Creatio Academy

Debugging

Version 7.18

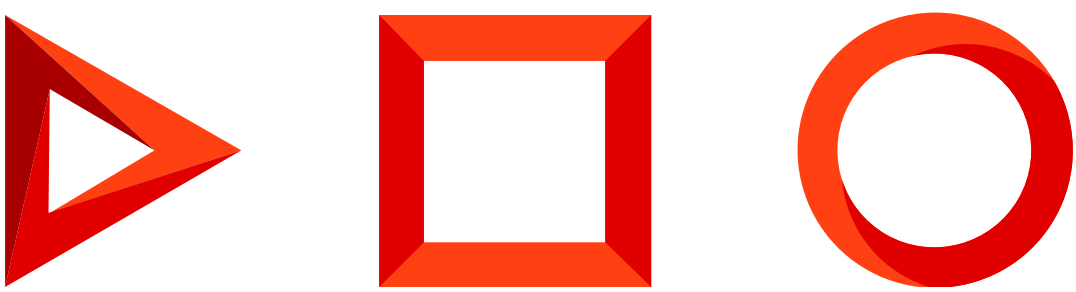


Table of Contents

Back-end debugging

Advanced

Back-end debugging is debugging of C# code schemas, for example, existing base schemas, custom configuration classes, web services, etc.

Execute back-end debugging

Debug the C# code schemas using the integrated debugging functionality of **external IDEs**, for example, Visual Studio Code.

External IDE debuggers let you execute the following **actions**:

- suspend the execution of methods
- check the values of variables
- change the values of variables

View an example that executes back-end debugging in the Visual Studio Code external IDE below.

To execute debugging in Visual Studio Code, ensure the following **requirements** are met:

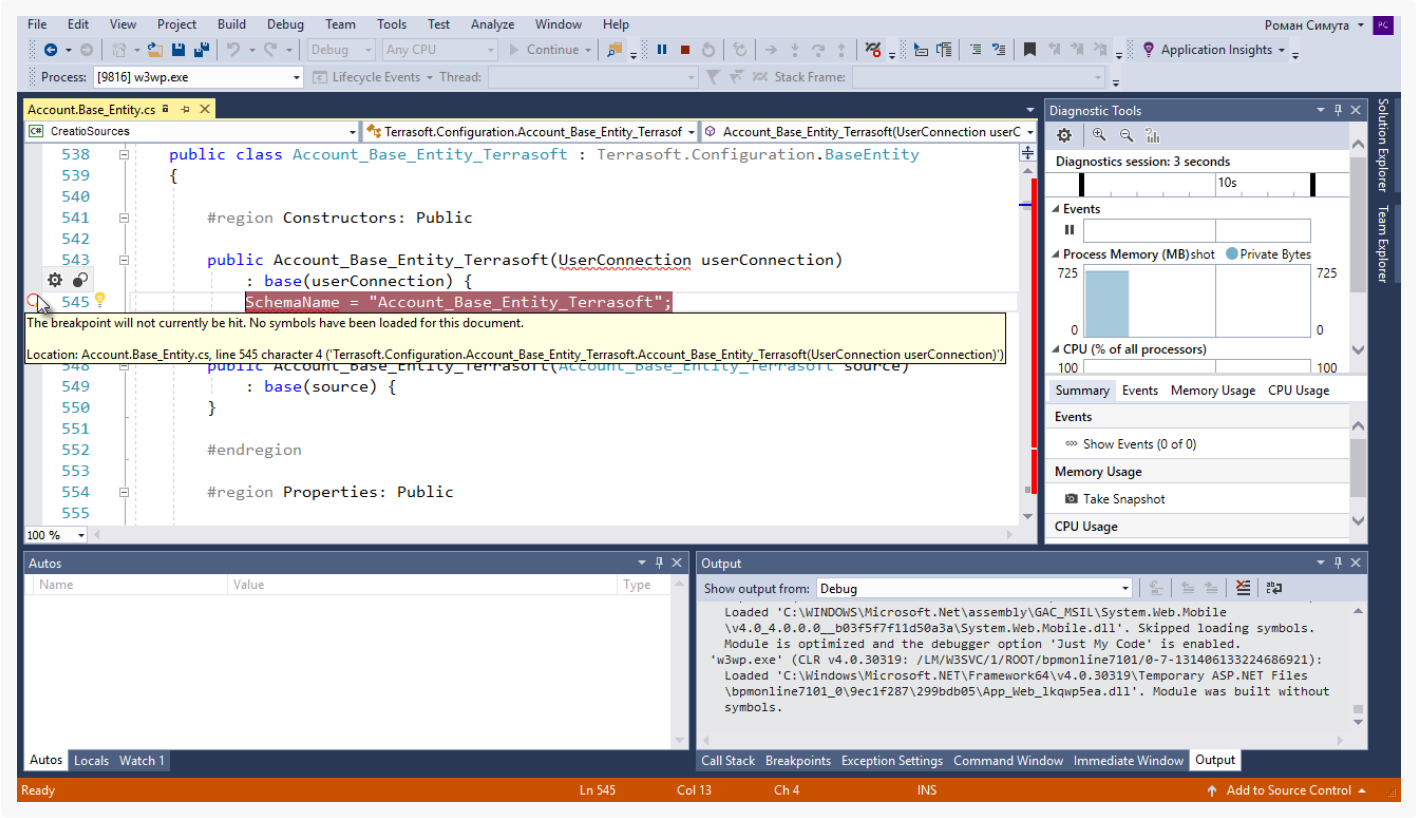
- Use Creatio on-site.
- Turn off the file system development mode.
- Select the [*Suppress JIT Optimization*] checkbox in Visual Studio Code ([*Options*] menu → [*Debugging*] item → [*General*] block). The checkbox lets you control the values of variables while debugging. Learn more about the impact of optimized and unoptimized code on debugging in the official Visual Studio documentation.

To **execute back-end debugging**:

1. Export the C#-code schemas to the local directory files.
2. Create a Visual Studio Code project for debugging.
3. Add the exported files to the Visual Studio Code project.
4. Connect the project to the IIS server workflow and execute debugging.

Solve debugging issues

Issue 1. The breakpoint is displayed as a white circle with a red border. This breakpoint does not stop the script execution. Hold the pointer over the inactive breakpoint character to view the tooltip about the issue.

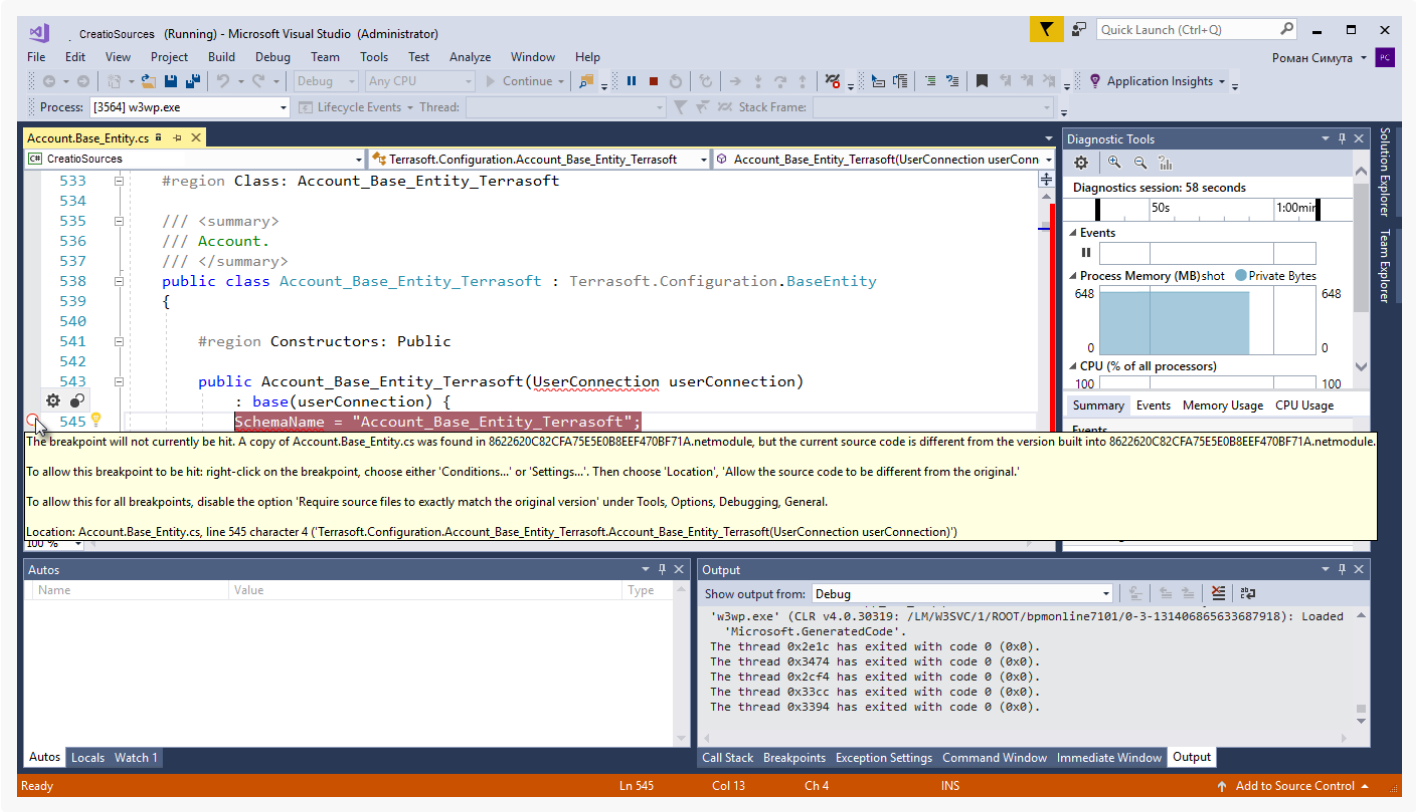


Solution:

1. Stop debugging ([*Debug*] menu → [*Stop Debugging*] item).
2. Close the file that contains the debugged source code.
3. Execute the [*Compile all*] action in the [*Configuration*] Creatio section.
4. Reconnect the project to the IIS process when compiling and reexporting the source code files.
5. Reopen the debugged source code file after the compilation finishes.

You can also recompile code without closing the file and reconnecting the project to the IIS process.

Issue 2. Message about non-uniform end-of-line characters appears after you reopen the source code file.



Solution:

1. Click [*No*] to cancel character normalization. If you click [*Yes*] to normalize the characters, the breakpoint might become inactive again.
2. Execute one of the suggested solution options from the tooltip that also specifies the cause of the issue (file version mismatch).

Debug C# code

Advanced

1. Export the C# code schemas to the local directory files

1. Modify the Creatio configuration files to debug C# code.

   a. Set the `debug` attribute of the `<compilation>` element to `true` in the `Web.config` configuration file located in the Creatio root directory.

   Web.config

   ```
   <compilation debug="true" targetFramework="4.5" />
   ```

   Save the file.

   b. Set the following values in the `Web.config` configuration file located in the `...\Terrasoft.WebApp` directory:

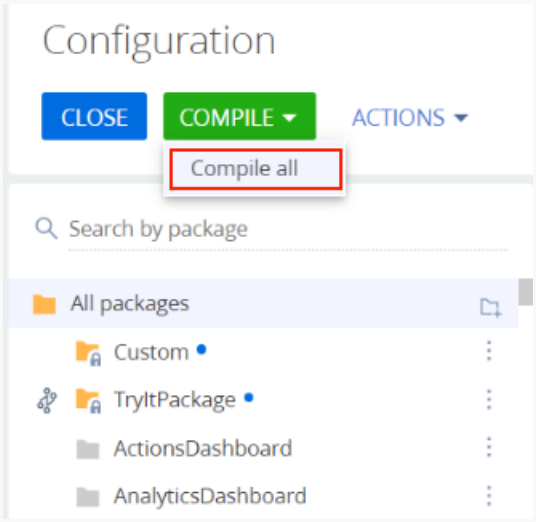
- Set `IncludeDebugInformation` to "true."
- Set `ExtractAllCompilerSources` to "true" (if you need to export all schemas while executing the [*Compile*] action in the [*Configuration*] section) or "false" (if you need to export only modified schemas, the default value).

`...\Terrasoft.WebApp\Web.config`

```
<add key="IncludeDebugInformation" value="true" />
<add key="ExtractAllCompilerSources" value="false" />
```

Save the file.

2. Execute the [*Compile all*] action in the [*Configuration*] section.



As a result of compilation, Creatio will export the files that contain the source code of configuration schemas to the `../Terrasoft.WebApp/Terrasoft.Configuration/Autogenerated/Src` directory. For example, configuration libraries, their modules, and files that contain debugging information (*.pdb files). Creatio exports the source code of schemasupon each compilation.

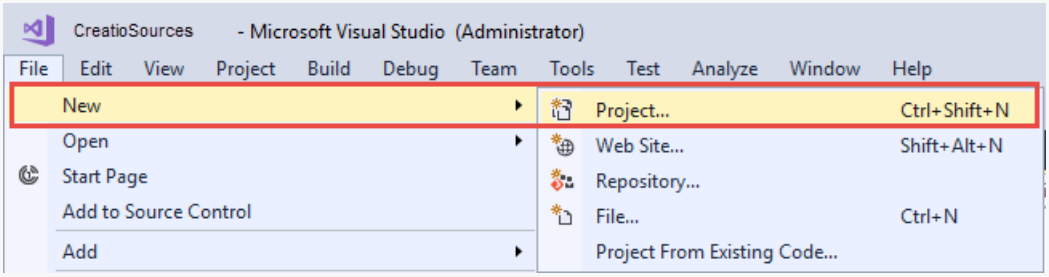
The file names of exported schemas follow this pattern: `SchemaNameInConfiguration.PackageName_SchemaType.cs`, for example, `Contact.Base_Entity.cs`, `ContractReport.Base_Report.cs`.

2. Create a Visual Studio Code project for debugging

Attention. You do not have to create a Visual Studio project to debug the source code. You can simply open the needed files in Visual Studio. If you debug files often or need to manage a large number of files at the same time, create a project streamline the workflow.

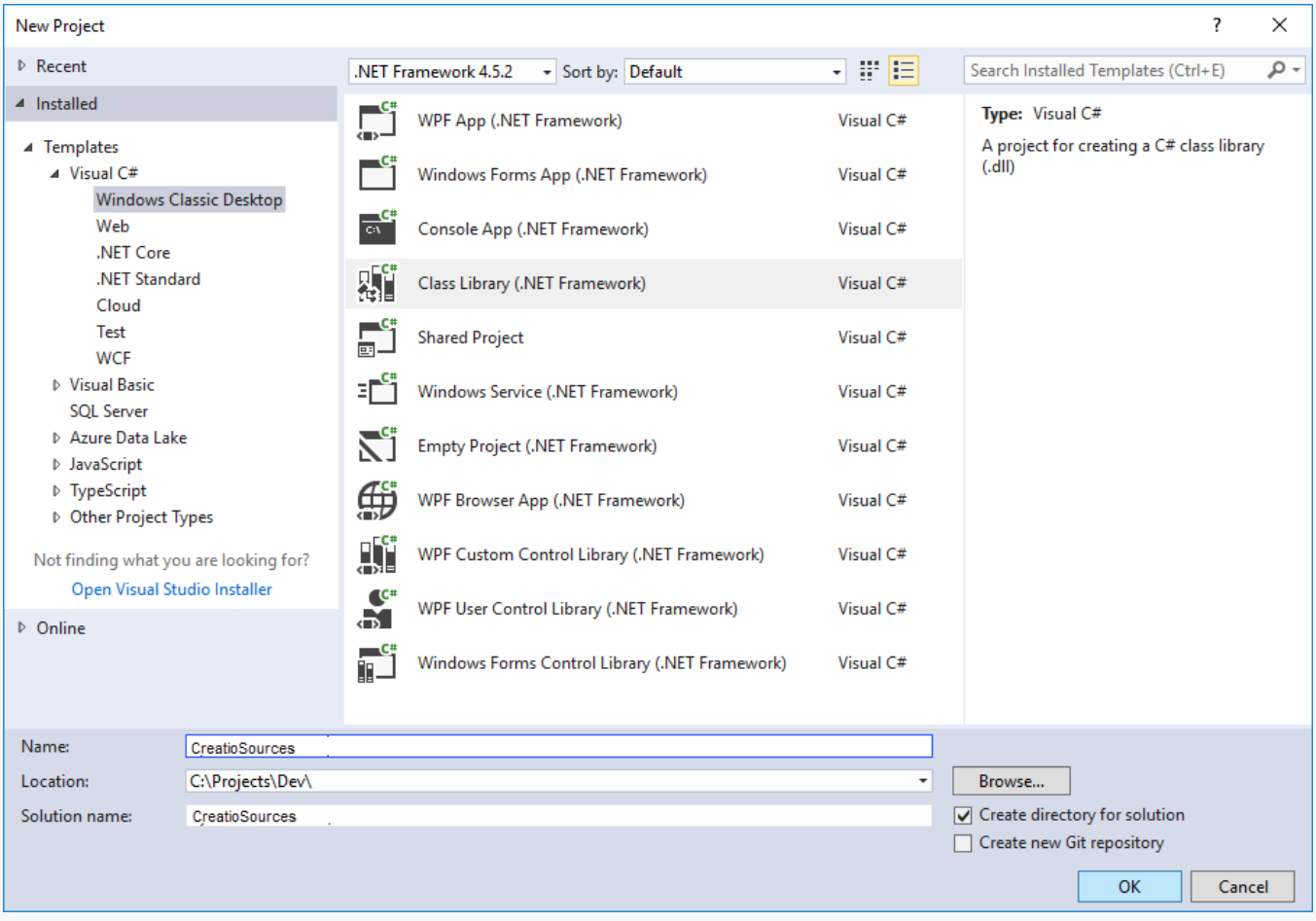
To **create a Visual Studio Code project for debugging**:

1. Click [*File*] → [*New*] → [*Project*] on the Visual Studio toolbar.



2. Fill out the **project properties**:

   - Select the "`Class Library (.NET Framework)`" project type.
   - Enter the project name.
   - Enter the project location.

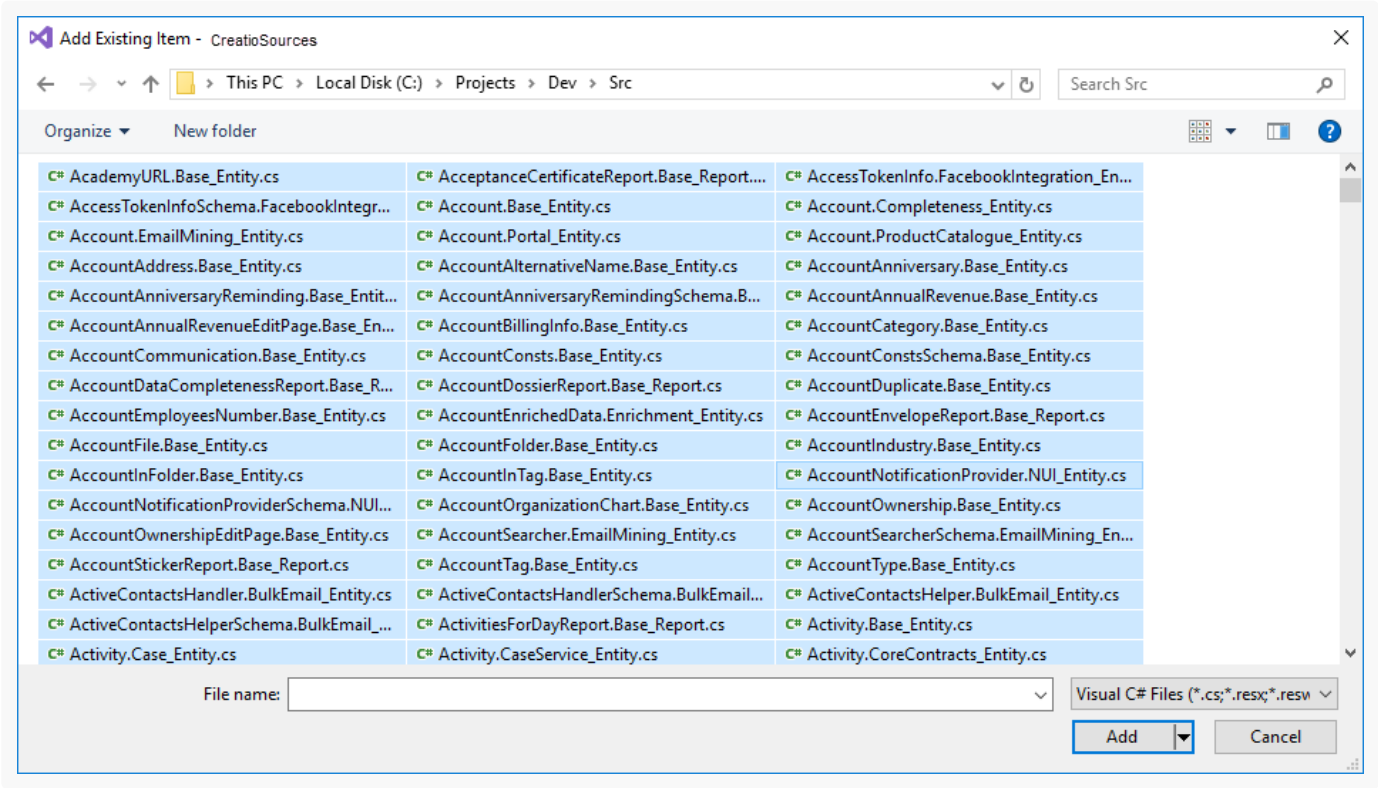


3. Delete the `Class1.cs` file from the project after creating it and save the project.

3. Add the exported files to the Visual Studio Code project

1. Click [*Add*] → [*Existing Item*] in the context menu of the project in the solution explorer.

2. Select all files in the exported files directory.

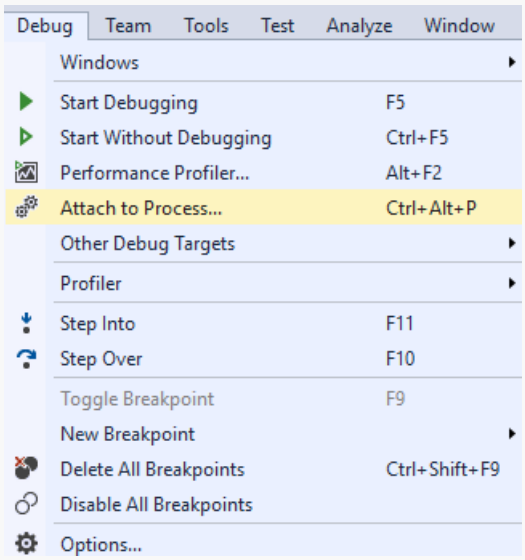


Note. You can only add files needed to debug to the Visual Studio project. In this case, the transition among methods while debugging is limited to the class methods implemented in the added project files.

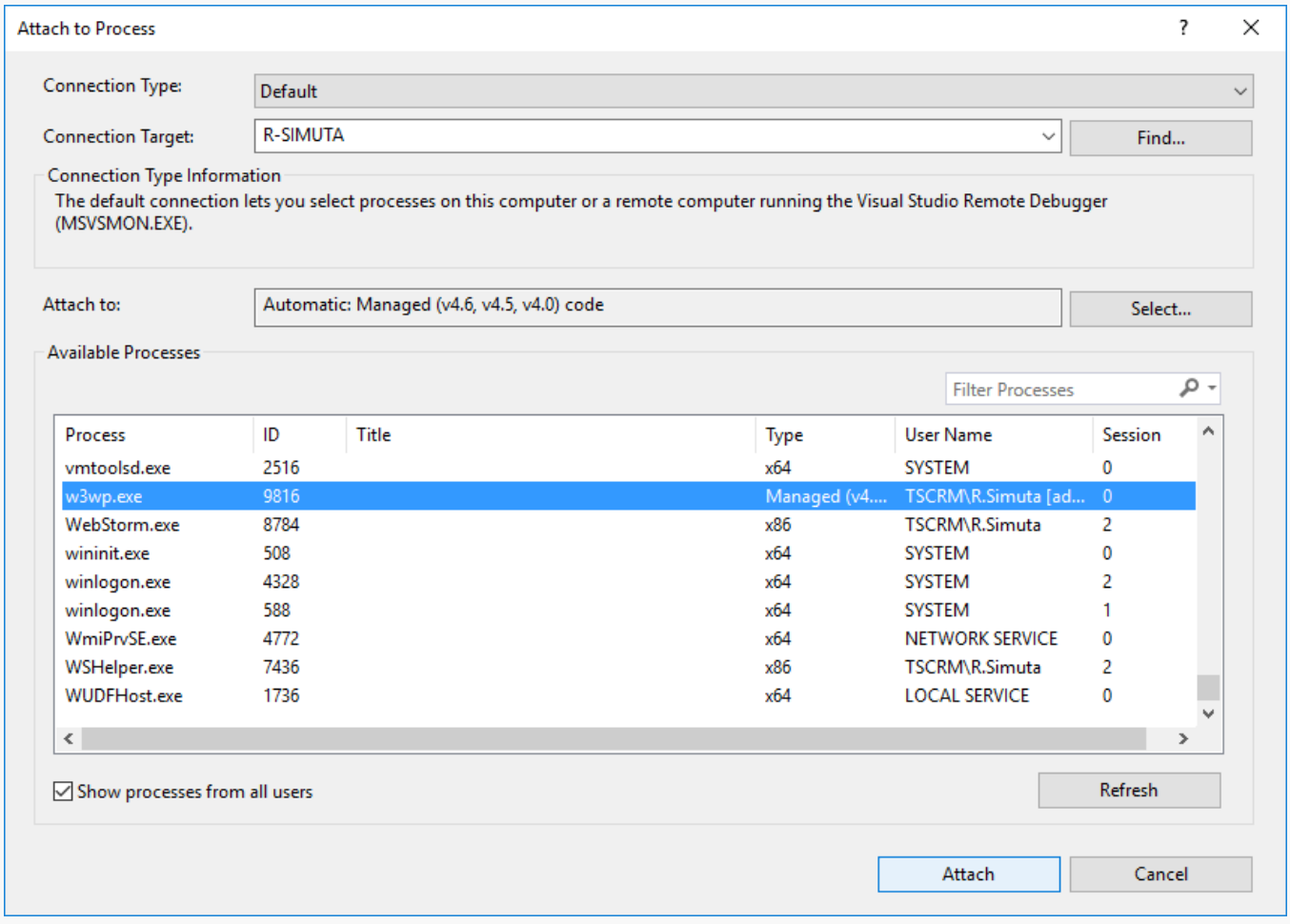
3. Save the project.

4. Connect the project to the IIS server workflow and debug code

1. Click [*Debug*] → [*Attach to process*] on the Visual Studio toolbar.



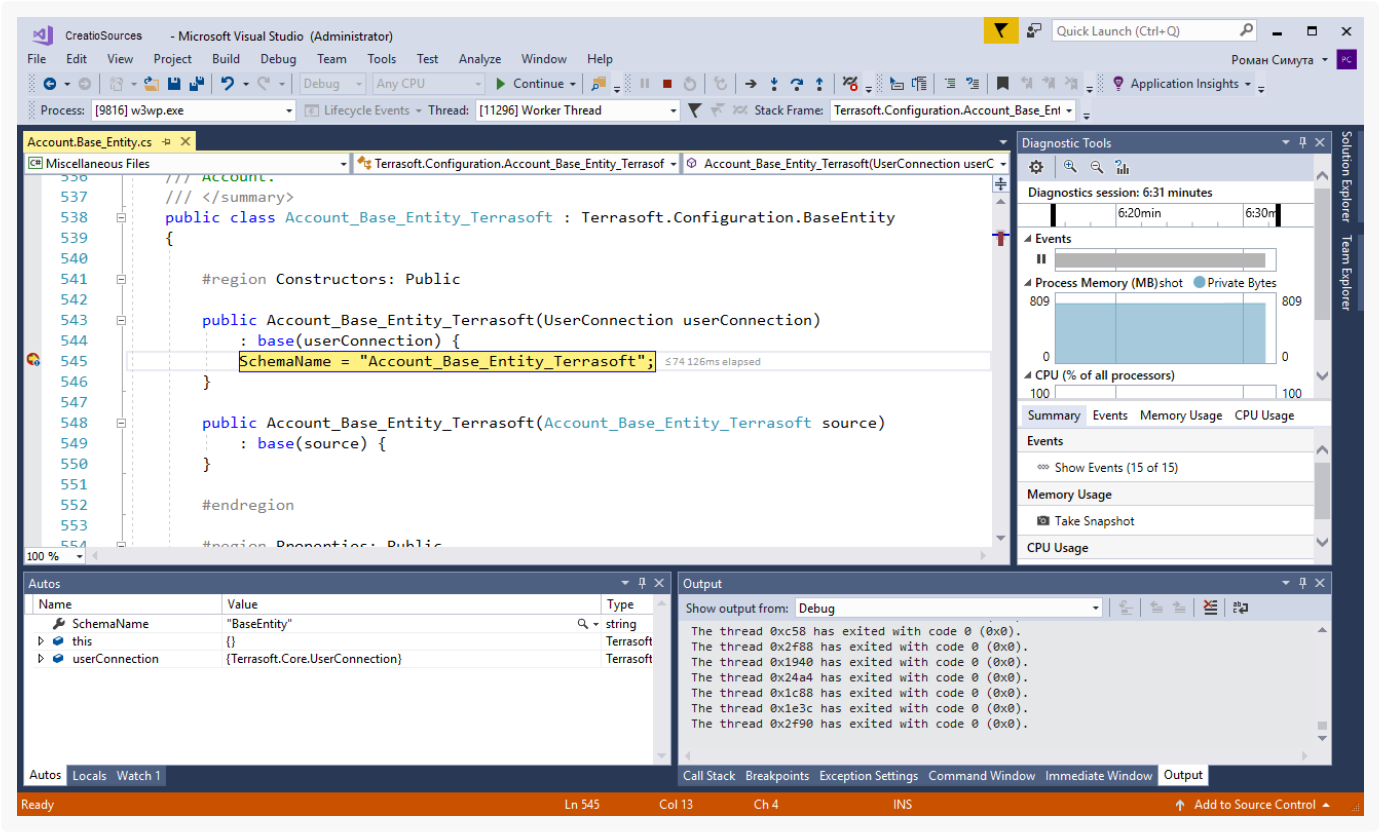
2. Select the IIS workflow where the Creatio instance pool runs from the process list.



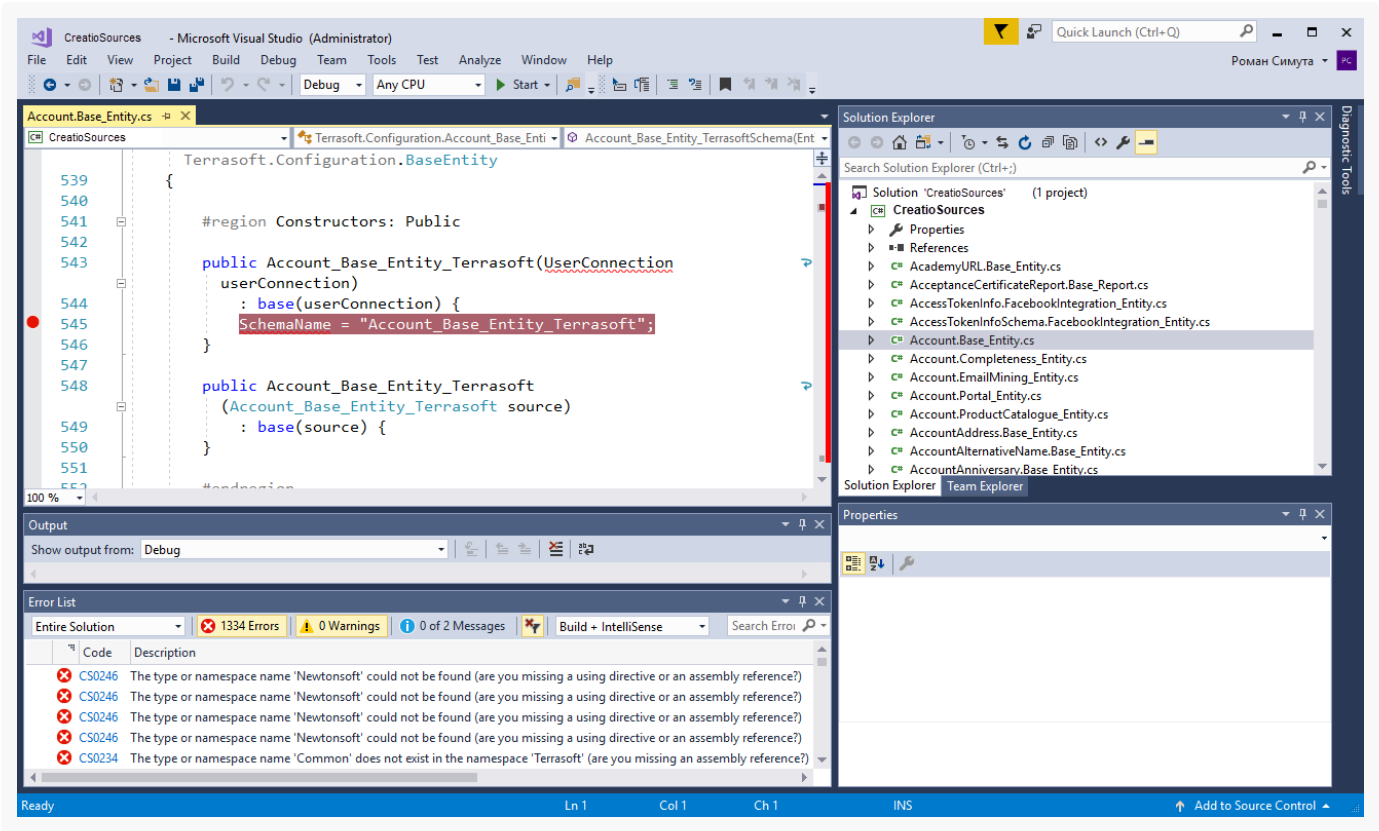
Attention. The workflow name depends on the current IIS server configuration and might differ. The process name for the regular IIS Web Server is `w3wp.exe` and `iisexpress.exe` for IIS Express.

By default, IIS runs the working process as the account whose name matches the name of the application pool. To display processes from all users, select the [*Show processes from all users*] checkbox.

3. Open the source code file and set a breakpoint.



As soon Visual Studio Code runs the method that has a breakpoint set, the program execution is stopped. You can check the current state of the variables and trace the code after stopping.



Front-end debugging

Advanced

Front-end debugging is debugging of Creatio front-end implemented in configuration element schemas of the [*Client module*] type. Learn more in a separate article: Client module.

Integrated debugging tools

Debug the front-end from the browser directly. To do this, use integrated developer tools provided by all supported browsers. All browsers provide similar debugging tools. Learn more in the official Google Chrome documentation, official Microsoft Firefox documentation, official Microsoft Edge documentation, and official Apple Safari documentation.

You can open the debugging tools in the following **ways**:

- Press `F12` or `Ctrl+Shift+I` in Google Chrome
- Press `F12` in Microsoft Firefox
- Press `F12` in Microsoft Edge

Scripts and breakpoints

The development tools let you view the complete list of scripts that are connected to the page and downloaded to the client. You can set a **breakpoint** wherever you want the code to stop.

To **set a breakpoint**:

1. Find the needed script file (you can do it by pressing `Ctrl+O` or `Ctrl+P`) and open it.
2. Go to the code line to set the breakpoint. For example, you can search for a method by name.
3. Set a breakpoint.

   You can set a breakpoint in the following **ways**:

   - click the line number
   - press `F9`
   - click [*Add breakpoint*] in the context menu

You can also set a **conditional breakpoint** which lets you set acondition that triggers the breakpoint.

Use the `debugger` command to **suspend the execution of a script directly from the code**.

Example that suspends the execution of a script from the code

```
function customFunc (args) {
    ...
    debugger; // <-- Debugger stops here.
    ...
}
```

You can view the current values of the variables, execute commands, etc. after the code execution is suspended.

Manage front-end debugging

Check the values of the call stack variables after the code execution is suspended. Then, trace code for fragments where the actual program behavior differs from the expected.

View the browser **debugger commands** that let you navigate the code step by step in the table below.

Browser debugger commands

Command	Browser		
	Google Chrome 	Microsoft Firefox 	Microsoft Edge 
Pause script execution (1)	+	+	+
Step over next function call (2)	+	+	+
Step into next function call (3)	+	+	+
Step out of current function (4)	+	+	+
Deactivate breakpoints (5)	+	–	–
Pause on exceptions (6)	+	–	–

Learn more about the features and commands of the browser navigation bar in the official documentation.

Browser console actions

You can execute the following **actions** at the browser console:

- call JavaScript commands
- display debugging information
- display trace information
- measure and profile code

To do this, use the `console` object.

Call JavaScript commands

1. Open the browser console. To do this, open the [*Console*] tab. To open the [*Console*] tab besides the debugger, click `Esc`.
2. Enter JavaScript commands at the console.
3. Press `Enter` to execute the commands.

Display debugging information

The console can display the following **debugging information**:

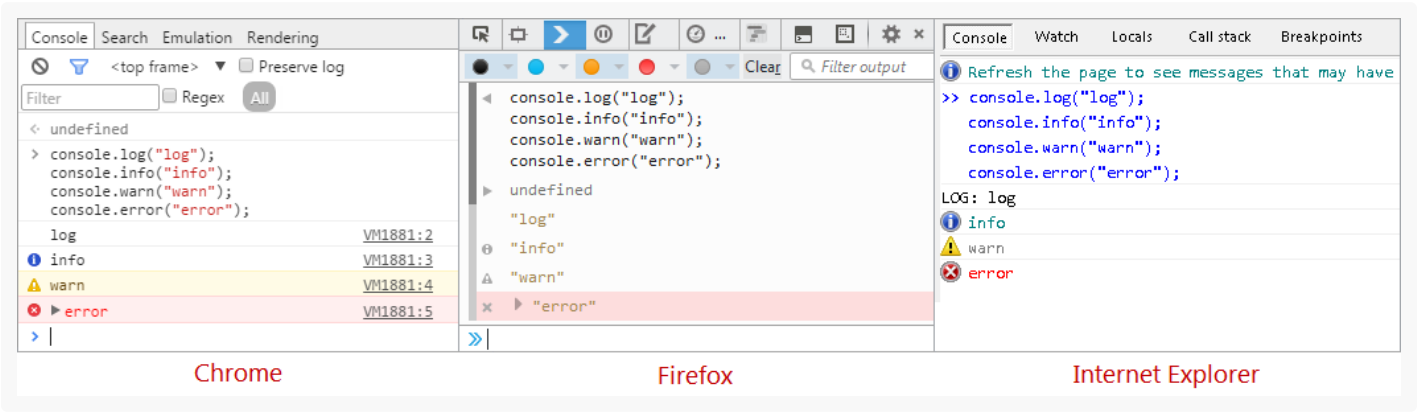
- information messages
- warnings
- error messages

To **display debugging information**, use the corresponding methods of the `console` object listed in the table below.

Methods of the `console` object

Method	Description	Browser		
		Google Chrome	Microsoft Firefox	Microsoft Edge
`console.log(object [, object, ...])`	Displays comma-separated parameters at the console. Required to display various general-purpose messages.	+	+	+
`console.info(object [, object, ...])`	Similar to the `log()` method, but displays messages in a different format. This focuses attention on their importance.	+	+	+
`console.warn(object [, object, ...])`	Displays a warning at the console.	+	+	+
`console.error(object [, object, ...])`	Displays an error message at the console.	+	+	+ (8+)

The console applies a unique style to each message type.



The methods of the `console` object let you format console messages. You can use special **control sequences** (templates) in the message text. The templates are replaced by the corresponding values when displayed. The

values are also transferred to the function in order of appearance.

View the **formatting templates** for the methods of the `console` object in the table below.

Formatting templates for the methods of the `console` object

Template	Data type	Use case
`%s`	String	`console.log("%s is one of the base Creatio products %s", "Creatio sales", "Creatio");`
`%d`, `%i`	Number	`console.log("%s application was originally released in %d", "Creatio", 2011);`
`%f`	Float	`console.log("Pi character is equal to %f", Math.PI);`
`%o`	DOM element (not supported by Microsoft Edge)	`console.log("DOM-View of item <body/>: %o", document.getElementsByTagName("body")[0]);`
`%O`	JavaScript object (not supported by Microsoft Edge, Microsoft Firefox)	`console.log("Object: %O", {a:1, b:2});`
`%c`	CSS style (not supported by Microsoft Edge)	`console.log("%cGreen text, %cRed Text on a blue background, %cCapital letters, %cPlain text", "color:green;", "color:red; background:blue;", "font-size:20px;", "font:normal; color:normal; background:normal");`

Display trace information

View the browser console methods that let you trace and verify expressions in the table below.

Browser console methods that trace and verify expressions

Method	Description	Browser		
		Google Chrome	Microsoft Firefox	Microsoft Edge
`console.trace()`	Displays the call stack from the breakpoint and calls the method. The call stack contains file names, line numbers, and a count of `trace()` method calls from the same breakpoint.	+	+	+ (11+)
`console.assert(expression[, object, ...])`	Checks the expression passed as the `expression` parameter. If the expression is invalid, the console displays the error along with the call stack (`console.error()`). Otherwise, it displays nothing. This ensures the code rules are followed and lets you verify that the actual results of code execution are as expected. The method lets you **test the code**. If the execution result is false, an exception is thrown.	+	+ (28+)	+

Example that uses the `console.assert()` method to test results

```
var a = 1, b = "1";
console.assert(a === b, "A is not equal to B");
```

Measure and profile code

View the browser console methods that let you measure code execution time in the table below.

Browser console methods that measure code execution time

Method	Description	Browser		
		Google Chrome	Microsoft Firefox	Microsoft Edge
`console.time(label)`	Activates a millisecond counter that has the `label` tag.	+	+	+ (11+)
`console.timeEnd(label)`	Stops the millisecond counter that has the `label` tag and displays the results at the console.	+	+	+ (11+)

Example that uses the `console.time()` and `console.timeEnd()` methods

```
var myArray = new Array();
/* Activate the counter with Initialize myArray tag. */
console.time("Initialize myArray");
myArray[0] = myArray[1] = 1;
for (i = 2; i<10; i++) {
    myArray[i] = myArray[i-1] + myArray[i-2];
}
/* Deactivate the counter with Initialize myArray tag. */
console.timeEnd("Initialize myArray");
```

View the browser console methods that let you profile code and display the profiling stack in the table below. The profiling stack contains detailed information about the time it takes the browser to execute the operation.

Browser console methods that measure code execution time

Method	Description	Browser		
		Google Chrome	Microsoft Firefox	Microsoft Edge
`console.profile(label)`	Runs a JavaScript profiler, then displays the result using the `label` tag.	+	+ (if the DevTools panel is open)	+ (10+)
`console.profileEnd(label)`	Stops the JavaScript profiler.	+	+ (if the DevTools panel is open)	+ (10+)

The profiling results are displayed in the following **browser tabs**:

- [*Profiles*] in Google Chrome
- [*Performance*] in Microsoft Firefox
- [*Profiler*] in Microsoft Edge

Front-end debugging mode `isDebug`

The front-end debugging mode `isDebug` lets you retrieve detailed information about Creatio errors and track them in the code.

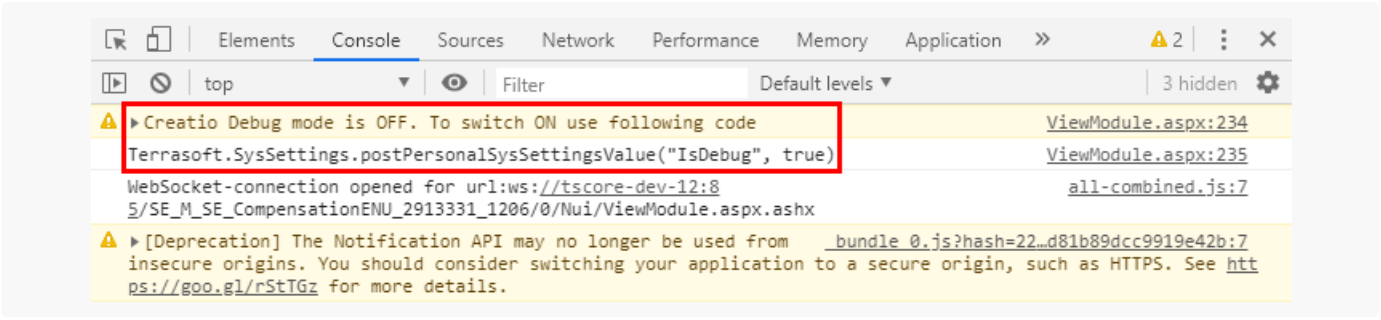
Browser minifies code in regular mode. This means client scripts are assembled in the `all-combined.js` file. The file is assembled when the assembly is created and contains the entire functionality. If you enable `isDebug` mode, the assembly and minification of *.js files are turned off. Client scripts are displayed as separate files.

> **Note.** Front-end debugging mode affects Creatio performance. For example, it increases the time it takes to open pages.

To **configure front-end debugging mode**:

1. Identify the current status of the front-end debugging mode. To do this, press `F12` or `Ctrl+Shift+I` in Google Chrome.

   Besides the status of the front-end debugging mode, the console displays a code to activate or deactivate it.



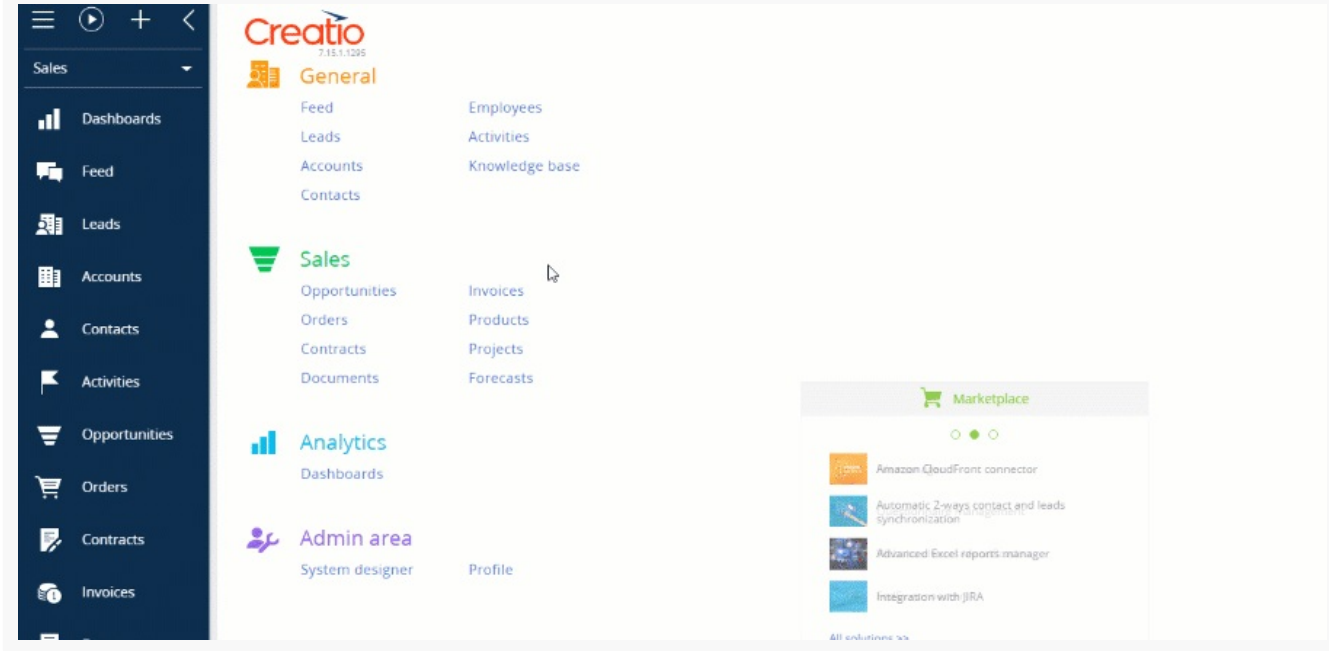
2. Enable front-end debugging mode.

   You can enable front-end debugging mode in the following **ways**:

   - Run code below at the browser console:

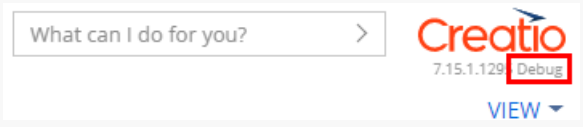
     ```
     Terrasoft.SysSettings.postPersonalSysSettingsValue("IsDebug", true)
     ```

   - Change the value of the [*Debug mode*] (`isDebug` code) system setting.



3. Refresh the page or press `F5` to apply the changes.

Creatio displays a `Debug` **indicator** next to the version number after you turn on front-end debugging mode.

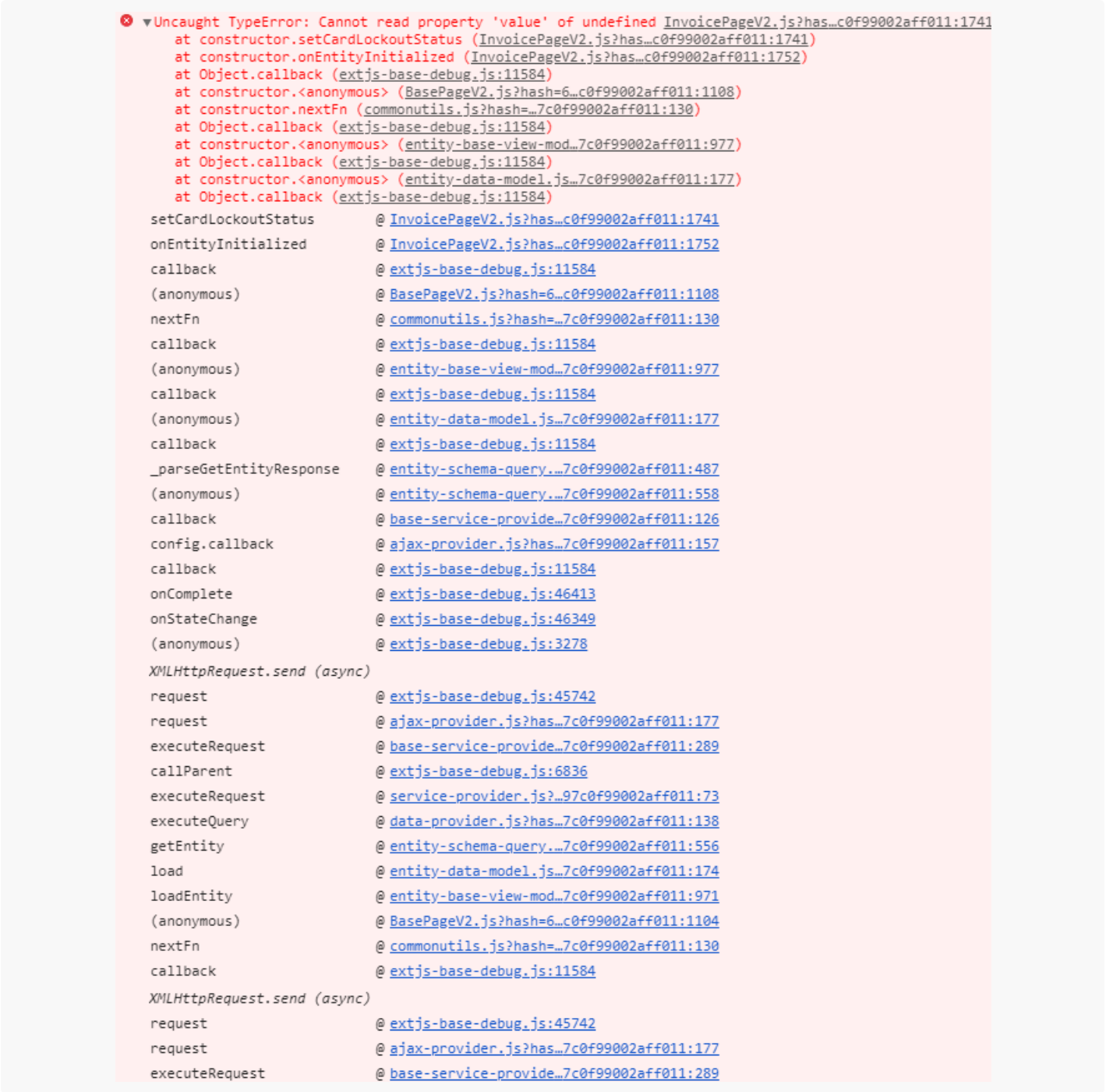


View the examples that display error information in the console when `isDebug` mode is turned on and off on the figures below.

Error information (`isDebug` is turned off)

```
▼Uncaught TypeError: Cannot read property 'value' of undefined InvoicePageV2.js?has…c0f99002aff011:1741
    at i.setCardLockoutStatus (InvoicePageV2.js?has…c0f99002aff011:1741)
    at i.onEntityInitialized (InvoicePageV2.js?has…c0f99002aff011:1752)
    at Object.callback (all-combined.js:6)
    at i.<anonymous> (BasePageV2.js?hash=6…c0f99002aff011:1108)
    at i.e (all-combined.js:7)
    at Object.callback (all-combined.js:6)
    at i.<anonymous> (all-combined.js:7)
    at Object.callback (all-combined.js:6)
    at i.<anonymous> (all-combined.js:7)
    at Object.callback (all-combined.js:6)
setCardLockoutStatus          @ InvoicePageV2.js?has…c0f99002aff011:1741
onEntityInitialized           @ InvoicePageV2.js?has…c0f99002aff011:1752
callback                      @ all-combined.js:6
(anonymous)                   @ BasePageV2.js?hash=6…c0f99002aff011:1108
e                             @ all-combined.js:7
callback                      @ all-combined.js:6
(anonymous)                   @ all-combined.js:7
callback                      @ all-combined.js:6
(anonymous)                   @ all-combined.js:7
callback                      @ all-combined.js:6
_parseGetEntityResponse       @ all-combined.js:7
(anonymous)                   @ all-combined.js:7
callback                      @ all-combined.js:7
e.callback                    @ all-combined.js:7
callback                      @ all-combined.js:6
onComplete                    @ all-combined.js:6
onStateChange                 @ all-combined.js:6
(anonymous)                   @ all-combined.js:6
XMLHttpRequest.send (async)
request                       @ all-combined.js:6
request                       @ all-combined.js:7
executeRequest                @ all-combined.js:7
callParent                    @ all-combined.js:6
executeRequest                @ all-combined.js:7
executeQuery                  @ all-combined.js:7
getEntity                     @ all-combined.js:7
load                          @ all-combined.js:7
loadEntity                    @ all-combined.js:7
(anonymous)                   @ BasePageV2.js?hash=6…c0f99002aff011:1104
e                             @ all-combined.js:7
callback                      @ all-combined.js:6
XMLHttpRequest.send (async)
request                       @ all-combined.js:6
request                       @ all-combined.js:7
executeRequest                @ all-combined.js:7
```

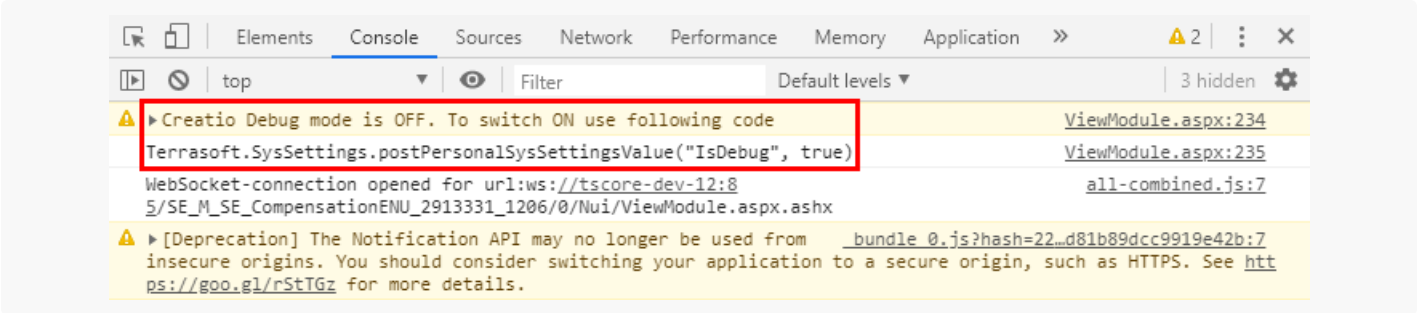
Error information (`isDebug` is turned on)



Debug the server code

Easy

To check the current status of the client debugging mode, open the browser console by pressing the F12 key or pressing Ctrl+Shift+I. Aside from the current status of the client debugging mode, the console will display the code to enable or disable debugging.

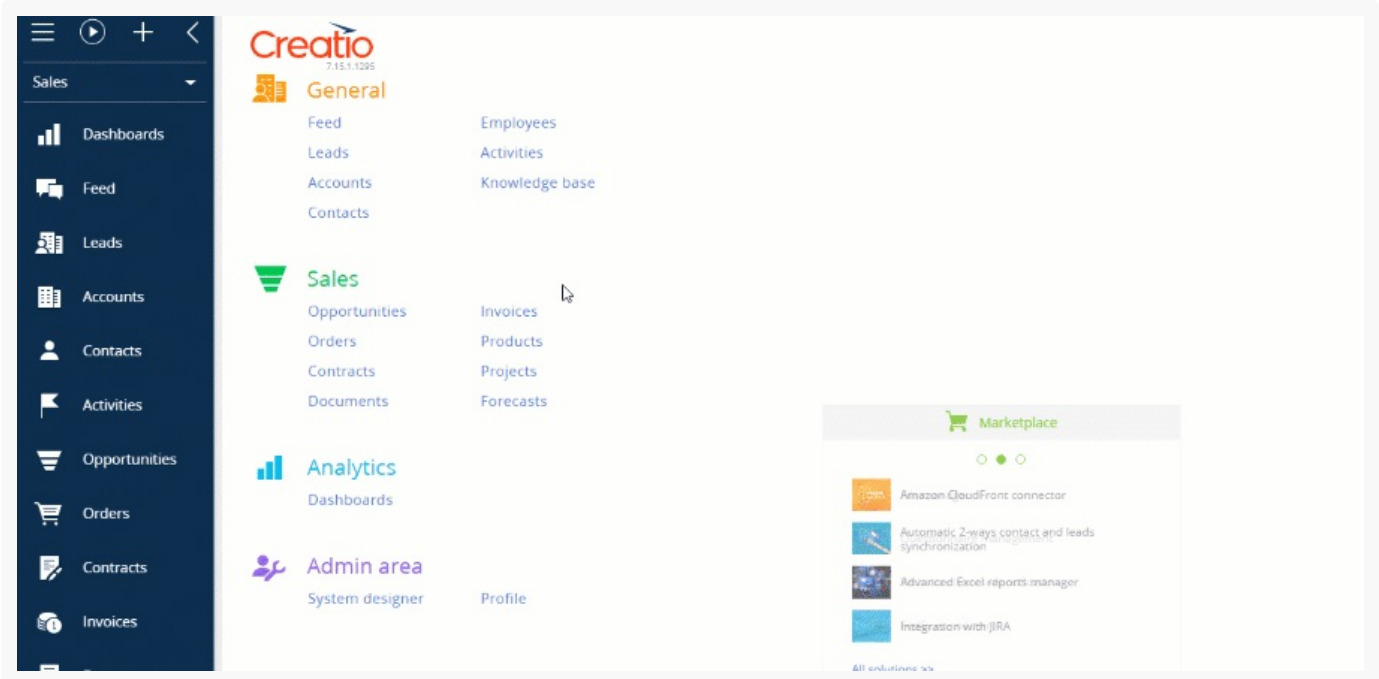


You can enable the client debugging mode using the following methods:

- Execute the following code in the browser console

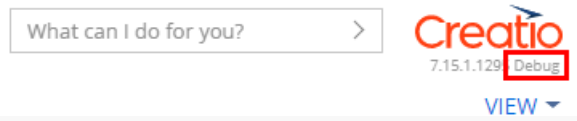
```
Terrasoft.SysSettings.postPersonalSysSettingsValue("IsDebug", true)
```

- Change the value of the [Debug mode] system setting.



To apply the changes, refresh the page or hit F5.

Upon activating the client debugging mode, you will see the Debug indicator next to the site's version number.



Note. Enabling the client debugging mode will affect site performance. For instance, it can increase the time needed for the pages to load.

The figures below show examples of errors displayed in the console with the ‘isDebug’ mode disabled and enabled.

Displaying an error (‘isDebug’ disabled)

Displaying an error (‘isDebug’ enabled)