

Файловый контент пакетов

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

Файловый контент пакетов	4
Структура хранения файлового контента пакета	4
Bootstrap-файлы пакета	6
Версионность файлового контента	7
Генерация вспомогательных файлов	7
Предварительная генерация статического файлового контента	8
Генерация файлового контента	8
Совместимость с режимом разработки в файловой системе	10
Перенос изменений между рабочими средами	11
Локализовать файловый контент с помощью конфигурационных ресурсов	11
1. Создать модуль с локализуемыми ресурсами	11
2. Импортировать модуль локализуемых ресурсов	12
Локализовать файловый контент с помощью плагина i18n	12
1. Добавить плагин	12
2. Создать папку с локализуемыми ресурсами	12
3. Создать папки культур	13
4. Добавить файлы с локализуемыми ресурсами	13
5. Отредактировать файл bootstrap.js	14
6. Использовать ресурсы в клиентском модуле	15
Использовать TypeScript при разработке клиентской функциональности	15
1. Установить TypeScript	16
2. Перейти в режим разработки в файловой системе	16
3. Создать структуру хранения файлового контента	17
4. Реализовать валидацию на языке TypeScript	18
5. Выполнить компиляцию исходных кодов TypeScript в исходные коды JavaScript	19
6. Выполнить генерацию вспомогательных файлов	21
7. Проверить результат выполнения примера	21
Создать Angular-компонент для использования в Creatio	24
Создание пользовательского Angular-компонента	24
Подключение Custom Element в Creatio	28
Работа с данными	29
Использование Shadow DOM	31

Файловый контент пакетов



Файловый контент пакетов — файлы (*.js, *.css, изображения и др.), добавленные в пользовательские пакеты приложения. Файловый контент является статическим и не обрабатывается веб-сервером, что позволяет повысить скорость работы приложения.

Виды файлового контента:

- Клиентский контент, генерируемый в режиме реального времени.
- Предварительно сгенерированный клиентский контент.

Особенности использования клиентского контента, генерируемого в режиме реального времени:

- Нет необходимости предварительно генерировать клиентский контент.
- При вычислении иерархии пакетов, схем и формировании контента присутствует нагрузка на процессор (CPU).
- При получении иерархии пакетов, схем и формировании контента присутствует нагрузка на базу данных.
- Потребление памяти для кэширования клиентского контента.

Особенности использования предварительно сгенерированного клиентского контента:

- Присутствует минимальная нагрузка на процессор.
- Необходимо предварительно генерировать клиентский контент.
- Отсутствуют запросы в базу данных.
- Клиентский контент кэшируется средствами IIS.

Структура хранения файлового контента пакета

Файловый контент является частью пакета. Для повышения производительности приложения и снижения нагрузки на базу данных весь файловый контент можно предварительно сгенерировать в специальной папке приложения `...\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\[Имя пакета]\Files`. При запросе сервер IIS ищет запрашиваемый контент в этой папке и сразу же отправляет его приложению. В пакет могут быть добавлены любые файлы, однако использоваться будут только файлы, необходимые для клиентской части Creatio.

Рекомендуется использовать **структуру** папки `Files`, приведенную ниже.

Рекомендуемая структура папки `Files`

```
-PackageName
...
-Files
```

```

    -src
      -js
        bootstrap.js
        [другие *.js-файлы]
      -css
        [*.css-файлы]
      -less
        [*.less-файлы]
      -img
        [файлы изображений]
      -res
        [файлы ресурсов]
    descriptor.json
  ...
descriptor.json

```

`js` — папка с *.js-файлами исходных кодов на языке JavaScript.

`css` — папка с *.css-файлами стилей.

`less` — папка с *.less-файлами стилей.

`img` — папка с изображениями.

`res` — папка с файлами ресурсов.

`descriptor.json` — дескриптор файлового контента, который хранит информацию о bootstrap-файлах пакета.

Структура файла `descriptor.json` представлена ниже.

Структура файла `descriptor.json`

```

{
  "bootstraps": [
    ... // Массив строк, содержащих относительные пути к bootstrap-файлам.
  ]
}

```

Пример файла `descriptor.json`

```

{
  "bootstraps": [
    "src/js/bootstrap.js",
    "src/js/anotherBootstrap.js"
  ]
}

```

Чтобы добавить файловый контент в пакет, необходимо поместить файл в соответствующую вложенную папку папки `Files` необходимого пакета.

Bootstrap-файлы пакета

Bootstrap-файлы пакета — это *.js-файлы, которые позволяют управлять загрузкой клиентской конфигурационной логики. Структура файла может варьироваться.

Структура файла bootstrap.js

```
(function() {
  require.config({
    paths: {
      "Название модуля": "Ссылка на файловый контент",
      ...
    }
  });
})();
```

Пример файла bootstrap.js

```
(function() {
  require.config({
    paths: {
      "MyPackage1-ContactSectionV2": Terrasoft.getFileContentUrl("MyPackage1", "src/js/Cor
      "MyPackage1-Utilities": Terrasoft.getFileContentUrl("MyPackage1", "src/js/Utilities.
    }
  });
})();
```

Bootstrap-файлы загружаются асинхронно после загрузки ядра, но до загрузки конфигурации. Для корректной загрузки bootstrap-файлов в папке статического контента генерируется вспомогательный файл `_FileContentBootstraps.js`, который содержит информацию о bootstrap-файлах всех пакетов.

Пример содержимого файла `_FileContentBootstraps.js`

```
var Terrasoft = Terrasoft || {};
Terrasoft.configuration = Terrasoft.configuration || {};
Terrasoft.configuration.FileContentBootstraps = {
  "MyPackage1": [
    "src/js/bootstrap.js"
  ]
};
```

Версионность файлового контента

Для корректной работы версионности файлового контента в папке статического контента генерируется вспомогательный файл `_FileContentDescriptors.js`. Это файл, в котором в виде коллекции "ключ-значение" содержится информация о файлах в файловом контенте всех пакетов. Каждому ключу (названию файла) соответствует значение — уникальный хэш-код. Таким образом обеспечивается гарантированная загрузка в браузер актуальной версии файла.

Пример содержимого файла `_FileContentDescriptors.js`

```
var Terrasoft = Terrasoft || {};
Terrasoft.configuration = Terrasoft.configuration || {};
Terrasoft.configuration.FileContentDescriptors = {
  "MyPackage1/descriptor.json": {
    "Hash": "5d4e779e7ff24396a132a0e39cca25cc"
  },
  "MyPackage1/Files/src/js/Utilities.js": {
    "Hash": "6d5e776e7ff24596a135a0e39cc525gc"
  }
};
```

Генерация вспомогательных файлов

Для **генерации вспомогательных файлов** (`_FileContentBootstraps.js` и `FileContentDescriptors.js`) необходимо с помощью [утилиты WorkspaceConsole](#) выполнить операцию `BuildConfiguration`.

Параметры операции `BuildConfiguration`

Параметр	Описание
<code>operation</code>	Название операции. Необходимо установить значение <code>BuildConfiguration</code> — операция компиляции конфигурации.
<code>useStaticFileContent</code>	Признак использования статического контента. Необходимо установить значение <code>false</code> .
<code>usePackageFileContent</code>	Признак использования файлового контента пакетов. Необходимо установить значение <code>true</code> .

Генерация вспомогательных файлов

```
Terrasoft.Tools.WorkspaceConsole.exe -operation=BuildConfiguration -workspaceName=Default -desti
```

В результате выполнения операции в папке со статическим контентом `...\Terrasoft.WebApp\conf\content` будут сгенерированы вспомогательные файлы `_FileContentBoostraps.js` и `_FileContentDescriptors.js`.

Описание параметров утилиты `WorkspaceConsole` содержится в статье [Параметры утилиты WorkspaceConsole](#).

Предварительная генерация статического файлового контента

Файловый контент генерируется в специальную папку `.\Terrasoft.WebApp\conf`, которая содержит *.js-файлы с исходным кодом схем, *.css-файлы стилей и *.js-файлы ресурсов для всех культур приложения, а также изображения.

Важно. Для генерации статического контента папки `.\Terrasoft.WebApp\conf` пользователю пула IIS, в котором запущено приложение, необходимо иметь права на модификацию. Права настраиваются на уровне сервера в секции `Handler Mappings`. Подробнее описано в статье [Настроить сервер приложения на IIS](#).

Имя пользователя пула IIS устанавливается в свойстве `[Identity]`. Доступ к этому свойству можно получить в менеджере IIS на вкладке `[Application Pools]` через команду `[Advanced Settings]`.

Условия для выполнения первичной или повторной генерации статического файлового контента:

- Сохранение схемы через дизайнеры клиентских схем и объектов.
 - Сохранение через мастера разделов и деталей.
 - Установка и удаление приложений из Marketplace и *.zip-архива.
 - Применение переводов.
 - Действия `[Компилировать]` (`[Compile]`) и `[Перекомпилировать все]` (`[Compile all]`) раздела `[Конфигурация]` (`[Configuration]`).
- Эти действия необходимо выполнять при **удалении схем или пакетов** из раздела `[Конфигурация]` (`[Configuration]`).

Действие `[Перекомпилировать все]` (`[Compile all]`) необходимо выполнять при **установке или обновлении пакета** из системы контроля версий [SVN](#).

На заметку. Действие `[Перекомпилировать все]` (`[Compile all]`) выполняет полную регенерацию файлового статического контента. Остальные действия в системе выполняют регенерацию только измененных схем.

Генерация файлового контента

Для **генерации файлового контента** необходимо с помощью утилиты `WorkspaceConsole` выполнить операцию `BuildConfiguration`.

Параметры операции BuildConfiguration

Параметр	Описание
<code>workspaceName</code>	Название рабочего пространства. По умолчанию — <code>Default</code> .
<code>destinationPath</code>	Папка, в которую будет сгенерирован статический контент.
<code>webApplicationPath</code>	<p>Путь к веб-приложению, из которого будет вычитана информация по соединению с базой данных.</p> <p>Необязательный параметр. Если значение не указано, то соединение будет установлено с базой данных, указанной в строке соединения в файле <code>Terrasoft.Tools.WorkspaceConsole.config</code>. Если значение указано, то соединение будет установлено с базой данных из файла <code>ConnectionStrings.config</code> веб-приложения.</p>
<code>force</code>	<p>Необязательный параметр. По умолчанию — <code>false</code> (выполняется генерация файлового контента для измененных схем).</p> <p>Если установлено значение <code>true</code>, то выполняется генерация файлового контента по всем схемам.</p>

Генерация файлового контента (вариант 1)

```
Terrasoft.Tools.WorkspaceConsole.exe -operation=BuildConfiguration -workspaceName=Default -desti
```

Генерация файлового контента (вариант 2)

```
Terrasoft.Tools.WorkspaceConsole.exe -operation=BuildConfiguration -workspaceName=Default -webAp
```

Генерация клиентского контента при добавлении новой культуры

После **добавления новых культур** из интерфейса приложения необходимо в разделе [*Конфигурация*] ([*Configuration*]) выполнить действие [*Перекомпилировать все*] ([*Compile all*]).

Важно. Если пользователь не может войти в систему после добавления новой культуры, то необходимо перейти в раздел [*Конфигурация*] ([*Configuration*]) по ссылке `http://[Путь к приложению]/0/dev`.

Получение URL изображения

Изображения в клиентской части Creatio запрашиваются браузером по URL, который устанавливается в атрибуте `src` html-элемента `img`. Для формирования этого URL в Creatio используется модуль `Terrasoft.ImageUrlBuilder (imageurlbuilder.js)`, в котором реализован `getUrl(config)` — публичный метод для получения URL изображения. Этот метод принимает конфигурационный JavaScript-объект `config`, в свойстве `params` которого содержится объект параметров. На основе этого свойства формируется URL изображения для вставки на страницу.

Структура объекта `params`

```
config: {
  params: {
    schemaName: "",
    resourceItemName: "",
    hash: "",
    resourceItemExtension: ""
  }
}
```

`schemaName` — название схемы (строка).

`resourceItemName` — название изображения в Creatio (строка).

`hash` — хэш изображения (строка).

`resourceItemExtension` — расширение файла изображения (например, ".png").

Пример формирования конфигурационного объекта параметров для получения URL статического изображения представлен ниже.

Пример формирования конфигурационного объекта параметров

```
var localizableImages = {
  AddButtonImage: {
    source: 3,
    params: {
      schemaName: "ActivityMiniPage",
      resourceItemName: "AddButtonImage",
      hash: "c15d635407f524f3bbe4f1810b82d315",
      resourceItemExtension: ".png"
    }
  }
}
```

Совместимость с режимом разработки в файловой системе

Режим разработки в файловой системе несовместим с получением клиентского контента из

предварительно сгенерированных файлов. Для корректной работы с режимом разработки в файловой системе необходимо **отключить получение статического клиентского контента** из файловой системы. Для отключения данной функциональности необходимо в файле `web.config` для флага `UseStaticFileContent` установить значение `false`.

Отключить получение статического клиентского контента из файловой системы

```
<fileDesignMode enabled="true" />
...
<add key="UseStaticFileContent" value="false" />
```

Перенос изменений между рабочими средами

Файловый контент является неотъемлемой частью пакета. Он фиксируется в хранилище системы контроля версий наравне с остальным содержимым пакета. В дальнейшем файловый контент может быть **перенесен на другую рабочую среду**:

- Для переноса изменений на [среду разработки](#) рекомендуется использовать систему контроля версий SVN.
- Для переноса изменений на [предпромышленную](#) и [промышленную](#) среды рекомендуется использовать механизм [экспорта и импорта](#) Creatio IDE.

Важно. При установке пакетов папка `Files` будет создана только в том случае, если она не пустая. Если эта папка не была создана, то для начала разработки ее необходимо создать вручную.

Локализовать файловый контент с помощью конфигурационных ресурсов

 Сложный

1. Создать модуль с локализуемыми ресурсами

Для **перевода ресурсов** на разные языки рекомендуется использовать отдельный модуль с локализуемыми ресурсами, созданный встроенными инструментами Creatio в разделе [*Конфигурация*] ([*Configuration*]).

Пример модуля с локализуемыми ресурсами

```
define("Module1", ["Module1Resources"], function(res) {
    return res;
});
```

2. Импортировать модуль локализуемых ресурсов

Чтобы из клиентского модуля **получить доступ к модулю локализуемых ресурсов**, необходимо в качестве зависимости импортировать модуль локализуемых ресурсов в клиентский модуль.

Пример подключения локализуемых ресурсов в модуль

```
define("MyPackage-MyModule", ["Module1"], function(module1) {
  console.log(module1.localizableStrings.MyString);
});
```

Локализовать файловый контент с помощью плагина i18n



i18n — это плагин для AMD-загрузчика (например, RequireJS), предназначенный для загрузки локализуемых строковых ресурсов. Исходный код плагина можно найти в [GitHub-репозитории](#).

1. Добавить плагин

Добавьте плагин в папку с *.js-файлами исходных кодов

```
..\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\MyPackage1\content\js\i18n.js .
```

Здесь `MyPackage1` — рабочая папка пакета `MyPackage1`.

2. Создать папку с локализуемыми ресурсами

Создайте папку `..\MyPackage1\content\nls` и добавьте в нее *.js-файл с локализуемыми ресурсами.

Можно добавлять несколько *.js-файлов с локализуемыми ресурсами. Имена файлов могут быть произвольными. Содержимое файлов — AMD модули, которые содержат объекты.

Структура объектов AMD модулей:

- **Поле** `root`.

Поле содержит коллекцию "ключ-значение", где ключ — это название локализуемой строки, а значение — локализуемая строка на языке по умолчанию. Значение будет использоваться, если запрашиваемый язык не поддерживается.

- **Поля культур.**

В качестве имен полей установите стандартные коды поддерживаемых культур (например, `en-US`, `ru-RU`), а значение имеет логический тип (`true` — поддерживаемая культура включена, `false` — поддерживаемая культура отключена).

Ниже представлен пример файла `..\MyPackage1\content\js\nls\ContactSectionV2Resources.js`.

Пример файла ContactSectionV2Resources.js

```
define({
  "root": {
    "FileContentActionDescr": "File content first action (Default)",
    "FileContentActionDescr2": "File content second action (Default)"
  },
  "en-US": true,
  "ru-RU": true
});;
```

3. Создать папки культур

В папке `..\MyPackage1\content\nls` создайте папки культур. В качестве имен папок установите код той культуры, локализация которой будет в них размещена (например, `en-US`, `ru-RU`).

Структура папки `MyPackage1` с русской и английской культурами представлена ниже.

Структура папки MyPackage1

```
content
  nls
    en-US
    ru-RU
```

4. Добавить файлы с локализуемыми ресурсами

В каждую созданную папку локализации поместите такой же набор *.js-файлов с локализуемыми ресурсами, как и в корневой папке `..\MyPackage1\content\nls`. Содержимое файлов — AMD модули, объекты которых являются коллекциями "ключ-значение", где ключ — это наименование локализуемой строки, а значение — строка на языке, соответствующем названию папки (коду культуры).

Например, если поддерживаются только русская и английская культуры, то создайте два файла

`ContactSectionV2Resources.js`.

Файл `ContactSectionV2Resources.js`, соответствующий английской культуре

```
define({
  "FileContentActionDescr": "File content first action",
  "FileContentActionDescr2": "File content second action"
});;
```

Файл ContactSectionV2Resources.js, соответствующий русской культуре

```
define({
  "FileContentActionDescr": "Первое действие файлового контента"
});
```

Поскольку для русской культуры перевод строки "FileContentActionDescr2" не указан, то будет использовано значение по умолчанию — "File content second action (Default)".

5. Отредактировать файл bootstrap.js

Чтобы отредактировать файл `bootstrap.js`:

1. Подключите плагин `i18n`, указав его название в виде псевдонима `i18n` в конфигурации путей `RequireJS` и прописав соответствующий путь к нему в свойстве `paths`.
2. Укажите плагину культуру, которая является текущей для пользователя. Для этого свойству `config` объекта конфигурации библиотеки `RequireJS` присвойте объект со свойством `i18n`, которому, в свою очередь, присвойте объект со свойством `locale` и значением, полученным из глобальной переменной `Terrasoft.currentUserCultureName` (код текущей культуры).
3. Для каждого файла с ресурсами локализации укажите соответствующие псевдонимы и пути в конфигурации путей `RequireJS`. При этом псевдоним должен являться URL-путем относительно директории `nls`.

Пример файла `..\MyPackage1\content\js\bootstrap.js`

```
(function() {
  require.config({
    paths: {
      "MyPackage1-Utilities": Terrasoft.getFileContentUrl("MyPackage1", "content/js/Utilit
      "MyPackage1-ContactSectionV2": Terrasoft.getFileContentUrl("MyPackage1", "content/js
      "MyPackage1-CSS": Terrasoft.getFileContentUrl("MyPackage1", "content/css/MyPackage.c
      "MyPackage1-LESS": Terrasoft.getFileContentUrl("MyPackage1", "content/less/MyPackage
      "i18n": Terrasoft.getFileContentUrl("MyPackage1", "content/js/i18n.js"),
      "nls/ContactSectionV2Resources": Terrasoft.getFileContentUrl("MyPackage1", "content/
      "nls/ru-RU/ContactSectionV2Resources": Terrasoft.getFileContentUrl("MyPackage1", "cc
      "nls/en-US/ContactSectionV2Resources": Terrasoft.getFileContentUrl("MyPackage1", "c
    },
    config: {
      i18n: {
        locale: Terrasoft.currentUserCultureName
      }
    }
  });
})();
```

6. Использовать ресурсы в клиентском модуле

Чтобы использовать ресурсы в клиентском модуле, укажите в массиве зависимостей модуль с ресурсами с префиксом "i18n!".

Ниже представлен пример использования локализуемой строки `FileContentActionDescr` в качестве заголовка для нового действия раздела [*Контакты*] ([*Contacts*]).

Пример файла `..\MyPackage1\content\js\ContactSectionV2.js`

```
define("MyPackage1-ContactSectionV2", ["i18n!nls/ContactSectionV2Resources",
  "css!MyPackage1-CSS", "less!MyPackage1-LESS"], function(resources) {
  return {
    methods: {
      getSectionActions: function() {
        var actionMenuItems = this.callParent(arguments);
        actionMenuItems.addItem(this.getButtonMenuItem({"Type": "Terrasoft.MenuSeparator"}));
        actionMenuItems.addItem(this.getButtonMenuItem({
          "Click": {"bindTo": "onFileContentActionClick"},
          "Caption": resources.FileContentActionDescr
        }));
        return actionMenuItems;
      },
      onFileContentActionClick: function() {
        console.log("File content clicked!")
      }
    },
    diff: /**SCHEMA_DIFF*/[/**SCHEMA_DIFF*/
  ]
});
```

Использовать TypeScript при разработке клиентской функциональности



При разработке клиентской функциональности файловый контент позволяет использовать компилируемые в JavaScript языки, например, **TypeScript**. Подробнее о TypeScript можно узнать на официальном [сайте TypeScript](#).

Пример. При сохранении записи контрагента выводить для пользователя сообщение о правильности заполнения поля [*Альтернативные названия*] ([*Also known as*]). Поле должно содержать только буквенные символы. Логика валидации поля реализовать на языке TypeScript.

1. Установить TypeScript

Одним из способов установки TypeScript является использование **менеджера пакетов NPM** для `Node.js`.

Чтобы **установить TypeScript**:

1. Проверьте наличие среды выполнения `Node.js` в вашей операционной системе.
Скачать инсталлятор можно на сайте <https://nodejs.org>.
2. В консоли Windows выполните команду:

Команда для установки TypeScript

```
npm install -g typescript
```

2. Перейти в режим разработки в файловой системе

Чтобы **настроить Creatio для работы в файловой системе**:

1. **Включите режим разработки в файловой системе.**

В файле `web.config`, который находится в корневом каталоге приложения, установите значение `true` для атрибута `enabled` элемента `fileDesignMode`.

2. **Отключите получение статического клиентского контента из файловой системы.**

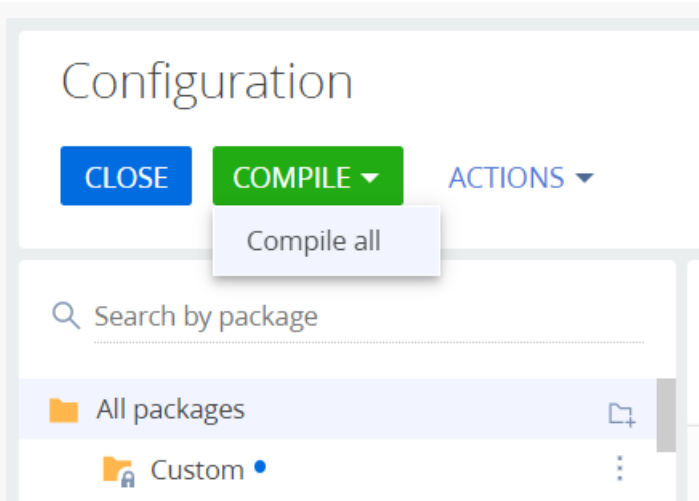
В файле `web.config`, который находится в корневом каталоге приложения, установите значение `false` для флага `UseStaticFileContent`.

Web.config

```
<filedesignmode enabled="true"/>
...
<add key="UseStaticFileContent" value="false"/>
```

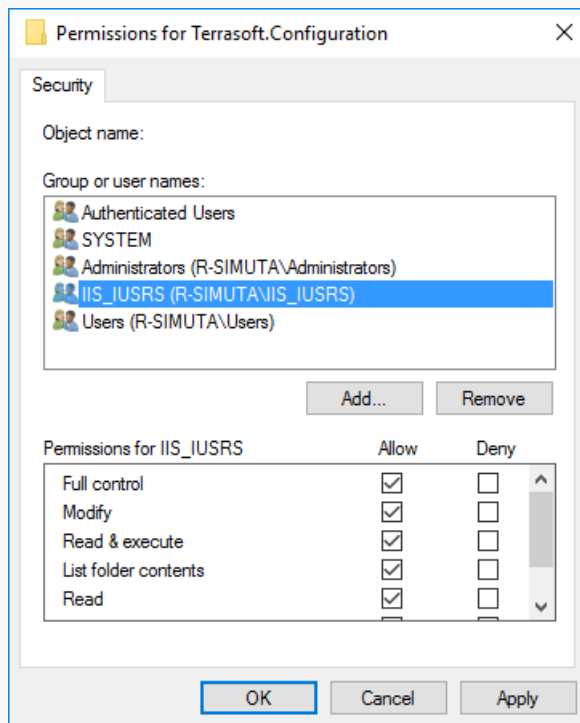
3. **Скомпилируйте приложение.**

В разделе [*Конфигурация*] ([*Configuration*]) выполните действие [*Компилировать все*] ([*Compile all items*]).



4. Предоставьте доступ IIS к каталогу конфигурации.

Чтобы приложение могло корректно работать с конфигурационным проектом, необходимо предоставить полный доступ пользователю операционной системы, от имени которого запущен пул приложений IIS, к каталогу `[Путь к приложению]\Terrasoft.WebApp\Terrasoft.Configuration`. Как правило, это встроенный пользователь `IIS_IUSRS`.



Режим разработки в файловой системе описан в статье [Внешние IDE](#).

3. Создать структуру хранения файлового контента

Чтобы **создать структуру хранения файлового контента**:

1. В пользовательском пакете, выгруженном в файловую систему, создайте каталог `Files`.

- В каталоге `Files` создайте вложенный каталог `src`.
- В каталоге `src` создайте вложенный каталог `js`.
- В каталоге `Files` создайте файл `descriptor.json`.

`descriptor.json`

```
{
  "bootstraps": [
    "src/js/bootstrap.js"
  ]
}
```

- В каталоге `Files\src\js` создайте файл `bootstrap.js`.

`bootstrap.js`

```
(function() {
  require.config({
    paths: {
      "LettersOnlyValidator": Terrasoft.getFileContentUrl("sdkTypeScript", "src/js/Lett
    }
  });
})();
```

4. Реализовать валидацию на языке TypeScript

Чтобы **реализовать валидацию на языке TypeScript**:

- В каталоге `Files\src\js` создайте файл `Validation.ts`, в котором объявите интерфейс `StringValidator`.

`Validation.ts`

```
interface StringValidator {
  isAcceptable(s: string): boolean;
}
export = StringValidator;
```

- В каталоге `Files\src\js` создайте файл `LettersOnlyValidator.ts`. Объявите в нем класс `LettersOnlyValidator`, реализующий интерфейс `StringValidator`.

`LettersOnlyValidator.ts`

```
// Импорт модуля, в котором реализован интерфейс StringValidator.
import StringValidator = require("Validation");

// Создаваемый класс должен принадлежать пространству имен (модулю) Terrasoft.
module Terrasoft {
    // Объявление класса валидации значений.
    export class LettersOnlyValidator implements StringValidator {
        // Регулярное выражение, допускающее использование только буквенных символов.
        lettersRegexp: any = /^[A-Za-z]+$/;
        // Валидирующий метод.
        isAcceptable(s: string) {
            return !Ext.isEmpty(s) && this.lettersRegexp.test(s);
        }
    }
}

// Создание и экспорт экземпляра класса для require.
export = new Terrasoft.LettersOnlyValidator();
```

5. Выполнить компиляцию исходных кодов TypeScript в исходные коды JavaScript

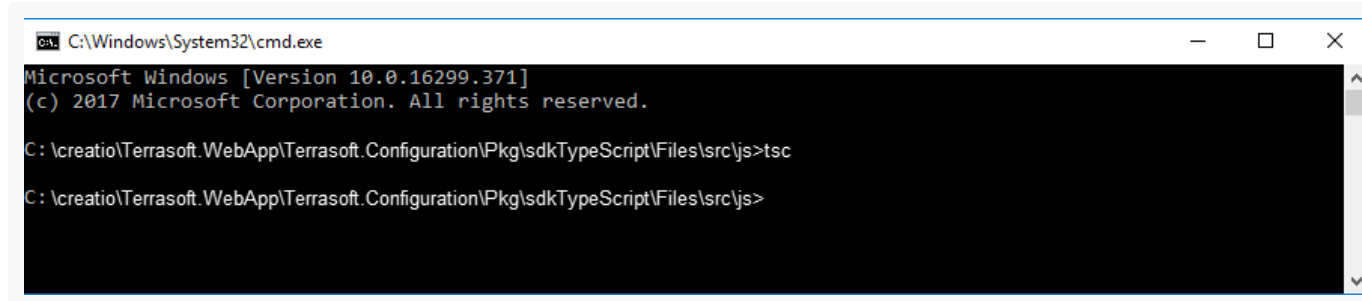
Чтобы **выполнить компиляцию исходных кодов TypeScript в исходные коды JavaScript**:

1. Для настройки компиляции добавьте в каталог `Files\src\js` конфигурационный файл `tsconfig.json`.

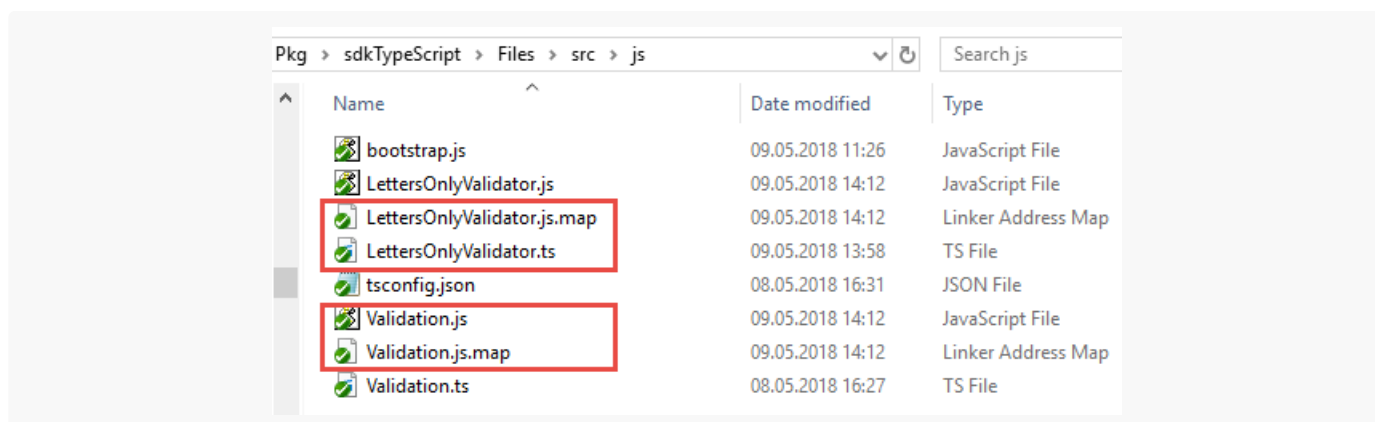
`tsconfig.json`

```
{
  "compilerOptions":
  {
    "target": "es5",
    "module": "amd",
    "sourceMap": true
  }
}
```

2. В консоли Windows перейдите в каталог `Files\src\js` и выполните команду `tsc`.



В результате выполнения компиляции в каталоге `Files\src\js` будут созданы JavaScript-версии файлов `Validation.ts` и `LettersOnlyValidator.ts`, а также *.map-файлы, облегчающие отладку в браузере.



Содержимое файла `LettersOnlyValidator.js`, который будет использоваться в Creatio, получено автоматически.

LettersOnlyValidator.js

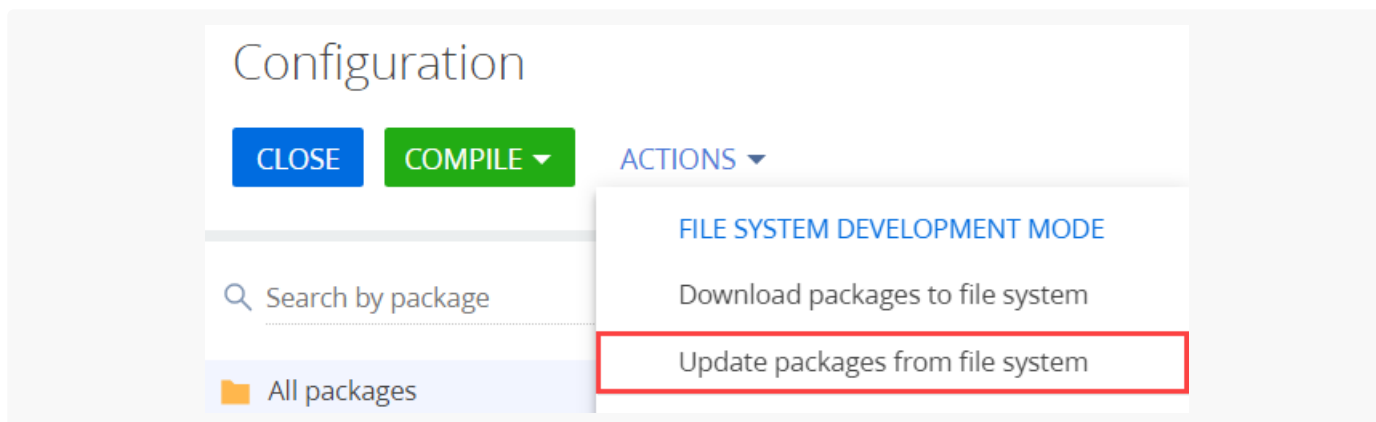
```
define(["require", "exports"], function (require, exports) {
    "use strict";
    var Terrasoft;
    (function (Terrasoft) {
        var LettersOnlyValidator = /** @class */ (function () {
            function LettersOnlyValidator() {
                this.lettersRegexp = /^[A-Za-z]+$/;
            }
            LettersOnlyValidator.prototype.isAcceptable = function (s) {
                return !Ext.isEmpty(s) && this.lettersRegexp.test(s);
            };
            return LettersOnlyValidator;
        })();
        Terrasoft.LettersOnlyValidator = LettersOnlyValidator;
    })(Terrasoft || (Terrasoft = {}));
    return new Terrasoft.LettersOnlyValidator();
});
```

```
//# sourceMappingURL=LettersOnlyValidator.js.map
```

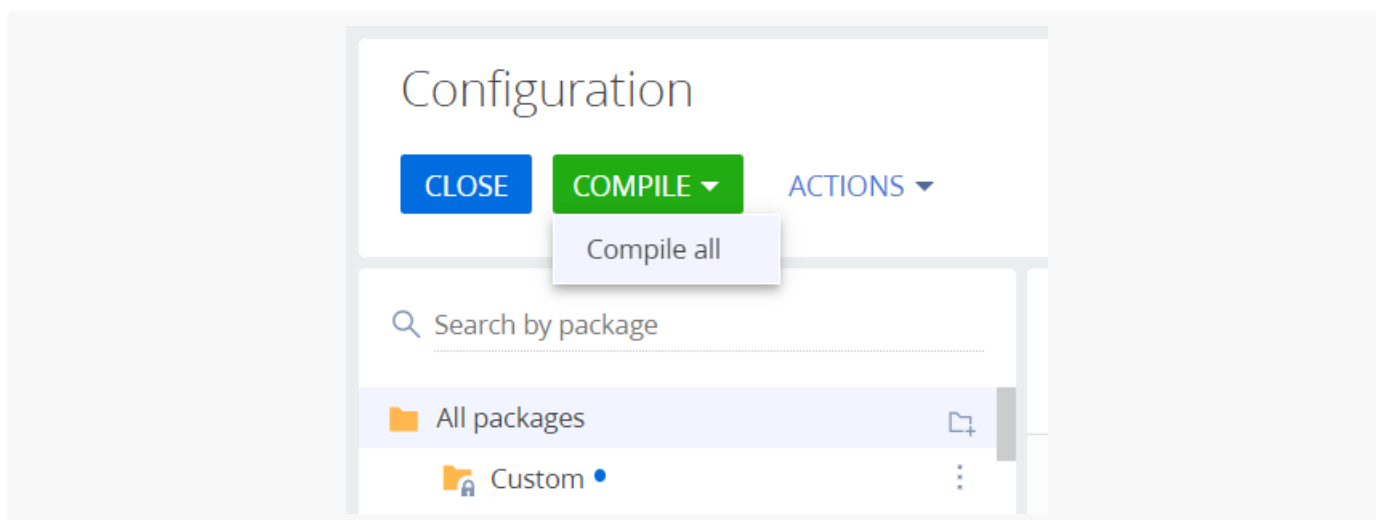
6. Выполнить генерацию вспомогательных файлов

Чтобы **выполнить генерацию вспомогательных файлов** `_FileContentBootstraps.js` и `FileContentDescriptors.js`:

1. Перейдите в раздел [Конфигурация] ([*Configuration*]).
2. Выполните загрузку пакетов из файловой системы (действие [Обновить пакеты из файловой системы] ([*Update packages from file system*])).



3. Для применения изменений в файле `bootstrap.js` выполните компиляцию приложения (действие [Компилировать все] ([*Compile all items*])).

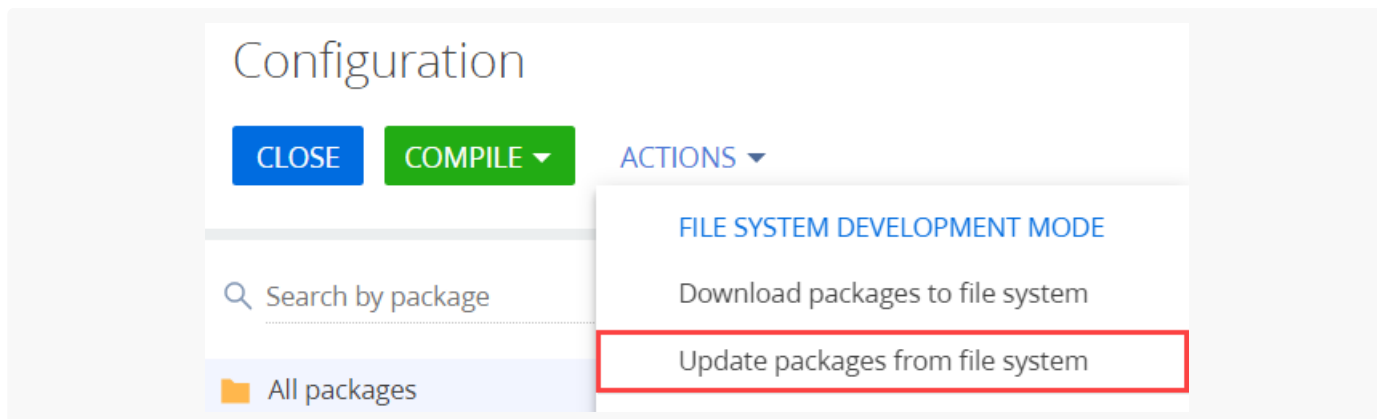


7. Проверить результат выполнения примера

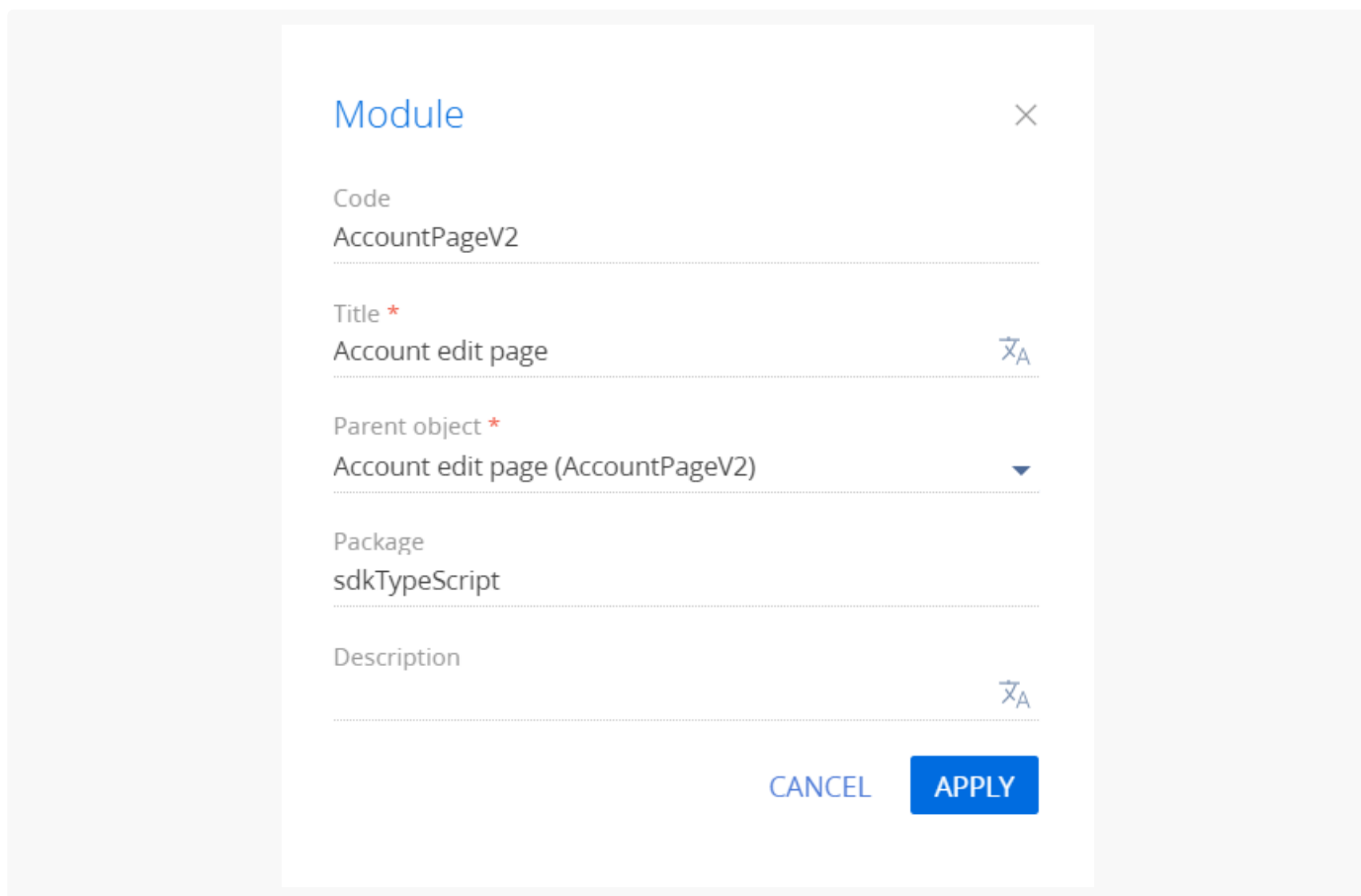
Чтобы **использовать валидацию**:

1. Перейдите в раздел [Конфигурация] ([*Configuration*]).

2. Выполните загрузку пакетов из файловой системы (действие [Обновить пакеты из файловой системы] ([Update packages from file system])).



3. Создайте [схему замещающей модели представления](#) страницы контрагента.



4. Выполните выгрузку пакетов в файловую систему (действие [Выгрузить пакеты в файловую систему] ([Download packages to file system])).
5. В файловой системе измените файл `..\sdkTypeScript\Schemas\AccountPageV2\AccountPageV2.js`.

```
..\sdkTypeScript\Schemas\AccountPageV2\AccountPageV2.js
```

```
// Объявление модуля и его зависимостей.
```

```

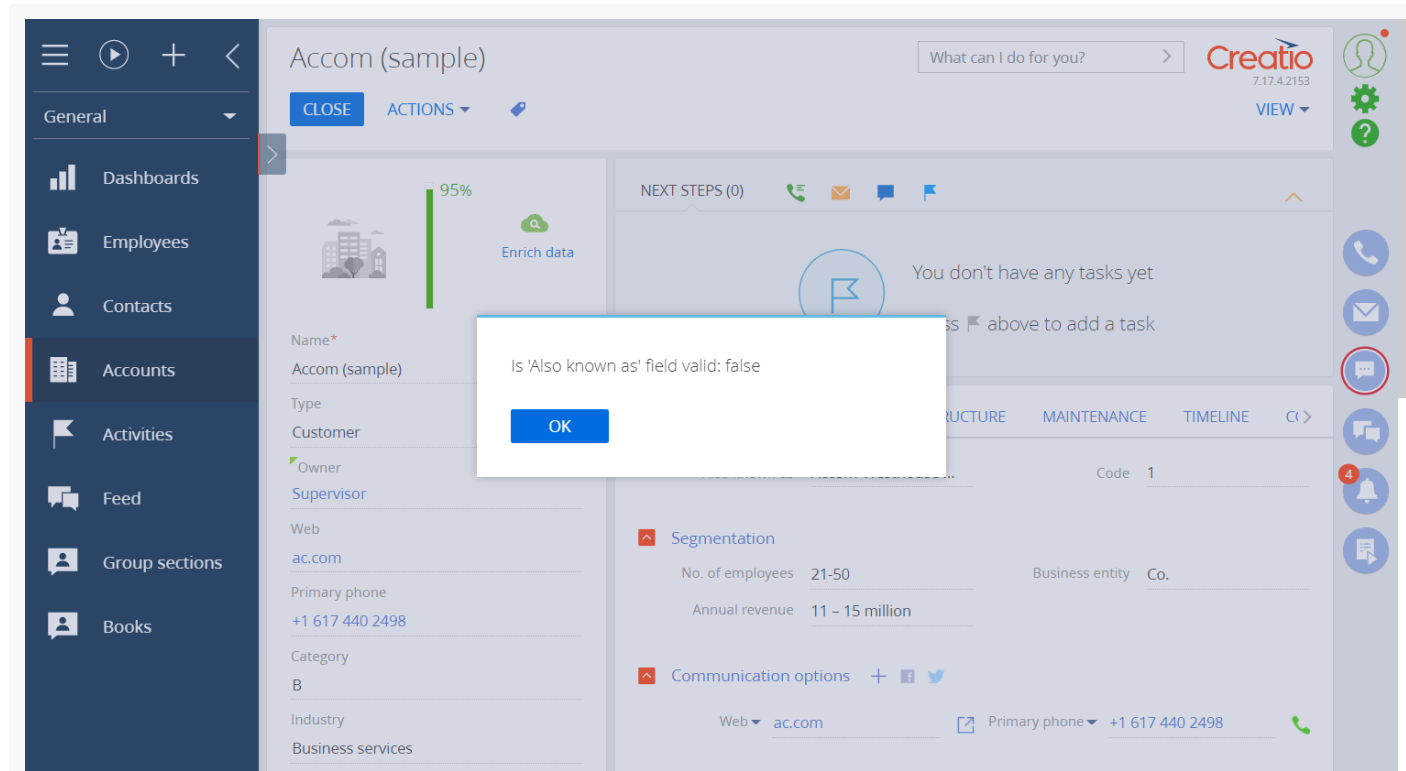
define("AccountPageV2", ["LettersOnlyValidator"], function(LettersOnlyValidator) {
    return {
        entitySchemaName: "Account",
        methods: {
            // Метод валидации.
            validateMethod: function() {
                // Определение правильности заполнения колонки AlternativeName.
                var res = LettersOnlyValidator.isAcceptable(this.get("AlternativeName"));
                // Вывод результата пользователю.
                Terrasoft.showInformation("Is 'Also known as' field valid: " + res);
            },
            // Переопределение метода родительской схемы, вызываемого при сохранении записи.
            save: function() {
                // Вызов метода валидации.
                this.validateMethod();
                // Вызов базовой функциональности.
                this.callParent(arguments);
            }
        },
        diff: /**SCHEMA_DIFF*/ [] /**SCHEMA_DIFF*/
    };
});

```

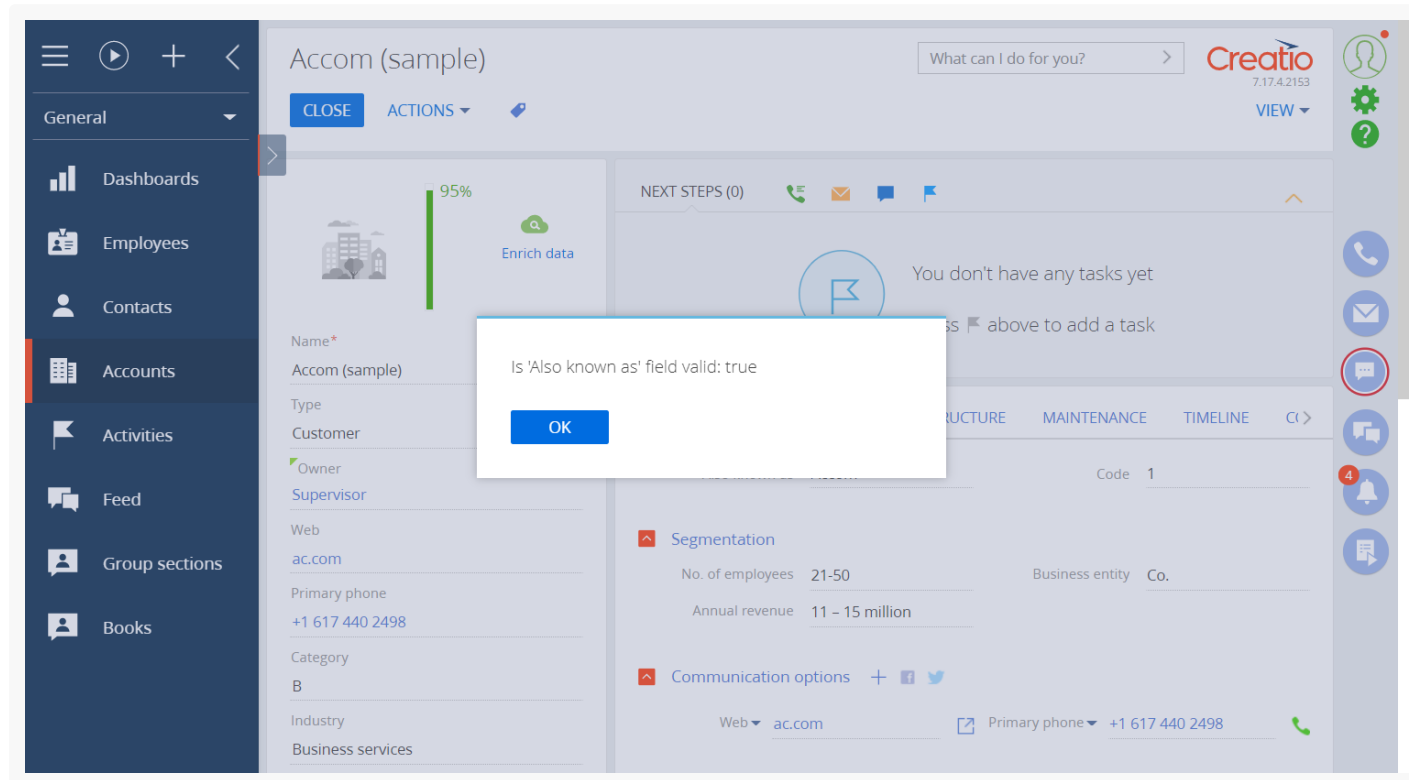
6. Сохраните файл с исходным кодом схемы и обновите страницу контрагента.

При сохранении записи будет выполняться [валидация](#) и отображаться соответствующее сообщение.

Сообщение о некорректно заполненном поле



Сообщение о корректно заполненном поле



Создать Angular-компонент для использования в Creatio

 Сложный

Для встраивания Angular-компонентов в приложение Creatio используется функциональность Angular Elements. **Angular Elements** — это `npm`-пакет, который позволяет упаковывать Angular-компоненты в Custom Elements и определять новые HTML-элементы со стандартным поведением (Custom Elements является частью стандарта Web-Components).

Создание пользовательского Angular-компонента

1. Настроить окружение для разработки компонентов средствами Angular CLI

Для этого установите:

1. [Node.js® и npm package manager](#).
2. Angular CLI.

Чтобы установить Angular CLI выполните в системной консоли команду:

Установка Angular CLI

```
npm install -g @angular/cli
```

Пример установки Angular CLI версии 8

```
npm install -g @angular/cli@8
```

2. Создать Angular приложение

Выполните в консоли команду `ng new` и укажите имя приложения, например `angular-element-test`.

Создание Angular приложения

```
ng new angular-element-test --style=scss
```

3. Установить пакет Angular Elements

Из папки приложения, созданного на предыдущем шаге, выполните в консоли команду.

Установка пакета Angular Elements

```
ng add @angular/elements
```

4. Создать компонент Angular

Чтобы создать компонент выполните в консоли команду.

Создание компонента Angular

```
ng g c angular-element
```

5. Зарегистрировать компонент как Custom Element

Чтобы настроить трансформацию компонента в пользовательский HTML-элемент, необходимо внести изменения в файл `app.module.ts`:

1. Добавьте импорт модуля `createCustomElement`.
2. В модуле в секции `entryComponents` укажите имя компонента.

3. В методе `ngDoBootstrap` зарегистрируйте компонент под HTML-тегом.

`app.module.ts`

```
import { BrowserModule } from "@angular/platform-browser";
import { NgModule, DoBootstrap, Injector, ApplicationRef } from "@angular/core";
import { createCustomElement } from "@angular/elements";
import { AppComponent } from "../app.component";
@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule],
  entryComponents: [AngularElementComponent]
})
export class AppModule implements DoBootstrap {
  constructor(private injector: Injector) {
  }
  ngDoBootstrap(appRef: ApplicationRef): void {
    const el = createCustomElement(AngularElementComponent, { injector: this._injector });
    customElements.define('angular-element-component', el);
  }
}
```

6. Выполнить сборку приложения

1. При сборке проекта сгенерируются несколько *.js-файлов. Для простоты дальнейшего использования веб-компонента в Creatio, созданные после сборки файлы рекомендуется поставлять в одном файле. Для этого необходимо в корне приложения создать скрипт `build.js`.

Пример `build.js`

```
const fs = require('fs-extra');
const concat = require('concat');
const componentPath = './dist/angular-element-test/angular-element-component.js';

(async function build() {
  const files = [
    './dist/angular-element-test/runtime.js',
    './dist/angular-element-test/polyfills.js',
    './dist/angular-element-test/main.js',
    './tools/lodash-fix.js',
  ].filter((x) => fs.pathExistsSync(x));
  await fs.ensureFile(componentPath);
  await concat(files, componentPath);
})();
```

Если в веб-компоненте используется библиотека `lodash`, то для ее работы в Creatio необходимо `main.js` (и при необходимости `styles.js`) объединять со скриптом, устраняющим конфликты по `lodash`. Для этого в корне Angular-проекта создаем папку `tools` и файл `lodash-fix.js`.

```
lodash-fix.js
```

```
window._.noConflict();
```

Важно. Если Вы не используете библиотеку `lodash`, то файл `lodash-fix.js` создавать не нужно и строку `'./tools/lodash-fix.js'` из массива `files` необходимо убрать.

Дополнительно для выполнения скрипта в `build.js` необходимо установить в проекте пакеты `concat` и `fs-extra` как dev-dependency. Для этого выполните в командной строке команды:

Установка дополнительных пакетов

```
npm i concat -D
npm i fs-extra -D
```

По умолчанию для созданного приложения могут быть установлены настройки файла `browserslist`, которые создают сразу несколько сборок для браузеров, которые поддерживают ES2015, и для тех, которым нужен ES5. Для данного примера мы собираем Angular элемент для современных браузеров.

Пример `browserslist`

```
# This file is used by the build system to adjust CSS and JS output to support the specified
# For additional information regarding the format and rule options, please see:
# https://github.com/browserslist/browserslist#queries
```

```
# You can see what browsers were selected by your queries by running:
# npx browserslist
```

```
last 1 Chrome version
last 1 Firefox version
last 2 Edge major versions
last 2 Safari major versions
last 2 iOS major versions
Firefox ESR
not IE 11
```

Важно. Если Вам необходимо поставлять веб-компонент в браузеры, которые не поддерживают ES2015, нужно либо править массив файлов в `build.js`, либо изменить `target` в `tsconfig.json` (`target: "es5"`). Внимательно проверяйте названия файлов после сборки в папке `dist`. Если они не совпадают с названиями в массиве `build.js`, их нужно изменить в файле.

- Добавьте в `package.json` команды, которые отвечают за сборку элемента. В результате их выполнения, вся бизнес логика помещается в один файл `angular-element-component.js`, с которым мы будем работать далее.

`package.json`

```
....
"build-ng-element": "ng build --output-hashing none && node build.js",
"build-ng-element:prod": "ng build --prod --output-hashing none && node build.js",
...
```

Важно. Рекомендуем при разработке, выполнять сборку приложения без параметра `--prod`.

Подключение Custom Element в Creatio

Созданный в результате сборки файл `angular-element-component.js` необходимо встроить в пакет Creatio как [файловый контент](#).

1. Разместить файл в статическом контенте пакета

Для этого скопируйте файл в папку `Название пользовательского пакета\Files\src\js`, например, `MyPackage\Files\src\js`.

2. Встроить билд в Creatio

Для этого необходимо в файле `bootstrap.js` (пакета Creatio, куда Вы хотите загрузить веб-компонент) настроить конфиг с указанием пути к билду.

Настройка конфига

```
(function() {
  require.config({
    paths: {
      "angular-element-component": Terrasoft.getFileContentUrl("MyPackageName", "src/js/ar
    }
  });
})();
```

Для загрузки `bootstrap` укажите путь к данному файлу. Для этого создайте `descriptor.json` в `Название пользовательского пакета\Files`.

descriptor.json

```
{
  "bootstraps": [
    "src/js/bootstrap.js"
  ]
}
```

Выполните загрузку из файловой системы и компиляцию.

3. Выполнить загрузку компонента в необходимой схеме/модуле

Создайте в пакете схему или модуль, в котором должен быть использован созданный пользовательский элемент, и выполните его загрузку в блоке подключения зависимостей модуля.

Выполнение загрузки компонента

```
define("MyModuleName", ["angular-element-component"], function() {
```

4. Создать HTML-элемент и добавить его в модель DOM

Пример добавления пользовательского элемента `angular-element-component` в модель DOM ...

```
/**
 * @inheritDoc Terrasoft.BaseModule#render
 * @override
 */
render: function(renderTo) {
  this.callParent(arguments);
  const component = document.createElement("angular-element-component");
  component.setAttribute("id", this.id);
  renderTo.appendChild(component);
}
```

Работа с данными

Передача данных в Angular-компонент выполняется через публичные свойства/поля, помеченные декоратором `@Input`.

Важно. Описанные в camelCase свойства без указания в декораторе явного имени будут переведены в HTML-атрибуты в kebab-case.

Пример создания свойства компонента (`app.component.ts`)

```
@Input('value')
public set value(value: string) {
    this._value = value;
}
```

Пример передачи данных в компонент (`CustomModule.js`)

```
/**
 * @inheritDoc Terrasoft.BaseModule#render
 * @override
 */
render: function(renderTo) {
    this.callParent(arguments);
    const component = document.createElement("angular-element-component");
    component.setAttribute("value", 'Hello');
    renderTo.appendChild(component);
}
```

Получение данных от компонента реализовано через механизм событий. Для этого необходимо публичное поле (тип `EventEmitter<T>`) пометить декоратором `@Output` . Для инициализации события необходимо у поля вызвать метод `emit(T)` и передать необходимые данные.

Пример реализации события в компоненте (`app.component.ts`)

```
/**
 * Emits btn click.
 */
@Output() btnClicked = new EventEmitter<any>();

/**
 * Handles btn click.
 * @param eventData - Event data.
 */
public onBtnClick(eventData: any) {
    this.btnClicked.emit(eventData);
}
```

Добавьте кнопку в `angular-element.component.html`.

Пример добавления кнопки в `angular-element.component.html`

```
<button (click)="onBtnClick()">Click me</button>
```

Пример обработки события в Creatio (`CustomModule.js`)

```
/**
 * @inheritDoc Terrasoft.Component#initDomEvents
 * @override
 */
initDomEvents: function() {
  this.callParent(arguments);
  const el = this.component;
  if (el) {
    el.on("itemClick", this.onItemClickHandler, this);
  }
}
```

Использование Shadow DOM

Некоторые компоненты, созданные с помощью Angular и встроенные в Creatio могут быть сконфигурированы так, чтобы реализация компонента была закрыта от внешнего окружения так называемым Shadow DOM.

Shadow DOM — это механизм инкапсуляции компонентов внутри DOM. Благодаря ему, в компоненте есть собственное «теневое» DOM-дерево, к которому нельзя просто так обратиться из главного документа, у него могут быть изолированные CSS-правила и т. д.

Для использования Shadow DOM необходимо в декоратор компонента добавить свойство

```
encapsulation: ViewEncapsulation.ShadowDom.
```

`angular-element.component.ts`

```
import { Component, OnInit, ViewEncapsulation } from '@angular/core';

@Component({
  selector: 'angular-element-component',
  templateUrl: './angular-element-component.html',
  styleUrls: [ './angular-element-component.scss' ],
  encapsulation: ViewEncapsulation.ShadowDom,
})
export class AngularElementComponent implements OnInit {
}
```

Создание Acceptance Tests для Shadow DOM

Shadow DOM создает проблему для тестирования компонентов в приложении с помощью приемочных cucumber тестов. К компонентам внутри Shadow DOM нельзя обратиться через стандартные селекторы из корневого документа.

Для этого необходимо использовать `shadow root` как корневой документ и через него обращаться к элементам компонента.

Shadow root — корневая нода компонента внутри Shadow DOM.

Shadow host — нода компонента, внутри которой размещается Shadow DOM.

В классе `BPMonline.BaseItem` реализованы базовые методы по работе с Shadow DOM.

Важно. В большинстве методов необходимо передавать селектор компонента, в котором находится Shadow DOM — `shadow host`.

Метод	Описание
<code>clickShadowItem</code>	Нажать на элемент внутри Shadow DOM компонента.
<code>getShadowRootElement</code>	По заданному css-селектору Angular компонента возвращает его <code>shadow root</code> , который можно использовать для дальнейших выборов элементов.
<code>getShadowWebElement</code>	Возвращает экземпляр элемента внутри Shadow DOM по заданному css-селектору. В зависимости от параметра <code>waitForVisible</code> ожидает его появления либо нет.
<code>getShadowWebElements</code>	Возвращает экземпляры элементов внутри Shadow DOM по заданному css-селектору.
<code>mouseOverShadowItem</code>	Навести курсор на элемент внутри Shadow DOM.
<code>waitForShadowItem</code>	Ожидает появления элемента внутри Shadow DOM компонента и возвращает его экземпляр.
<code>waitForShadowItemExist</code>	Ожидает появления элемента внутри Shadow DOM компонента.
<code>waitForShadowItemHide</code>	Ожидает скрытие элемента внутри Shadow DOM компонента.

К сведению. Примеры использования методов можно найти в классе `BPMonline.pages.ForecastTabUIV2`.

