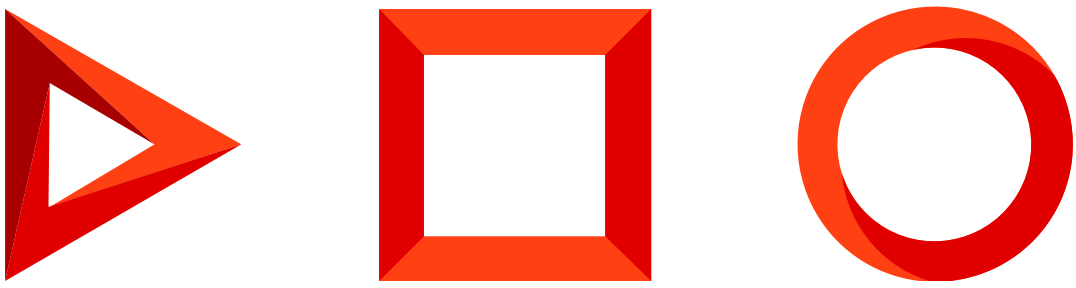
Creatio Academy

Модули

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

Понятие модуля

Сложный

Концепция AMD

Front-end часть приложения Creatio представляет собой набор блоков функциональности, каждый из которых реализован в отдельном **модуле**. Согласно **концепции** Asynchronous Module Definition (AMD), в процессе работы приложения выполняется асинхронная загрузка модулей и их зависимостей. Таким образом, концепция AMD позволяет подгружать только те данные, которые необходимы для работы в текущий момент.

Концепция AMD поддерживается различными JavaScript-фреймворками. В Creatio для работы с модулями используется **загрузчик** RequireJS.

Модульная разработка в Creatio

Модуль — фрагмент кода, инкапсулированный в обособленный блок, который может быть загружен и самостоятельно выполнен.

Создание модулей в специфике JavaScript декларируется **паттерном программирования** "Модуль". Классическая **реализация паттерна** — использование анонимных функций, возвращающих определенное значение (объект, функцию и т. д.), которое ассоциируется с модулем. При этом значение модуля экспортируется в глобальный объект.

Пример экспорта значения модуля в глобальный объект

```
// Немедленно вызываемое функциональное выражение (IIFE). Анонимная функция,
// которая инициализирует свойство myGlobalModule глобального объекта функцией,
// возвращающей значение модуля. Таким образом, фактически происходит загрузка модуля,
// к которому в дальнейшем можно обращаться через глобальное свойство myGlobalModule.
(function () {
    // Обращение к некоторому модулю, от которого зависит текущий модуль.
    // Этот модуль на момент обращения к нему должен быть загружен
    // в глобальную переменную SomeModuleDependency.
    // Контекст this в данном случае – глобальный объект.
    var moduleDependency = this.SomeModuleDependency;
    // Объявление в свойстве глобального объекта функции, возвращающей значение модуля.
    this.myGlobalModule = function () { return someModuleValue; };
}());
```

Интерпретатор, обнаруживая в коде такое функциональное выражение, сразу вычисляет его. В результате выполнения в свойство `myGlobalModule` глобального объекта будет помещена функция, которая будет возвращать значение модуля.

Особенности подхода:

- Сложность декларирования и использования модулей-зависимостей.
- В момент выполнения анонимной функции все зависимости модуля должны быть загружены.
- Загрузка модулей-зависимостей выполняется в заголовке страницы через HTML-элемент `<script>`. Обращение к модулям-зависимостям осуществляется через имена глобальных переменных. При этом разработчик должен четко представлять и реализовывать порядок загрузки модулей-зависимостей.
- Как следствие предыдущего пункта — модули загружаются до начала рендеринга страницы, поэтому в модулях нельзя обращаться к элементам управления страницы для реализации пользовательской логики.

Особенности использования подхода в Creatio:

- Отсутствие возможности динамической загрузки модулей.
- Применение дополнительной логики при загрузке модулей.
- Сложность управления большим количеством модулей со многими зависимостями, которые могут перекрывать друг друга.

Загрузчик RequireJS

Загрузчик RequireJS предоставляет механизм объявления и загрузки модулей, базирующийся на концепции AMD, и позволяющий избежать перечисленных выше недостатков.

Принципы работы механизма загрузчика RequireJS:

- Объявление модуля выполняется в функции `define()`, которая регистрирует функцию-фабрику для инстанцирования модуля, но при этом не загружает его в момент вызова.
- Зависимости модуля передаются как массив строковых значений, а не через свойства глобального объекта.
- Загрузчик выполняет загрузку модулей-зависимостей, переданных в качестве аргументов в функции `define()`. Модули загружаются асинхронно, при этом порядок их загрузки произвольно определяется загрузчиком.
- После загрузки указанных зависимостей модуля будет вызвана функция-фабрика, которая вернет значение модуля. При этом в нее в качестве аргументов будут переданы загруженные модули-зависимости.

Пример объявления модуля

Основы

Пример использования функции `define()` для объявления модуля `SumModule`

```
// Модуль с именем SumModule реализует функциональность суммирования двух чисел.
// SumModule не имеет зависимостей.
// Поэтому в качестве второго аргумента передается пустой массив, а
// анонимной функции-фабрике не передаются никакие параметры.
define("SumModule", [], function () {
```

```
    // Тело анонимной функции содержит внутреннюю реализацию функциональности модуля.
    var calculate = function (a, b) { return a + b; };
    // Возвращаемое функцией значение — объект, которым является модуль для системы.
    return {
        // Описание объекта. В данном случае модуль представляет собой объект со свойством summ.
        // Значение этого свойства — функция с двумя аргументами, которая возвращает сумму этих
        summ: calculate
    };
});
```

Функция define()

Основы

Назначение функции `define()` — объявление асинхронного модуля в исходном коде, с которым будет работать загрузчик.

Объявление модуля

```
define(
        ModuleName,
        [dependencies],
        function (dependencies) {
        }
);
```

Параметры

`ModuleName`

Строка с именем модуля. Необязательный параметр.

Если не указать параметр, загрузчик самостоятельно присвоит модулю имя в зависимости от его расположения в дереве скриптов приложения. Для обращения к модулю из других частей приложения (в том числе для асинхронной загрузки как зависимости другого модуля), имя модуля должно быть однозначно определено.

`dependencies`

Массив имен модулей, от которых зависит модуль. Необязательный параметр.

RequireJS выполняет асинхронную загрузку зависимостей, переданных в массиве. Порядок перечисления зависимостей в массиве `dependencies` должен соответствовать порядку перечисления параметров, передаваемых в фабричную функцию. Фабричная функция будет вызвана после загрузки зависимостей, перечисленных в `dependencies`.

`function(dependencies)`

Анонимная фабричная функция, которая инстанцирует модуль. Обязательный параметр.

В качестве параметров в функцию передаются объекты, которые ассоциируются загрузчиком с модулями-зависимостями, перечисленными в аргументе `dependencies`. Через эти аргументы осуществляется доступ к свойствам и методам модулей-зависимостей внутри создаваемого модуля. Порядок перечисления модулей в `dependencies` должен соответствовать порядку параметров фабричной функции.

Фабричная функция должна возвращать значение, которое загрузчик будет ассоциировать как экспортируемое значение создаваемого модуля.

Виды возвращаемого значения фабричной функции:

- Объект, которым является модуль для системы. Модуль сохраняется в кэше браузера после первичной загрузки на клиент. Если объявление модуля было изменено после загрузки на клиент (например, в процессе реализации конфигурационной логики), то необходимо очистить кэш и заново загрузить модуль.
- Функция-конструктор модуля. В качестве аргумента в конструктор передается объект контекста, в котором будет создаваться модуль. Загрузка модуля приведет к созданию на клиенте экземпляра модуля (**инстанцируемого модуля**). Повторная загрузка модуля на клиент функцией `require()` приведет к созданию еще одного экземпляра модуля. Эти экземпляры одного и того же модуля система будет воспринимать как два самостоятельных модуля. Примером объявления инстанцируемого модуля является модуль `CardModule` пакета `NUI`.

Виды модулей

Легкий

Базовые модули

В Creatio реализованы **базовые модули**:

- `ext-base` — реализует функциональность фреймворка `ExtJs`.
- `terrasoft` пространства имен и объектов `Terrasoft` — реализует доступ к системным операциям, переменным ядра и т. д.
- `sandbox` — реализует механизм обмена сообщениями между модулями.

Доступ к модулям `ext-base`, `terrasoft` и `sandbox`

```
// Определение модуля и получение ссылок на модули-зависимости.
define("ExampleModule", ["ext-base", "terrasoft", "sandbox"],
    // Ext – ссылка на объект, дающий доступ к возможностям фреймворка ExtJs.
    // Terrasoft – ссылка на объект, дающий доступ к системным переменным, переменным ядра и т.д
    // sandbox – используется для обмена сообщениями между модулями.
    function (Ext, Terrasoft, sandbox) {
});
```

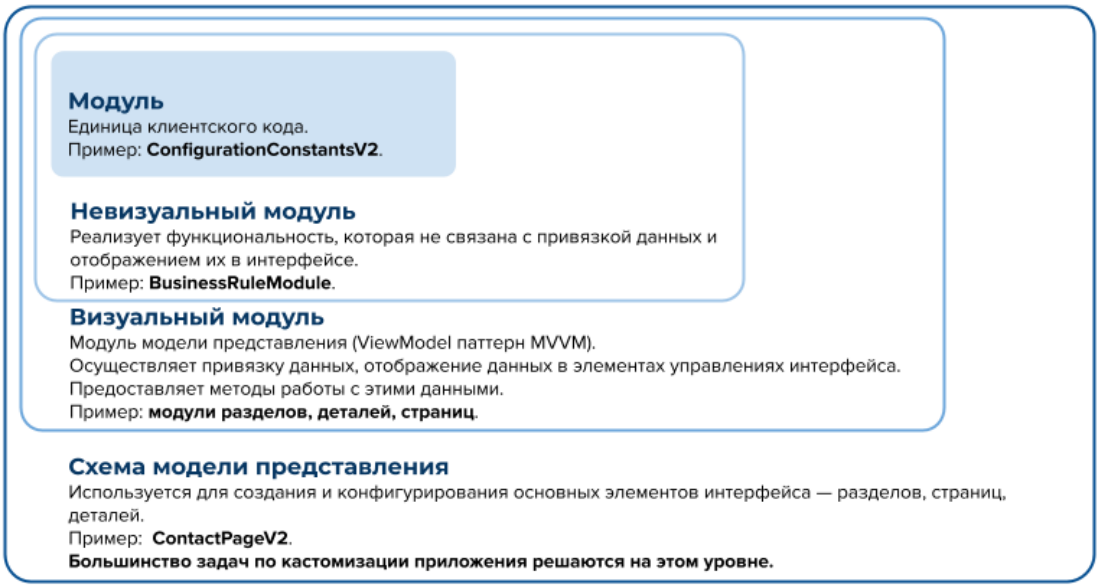
Базовые модули используются в большинстве клиентских модулей. Указывать базовые модули в зависимостях `["ext-base", "terrasoft", "sandbox"]` не обязательно. После создания объекта класса модуля объекты `Ext`, `Terrasoft` и `sandbox` будут доступны как свойства объекта `this.Ext`, `this.Terrasoft`, `this.sandrenderbox`.

Клиентские модули

Виды клиентских модулей

- Невизуальный модуль (схема модуля).
- Визуальный модуль (схема модели представления).
- Модуль расширения и замещающий клиентский модуль (схема замещающей модели представления).

Иерархия клиентских модулей представлена на рисунке ниже.



Разработка клиентских модулей описана в статье Разработка конфигурационных элементов.

Невизуальный модуль

Назначение невизуального модуля — реализация функциональности системы, которая, как правило, не сопряжена с привязкой данных и отображением их в интерфейсе. Примерами невизуальных модулей в системе являются модули бизнес-правил (`BusinessRuleModule`) и утилитные модули, которые реализуют

служебные функции.

Разработка невизуального модуля описана в статье Схема модуля.

Визуальный модуль

К визуальным относятся модули, которые реализуют в системе модели представления (`ViewModel`) согласно шаблону MVVM.

Назначение визуального модуля — инкапсуляция данных, которые отображаются в элементах управления графического интерфейса, и методов работы с данными. Примерами визуальных модулей в системе являются модули разделов, деталей, страниц.

Разработка визуального модуля описана в статье Схема модели представления.

Модуль расширения и замещающий клиентский модуль

Назначение модулей расширения и замещающих клиентских модулей — расширение функциональности базовых модулей.

Разработка модуля расширения и замещающего клиентского модуля описана в статье Схема замещающей модели представления.

Особенности клиентских модулей

Методы клиентских модулей

- `init()` — метод реализует логику, выполняемую при загрузке модуля. При загрузке модуля клиентское ядро автоматически вызывает этот метод первым. Как правило, в методе `init()` выполняется подписка на события других модулей и инициализация значений.
- `render(renderTo)` — метод реализует логику визуализации модуля. При загрузке модуля и наличии метода клиентское ядро автоматически его вызывает. Для корректного отображения данных перед их визуализацией должен отработать механизм связывания представления (`View`) и модели представления (`ViewModel`). Поэтому, как правило, в методе `render()` выполняется запуск механизма — вызов у объекта представления метода `bind()`. Если модуль загружается в контейнер, то в качестве аргумента метода `render()` будет передана ссылка на этот контейнер. Метод `render()` в обязательном порядке должен реализовываться визуальными модулями.

Вызов одного модуля из другого. Утилитные модули

Несмотря на то что модуль является изолированной программной единицей, он может использовать функциональность других модулей. Для этого достаточно в качестве зависимости импортировать тот модуль, функциональность которого предполагается использовать. Доступ к экземпляру модуля-зависимости осуществляется через аргумент фабричной функции.

Так, в процессе разработки вспомогательные и служебные методы общего назначения можно группировать в отдельные **утилитные модули** и затем импортировать их в те модули, в которых необходима эта функциональность.

Работа с ресурсами

Ресурсы — дополнительные свойства схемы. Ресурсы добавляются в клиентскую схему на панели свойств дизайнера (кнопка +). Чаще всего используемые ресурсы — локализуемые строки (свойство [*Локализуемые строки*] ([*Localizable strings*])) и изображения (свойство [*Изображения*] ([*Images*])). Ресурсы содержатся в специальном модуле с именем `[ИмяКлиентскогоМодуля]Resources`, который автоматически генерирует ядро приложения для каждого клиентского модуля.

Чтобы получить доступ к модулю ресурсов из клиентского модуля, необходимо в качестве зависимости импортировать модуль ресурсов в клиентский модуль. В коде модуля необходимо использовать локализуемые ресурсы, а не строковые литералы или константы.

Использование модулей расширения

Модули расширения базовой функциональности не поддерживают наследование в традиционном его представлении.

Особенности создания расширяющего модуля:

1. Скопировать программный код расширяемого модуля.
2. Внести изменения в функциональность.

В замещающем модуле нельзя использовать ресурсы замещаемого модуля — все ресурсы должны заново создаваться в замещающей схеме.

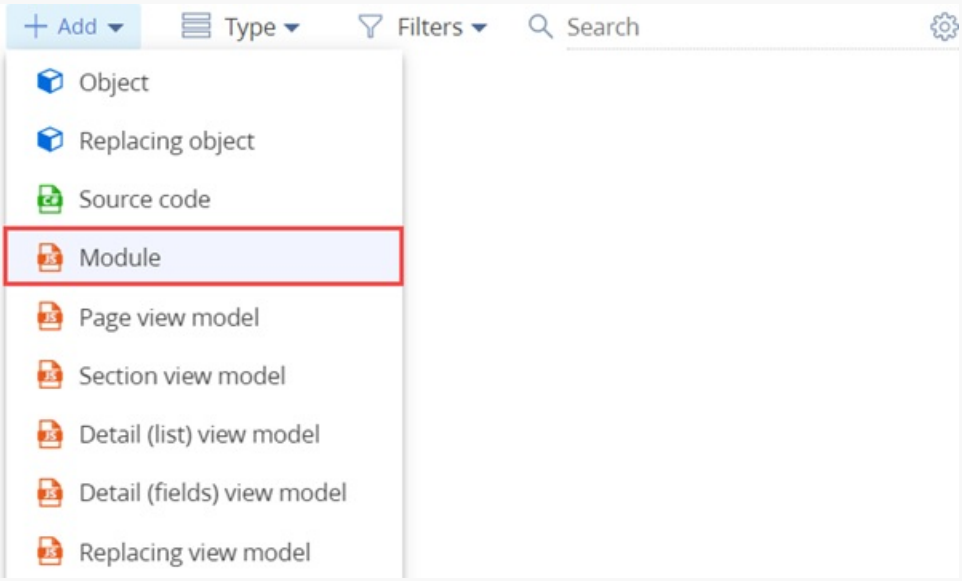
Создать стандартный модуль

Средний

Пример. Создать стандартный модуль, который содержит методы `init()` и `render()`. Каждый метод должен отображать информационное сообщение. При загрузке модуля на клиент ядро сначала вызовет метод `init()`, а затем — метод `render()`, о чем должны оповестить соответствующие информационные сообщения.

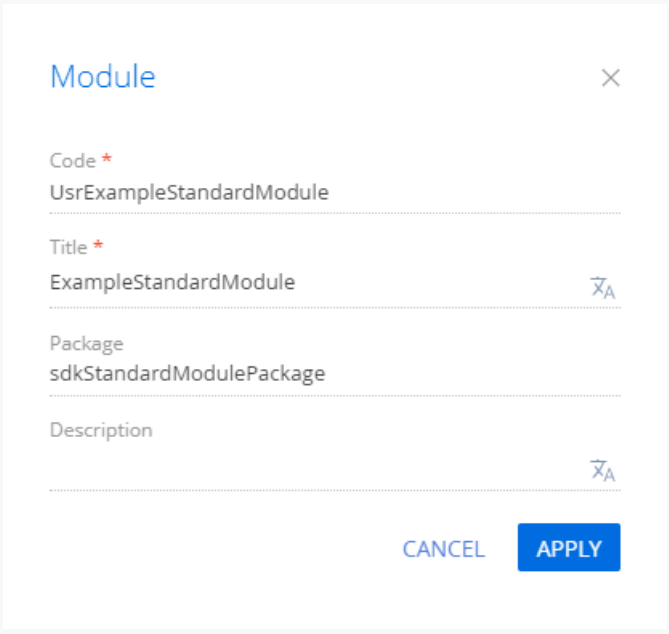
1. Создать визуальный модуль

1. Перейдите в раздел [*Конфигурация*] ([*Configuration*]) и выберите пользовательский пакет, в который будет добавлена схема.
2. На панели инструментов реестра раздела нажмите [*Добавить*] —> [*Модуль*] ([*Add*] —> [*Module*]).



3. В дизайнере схем заполните свойства схемы:

   - [*Код*] ([*Code*]) — "`UsrExampleStandardModule`".
   - [*Заголовок*] ([*Title*]) — "`ExampleStandardModule`".



   Для применения заданных свойств нажмите [*Применить*] ([*Apply*]).

4. В дизайнере схем добавьте исходный код.

Исходный код модуля

```
// Объявление модуля с именем UsrExampleStandartModule. Модуль не имеет никаких зависимостей,
// поэтому в качестве второго параметра передается пустой массив.
define("UsrExampleStandardModule", [],
```

```
// Функция-фабрика возвращает объект модуля с двумя методами.
function () {
    return {
        // Метод будет вызван ядром самым первым, сразу после загрузки на клиент.
        init: function () {
            alert("Calling the init() method of the UsrExampleStandardModule module");
        },
        // Метод будет вызван ядром при загрузке модуля в контейнер. Ссылка на контейнер
        // в качестве параметра renderTo. В информационном сообщении будет выведен id эле
        // в котором должны отображаться визуальные данные модуля. По умолчанию — centerP
        render: function (renderTo) {
            alert("Calling the render() method of the UsrExampleStandardModule module. Th
        }
    };
});
```

5. На панели инструментов дизайнера нажмите [*Сохранить*] ([*Save*]).

2. Проверить визуальный модуль

Базовая версия Creatio позволяет проверить визуальный модуль, выполнить его загрузку на клиент и визуализацию. Для этого сформируйте адресную строку запроса.

Адресная строка запроса

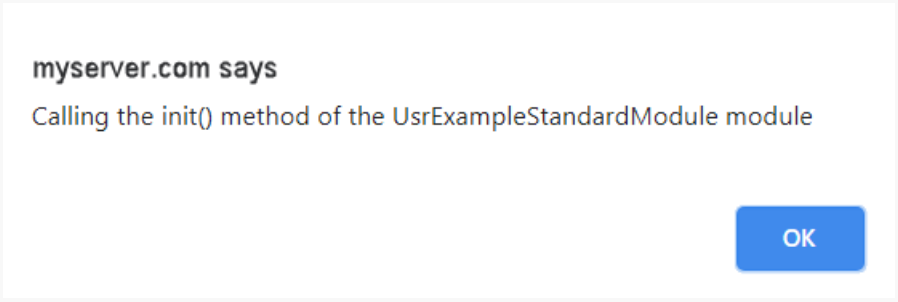
```
[АдресПриложения]/[НомерКонфигурации]/0/NUI/ViewModule.aspx#[ИмяМодуля]
```

Пример адресной строки запроса

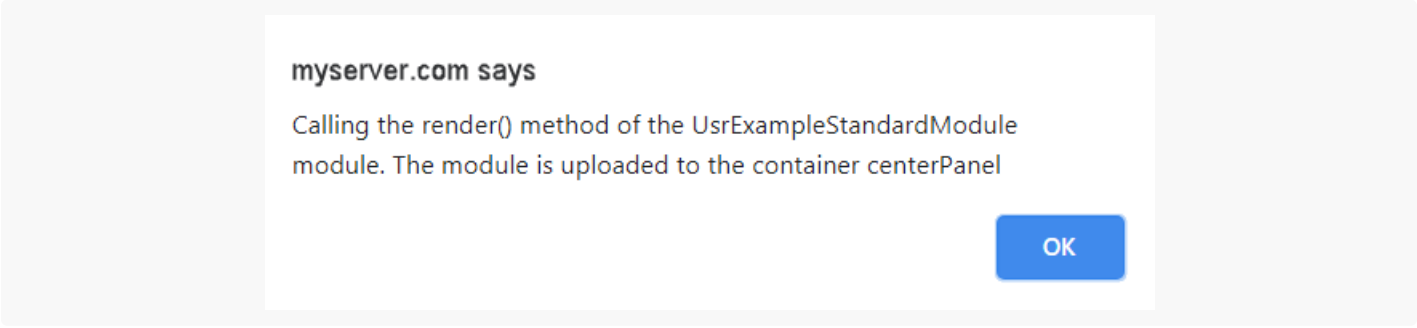
```
http://myserver.com/CreatioWebApp/0/NUI/ViewModule.aspx#UsrExampleStandardModule
```

На клиент будет возвращен модуль `UsrExampleStandardModule`.

Вызов метода `init()` модуля `UsrExampleStandardModule`



Вызов метода `render()` модуля `UsrExampleStandardModule`



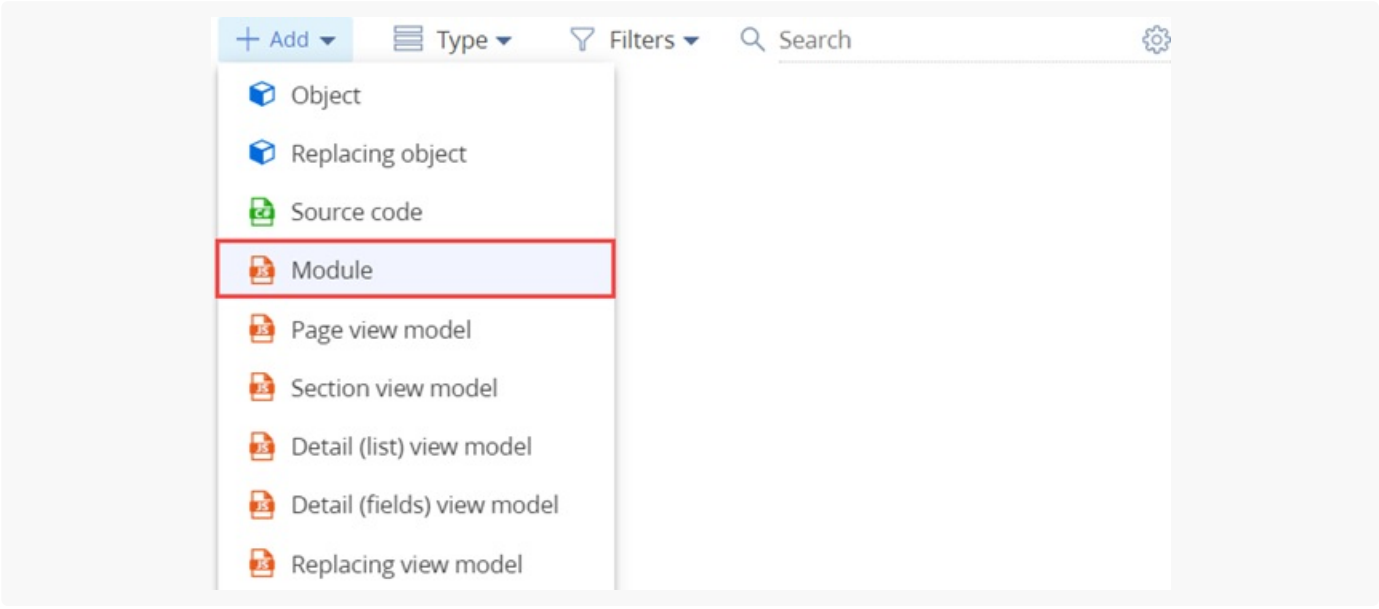
Создать утилитный модуль

Средний

Пример. Создать стандартный модуль, который содержит методы `init()` и `render()`. Каждый метод должен отображать информационное сообщение. При загрузке модуля на клиент ядро сначала вызовет метод `init()`, а затем — метод `render()`, о чем должны оповестить соответствующие информационные сообщения. Метод отображения информационного окна необходимо вынести в отдельный утилитный модуль.

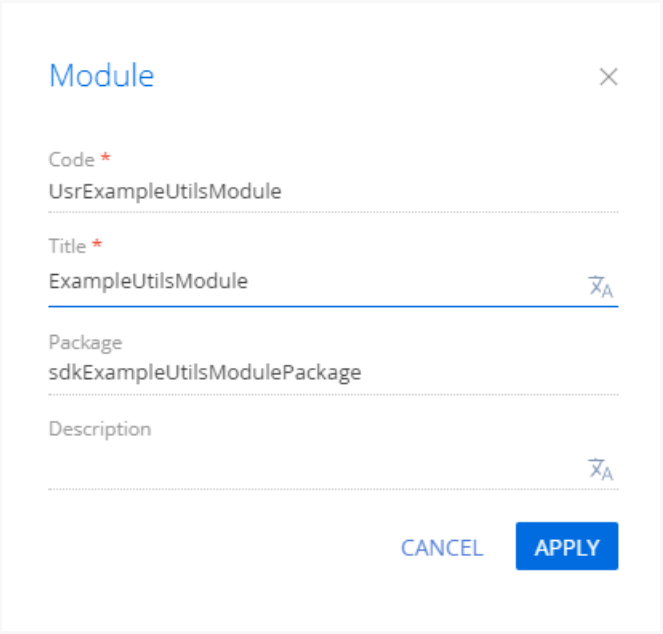
1. Создать утилитный модуль

1. Перейдите в раздел [*Конфигурация*] ([*Configuration*]) и выберите пользовательский пакет, в который будет добавлена схема.
2. На панели инструментов реестра раздела нажмите [*Добавить*] —> [*Модуль*] ([*Add*] —> [*Module*]).



3. В дизайнере схем заполните свойства схемы:

- [*Код*] ([*Code*]) — "`UsrExampleUtilsModule`".
- [*Заголовок*] ([*Title*]) — "`ExampleUtilsModule`".



Для применения заданных свойств нажмите [*Применить*] ([*Apply*]).

4. В дизайнере схем добавьте исходный код.

Исходный код утилитного модуля

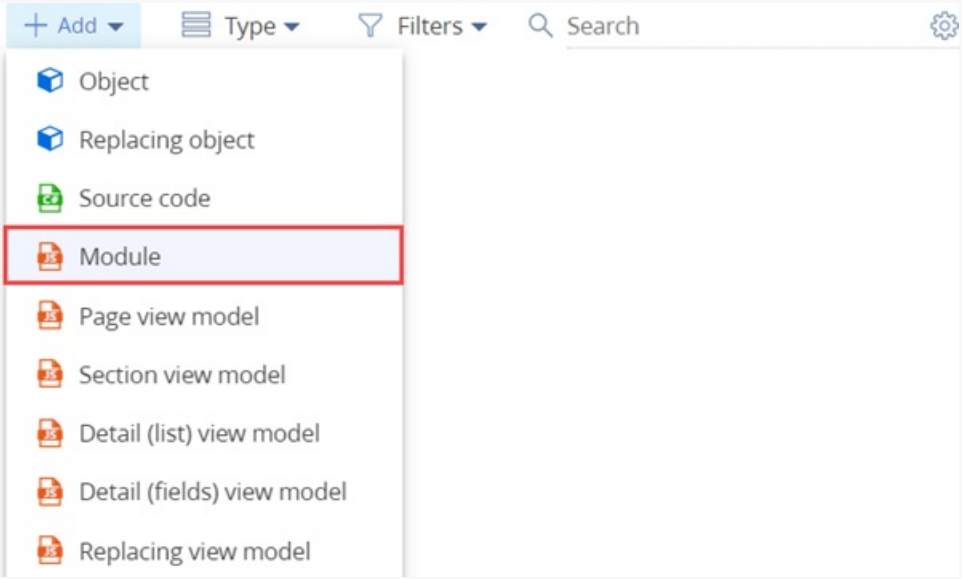
```
// Объявление утилитного модуля. Модуль не имеет зависимостей и содержит один метод
// для отображения информационного сообщения.
define("UsrExampleUtilsModule", [],
    function () {
        return {
            // Метод, который отображает информационное окно с сообщением. Сообщение, выводим
            // передается в метод в качестве аргумента information.
            showInformation: function (information) {
                alert(information);
            }
        };
    });
```

5. На панели инструментов дизайнера нажмите [*Сохранить*] ([*Save*]).

2. Создать визуальный модуль

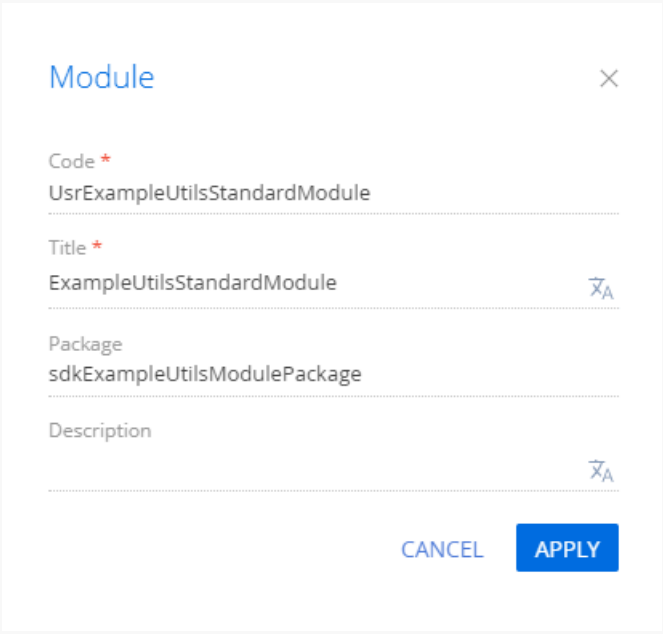
1. Перейдите в раздел [*Конфигурация*] ([*Configuration*]) и выберите пользовательский пакет, в который будет добавлена схема.

2. На панели инструментов реестра раздела нажмите [*Добавить*] —> [*Модуль*] ([*Add*] —> [*Module*]).



3. В дизайнере схем заполните свойства схемы:

   - [*Код*] ([*Code*]) — "`UsrExampleUtilsStandardModule`".
   - [*Заголовок*] ([*Title*]) — "`ExampleUtilsStandardModule`".



   Для применения заданных свойств нажмите [*Применить*] ([*Apply*]).

4. В дизайнере схем добавьте исходный код.

Исходный код модуля

```
// В модуль импортируется модуль-зависимость UsrExampleUtilsModule для доступа к утилитному м
```

```
// Аргумент функции-фабрики – ссылка на загруженный утилитный модуль.
define("UsrExampleUtilsStandardModule", ["UsrExampleUtilsModule"],
    function (UsrExampleUtilsModule) {
        return {
            // В функциях init() и render() вызывается утилитный метод для отображения информ
            // с сообщением, которое передается в качестве аргумента утилитному методу.
            init: function () {
                UsrExampleUtilsModule.showInformation("Calling the init() method of the UsrEx
            },
            render: function (renderTo) {
                UsrExampleUtilsModule.showInformation("Calling the render() method of the Usr
            }
        };
    });
```

5. На панели инструментов дизайнера нажмите [*Сохранить*] ([*Save*]).

3. Проверить визуальный модуль

Базовая версия Creatio позволяет проверить визуальный модуль, выполнить его загрузку на клиент и визуализацию. Для этого сформируйте адресную строку запроса.

Адресная строка запроса

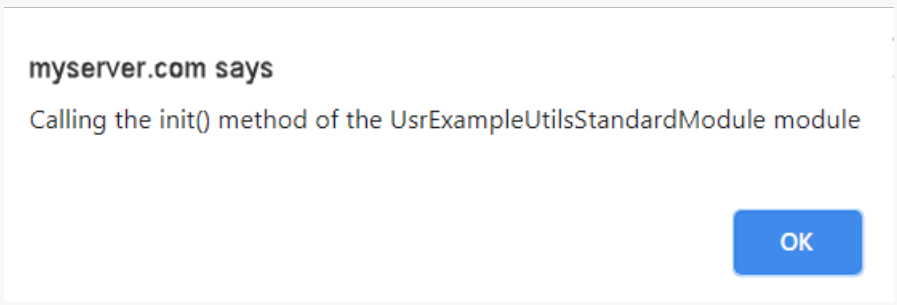
```
[АдресПриложения]/[НомерКонфигурации]/0/NUI/ViewModule.aspx#[ИмяМодуля]
```

Пример адресной строки запроса

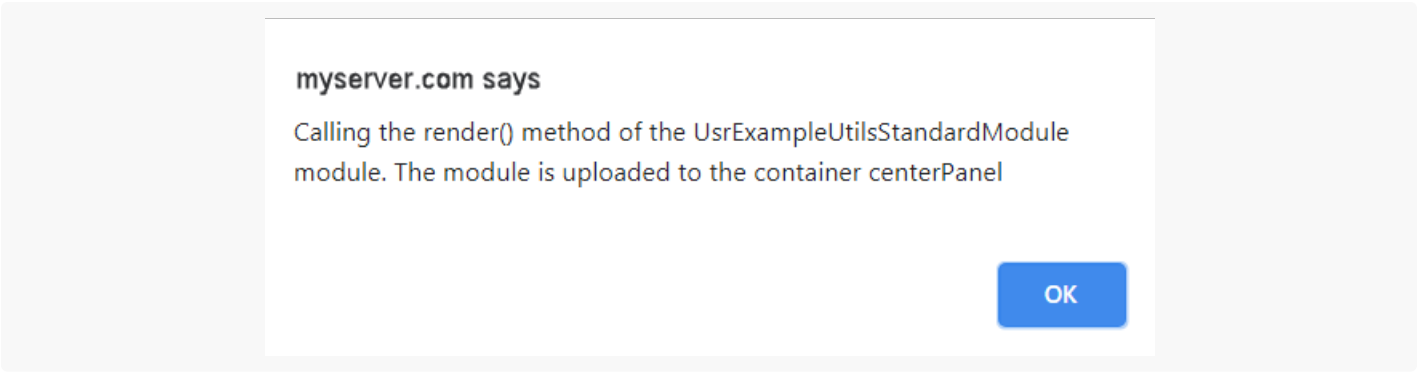
```
http://myserver.com/CreatioWebApp/0/NUI/ViewModule.aspx#UsrExampleUtilsStandardModule
```

На клиент будет возвращен модуль `UsrExampleUtilsStandardModule`.

Вызов метода `init()` модуля `UsrExampleUtilsStandardModule`



Вызов метода `render()` модуля `UsrExampleUtilsStandardModule`



Класс модуля

Средний

Объявление класса модуля

Объявление классов — функция JavaScript-фреймворка `ExtJs`. Для объявления классов используется стандартный механизм библиотеки — метод `define()` глобального объекта `Ext`.

Пример объявления класса с помощью метода `define()`

```
// Название класса с соблюдением пространства имен.
Ext.define("Terrasoft.configuration.ExampleClass", {
    // Сокращенное название класса.
    alternateClassName: "Terrasoft.ExampleClass",
    // Название класса, от которого происходит наследование.
    extend: "Terrasoft.BaseClass",
    // Блок для объявления статических свойств и методов.
    static: {
        // Пример статического свойства.
        myStaticProperty: true,

        // Пример статического метода.
        getMyStaticProperty: function () {
            // Пример доступа к статическому свойству.
            return Terrasoft.ExampleClass.myStaticProperty;
        }
    },
    // Пример динамического свойства.
    myProperty: 12,
    // Пример динамического метода класса.
    getMyProperty: function () {
        return this.myProperty;
    }
});
```

Примеры создания экземпляров класса представлены ниже.

Пример создания экземпляра класса по полному имени

```
// Создание экземпляра класса по полному имени.
var exampleObject = Ext.create("Terrasoft.configuration.ExampleClass");
```

Пример создания экземпляра класса по псевдониму

```
// Создание экземпляра класса по сокращенному названию – псевдониму.
var exampleObject = Ext.create("Terrasoft.ExampleClass");
```

Наследование класса модуля

В большинстве случаев класс модуля правильно наследовать от `Terrasoft.configuration.BaseModule` или `Terrasoft.configuration.BaseSchemaModule`, в которых реализованы **методы**:

- `init()` — метод инициализации модуля. Отвечает за инициализацию свойств объекта класса, а также за подписку на сообщения.
- `render(renderTo)` — метод отрисовки представления модуля в DOM. Возвращает представление модуля. Принимает единственный аргумент `renderTo`, в который будет добавлено представление объекта модуля.
- `destroy()` — метод, отвечающий за удаление представления модуля, удаление модели представления, отписку от ранее подписанных сообщений и уничтожение объекта класса модуля.

Ниже приведен пример класса модуля, наследуемого от `Terrasoft.BaseModule`. Данный модуль добавляет кнопку в DOM. При клике на кнопку выводится сообщение, после чего она удаляется из DOM.

Пример класса модуля, наследуемого от `Terrasoft.BaseModule`

```
define("ModuleExample", [], function () {
    Ext.define("Terrasoft.configuration.ModuleExample", {
        // Короткое название класса.
        alternateClassName: "Terrasoft.ModuleExample",
        // Класс, от которого происходит наследование.
        extend: "Terrasoft.BaseModule",
        /* Обязательное свойство. Если не определено, будет сгенерирована ошибка на уровне
           "Terrasoft.core.BaseObject", так как класс наследуется от "Terrasoft.BaseModule".*/
        Ext: null,
        /* Обязательное свойство. Если не определено, будет сгенерирована ошибка на уровне
           "Terrasoft.core.BaseObject", так как класс наследуется от "Terrasoft.BaseModule".*/
        sandbox: null,
```

```
        /* Обязательное свойство. Если не определено, будет сгенерирована ошибка на уровне
           "Terrasoft.core.BaseObject", так как класс наследуется от "Terrasoft.BaseModule".*/
        Terrasoft: null,
        // Модель представления.
        viewModel: null,
        // Представление. В качестве примера используется кнопка.
        view: null,
        /* Если не реализовать метод init() в текущем классе,
           то при создании экземпляра текущего класса будет вызван метод
           init() класса-родителя Terrasoft.BaseModule.*/
        init: function () {
            // Вызывает выполнение логики метода init() класса-родителя.
            this.callParent(arguments);
            this.initViewModel();
        },
        // Инициализирует модель представления.
        initViewModel: function () {
            /* Сохранение контекста класса модуля
               для доступа к нему из модели представления.*/
            var self = this;
            // Создание модели представления.
            this.viewModel = Ext.create("Terrasoft.BaseViewModel", {
                values: {
                    // Заголовок кнопки.
                    captionBtn: "Click Me"
                },
                methods: {
                    // Обработчик нажатия на кнопку.
                    onClickBtn: function () {
                        var captionBtn = this.get("captionBtn");
                        alert(captionBtn + " button was pressed");
                        /* Вызывает метод выгрузки представления и модели представления,
                           что приводит к удалению кнопки из DOM.*/
                        self.destroy();
                    }
                }
            });
        },
        /* Создает представление (кнопку),
           связывает ее с моделью представления и вставляет в DOM.*/
        render: function (renderTo) {
            // В качестве представления создается кнопка.
            this.view = this.Ext.create("Terrasoft.Button", {
                // Контейнер, в который будет помещена кнопка.
                renderTo: renderTo,
                // HTML-атрибут id.
                id: "example-btn",
                // Название класса.
                className: "Terrasoft.Button",
```

```
                // Заголовок.
                caption: {
                    // Связывает заголовок кнопки
                    // со свойством captionBtn модели представления.
                    bindTo: "captionBtn"
                },
                // Метод-обработчик события нажатия на кнопку.
                click: {
                    /* Связывает обработчик события нажатия на кнопку
                       с методом onClickBtn() модели представления.*/
                    bindTo: "onClickBtn"
                },
                /* Стиль кнопки. Возможные стили определены в перечислении
                   Terrasoft.controls.ButtonEnums.style.*/
                style: this.Terrasoft.controls.ButtonEnums.style.GREEN
            });
            // Связывает представление и модель представления.
            this.view.bind(this.viewModel);
            // Возвращает представление, которое будет вставлено в DOM.
            return this.view;
        },
        // Удаляет неиспользуемые объекты.
        destroy: function () {
            // Уничтожает представление, что приводит к удалению кнопки из DOM.
            this.view.destroy();
            // Удаляет неиспользуемую модель представления.
            this.viewModel.destroy();
        }
    });
    // Возвращает объект модуля.
    return Terrasoft.ModuleExample;
});
```

Мониторинг переопределения членов класса модуля

При наследовании класса модуля в классе-наследнике могут быть переопределены как публичные, так и приватные свойства и методы базового модуля.

В Creatio **приватными свойствами или методами класса** считаются те, названия которых начинаются с нижнего подчеркивания, например, `_privateMemberName`.

В Creatio присутствует функциональность **мониторинга переопределения приватных членов класса** — класс `Terrasoft.PrivateMemberWatcher`.

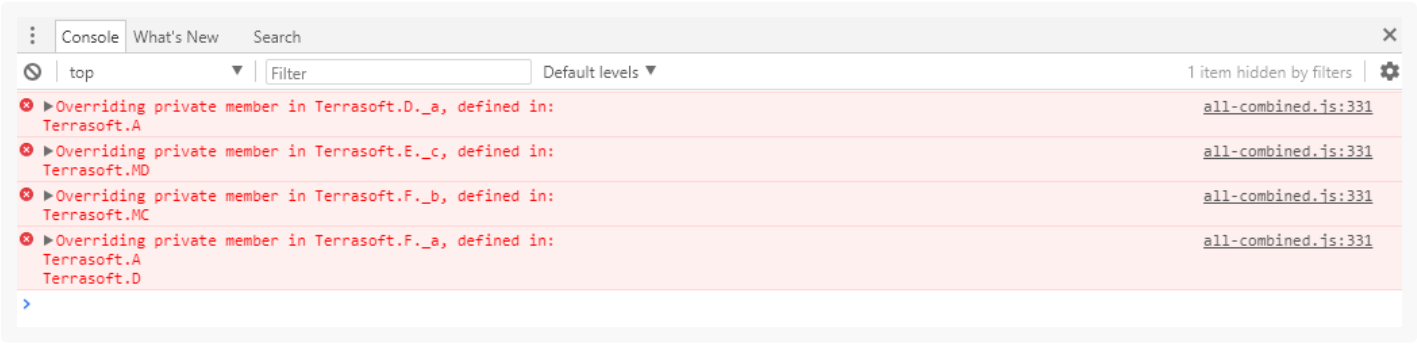
Назначение мониторинга — при определении пользовательского класса проверять выполнение переопределений приватных свойств или методов, которые объявлены в родительских классах. При этом в режиме отладки отображается предупреждение в консоли браузера.

Например, в пользовательский пакет добавлена схема модуля, исходный код которой приведен ниже.

Пример переопределения приватных свойств класса модуля

```
define("UsrPrivateMemberWatcher", [], function() {
    Ext.define("Terrasoft.A", {_a: 1});
    Ext.define("Terrasoft.B", {extend: "Terrasoft.A"});
    Ext.define("Terrasoft.MC", {_b: 1});
    Ext.define("Terrasoft.C", {extend: "Terrasoft.B", mixins: {ma: "Terrasoft.MC"}});
    Ext.define("Terrasoft.MD", {_c: 1});
    // Переопределение свойства _a.
    Ext.define("Terrasoft.D", {extend: "Terrasoft.C", _a: 3, mixins: {mb: "Terrasoft.MD"}});
    // Переопределение свойства _c.
    Ext.define("Terrasoft.E", {extend: "Terrasoft.D", _c: 3});
    // Переопределение свойств _a и _b.
    Ext.define("Terrasoft.F", {extend: "Terrasoft.E", _b: 3, _a: 0});
});
```

После загрузки этого модуля в консоли отобразятся предупреждения о том, что приватные члены базовых классов были переопределены.



Инициализация экземпляра класса модуля

Синхронная инициализация

Модуль инициализируется синхронно, если при его загрузке явно не указано свойство `isAsync: true` конфигурационного объекта, передаваемого в качестве параметра метода `loadModule()`. Например, при выполнении

```
this.sandbox.loadModule([moduleName])
```

методы класса модуля будут загружены синхронно. Первым будет вызван метод `init()`, после которого сразу же будет вызван метод `render()`.

Пример реализации синхронно инициализируемого модуля

```
define("ModuleExample", [], function () {
    Ext.define("Terrasoft.configuration.ModuleExample", {
        alternateClassName: "Terrasoft.ModuleExample",
        Ext: null,
        sandbox: null,
        Terrasoft: null,
        init: function () {
            // При инициализации выполнится первым.
        },
        render: function (renderTo) {
            // При инициализации модуля выполнится сразу же после метода init().
        }
    });
});
```

Асинхронная инициализация

Модуль инициализируется асинхронно, если при его загрузке явно указано свойство `isAsync: true` конфигурационного объекта, передаваемого в качестве параметра метода `loadModule()`. Например, при выполнении

```
this.sandbox.loadModule([moduleName], {
    isAsync: true
})
```

первым будет вызван метод `init()`, в который будет передан единственный параметр — callback-функция с контекстом текущего модуля. При вызове callback-функции будет вызван метод `render()` загружаемого модуля. Представление будет добавлено в DOM только после выполнения метода `render()`.

Пример реализации асинхронно инициализируемого модуля

```
define("ModuleExample", [], function () {
    Ext.define("Terrasoft.configuration.ModuleExample", {
        alternateClassName: "Terrasoft.ModuleExample",
        Ext: null,
        sandbox: null,
        Terrasoft: null,
        // При инициализации модуля выполнится первым.
        init: function (callback) {
            setTimeout(callback, 2000);
        },
        render: function (renderTo) {
            // Метод выполнится с задержкой в 2 секунды,
```

```
            // Задержка указана в аргументе функции setTimeout() в методе init().
        }
    });
});
```

Цепочки модулей

Цепочки модулей — механизм, который позволяет отобразить представление одной модели на месте представления другой модели. Например, для установки значения поля на текущей странице необходимо отобразить страницу `SelectData` для выбора значения из справочника. То есть, на месте контейнера модуля текущей страницы должно отобразиться представление модуля страницы выбора из справочника.

Для построения цепочки необходимо добавить свойство `keepAlive` в конфигурационный объект загружаемого модуля. Например, в модуле текущей страницы `CardModule` необходимо вызвать модуль выбора из справочника `selectDataModule`.

Пример вызова модуля выбора из справочника `selectDataModule` в модуле страницы `CardModu`

```
sandbox.loadModule("selectDataModule", {
    // Id представления загружаемого модуля.
    id: "selectDataModule_id",
    // Представление будет добавлено в контейнер текущей страницы.
    renderTo: "cardModuleContainer",
    // Указывает, чтобы текущий модуль не выгружался.
    keepAlive: true
});
```

После выполнения кода будет построена цепочка из модуля текущей страницы и модуля страницы выбора из справочника. Добавление еще одного элемента в цепочку позволяет из модуля текущей страницы `selectData` по нажатию на кнопку [*Добавить новую запись*] ([*Add new record*]) открыть новую страницу. Добавив в цепочку еще один элемент, из модуля текущей страницы `selectData` по нажатию на кнопку [*Добавить новую запись*] ([*Add new record*]) откроется новая страница. Таким образом, в цепочку модулей можно добавлять неограниченное количество экземпляров модулей.

Активный модуль (тот, который отображен на странице) — последний элемент цепочки. Если установить активным элемент из середины цепочки, то будут уничтожены все элементы, находящиеся в цепочке после него. Активация элемента цепочки выполняется путем вызова функции `loadModule()`, в параметр которой необходимо передать идентификатор модуля.

Пример вызова функции `loadModule()`

```
sandbox.loadModule("someModule", {
    id: "someModuleId"
});
```

Ядро уничтожит все элементы цепочки, после чего вызовет методы `init()` и `render()`. При этом в метод `render()` будет передан контейнер, который содержит предыдущий активный модуль. Модули цепочки могут работать, как и раньше — принимать и отправлять сообщения, сохранять данные и т. д.

Если при вызове метода `loadModule()` в конфигурационный объект не добавлять свойство `keepAlive` или добавить его со значением `keepAlive: false`, то цепочка модулей будет уничтожена.