Code Editor Reference

Work with the built-in code editor.

Reference **Modules** **Scripts** **UI** Code Editor | Classes | Expressions | Monitor | References | Tasks

---

Code Editor (Reference) provides a unified development environment for writing scripts, display code behind, SQL queries, and report documents across the FrameworX platform. It supports:

- **Script Languages**: VB.NET, C#, Python
- **Display Code Behind**: .NET and JavaScript
- **Datasets Module**: SQL query statements
- **Reports Module**: JSON, XML, and HTML5 documents

The editor provides IntelliSense, automatic compilation, integrated debugging, and language conversion capabilities.

Accessing the Code Editor
Editing Code
Toolbar Commands
Language Features
Namespace Declarations
Compilation
Debugging
Best Practices
Troubleshooting
Appendix: Debugging Commands {#debugging-appendix}
Editing Code
CodeEditor user Interface
Scripts Compilation
Debugging and Error Handling

Accessing the Code Editor

The Code Editor is accessible from multiple modules:

Module	Path	Content Type
Scripts	Scripts Code Editor	VB.NET, C#, Python
Displays	Display Editor Code Behind	.NET, JavaScript
Datasets	Datasets Query Editor	SQL
Reports	Reports Document Editor	JSON, XML, HTML5

Editing Code

1. Navigate to **Scripts Code Editor**
2. Select task or class from dropdown in toolbar
3. Choose language (VB.NET/C# conversion available)
4. Edit code with IntelliSense support
5. Format code using **Auto Format** button
6. Save using toolbar **Save** icon. From FrameworX 10.1.6, you also save with **Ctrl+S** while the cursor is in the editor.

Toolbar Commands

Editing Tools

Icon	Command	Description
	Compile	Compile code (automatic on save)
	Namespace Declarations	Add/view namespace references
	Outdent	Add tab to selected lines
	Indent	Remove tab from selected lines
	Comment Out	Comment selected lines
	Uncomment	Remove comments
	Format Document	Format entire document
	Format Selection	Format selected lines

Language Features

Language Conversion (VB.NET C#)

The editor can automatically convert between VB.NET and C#, with important considerations:

Type Safety Differences:

- **C#** is strongly typed - requires explicit casting for type conversions
- **VB.NET** is loosely typed - performs implicit conversions

Example: Converting double to integer

- C#: `int value = (int)doubleValue;` (explicit cast required)
- VB.NET: `Dim value As Integer = doubleValue` (automatic conversion)

Recommendation: C# VB.NET usually succeeds, but VB.NET C# may require manual fixes for type conversions.

Language conversion is intended for:

- One-time migration between languages
- Sharing code with teams using different languages
- NOT for frequent back-and-forth conversion

IntelliSense

IntelliSense provides intelligent code completion for:

FrameworX Namespaces

Type `@` to see all platform namespaces:

- `@Tag` - Most common, access all tags
- `@Alarm` - Alarm system objects
- `@Display` - Display and screen objects (client-side only)
- `@Script` - Script classes and tasks
- `@Dataset` - Database operations
- See [Programming (Reference)] for complete namespace list

Tag Autocomplete

Available from FrameworX 10.1.6. Tag autocomplete groups tags by folder and shows one level of the path at a time. Folders are separated by a slash (`/`), for example `@Tag.Plant/Area2/Pump1`. Insert a folder name to move one level down the path.

To find a tag in any folder, type any part of its name after `@Tag.`. The list shows the matching tags with their folder. Press **Tab**, or use the arrow keys and **Enter**, to insert a tag found by name. A period, a space or a semicolon does not insert it. This prevents inserting the wrong tag when the same tag name exists in more than one folder.

Tag link fields in the display property editor use the same folder grouping and name search. In 10.1.5, tag autocomplete shows one flat list of full tag paths.

Scope Awareness

- **Server-side scripts**: No client namespaces shown
- **Client-side scripts**: Full namespace access
- **Display Code Behind**: Display-specific context

.NET Framework

Standard .NET namespaces available after declaration:

- Type `System.` for .NET framework classes
- DataTable, collections, and other .NET types
- Full IntelliSense for declared namespaces

Go to Definition

Available from FrameworX 10.1.6. Hold **Ctrl** and click a symbol, such as a class, a method or a namespace member, to jump to its definition. While you hold Ctrl, the symbol under the pointer is underlined as a clickable link.

Toolkit Namespace (TK)

Quick access to common utility methods:

csharp

```csharp
// Type conversion
double value = TK.ConvertTo<double>("123.45");

// Tag operations
DataTable dt = TK.TagsToDataTable("Tag.*");

// Communication settings
TK.SetChannelSettings("Channel1", "192.168.1.100");
```

See [Namespace API (Reference)] for complete TK method list.

Namespace Declarations

Important: Cannot add `using` (C#) or `Import` (VB.NET) directly in code. Use the Namespace Declarations button to manage references.

Click **Namespace Declarations** button to:

- View default included namespaces
- Add custom namespace references
- Different defaults for server vs client

Common server-side defaults:

- System
- System.Collections.Generic
- System.Linq
- System.Data

Compilation

Incremental Compilation

- Automatic background compilation
- Only changed scripts recompile
- No manual compilation needed
- Improves performance

Build Verification

- Save triggers compilation for current item
- Full solution build before production deployment
- See [Build and Publish] for production builds

Debugging

The Code Editor includes integrated .NET debugging capabilities.

Enabling Debugging

1. Enable Debug Information in **Runtime Build and Publish**
2. Save source code to generate debug symbols (first time only)
3. Connect to runtime via **Runtime Startup**
4. Click **Attach .NET Debugger** in toolbar
5. Debugger stops at breakpoints and exceptions

Multiplatform Solutions

From FrameworX 10.1.6, the built-in debugger attaches to Script Tasks and Classes of a Multiplatform (.NET 10) solution, with the same breakpoints, stepping and variable values as a Windows (.NET Framework) solution. The Designer and the runtime run on the same Windows x64 computer, with the runtime as a 64-bit process. On other computers, for example Windows on ARM64, the Code Editor hides the Debug menu and the debugger buttons for Tasks and Classes of a Multiplatform solution. Debugging of display code-behind is unchanged on every computer.

In 10.1.5, the built-in debugger does not attach to Tasks and Classes of a Multiplatform solution. Use Microsoft Visual Studio for them, as described in Debugging Scripts Code with Microsoft Visual Studio.

Debug Features

- Set breakpoints on any line
- Step through code execution
- Inspect variables on hover
- Watch window for monitoring



For debugging using Microsoft Visual Studio go to Debugging Scripts Code with Microsoft Visual Studio.

Best Practices

1. **Choose appropriate language**: C# for type safety, VB.NET for simplicity
2. **Use IntelliSense**: Type `@` for platform objects, `.` for properties
3. **Manage namespaces properly**: Use dialog, not direct statements
4. **Format regularly**: Use Auto Format for readability
5. **Test incrementally**: Save often to trigger compilation
6. **Debug systematically**: Use breakpoints instead of trial-and-error

---

Troubleshooting

IntelliSense not showing:

- Verify namespace declarations
- Check server/client context
- Ensure previous compilation succeeded

Language conversion errors:

- Review type conversions
- Add explicit casts in C#
- Check for language-specific features

Debugging not working:

- Enable debug information
- Ensure runtime is connected
- Verify debug symbols generated

Appendix: Debugging Commands {#debugging-appendix}

The Code Editor includes Visual Studio-style debugging capabilities:

Toolbar Debugging Icons

Icon	Command	Shortcut	Description
	Set Breakpoint	F9	Toggle breakpoint on current line
	Continue	F5	Resume execution until next breakpoint
	Step Into	F11	Execute current line, enter functions
	Step Over	F10	Execute current line, skip functions
	Step Out	Shift+F11	Complete current function and return
	Restart	Ctrl+Shift+F5	Restart debugging session
	Stop Debugging	Shift+F5	End debugging session

Setting Breakpoints

- Click in the left margin next to any line
- Red dot appears indicating breakpoint
- Click again to remove

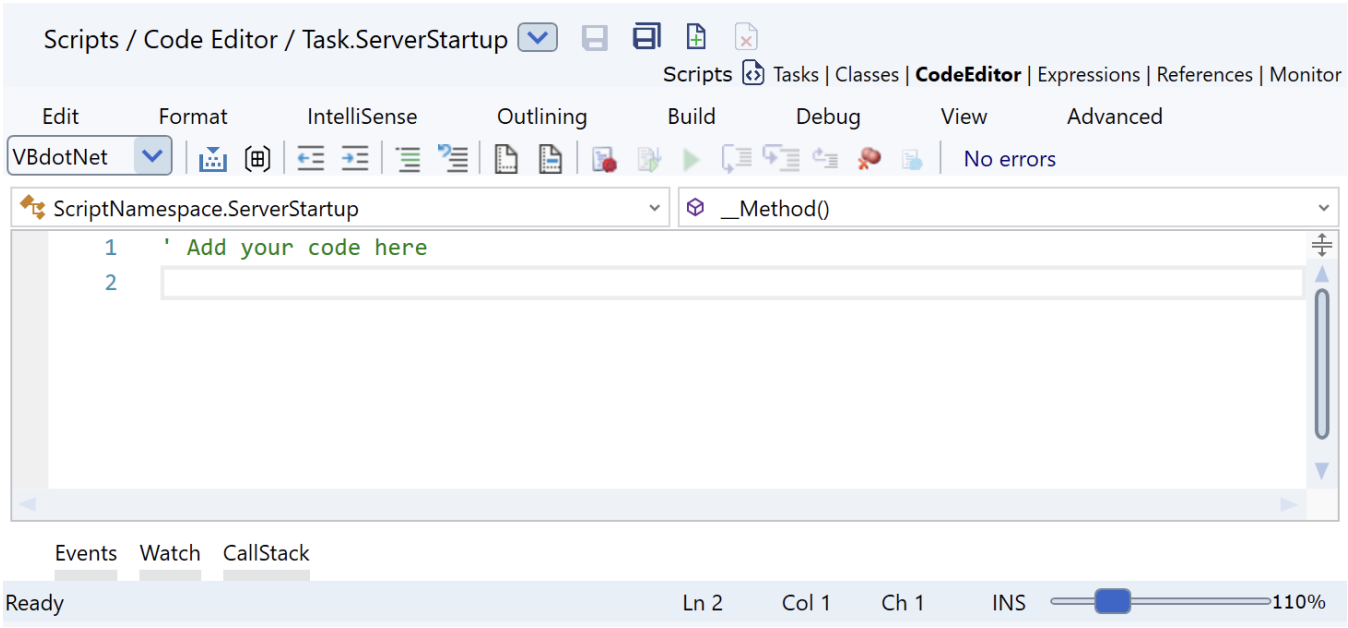
Inspecting Variables

- **Hover**: Mouse over variable to see current value
- **Watch Window**: Add variables to monitor continuously
    - Use @ prefix for solution objects (e.g., `@Tag.Temperature`)
    - Local .NET variables without prefix
- **Immediate Window**: Execute expressions during break

Important Notes

- Watch window updates only when execution is paused
- For real-time monitoring, use PropertyWatch diagnostic tool
- Debugging available for C# and VB.NET only (not Python)

In this section...



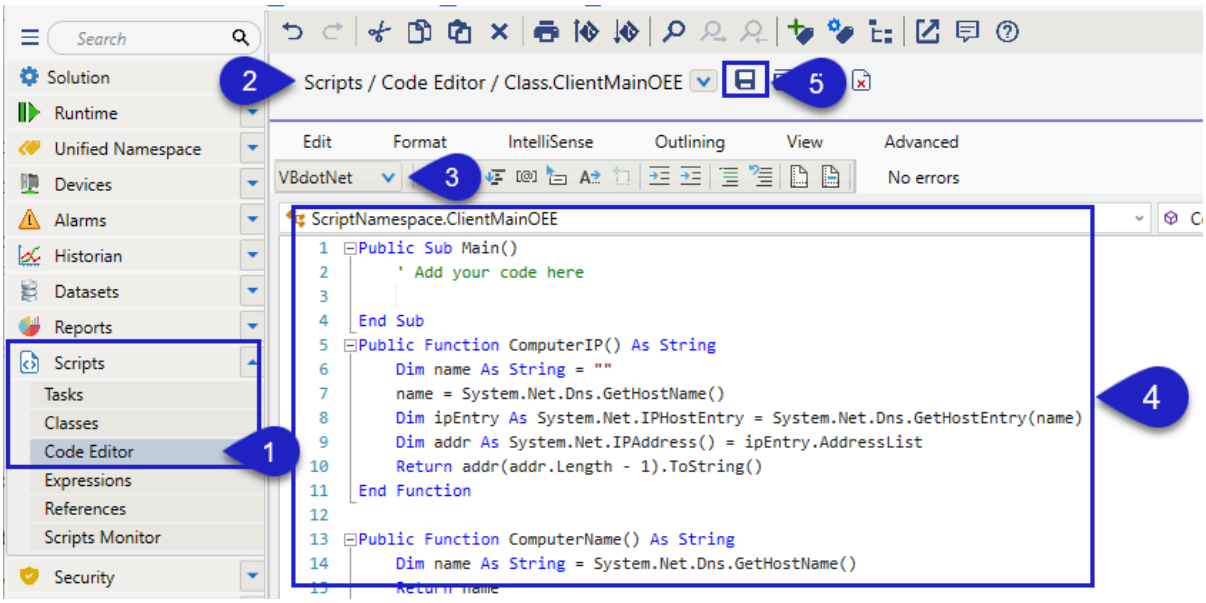
Editing Code

To edit code, follow the steps below:

1. Access **Scripts Code Editor**. The Code Editor environment displays the contents of the last selected Script Task or Script Class.
2. Select the task or class you want to edit from the drop-down list in the main toolbar. See the Scripts Module page for more details on creating tasks or classes.
3. Select the Language in the toolbar. If you change your language selection in the code editor, between **VB.NET** and **C#.** the system automatically converts existing code to the selected language. For other languages or document formats, this ComboBox is read-only.
4. Edit the code. You can also format your code for readability by clicking **Auto Format.**
5. Click the **Save icon** on the toolbar to save the changes.

The ability to switch between VB.Net and C# is a powerful feature that allows developers to leverage the strengths of both languages, but due differences on the languages, it is possible a VB.NET code when changed to C# will present errors, as C# is more restrictive on type conversion.

The image below illustrates the process described above.



Productivity Tools

The Scripts Modules offers tools to enhance your productivity when creating and editing scripts. The are as follow:

- **IntelliSense**: This is a code-completion aid that features listing the available properties and methods for the objects and auto-completing their names. In addition to the Properties and Methods of the .NET Framework, it supports all project entities, like tags and displays. Type a dot (**.**) after an element to show its available properties.
- **Toolkit Namespace**: The platform exposes many .NET libraries to the solution, but there are a few methods that are frequently required, such as type conversion, copying tags to DataTables and .NET objects, or dynamically changing the communication settings. Those methods are included in a library toolkit, to use then insert **TK** in the code editor, IntelliSense will respond with a list of all available methods and provide a summary documentation for any method you select. See the example below:

  ```
  double x = TK.ConvertTo<double>("123");
  ```

  Access the namespace T.Toolkit.TK class at the Namespaces API to see the list of all methods available.

CodeEditor user Interface

Main ToolBar

CodeEditor ToolBar		
Icon	**Command**	**Description**
	Compile	Compile the code. This operation is also executed automatically when saving the document.
	Namespace Declarations	Verify or add the namespaces added to this code. This editor don't allow to include C# 'using' or VB.Net 'Import' statements directly. For more information go to Scripts References Reference.
	Outdent	Insert a tab formatting to the selected lines.
	Indent	Remove a tab from the selecting lines.
	Comment Out	Comment the selected lines.
	Uncomment	Remove the comment from the selected lines.
	Format Document	Format the entire document.

	Formant Selection	Format the selected lines.
	Debugger	The remaining icons are using when debugging the code. (C# and VB.Net only.)

Scripts Compilation

Scripts Compilation uses **incremental compilation**, meaning scripts are automatically compiled in the background as they are created or modified — no user action required.

Only affected scripts are recompiled, improving performance and reducing build times.

When you edit a script, the system detects changes and compiles only the necessary scripts, avoiding the overhead of recompiling the entire solution.

While the code editor continuously compiles scripts in the background, we recommend performing a full **Build** operation before deploying a project to production.

Building ensures all scripts are fully up-to-date and minimizes the risk of runtime errors. For details, see the Build and Publish page.

Debugging and Error Handling

The platform has an integrated .NET code debugger. In order to use it, the local computer must have a cache with the necessary files to run the debugger. The steps to enable the debugger are:

- Enable the Debug Information on **Runtime Build and Publish.**
- As necessary, save the source code you want to debug. This will create the debug information. This step is only necessary the first time you open the project on the computer. Afterwards, the background compiling generates the necessary debugging information to enable the use of breakpoints and step execution.

Attaching the .NET debugger

In order to have a .NET debugging session, the engineering environment must be connected to the runtime and the .NET debugger must be attached to server or client process. Follow these steps.

- When Running the solution, go to **Runtime Startup** and enable the connect check box.
- If the solution is already running, go to **Runtime Startup**, according to the runtime you want to attach, and connect to the runtime system by pressing the connect button on those pages.
- Open any script that has debugging information and press the **Attach .NET debugger** button. A message on the bottom of the engineering workspace will show that a debugging session is active with the server components or the client components of the running project.
- When the .NET debugger is attached, the system will stop on the defined breakpoints and will stop automatically when any .NET Exception occurs.

Breakpoints, Steps, and Watch

In order to setup a breakpoint, open the desired code, select the line, and press Insert Breakpoint on the toolbar.

When the system stops on a breakpoint, you can perform step by step execution or click the continue button.

In order to inspect local .NET variables, tags, or project objects, select the text in the script editor. When the execution is stopped on a breakpoint, the toolkit will show the current value of the variable.

You can also add .NET variables or project objects to the Watch window. When adding tags or project objects, you need to use the at symbols @, like `@tag.tag1`. This will allow the system to know it is a project object and not a local .NET variable. Keep in mind that the Watch display is only updated when execution is stopped. If you want to have real-time values for tags and objects, open the PropertyWatch diagnostics tool.

In this section...