, без указания конкретного подписчика.
                    "mode": Terrasoft.MessageMode.BROADCAST,
                    // Направление сообщения – публикация.
                    "direction": Terrasoft.MessageDirectionType.PUBLISH
```

```
                }
            },
            methods: {
                // Инициализация схемы.
                init: function() {
                    // Вызов родительского метода.
                    this.callParent(arguments);
                    // Добавление нового конфигурационного объекта в коллекцию конфигурационных
                    this.addMessageConfig({
                        // Имя сообщения, получаемого по WebSocket.
                        sender: "NewUserSet",
                        // Имя сообщения, с которым оно будет разослано.
                        messageName: "NewUserSet"
                    });
                },
                // Метод, выполняемый после публикации сообщения.
                afterPublishMessage: function(
                    // Имя сообщения, с которым оно было разослано.
                    sandboxMessageName,
                    // Содержимое сообщения.
                    webSocketBody,
                    // Результат отправки сообщения.
                    result,
                    // Конфигурационный объект рассылки сообщения.
                    publishConfig) {
                    // Проверка, что сообщение соответствует добавленному в конфигурационный объ
                    if (sandboxMessageName === "NewUserSet") {
                        // Сохранение содержимого в локальные переменные.
                        var birthday = webSocketBody.birthday;
                        var name = webSocketBody.name;
                        // Вывод содержимого в консоль браузера.
                        window.console.info("Опубликовано сообщение: " + sandboxMessageName +
                            ". Данные: name: " + name +
                            "; birthday: " + birthday);
                    }
                }
            }
        };
    });
```

Здесь в свойстве `messages` к схеме привязывается широковещательное сообщение `NewUserSet`, которое может публиковаться внутри системы. В свойстве `methods` выполняется перегрузка родительского метода `init()` для добавления сообщения, полученного по `WebSocket` в конфигурационый объект сообщений схемы. Для того чтобы отслеживать момент рассылки сообщения, выполняется перегрузка родительского метода `afterPublishMessage`.

После сохранения схемы и обновления страницы в браузере, сообщения `NewUserSet`, полученные по WebSocket, будут рассылаться внутри системы, о чем будет сигнализировать сообщение в консоли

браузера в режиме отладки.

6. Реализовать подписку на сообщение

Чтобы получить объектпо WebSocket, подпишитесь на получение сообщения `NewUserSet` в любой схеме, например, в схеме страницы контакта. Для этого создайте замещающую модель представления (см. п. 5), указав в качестве родительского объекта "ContactPageV2".

Затем в схему добавьте следующий исходный код:

ContactPageV2.js

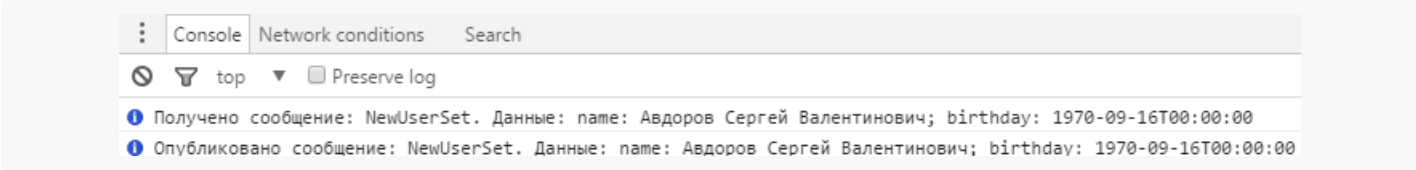
```
define("ContactPageV2", [],
    function(BusinessRuleModule, ConfigurationConstants) {
        return {
            // Имя схемы объекта.
            entitySchemaName: "Contact",
            messages: {
                // Имя сообщения.
                "NewUserSet": {
                    // Тип сообщения – широковещательное, без указания конкретного подписчика.
                    "mode": Terrasoft.MessageMode.BROADCAST,
                    // Направление сообщения – подписка.
                    "direction": Terrasoft.MessageDirectionType.SUBSCRIBE
                }
            },
            methods: {
                // Инициализация схемы.
                init: function() {
                    // Вызов родительского метода init().
                    this.callParent(arguments);
                    // Подписка на прием сообщения NewUserSet.
                    this.sandbox.subscribe("NewUserSet", this.onNewUserSet, this);
                },
                // Обработчик события получения сообщения NewUserSet.
                onNewUserSet: function(args) {
                    // Сохранение содержимого сообщения в локальные переменные.
                    var birthday = args.birthday;
                    var name = args.name;
                    // Вывод содержимого в консоль браузера.
                    window.console.info("Получено сообщение: NewUserSet. Данные: name: " +
                        name + "; birthday: " + birthday);
                }

            }
        };
    });
```

Здесь в свойстве `messages` к схеме привязывается широковещательное сообщение `NewUserSet`, на которое можно подписаться. В свойстве `methods` выполняется перегрузка родительского метода `init()` для подписки на сообщение `NewUserSet`, с указанием метода-обработчика `onNewUserSet()`, в котором выполняется обработка полученного в сообщении объекта и вывод результата в консоль браузера.

После добавления исходного кода сохраните схему и обновите страницу приложения в браузере.

Результат выполнения примера

Результатом выполнения примера будет получение двух информационных сообщений в консоли браузера после сохранения контакта.



Класс ClientMessageBridge JS

Сложный

Класс `ClientMessageBridge` используется для транслирования сообщений, полученных по WebSocket, подписчикам внутри системы.

Для дополнительной обработки сообщений перед публикацией и изменением имени, с которым сообщение будет рассылаться внутри системы, необходимо реализовать расширяющую схему. В расширяющей схеме, используя доступный API, можно сконфигурировать отправку для конкретного типа сообщений.

Свойства

`WebSocketMessageConfigs` `Array`

Коллекция конфигурационных объектов.

`LocalStoreName` `String`

Название хранилища, в котором сохраняется история сообщений.

`LocalStore` `Terrasoft.LocalStore`

Экземпляр класса, реализующий доступ к локальному хранилищу.

Методы

`init()`

Инициализирует значения по умолчанию.

`getSandboxMessageListenerExists(sandboxMessageName)`

Проверяет наличие слушателей сообщения с переданным именем. Возвращаемое значение: `Boolean` — результат проверки наличия слушателя сообщения.

Параметры

`sandboxMessageName: String`	Имя сообщения, с которым оно будет разослано внутри системы.

`publishMessageResult(sandboxMessageName, webSocketMessage)`

Публикует сообщение внутри системы. Возвращаемое значение: * — результат, полученный от обработчика сообщения.

Параметры

`sandboxMessageName: String`	Имя сообщения, с которым оно будет разослано внутри системы.
`webSocketMessage: Object`	Сообщение полученное по WebSocket.

`beforePublishMessage(sandboxMessageName, webSocketBody, publishConfig)`

Обработчик, вызываемый перед публикацией сообщения внутри системы.

Параметры

`sandboxMessageName: String`	Имя сообщения, с которым оно будет разослано внутри системы.
`webSocketBody: Object`	Сообщение, полученное по WebSocket.
`publishConfig: Object`	Конфигурационный объект рассылки сообщения.

`afterPublishMessage(sandboxMessageName, webSocketBody, result, publishConfig)`

Обработчик, вызываемый после публикации сообщения внутри системы.

Параметры

`sandboxMessageName: String`	Имя сообщения, с которым оно будет разослано внутри системы.
`webSocketBody: Object`	Сообщение, полученное по WebSocket. result: * — результат выполнения публикации сообщения внутри системы.
`publishConfig: Object`	Конфигурационный объект рассылки сообщения.

`addMessageConfig(config)`

Добавляет новый конфигурационный объект в коллекцию конфигурационных объектов.

Параметры

`config: Object`	Конфигурационный объект.

Пример конфигурационного объекта

```
{
    "sender": "webSocket sender key 1",
    "messageName": "sandbox message name 1",
    "isSaveHistory": true
}
```

Здесь:

- `sender: String` — имя сообщения, которое ожидается получить по WebSocket.
- `messageName: String` — имя сообщения, с которым оно будет разослано внутри системы.
- `isSaveHistory: Boolean` — признак, отвечающий за необходимость сохранения истории сообщений.

`saveMessageToHistory(sandboxMessageName, webSocketBody)`

Сохраняет сообщение в хранилище, если подписчик отсутствует, а в конфигурационном объекте указан признак необходимости сохранения.

Параметры

`sandboxMessageName: String`	Имя сообщения, с которым оно будет разослано внутри системы.
`webSocketBody: Object`	Сообщение, полученное по WebSocket.

`getMessagesFromHistory(sandboxMessageName)`

Возвращает массив сохраненных сообщений из хранилища.

Параметры

`sandboxMessageName: String`	Имя сообщения, с которым оно будет разослано внутри системы.

`deleteSavedMessages(sandboxMessageName)`

Удаляет из хранилища сохраненные сообщения.

Параметры

`sandboxMessageName: String`	Имя сообщения, с которым оно будет разослано внутри системы.