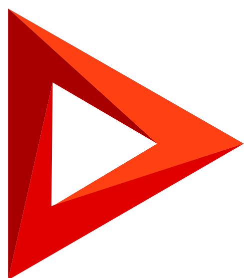
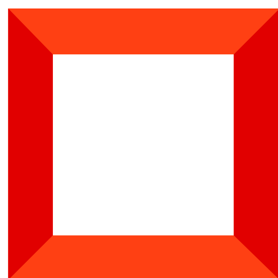
Creatio Academy

Packages

Packages file content

Version 7.17

This documentation is provided under restrictions on use and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this documentation, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

© 2023 Creatio. All rights reserved.

Table of Contents

Packages file content **4**
Storage structure of package file content 4
Package bootstrap files 6
File content versioning 6
Auxiliary file generation 7
Static file content pre-generation 8
File content generation 8
Compatibility with the file system development mode 10
Transfer changes between environments 11
Localize the file content using configuration resources **11**
1. Create a module with localizable resources 11
2. Import the module with localizable resources 11
Localize the file content using the i18n plug-in **12**
1. Add the plug-in 12
2. Create a directory with localizable resources 12
3. Create the culture directories 13
4. Add files with localizable resources 13
5. Modify the bootstrap.js file 14
6. Use the resources in a client module 14
Use TypeScript when developing custom functions **15**
1. Install TypeScript 15
2. Switch to the file system development mode 16
3. Establish the structure of the file content storage 17
4. Code the validation in TypeScript 18
5. Compile the TypeScript source code to JavaScript source code 18
6. Generate the auxiliary files 20
7. Verify the example implementation results 21
Create an Angular component for Creatio **24**
Create a custom Angular component 24
Connect the Custom Element to Creatio 28
Work with data 29
Use Shadow DOM 31

© 2023 Creatio. All rights reserved.

Packages file content

The **package file content** comprises files (*.js, *.css, images, etc.) added to custom Creatio packages. Since the file content is static, the webserver does not process it. This helps to improve Creatio performance.

The **types** of file content:

- Client content generated in real time.
- Pre-generated client content.

Special aspects of client content generated in real time:

- The pre-generation of client content is not needed.
- The calculation of package and schema hierarchy, as well as the generation of content, put a load on the CPU.
- The retrieval of package and schema hierarchy, as well as the generation of content, put a load on the database.
- The caching of client content takes up memory.

Special aspects of pre-generated client content:

- The load on the CPU is minimal.
- The pre-generation of client content is required.
- The database queries are not needed.
- The IIS tools cache the client content.

Storage structure of package file content

File content is an integral part of the package. You can pre-generate the file content in the dedicated `...\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\<PackageName>\Files` Creatio directory to improve Creatio performance and reduce the load on the database. If the IIS server receives a request, it will search for requested content in this directory and send the content to Creatio immediately. You can add any files to the package, however, Creatio will use only the files required for its client part.

We recommend **structuring** the `Files` directory as below.

Recommended `Files` directory structure

```
-PackageName
    ...
    -Files
        -src
            -js
                bootstrap.js
```

© 2023 Creatio. All rights reserved.

```
                [other *.js files]
            -css
                [*.css files]
            -less
                [*.less files]
            -img
                [image files]
            -res
                [resource files]
        descriptor.json
    ...
descriptor.json
```

`js` – the directory with JavaScript *.js source code files

`css` – the directory with *.css style files

`less` – the directory with *.less style files

`Img` – the directory with images

`res` – the directory with resource files

`descriptor.json` – the file content descriptor that stores the information about package bootstrap files

View the `descriptor.json` file structure below.

Descriptor.json file structure

```
{
    "bootstraps": [
        ... // The string array that contains relative paths to bootstrap files.
    ]
}
```

Descriptor.json file example

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

To add file content to a package, place the file in the corresponding subdirectory of the `Files` package directory.

© 2023 Creatio. All rights reserved.

Package bootstrap files

Package **bootstrap files** are *.js files that handle the loading of client configuration logic. The file structure may vary.

Bootstrap.js file structure

```
(function() {
    require.config({
        paths: {
            "Module name": "Link to the file content",
            ...
        }
    });
})();
```

Bootstrap.js file example

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

Creatio loads bootstrap files asynchronously after loading the core, but before loading the configuration. Creatio generates `_FileContentBootstraps.js` auxiliary file in the static content directory to ensure the bootstrap files are loaded correctly. The auxiliary file contains information about bootstrap files of all packages.

`_FileContentBootstraps.js` file content example

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

File content versioning

© 2023 Creatio. All rights reserved.

Creatio generates `_FileContentDescriptors.js` auxiliary file in the static file content directory to ensure the file content versioning works correctly. The auxiliary file contains information about files in the file content of all packages, displayed as a “key-value” collection. Each key (filename) corresponds to a unique hash code. This ensures the browser will receive the up-to-date file version.

`_FileContentDescriptors.js` file content example

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

Auxiliary file generation

To **generate auxiliary files** (`_FileContentBootstraps.js` and `FileContentDescriptors.js`), execute the `BuildConfiguration` operation using the WorkspaceConsole utility.

`BuildConfiguration` operation parameters

Parameter	Description
`-operation`	The name of the operation. Set the value to `BuildConfiguration` – the operation that compiles the configuration.
`-useStaticFileContent`	Enables the static content. Set the value to `false`.
`-usePackageFileContent`	Enables the package file content. Set the value to `true`.

Auxiliary files generation

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

As a result, Creatio will generate `_FileContentBootstraps.js` and `_FileContentDescriptors.js` auxiliary files in the `...\Terrasoft.WebApp\conf\content` static content directory.

Learn more about the parameters of the utility: WorkspaceConsole parameters.

© 2023 Creatio. All rights reserved.

Static file content pre-generation

Creatio generates the file content in the `.\Terrasoft.WebApp\conf` special directory. The directory contains *.js files with schema source code, *.css style files, *.js resource files of all Creatio cultures, as well as images.

Attention. The user of the IIS pool that runs Creatio must have modification permissions for static content generation in the `.\Terrasoft.WebApp\conf` directory to work properly. Set up permissions in the Handler Mappings section on the server level. Learn more: [Set up Creatio application server on IIS](#).

Set the name of the IIS pool user in the [*Identity*] property. Click [*Advanced Settings*] on the [*Application Pools*] tab of the IIS manager to open this property.

The following **events** trigger static file content generation or re-generation:

- Saving a schema in the Client Schema Designer and Client Objects Designer.
- Saving in the Section Wizard and Detail Wizard.
- Installing and deleting applications via Marketplace and *.zip archive.
- Applying translations.
- Running [*Compile*] and [*Compile all*] actions in the [*Configuration*] section.
  Run these actions after you **delete schemas or packages** from the [*Configuration*] section.

  Run the [*Compile all*] action after you **install or update a package** from [SVN](#) version control system.

Note. The [*Compile all*] action fully re-generates the static file content. Other Creatio actions re-generate only the modified schemas.

File content generation

To **generate the file content**, execute the `BuildConfiguration` operation using the [WorkspaceConsole utility](#).

© 2023 Creatio. All rights reserved.

`BuildConfiguration` operation parameters

Parameter	Description
`-workspaceName`	The workspace name. Set to `Default` by default.
`-destinationPath`	The directory to which to generate the static content.
`-webApplicationPath`	Path to the web application from which to read the information about the connection to the database. Optional. If you do not specify a value, Creatio will establish a connection to the database specified in the connection string of the `Terrasoft.Tools.WorkspaceConsole.config` file. If you specify a value, Creatio will establish the connection to the database from the `ConnectionStrings.config` file of the web application.
`-force`	Optional. Set to `false` by default. If you leave the value as is, Creatio will generate the file contents only for modified schemas. If you set the value to `true`, Creatio will generate the content for all schemas.

Generate the file content (option 1)

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

Generate the file content (option 2)

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

Client content generation when adding a new culture

After you **add new cultures** in the Creatio UI, run the [*Compile all*] action in the [*Configuration*] section.

Attention. If you cannot log in to Creatio after adding a new culture, open the [*Configuration*] section via the `http://[Creatio path]/0/dev` URL.

Retrieve the image URL

The browser requests images in the client part of Creatio with the URL specified in the `src` attribute of the `img` HTML element. Creatio generates this URL using the `Terrasoft.ImagesUrlBuilder (imagurlbuilder.js)` module with the `getUrl(config)` public method for retrieving the image URL. This method expects the `config` JavaScript

© 2023 Creatio. All rights reserved.

configuration object that contains the parameters object in the `params` property. Creatio generates the image URL to add to the page based on this property.

`params` object structure:

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

`schemaName` – the schema name (string)

`resourceItemName` – the image name in Creatio (string)

`Hash` – the image hash (string)

`resourceItemExtension` – the image file extension. For example, ".png."

View the example that generates the parameters configuration object needed to retrieve the static image URL below

Generate the parameters configuration object

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

Compatibility with the file system development mode

It is not possible to retrieve client content from pre-generated files in the file system development mode. To ensure the correct operation of the file system development mode, **disable the retrieval of static client content** from the file system. To disable this feature, set the `UseStaticFileContent` flag in the `Web.config` file to `false`.

Disable the retrieval of static file content from the file system

© 2023 Creatio. All rights reserved.

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

Transfer changes between environments

File content is an integral part of the package. The content is stored in the version control system repository along with other package content. You can **transfer the file content to another environment**:

- We recommend using the SVN version control system to transfer changes to the development environment.
- We recommend using the export and import mechanism in Creatio IDE to transfer changes to pre-production and production environments.

Attention. Creatio will add the `Files` directory when installing packages only if there are files to place in it. If there is no such directory, create it manually to begin development.

Localize the file content using configuration resources

Advanced

1. Create a module with localizable resources

If you want to **translate resources** to other languages, we recommend using a separate module with localizable resources. Create the module using the built-in Creatio tools in the [*Configuration*] section.

Module with localizable resources

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

2. Import the module with localizable resources

To **access the module with localizable resources** from the client module, import the module with localizable resources to the client module as a dependency.

Connect the needed resources to the module

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

© 2023 Creatio. All rights reserved.

```
});
```

Localize the file content using the i18n plug-in

Advanced

i18n is a plug-in for an AMD loader (for example, RequireJS). Use it to load the localizable string resources. Download the plug-in source code from the GitHub repository.

1. Add the plug-in

Place the plug-in in the `..\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\MyPackage1\content\js\i18n.js` directory with *.js source code files.

Where `MyPackage1` is the development directory of the `MyPackage1` package.

2. Create a directory with localizable resources

Create the `..\MyPackage1\content\nls` directory and place the *.js files with localizable resources there.

You can add one or more resource files. File names can be arbitrary. The files contain AMD modules, which, in turn, contain objects.

The **structure** of AMD module objects:

- `Root` **field**.

  The “root” field contains the “key-value” collection, where the “key” is the name of a localizable string and the “value” is the localizable string in the default language. The value will be used if the requested language is not supported.
- **Culture fields**.

  The field names must match the standard codes of supported cultures. For example, `en-US`, `de-DE`. The field values must be boolean: `true` means the culture is toggled on, `false` means the culture is toggled off.

View the `..\MyPackage1\content\js\nls\ContactSectionV2Resources.js` file example below.

`ContactSectionV2Resources.js` file example

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

© 2023 Creatio. All rights reserved.

3. Create the culture directories

Create the culture directories in the `..\MyPackage1\content\nls` directory. Set the code of the culture as the name of the directory where the relevant localization will be stored. For example, `en-US`, `de-DE`).

View the structure of the `MyPackage1` directory with German and English cultures below.

Structure of the `MyPackage1` directory

```
content
    nls
        en-US
        de-DE
```

4. Add files with localizable resources

Place the same set of *.js files with localizable resources in each localization directory as the set of files you placed in the `..\MyPackage1\content\nls` root directory. The files contain AMD modules whose objects are “key-value” collections, where the “key” is the name of a localizable string and the “value” is a string in the language that matches the directory name (the culture code).

For example, if you support only German and English cultures, create two `ContactSectionV2Resources.js` files.

ContactSectionV2Resources.js file for English culture

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

ContactSectionV2Resources.js file for German culture

```
define({
    "FileContentActionDescr": "Erste aktion des Dateiinhalts"
});
```

Since the translation of the “`FileContentActionDescr2`” string is not specified for the German culture, the “`File content second action (Default)`” default value will be used.

© 2023 Creatio. All rights reserved.

5. Modify the `bootstrap.js` file

To modify the `bootstrap.js` file:

1. Connect the `i18n` plug-in by specifying its name as the `i18n` alias in the RequireJS path configuration and specifying the corresponding path to the plug-in in the `paths` property.
2. Specify the user's current culture for the plug-in. To do this, assign the object with the `i18n` property to the `config` property of the RequireJS library's configuration object. In turn, assign the object with the `locale` property and the value retrieved from the `Terrasoft.currentUserCultureName` global variable (the code of the current culture) to the object with the `i18n` property.
3. Set the corresponding aliases and paths in the RequireJS path configuration for each file with the localization resources. The alias must be a URL path relative to the `nls` directory.

`..\MyPackage1\content\js\bootstrap.js` file example

```
(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/de-DE/ContactSectionV2Resources": Terrasoft.getFileContentUrl("MyPackage1", "co
            "nls/en-US/ContactSectionV2Resources":  Terrasoft.getFileContentUrl("MyPackage1", "c
        },
        config: {
            i18n: {
                locale: Terrasoft.currentUserCultureName
            }
        }
    });
})();
```

6. Use the resources in a client module

To use the resources in a client module, specify the resource module with the “`i18n!`” prefix in the dependency array.

View an example that uses the `FileContentActionDescr` localizable string as the heading for a new action in the [*Contacts*] section below.

`..\MyPackage1\content\js\bootstrap.js` file example

```
define("MyPackage1-ContactSectionV2", ["i18n!nls/ContactSectionV2Resources",
```

© 2023 Creatio. All rights reserved.

```
    "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*/
    }
});
```

Use TypeScript when developing custom functions

Advanced

You can use languages that can be compiled to JavaScript when developing the file contents of the custom functionality. For example, **TypeScript**. Learn more about TypeScript in documentation.

Example. When a user saves the account record, validate the value of the [*Also known as*] field and display the message with the validation results. The field must contain only letters. Code the validation logic in TypeScript.

1. Install TypeScript

One way to install TypeScript is to use the **NPM package manager** for `Node.js`.

To **install TypeScript**:

1. Make sure your OS has the `Node.js` runtime environment.
   Download the installer.
2. Run the following command at the Windows console:

TypeScript installation command

© 2023 Creatio. All rights reserved.

```
npm install -g typescript
```

2. Switch to the file system development mode

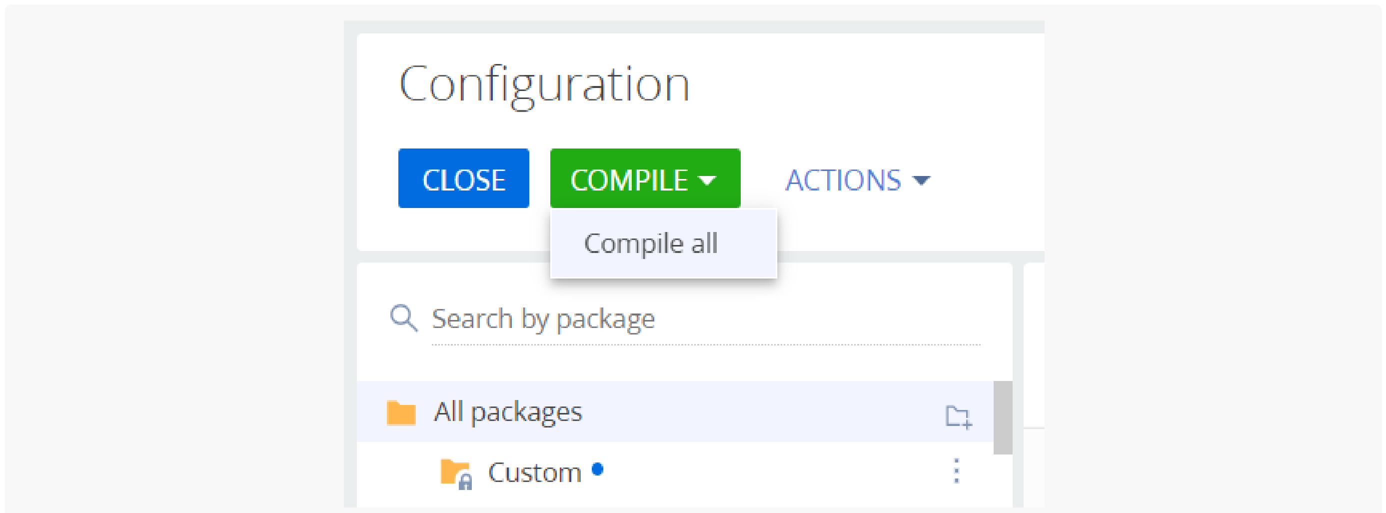
To **set up Creatio for file system development**:

1. **Enable the file system development mode.**
   Set the value of the `fileDesignMode` element's `enabled` attribute to `true` in the `Web.config` file in Creatio root directory.
2. **Disable the retrieval of static file content from the file system.**
   Set the `UseStaticFileContent` flag to `false` in the `Web.config` file in Creatio root directory.

`Web.config`

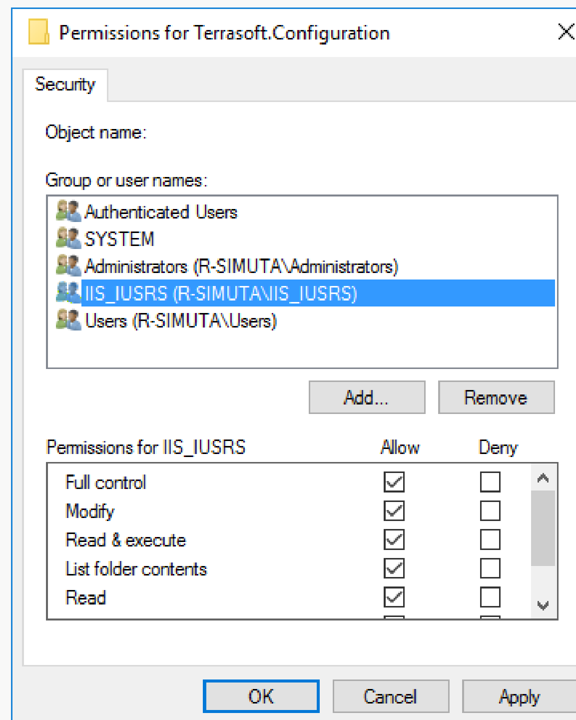
```
<filedesignmode enabled="true"/>
...
<add key="UseStaticFileContent" value="false"/>
```

3. **Compile Creatio.**
   Run the [*Compile all*] action in the [*Configuration*] section.



4. **Grant IIS access to the configuration directory.**
   To ensure Creatio works with the configuration project correctly, grant the OS user on whose behalf you run the IIS application pool full access to the `[Path to Creatio]\Terrasoft.WebApp\Terrasoft.Configuration` directory. Usually, this is a built-in `IIS_IUSRS` user.

© 2023 Creatio. All rights reserved.



Learn more about the file system development mode: External IDEs. Visual Studio.

3. Establish the structure of the file content storage

To **establish the structure of the file content storage**:

1. Create the `Files` directory in the custom package downloaded to the file system.
2. Create the `src` subdirectory in the `Files` directory.
3. Create the `js` subdirectory in the `src` directory.
4. Create the `descriptor.json` file in the `Files` directory.

descriptor.json

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

5. Create the `bootstrap.js` file in the `Files\src\js` directory.

bootstrap.js

```
(function() {
    require.config({
```

© 2023 Creatio. All rights reserved.

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

4. Code the validation in TypeScript

To **code the validation in TypeScript**:

1. Create the `Validation.ts` file in the `Files\src\js` directory and declare the `StringValidator` interface in the file.

Validation.ts

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

2. Create the `LettersOnlyValidator.ts` file in the `Files\src\js` directory. Declare the `LettersOnlyValidator` class that implements the `StringValidator` interface in the file.

LettersOnlyValidator.ts

```
// Import the module that implements the StringValidator interface.
import StringValidator = require("Validation");

// The new class must belong to the Terrasoft namespace (module).
module Terrasoft {
    // Declare the value validation class.
    export class LettersOnlyValidator implements StringValidator {
        // The regular expression that only accepts letters.
        lettersRegexp: any = /^[A-Za-z]+$/;
        // The validating method.
        isAcceptable(s: string) {
            return !Ext.isEmpty(s) && this.lettersRegexp.test(s);
        }
    }
}
// Create and export a class instance for require.
export = new Terrasoft.LettersOnlyValidator();
```

5. Compile the TypeScript source code to JavaScript source

© 2023 Creatio. All rights reserved.

code

To **compile the TypeScript source code to JavaScript source code**:

1. Add the `tsconfig.json` configuration file to the `Files\src\js` directory to set up the compilation.

tsconfig.json

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

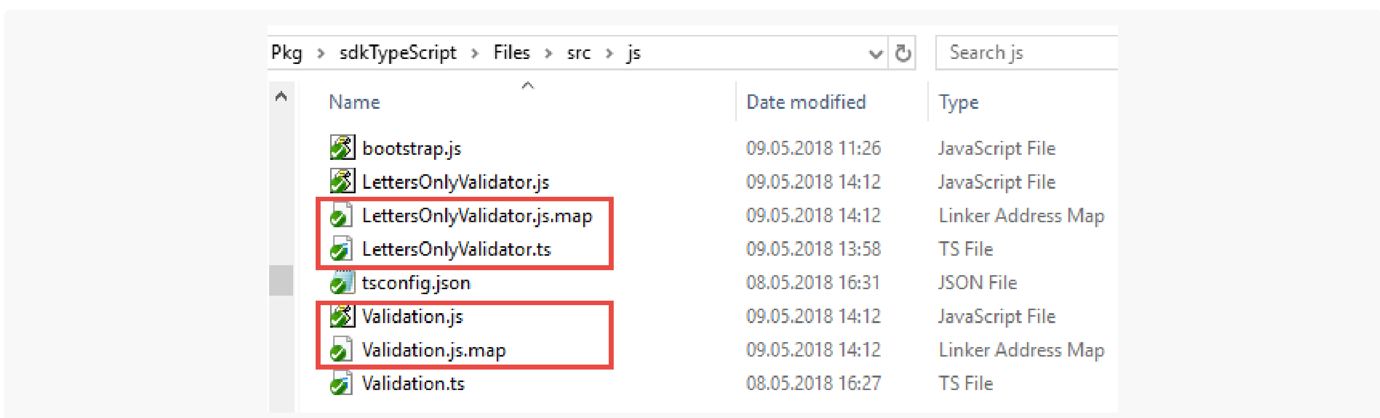
2. Go to the `Files\src\js` directory via the Windows console and run the `tsc` command.

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.16299.371]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\creatio\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\sdkTypeScript\Files\src\js>tsc

C:\creatio\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\sdkTypeScript\Files\src\js>
```

As a result, Windows will create the JavaScript version of the `Validation.ts` and `LettersOnlyValidator.ts` files, as well as the *.map files that streamline debugging in the browser, in the `Files\src\js` directory.



The contents of the automatically generated `LettersOnlyValidator.js` file to be used in Creatio.

LettersOnlyValidator.js

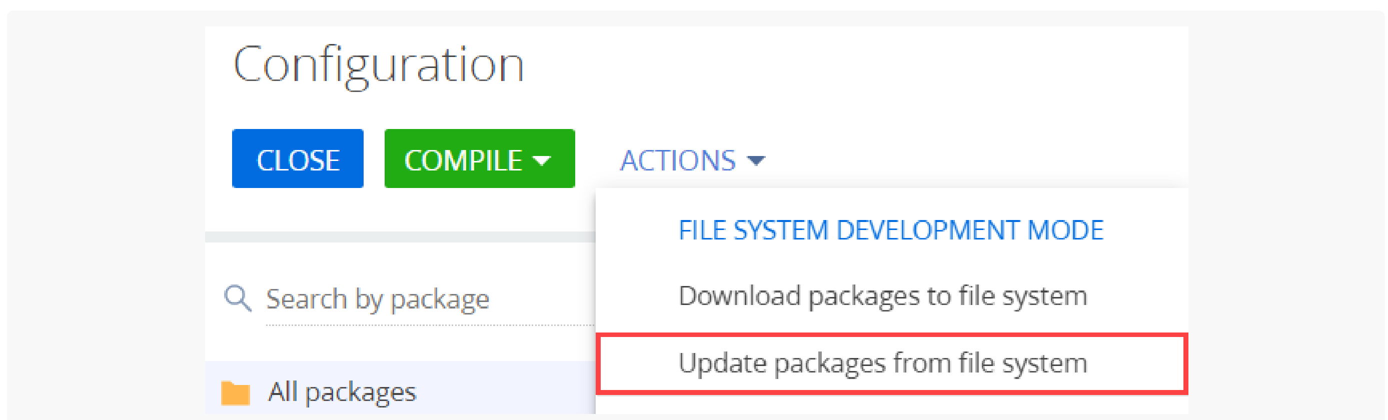
© 2023 Creatio. All rights reserved.

```
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. Generate the auxiliary files

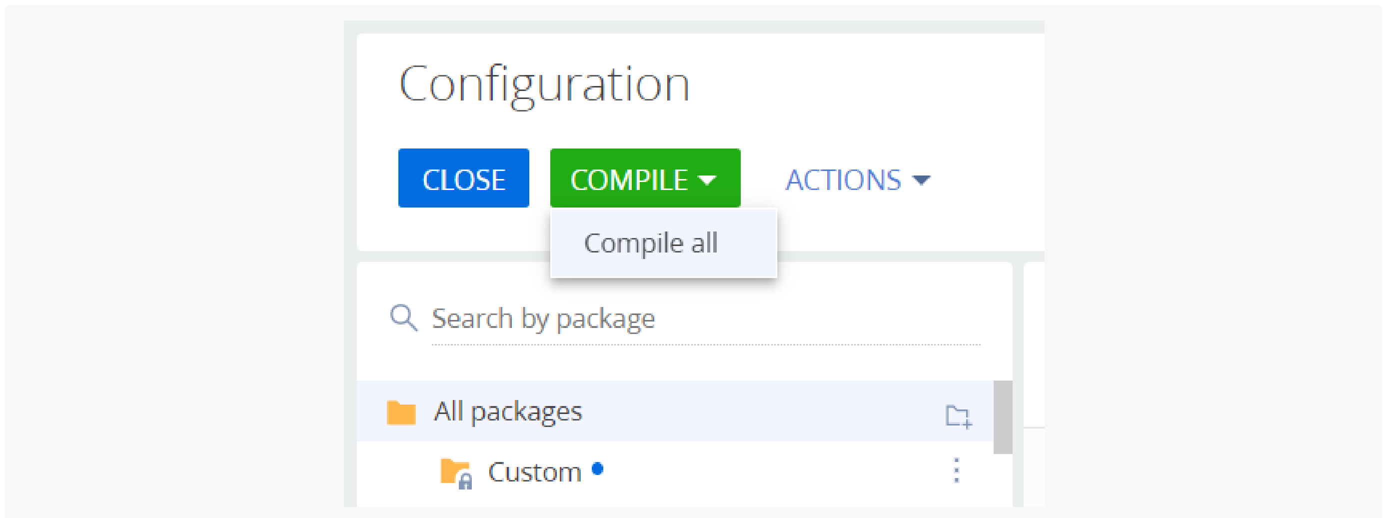
To **generate** the `_FileContentBootstraps.js` and `FileContentDescriptors.js` auxiliary files:

1. Go to the [*Configuration*] section.
2. Run the [*Update packages from file system*] action to upload packages from the file system.



3. Run the [*Compile all*] action to apply the changes to the `bootstrap.js` file.

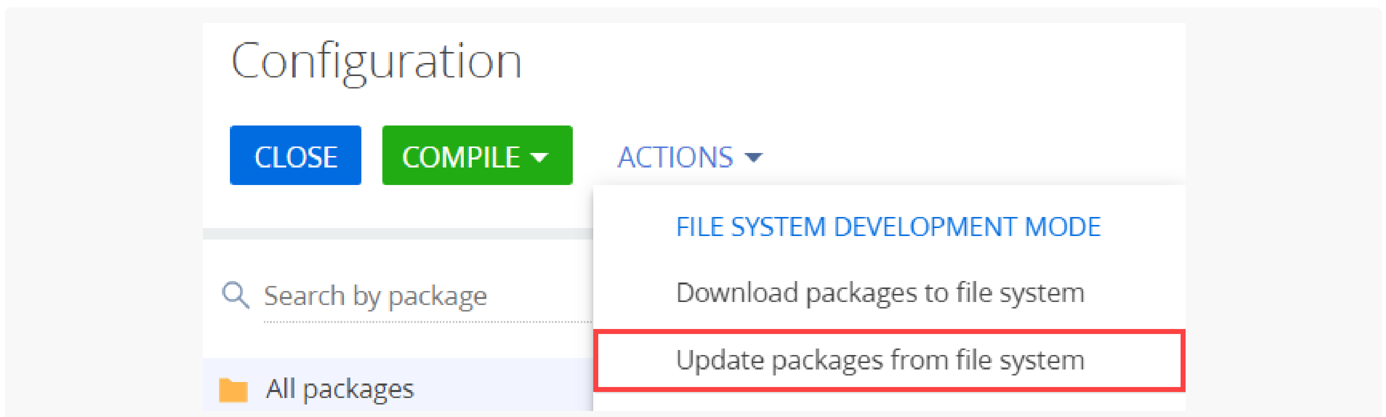
© 2023 Creatio. All rights reserved.



7. Verify the example implementation results

To **enable the validation**:

1. Go to the [*Configuration*] section.
2. Run the [*Update packages from file system*] action to upload packages from the file system.



3. Create the [replacing view model schema] for the account page.

© 2023 Creatio. All rights reserved.

Module

Code
AccountPageV2

Title *
Account edit page

Parent object *
Account edit page (AccountPageV2)

Package
sdkTypeScript

Description

CANCEL APPLY

4. Run the [*Download packages to the file system*] action to download packages to the file system.
5. Modify the `..\sdkTypeScript\Schemas\AccountPageV2\AccountPageV2.js` file in the file system.

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

```
// Declare the module and its dependencies.
define("AccountPageV2", ["LettersOnlyValidator"], function(LettersOnlyValidator) {
    return {
        entitySchemaName: "Account",
        methods: {
            // The validation method.
            validateMethod: function() {
                // Determine if the value of the AlternativeName column is valid.
                var res = LettersOnlyValidator.isAcceptable(this.get("AlternativeName"));
                // Display the results to the user.
                Terrasoft.showInformation("Is 'Also known as' field valid: " + res);
            },
            // Redefine the parent schema method called when saving the record.
            save: function() {
                // Call the validation method.
                this.validateMethod();
                // Call the base functionality.
                this.callParent(arguments);
            }
```

© 2023 Creatio. All rights reserved.

```
        },
        diff: /**SCHEMA_DIFF*/ [] /**SCHEMA_DIFF*/
    };
});
```

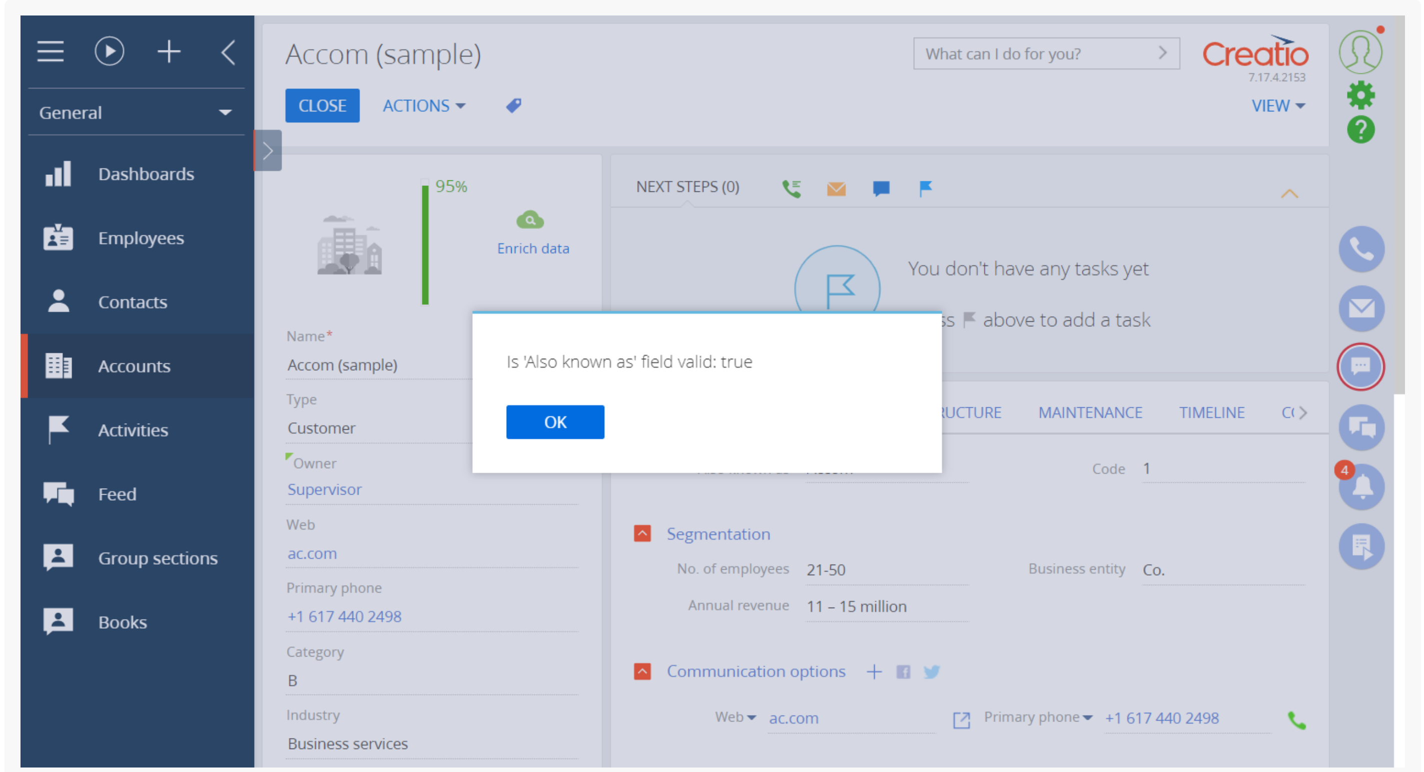
6. Save the file with the schema source code and refresh the account page.

Creatio will validate the field and display the message with the validation results when you save the record.

The field value is invalid

The field value is valid

© 2023 Creatio. All rights reserved.



Create an Angular component for Creatio

Advanced

Install Angular components in Creatio using the Angular Elements functionality. **Angular Elements** is an `npm` package that enables packing Angular components to Custom Elements and defining new HTML elements with standard behavior. Custom Elements is a part of the Web-Components standard.

Create a custom Angular component

1. Set up the Angular CLI development environment

To do this, install:

1. Node.js® and npm package manager.
2. Angular CLI.
   To install Angular CLI, run the following command at the command prompt:

```
Install Angular CLI

npm install -g @angular/cli
```

Angular CLI version 8 installation example

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

2. Create an Angular application

Run the `ng new` command at the command prompt and specify the application name. For example, `angular-element-test`.

Create an Angular application

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

3. Install the Angular Elements package

Go to the Creatio directory added on the previous step and run the following command at the command prompt.

Install the Angular Elements package

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

4. Create an Angular component

To create a component, run the following command at the command prompt.

Create an Angular component

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

5. Register the component as a Custom Element

To transform the component into a custom HTML element, modify the `app.module.ts` file:

1. Add the import of the `createCustomElement` module.
2. Specify the component name in the `entryComponents` section of the module.
3. Register the component under the HTML tag in the `ngDoBootstrap` method.

`app.module.ts`

© 2023 Creatio. All rights reserved.

```
import { BrowserModule } from "@angular/platform-browser";
import { NgModule, DoBootstrap, Injector, } from "@angular/core";
import { createCustomElement } from "@angular/elements";
import { AppComponent } from "./app.component";
import { AngularElementComponent } from './angular-element/angular-element.component';

@NgModule({
    declarations: [AppComponent, AngularElementComponent],
    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. Build the application

1. Several *.js files will be generated as part of the project build. We recommend deploying the generated files as a single file to streamline the use of the web component. To do this, create the `build.js` script in the root of Angular project.

`build.js` **example**

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

If the web component uses the lodash library, merge `main.js` (as well as `styles.js`, if necessary) with the script that resolves lodash conflicts to ensure the library's compatibility with Creatio. To do this, create the

© 2023 Creatio. All rights reserved.

`tools` directory and the `lodash-fix.js` file in the Angular project root.

`lodash-fix.js`

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

Attention. If you are not using the lodash library, do not create the `lodash-fix.js` file, rather, delete the `'./tools/lodash-fix.js'` string from the `files` array.

To execute the `build.js` script, install the `concat` and `fs-extra` packages in the project as dev-dependency. To do this, run the following commands at the command prompt:

Install additional packages

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

By default, you can set the settings of the `browserslist` file in the new application. These settings create several builds for browsers that support ES2015 and those that require ES5. For this example, build an Angular element for modern browsers.

`browserslist` **example**

```
# 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
```

Attention. If you must deploy the web component to browsers that do not support ES2015, either modify the file array in `build.js` or edit the `target` in `tsconfig.json` so that it reads `target: "es5"`. After you finish the build, review the filenames in the `dist` directory. If they do not match the names in

© 2023 Creatio. All rights reserved.

the `build.js` array, modify them in the file.

2. Add the element building commands to `package.json`. After the commands are executed, the business logic will be placed in a single `angular-element-component.js` file. Use this file going forward.

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",
...
```

Attention. We recommend building the application without the `--prod` parameter during the development.

Connect the Custom Element to Creatio

Install the `angular-element-component.js` file you built in a Creatio package as file content.

1. Place the file in the package static content

To do this, copy the file to the `Custom package name\Files\src\js` directory. For example, `MyPackage\Files\src\js`.

2. Install the build in Creatio

To do this, configure the build path in the bootstrap.js file of the package to which to upload the component.

Configure the build path

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

To upload `bootstrap`, specify the path to this file. To do this, create `descriptor.json` in `Custom package name\Files`.

descriptor.json

© 2023 Creatio. All rights reserved.

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

Upload the file from the file system and compile Creatio.

3. Load the component to the required schema/module

Create a schema or module in the package to which to load the custom element. Load the schema or module to the dependency load block of the module.

Load the component

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

4. Create an HTML element and add it to DOM

Add the `angular-element-component` custom element to Creatio page 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);
}
```

Work with data

The Angular component receives data using the public properties/fields marked with the `@Input` decorator.

Attention. The properties described in `camelCase` without the explicit name specified in the decorator will be transformed into HTML attributes in `kebab-case`.

Create the `app.component.ts` component property

© 2023 Creatio. All rights reserved.

```
private _value: string;

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

Pass data to the component(`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);
}
```

Data retrieval is implemented via the event functionality. To do this, mark the public field of the `EventEmiter<T>` type with the `@Output` decorator. To initialize an event, call the `emit(T)` field method and pass the required data.

Implement the event in the component (`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);
}
```

Add a button to `angular-element.component.html`.

Add a button to `angular-element.component.html`

© 2023 Creatio. All rights reserved.

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

Process the event in 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);
   }
}
```

Use Shadow DOM

Use Shadow DOM to block certain components created using Angular and installed in Creatio off the external environment.
The Shadow DOM mechanism encapsulates components within DOM. This mechanism adds a “shadow” DOM tree to the component, which cannot be addressed from the main document via the standard options. The tree may have isolated CSS rules, etc.

To toggle on Shadow DOM, add the `encapsulation: ViewEncapsulation.ShadowDom` property to the component decorator.

`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 {
}
```

Create Acceptance Tests for Shadow DOM

Shadow DOM creates test problems for application components using cucumber acceptance tests. It is not

© 2023 Creatio. All rights reserved.

possible to address the components within Shadow DOM from the root document via the standard selectors. Instead, use `shadow root` as the root document and address the component elements through it.

`Shadow root` – the root component node within Shadow DOM.
`Shadow host` – the component node that contains Shadow DOM.

The `BPMonline.BaseItem` class implements the base Shadow DOM operation methods.

Attention. You must pass the selector of the component that contains Shadow DOM — `shadow host` — in most methods.

Method	Description
`clickShadowItem()`	Click an element within the Shadow DOM component.
`getShadowRootElement()`	Returns the `shadow root` of the Angular component by the CSS selector. Use the shadow root to select other elements.
`getShadowWebElement()`	Returns the instance of the element within Shadow DOM by the CSS selector. Use the `waitForVisible` parameter to specify whether to wait for the instance to become visible.
`getShadowWebElements()`	Returns the instances of elements within Shadow DOM by the CSS selector.
`mouseOverShadowItem()`	Hover over the element within Shadow DOM.
`waitForShadowItem()`	Waits until the element within the Shadow DOM component becomes visible and returns its instance.
`waitForShadowItemExist()`	Waits until the element within the Shadow DOM component becomes visible.
`waitForShadowItemHide()`	Waits until the element within the Shadow DOM component becomes hidden.

Note. Review the method use examples in the `BPMonline.pages.ForecastTabUIV2` class.

© 2023 Creatio. All rights reserved.