AlteNET Scripter for WPF v7

Contents

Introduction

AlterNET Scripter is a component library designed to integrate C#/Visual Basic TypeScript/JavaScript and IronPython scripts into your .NET applications. It allows extending functionality of the application logic without recompiling and redeploying the application. An example of such an application is MS Office with ability to write and debug macros using Visual Basic for Application; Photoshop, SolidWorks and many other software packages provide some sort of SDK so developers or end-users can write their own scenarios to extend these platforms.

The AlterNET Scripter provides a framework to compile and execute user-defined scripts along with the set of debugging tools enabling application developers to expose application internals to the script writers so they can write user-defined scenarios for these applications.

What's included

The main component in the package is *ScriptRun*, which encapsulates functionality of running standalone script files or projects with forms and resources; it allows to reference third-party assemblies and register application-defined objects to be accessible in the scripts.

ScriptDebugger provides a fully-featured script debugging engine; it supports Start, Stop, Break and Continue commands, step by step execution, breakpoints, expression evaluation, viewing local variables and watches, stack tracing and multiple thread debugging.

ScriptDebugger Widgets – set of debugging widgets (output, compiler errors, call stack, threads, locals watches and breakpoints).

Script Debugger Demo and QuickStart projects – these projects show how to run scripts, execute specific methods, reference application objects, execute scripts asynchronously, debug scripts and projects by standalone debugger or embed Script debugging logic into your application.

TypeScript/JavaScript components – a separate set of ScriptRun implementation for execution of TypeScript/JavaScript code; a TypeScript Debugger – a full features debugging engine for TypeScript; TypeScript debugger demo and set of quick start projects.

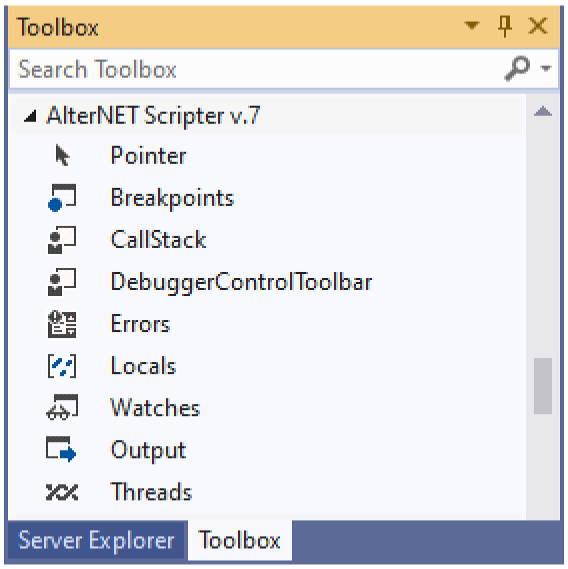
IronPython components – a separate set of ScriptRun implementations for execution of IronPython code; a IronPython Debugger – a full feature

Full Source Code - which comes with Universal edition.

Installation

AlterNET Scripter is installed as part of the AlterNET Studio installer program. Advanced installation options include platform selection (WinForms, WPF or both).

Installation requires .NET Framework 4.6.1 + and Visual Studio 2015, 2017 or 2019 to be installed on the target machine. Installation program will register Visual Studio extensions and place Script Debugger Widgets on the AlterNET Scripter tab in Visual Studio toolbox.



Other versions of .NET Framework 4.5.2,.NET Core 3.0,+ and Net 5.0 are available via NuGet packages. Complete list of NuGet packages can be found here:

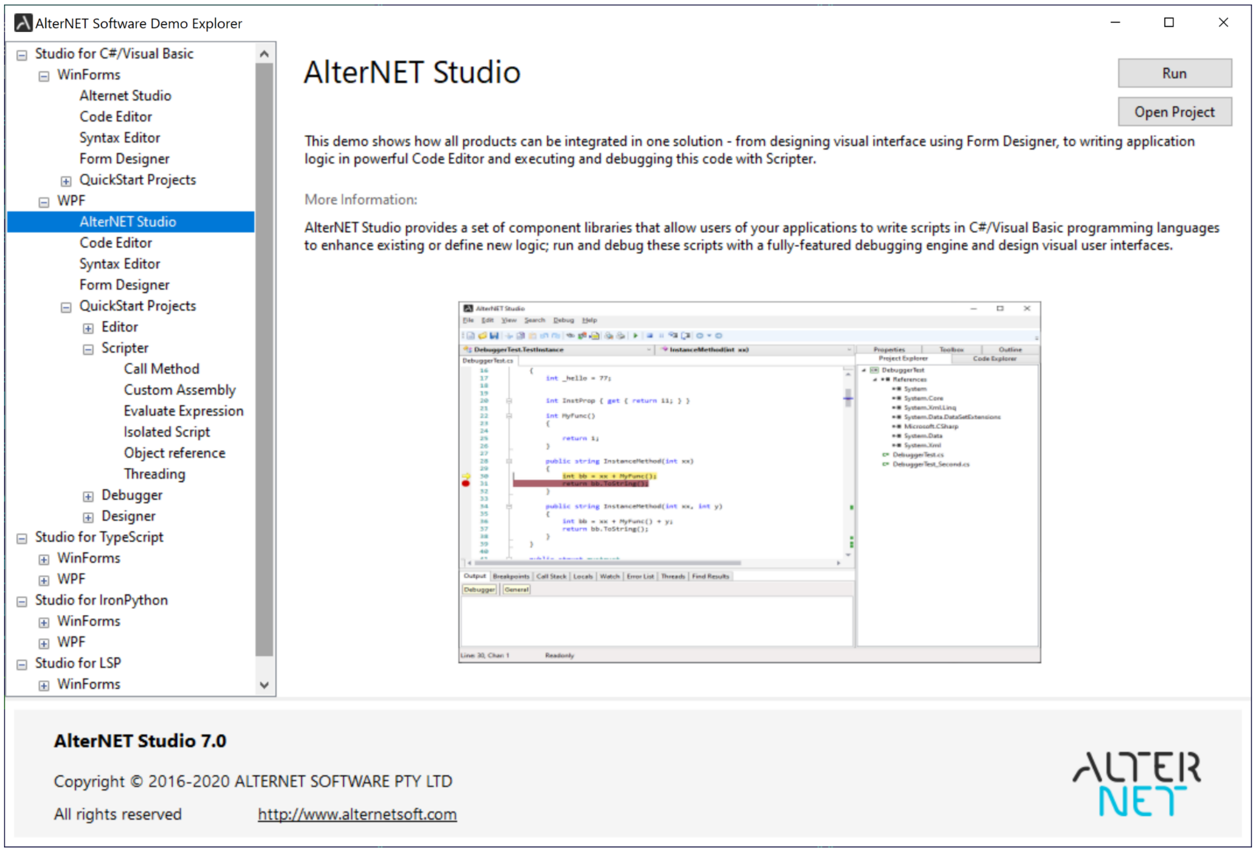
https://alternetsoft.com/download

Please note that building WPF applications containing XAML files under .NET Core require Microsoft Build tools for Microsoft Visual Studio 2015 or later installed on a target machine.

If you have a previous major version of AlterNET Studio, and decide to install the new one side-by-side, you will have two sets of Visual Studio Extensions and two sets of tabs, each one clearly displaying version number.

Getting Started

Once the product is installed, it might be good idea to explore quick start projects and Script Debugger demo first, either by compiling Alternet.Studio.AllDemos.Wpf.sln solution or accessing these demos through Demo Explorer tool which is added to Windows Start menu.



AlterNET Studio

This demo shows how all products can be integrated in one solution - from designing visual interface using Form Designer, to writing application logic in powerful Code Editor and executing and debugging this code with Scripter.

More Information:

AlterNET Studio provides a set of component libraries that allow users of your applications to write scripts in C#/Visual Basic programming languages to enhance existing or define new logic; run and debug these scripts with a fully-featured debugging engine and design visual user interfaces.

```
    {
        int _hello = 77;


        int InstProp { get { return 11; } }

        int MyFunc()
        {

            return 1;
        }

        public string InstanceMethod(int xx)
        {
            int bb = xx + MyFunc();
            return bb.ToString();
        }

        public string InstanceMethod(int xx, int y)
        {
            int bb = xx + MyFunc() + y;
            return bb.ToString();
        }
    }
```

AlterNET Studio 7.0

Below is brief overview of these projects:

AlterNET Studio– demonstrates how to run and debug script files and projects. Example files and projects are located in the demo\resources\scripter folder.

CallMethod - demonstrates how to execute script methods and pass application objects to the script.

CustomAssembly – demonstrates how to use external assemblies in the scripts

EvaluateExpression – shows how ScriptRun can evaluate expression, which again can access some objects defined in the application

Isolated Script - Shows how to load script in the separate AppDomain, so it can be unloaded afterwards and execute methods in it.

Object Reference – shows how application-defined objects can be accessed by bane from the script.

Threading – shows how scripts can be run asynchronously.

DebugWpfScript – demonstrates how scripts executed by application can be debugged by a separate Script Debugger tool.

DebuggerIntegration – Shows how debugger logic can be integrated in the application to debug application-independent scripts.

DebugRemoteScript – Shows how debugger logic can be embedded in the application to debug application-independent scripts or ones accessing application API indirectly.

Once you've done with demo projects, it's time to run the Hello World script yourself.

Essential steps are: Create *ScriptRun* component in a context of your WPF form, and write the following code in Button click event handler:

```
ScriptRun scriptRun1 = new ScriptRun();
scriptRun1.ScriptSource.FromScriptCode("public class ScriptTest { public static void
Main() { System.Windows.Forms.MessageBox.Show(\"Hello World \");} }");
scriptRun1.ScriptSource.WithDefaultReferences();
scriptRun1.Run();
```

The first line of the code creates an instance of *ScriptRun* component, the second populates Script source, and the third line adds references to most common System assemblies, and the third one runs the code.

Under the bonnet

The basic script execution workflow requires setting a script source, adding references to the assemblies used in the script; registering application-defined objects accessible to the script; compiling script to dynamically-linked library or standalone executable program and running some method in that dll or executing the program.

Setting up Script Source

All properties and methods required to set a script source are encapsulated in *ScriptSource* property of the *ScriptRun* class; below are the most essential ones:

Files - specifies collection of source files to be compiled and executed;

ScriptCode – specifies source in a form of text string;

ProjectName, *ProjectFileName* and *RootNamespace* - contain project-related information if *ScriptSource* is loaded from the project.

Imports - contains global namespaces in case Visual Basic is used so you do not need to specify them in the code;

Conditionals – contains lists conditional compilation symbols;

References - contains a list of assembly references for types used in the scripts; this can include reference to the calling application.

SearchPaths – contains search paths to look for the third-party references in case they're not supplied with a full path.

WpfResources- contains a list of XAML files.

FromScriptFile – loads Script Source from the single source file;

FromScriptCode – loads script from code in a form of a text string;

FromExpression – sets ScriptSource to the string expression.

FromScriptProject – loads code from Visual Studio Project

Adding assembly references:
In order to use types in the script, assemblies where these types are declared need to be properly referenced.

The following code populates references with most commonly used assemblies:

```
scriptRun1.ScriptSource.WithDefaultReferences();
```

In case technology is set to WPF, the references are populated with default WPF assemblies:

System, System.Core, System.Xaml, System.Xml, WindowBase, PresentationCore and *PresentationFramework.*

You can reference additional assemblies by adding it to the References property:

```
scriptRun1.ScriptSource.References.Add("System.Data");
```

This method accepts full path, you can also add reference to third-party assemblies in case script uses types from it.

Registering objects to be used in script
Application objects accessible by the script need to be added to the GlobalItems collection, along with the object's name which will be used in the script and object's type or object itself.

Object value itself is only required during script execution; for script compilation object name and type are sufficient.

ScriptRun adds references to the assemblies which contain types of the objects being added to *GlobalItems* automatically.

Please note that *AssemblyKind* property needs to be set to Dynamically Linked Library, for it to be loaded in the running application process and be able to access application-defined objects.

Below is sample code which registers application-defined objects in the script.

```
public class MyItem
  {
     public MyItem(string text)
     {
        this.Text = text;
     }
     public string Text;
  }
 scriptRun1.GlobalItems.Add(new ScriptGlobalItem("MyItem", obj: new MyItem("hello")))
```

Script Compilation and Execution

Once *ScriptSource* is set, next step is Compile the script; this step is performed implicitly when the script is run the first time, or when Script source is changed (this includes changes of script files externally)

Script compilation engine is implemented by *ScriptHost*; there are two implementations of *ScriptHost* provided, a legacy engine based on *CodeDOM* wrapper around command-line C# or Visual Basic compiler, or *RoslynScriptHost* based on new Microsoft Roslyn Code compiler technology – the last one is used by default and it allows some nice features such as referencing to other script source dynamically by using #load directive and gives more control on code parsing and compilation.

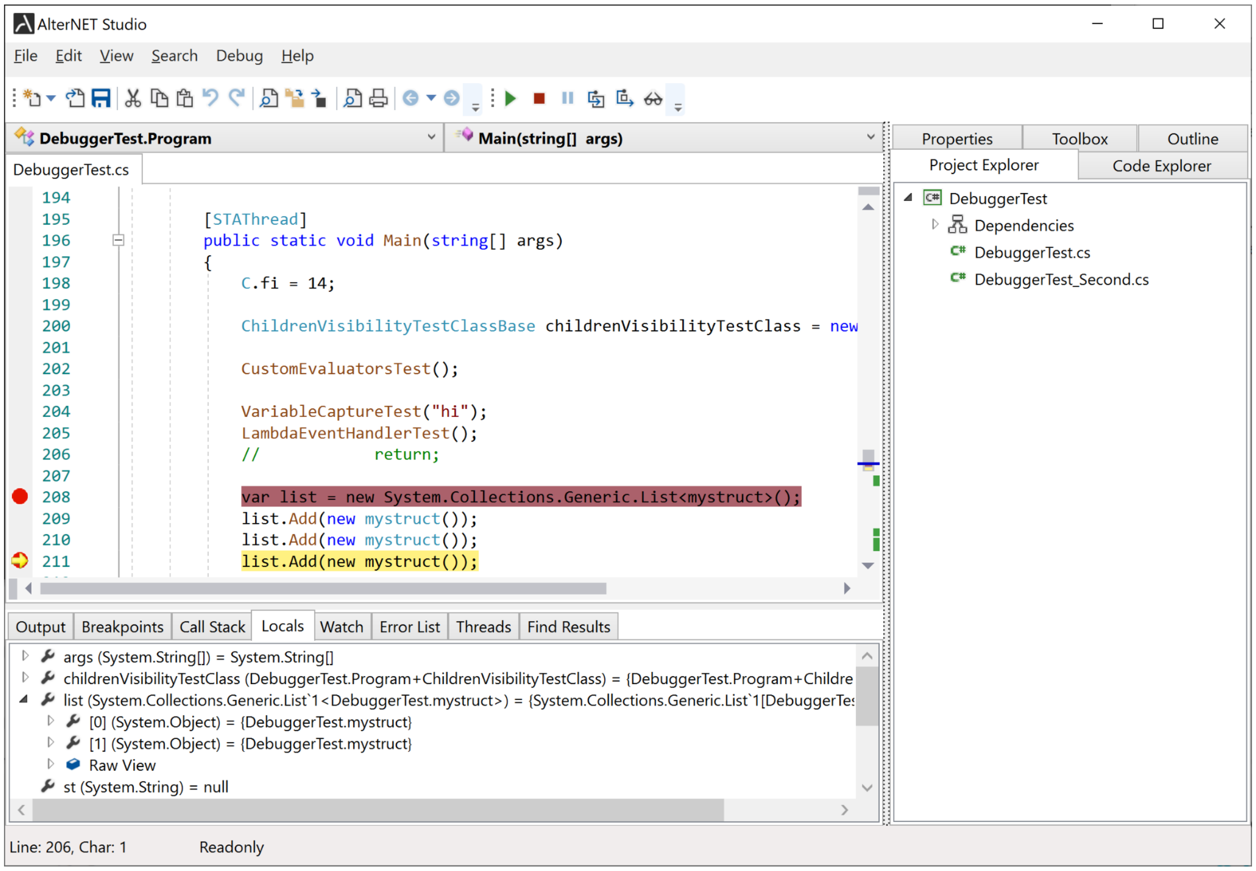
Script can be compiled into a dynamically-linked library or in a standalone executable; this is controlled by the *AssemblyKind* property. *ScriptHost.GenerateModulesOnDisk* allows to control whether assembly being compiled will reside in memory or on the disk; and *ScriptHost .ModulesDirectoryPath* specifies location of compiled assembly where compiled modules will be stored. Platform target (AnyCPU, AnyCpu32BitPreferred, x86, x64 or Auto) is controlled by *Platform* property (by default it's set to Auto and takes target platform from the application).

Once Compilation is executed, Compiled property will be set to true in case compilation was successful, and *ScriptHost's ScriptAssembly* property will point to the assembly being compiled from the script source. Otherwise *ScriptHost's properties CompileFailed* will be set to true and *CompilerErrors* will be populated with compiler errors. Please note, CompilerErrors may contain compiler warnings even in case of successful compilation.

Upon successful compilation you can subsequently call *Run*, *RunMethod*, or their asynchronous variants: *RunAsync* and *RunMethodAsync*; in case of standalone executable *RunProcess* should be used instead.

Script Debugging

Script writing is not complete fun without ability to debug what you write. We provide tools to debug script code and a set of UI widgets to build custom debugging interfaces.



```
        [STAThread]
        public static void Main(string[] args)
        {
            C.fi = 14;

            ChildrenVisibilityTestClassBase childrenVisibilityTestClass = new

            CustomEvaluatorsTest();

            VariableCaptureTest("hi");
            LambdaEventHandlerTest();
            //          return;

            var list = new System.Collections.Generic.List<mystruct>();
            list.Add(new mystruct());
            list.Add(new mystruct());
            list.Add(new mystruct());
```

```
args (System.String[]) = System.String[]
childrenVisibilityTestClass (DebuggerTest.Program+ChildrenVisibilityTestClass) = {DebuggerTest.Program+Childre
list (System.Collections.Generic.List`1<DebuggerTest.mystruct>) = {System.Collections.Generic.List`1[DebuggerTe:
    [0] (System.Object) = {DebuggerTest.mystruct}
    [1] (System.Object) = {DebuggerTest.mystruct}
    Raw View
st (System.String) = null
```

Line: 206, Char: 1   Readonly

Script Debugger engine is implemented in Alternet.Script.Debugger assembly and it is based on on CLR debugging COM interfaces low-level API to debug .NET applications.

https://msdn.microsoft.com/en-US/library/ms404484(v=vs.110).aspx

Main component of Script debugging is the ScriptDebugger class, which provides all commonly used debugging features like step by step execution, stopping on breakpoints, examining local variables, expression evaluations, etc.

Below is a summary of *ScriptDebugger* most essential properties, methods and events:

Methods:

StartDebugging – starts executing the program from the entry point.

AttachToProcess – attaches to the already started process which scripts are to be debugged.

StopDebugging – Stops debugging session.

Break – Causes the given process to pause its execution so that its current state can be analyzed.

Continue – Continues given process to the next breakpoint or until process finishes.

StepInto – Executes one statement of code; steps into the next function call, if possible.

StepOver – Executes one statement of code; steps over the next function call, if possible.

StepOut – Executes remaining lines of the function; steps out of the function currently being executed.

ActivateThread – Switches debugging to the specified thread.

SwitchToStackFrame – switches debugging to the given stack frame.

SetRunToPositionBreakpoint - causes debugger to stop at the specified position.

Following methods may take considerable amount of time, therefore they're implemented asynchronously:

EvaluateExpressionAsync – evaluates expression in the current stack frame, with or without child properties.

EvaluateCurrentExceptionAsync – evaluates exception being thrown by debugger.

GetStackFramesAsync – gets a list of method calls that are currently on a stack.

GetThreadsAsync – gets a list of active threads.

GetVariablesInScopeAsync – gets all local variables in the given stack frame.

TrySetNextStatementAsync – sets the execution point to the specified line.

GetExecutionPositionAsync – gets the current execution point.

Properties:

IsStarted – indicates whether the debug process has started.

State – gets current debugger state.

ScriptRun – in case Debugger used to debug standalone executable, contains all information required to compile and run the script.

GeneratedModulesPath – Specifies directory where assemblies for the scripts being debugged are located.

Breakpoints – returns collection of debugger breakpoints.

EventsSyncAction – A function which could be provided by the application to sync raised debugger events if required (for example, perform Control.Invoke)

***Events*:**

ActiveThreadChanged – occurs when thread to be debugged changes.

DebuggerErrorOccured – occurs when debugger encounters error during debugging session.

DebuggingStarted – occurs when debugging session is started.

DebuggingStopped – occurs when the debugging session is stopped.

ExecutionResumed – occurs when debugging is resumed after being paused.

ExecutionStopped – occurs when debugging is paused.

LogMessageReceived – occurs when a debug message is received.

StackFrameSwitched – occurs when the debugger is switched to the stack frame.

StateChanged – occurs when debugging state is changed (when debugger is started, stopped or paused)

Script Debugging best practices

The main issue that we've faced with debugging is that it's not quite possible to embed debugging logic in the same process where scripts are being executed, as the debugger process will need to freeze itself when debugging. Please refer to the following blog for more details:

https://blogs.msdn.microsoft.com/jmstall/2005/11/05/you-cant-debug-yourself/

Therefore we see two main options for script debugging to work:

1. Script is compiled as a dynamically linked library and is linked to the calling application (which is the most straightforward way of scripts to be able to access application-defined objects). In this case Script debugger must be a separate process which attaches to the main application process and allows to debug script code in it. The script debugger can be made look like it belongs to the same application (which is outside of the scope of this tutorial), but it has to be in a separate process.

   In this mode Script Debugger does not compile or execute script itself; instead it relies on the main application to do so. It receives the main application process id, source and project file along with the name of assembly to be debugged via command-line arguments; attaches to the main process and communicates with it send Start Debug or Stop Commands and to receive a list of compilation errors or script completion events.

   Please refer to DebugWpfScript quickstart projects for more details.

   Please note that you cannot debug the main application under Visual Studio and have Script Debugger to attach to it at the same time, as Visual Studio will attach its own debugger.

   Please also note that the target platform of debugger and debugee process need to be the same (for ScriptDebugger demo it's set to AnyCPU, 32-bit preferred).

2. Script to be compiled in the separate executable; and debugging logic is embedded in the application itself. This option requires either script to be application-independent (which is not useful if scripts are intended to extend application logic), or access application-defined objects via interprocess-communication. Please refer to our DebuggerIntegration/DebugRemoteScript quickstart projects for more details.

Yet another option would be the one mentioned in the blog, which is not currently implemented, but we've done some experiments in it. In essence Script Debugger UI can be embedded in the main application, while script being debugged will be executed in the separate thread. This will also require putting the debugger engine in a hidden proxy process and have debugger UI to communicate with this process. We might bring this feature in the subsequent releases provided that we can make it work reliably; however it will impose some limitations to the script writers (such as the script itself will need to be thread-safe in order to be able to run in the separate application thread).

Script Debugging Widgets

We've developed a set of debugger widgets, with the intent to make it easier customizing Script Debugger tool appearance so it matches the theme of the application which scripts are to be debugged.

These widgets are implemented both for WinForms and WPF; these widgets and a set of demo projects represent the difference between WinForms and WPF editions, as the *ScriptRun* and *ScriptDebugger* themselves are non-visual and therefore platform-agnostic.

These widgets include:

Output – to log debugger events or application-specific messages

Errors – to display a list of compilation errors.

Breakpoints – to display and navigate through the list of breakpoints set in the source;

CallStack – to display and navigate through the list of method calls that are currently on stack.

Locals – to examine values of local variables once debugging code step-by-step.

Watches – to examine values of watch expressions when debugging

Threads – to display active threads and switch debugging between them.

DebuggerControlToolbar - a toolbar with buttons executing Run/Stop/StepInto/StepOver commands.

Demo projects include TextEditor- based code editor with syntax highlighting and ability to display breakpoints and tracing styles and auxiliary code for displaying project tree, code explorer and navigating through methods and classes defined in the source.

TypeScript/JavaScript support.

Alongside with component libraries for .NET-based script compilation, execution and debugging, we provide a very similar package for TypeScript/JavaScript.

Script execution is based on Microsoft ClearScript which provides v8 high-performance open-source JavaScript engine. It supports executing JavaScript code and accessing .NET types and objects of the host application from the script.

TypeScript ScriptRun provides a very similar interface to .NET ScriptRun; the main difference is that it does not create .NET assembly, and execute JavaScript code using the ClearScript engine.

The main difference in API is that unlike .NET Script Runner, the collection of referenced objects, types and .NET assemblies is specified via HostItemsConfiguration property; as opposed to

GlobalItems/References properties; *RunMethod/RunMethodAsync* are replaced with *RunFunction/RunFunctionAsync.*

The following code adds references to most commonly used assemblies and registers RunButton to be accessible from the script:

```
  scriptRun.ScriptHost.HostItemsConfiguration.AddSystemAssemblies()
     .AddObject("RunButton", btNETFromScript);
```

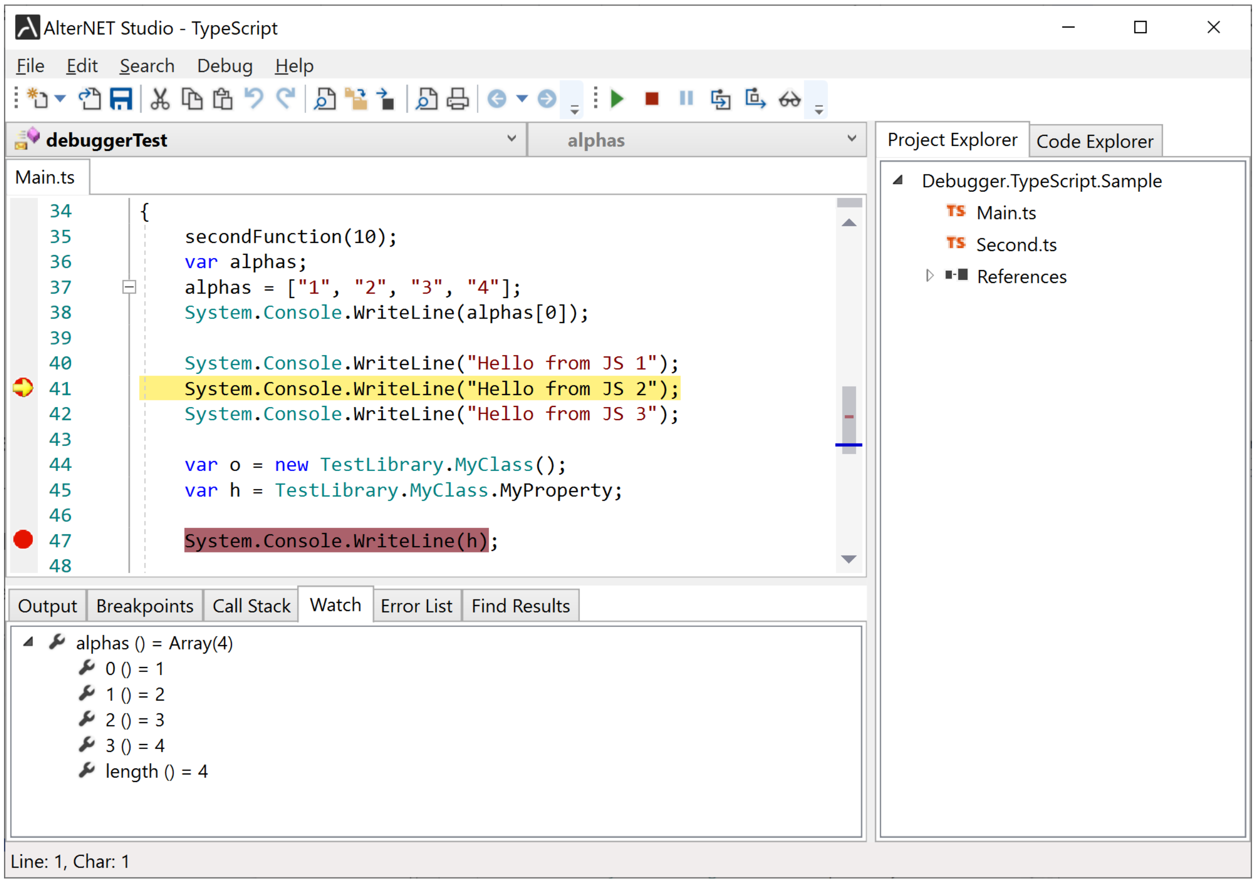
Like .NET ScriptRun, TypeScript Runner can execute single files, typescript projects (which can be loaded/saved to json file), or evaluate TypeScript/JavaScript expressions.

Note that order of TypeScript/JavaScript files in a project is important, as they get executed one by one.

TypeScript compilation service uses host configuration to automatically create all support files containing typescript definitions. The following line needs to be placed on top of user's script to access .NET types and objects from host configuration:

```
///<reference path="clr.d.ts" />
```

TypeScript ScriptDebugger is based on Google Chrome debugging development tools; it does not have a limitation of debugger and script to be debugged running in the separate application processes. It has most of the functionality that .NET Script debugger provides; except for multi-threaded debugging and automatic retrieval/evaluation of local variables.



```
{
    secondFunction(10);
    var alphas;
    alphas = ["1", "2", "3", "4"];
    System.Console.WriteLine(alphas[0]);

    System.Console.WriteLine("Hello from JS 1");
    System.Console.WriteLine("Hello from JS 2");
    System.Console.WriteLine("Hello from JS 3");

    var o = new TestLibrary.MyClass();
    var h = TestLibrary.MyClass.MyProperty;

    System.Console.WriteLine(h);
```

```
alphas () = Array(4)
    0 () = 1
    1 () = 2
    2 () = 3
    3 () = 4
    length () = 4
```

IronPython support

Another package that implements script compilation, execution and debugging, is IronPython ScriptRun and ScriptDebugger components.

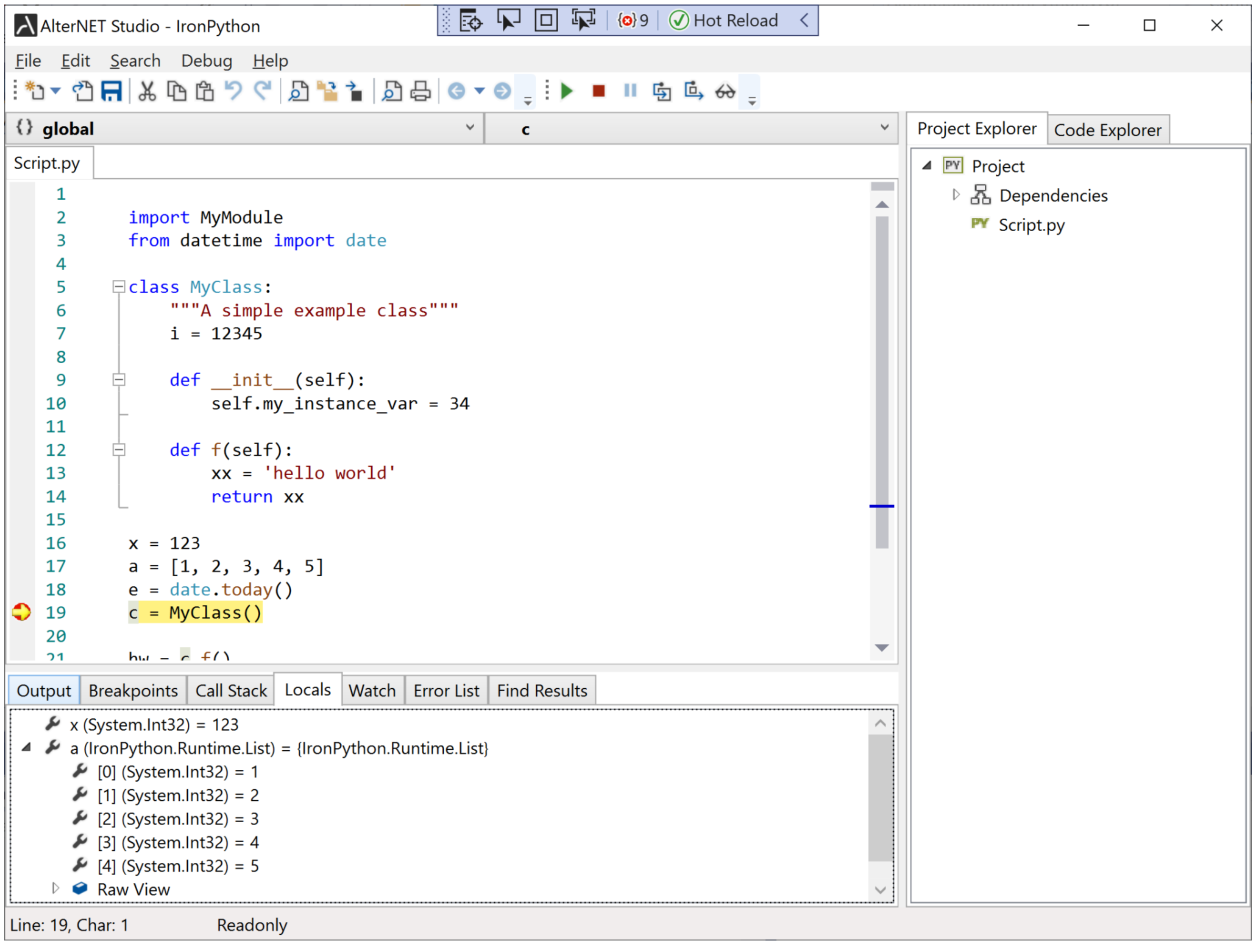
These components are installed on the AlterNet Scripter.IronPython tab in Visual Studio.

Script execution is based on IronPython.net script engine which is an open-source implementation of the Python programming language, tightly integrated with .NET. It supports executing Python code and accessing .NET types and objects of the host application from the script.

IronPython ScriptRun provides a very similar interface to .NET ScriptRun; the main difference is that it does not create .NET assembly, and executes Python code IronPython scripting engine.

Like .NET ScriptRun, IronPython ScriptRun can execute single files, typescript projects (which can be loaded/saved to .pyproj file), or evaluate Python expressions.

IronPython Script Debugger is based on Microsoft.Scripting debugging engine; it does not have a limitation of debugger and script to be debugged running in the separate application processes. It has most of the functionality that .NET Script debugger provides; except for multi-threaded debugging..



```
import MyModule
from datetime import date

class MyClass:
    """A simple example class"""
    i = 12345

    def __init__(self):
        self.my_instance_var = 34

    def f(self):
        xx = 'hello world'
        return xx

x = 123
a = [1, 2, 3, 4, 5]
e = date.today()
c = MyClass()
```

```
x (System.Int32) = 123
a (IronPython.Runtime.List) = {IronPython.Runtime.List}
    [0] (System.Int32) = 1
    [1] (System.Int32) = 2
    [2] (System.Int32) = 3
    [3] (System.Int32) = 4
    [4] (System.Int32) = 5
    Raw View
```

Line: 19, Char: 1   Readonly

Integration with other AlterNET Studio components.

Script writing requires an editor of some sort to allow users to write the script code. The AlterNET Studio package includes Code Editor specialy tailored to edit C#, Visual Basic, TypeScript, JavaScript and Python code.It also provides some additional features like expression evaluation when hovering mouse over symbols in debug mode. However you're not required to use it; you can choose any other external text editing components instead. Our demos are written the way it should be relatively easy to integrate them with other text editors by implementing *IScriptEdit* and *IDebugEdit* interfaces.

If you'd like to make the scripting engine in your application a bit more advanced, you might consider giving users the ability to write simple UI for their scripts; such as design option dialog that would control certain behaviors of the script itself. We have developed a Form designer product, which along with the Scripter and Code Editor form our AlterNET Studio solution.

Please refer to our AlterNET Studio demo to see how Code Editor, Form Designer and Scripter/Debugger work together.

Licensing

We require a valid license to be installed for developing with AlterNET software products. We supply evaluation-licenses upon AlterNET Studio installation; these licenses are based on licx files technology provided by Microsoft and are valid for 30 days since first use.

Upon ordering a paid version of our product, you will receive a License key and will be able to activate it on your computer. This key will support a number of activations and it's not transferable between development machines otherwise.

Below is more information about Microsoft license compiler and some discussions related to intent and purpose of licx files, using them with source code control systems, etc.

https://msdn.microsoft.com/en-us/library/ha0k3c9f(v=vs.110).aspx

http://stackoverflow.com/questions/5628969/how-licenses-licx-file-is-used

http://stackoverflow.com/questions/51363/how-does-the-licenses-licx-based-net-component-licensing-model-work

Since *ScripRun* or *ScriptDebugger* are non-visual components, you can't drag them onto your WPF form; but in order for them to function correctly, you need to create a licx file with the following content and add it to your project under Properties folder:

Alternet.Scripter.ScriptRun, Alternet.Scripter.v7, Version=7.1.0.0, Culture=neutral, PublicKeyToken=8032721e70924a63

In case you create a ScripRun or *ScriptDebugger* component from code, please make sure licx file with such content is added to your project – you can use one from one of our demo projects if needed.

The design-time license is being checked when you work with this component at design-time, or when you compile your project; and the nag screen reminding you about the evaluation mode will appear once in a while. If you run a project compiled with an evaluation version of Code Editor (without launching it from Microsoft Visual Studio debugger), you will see a screen suggesting that the application was created with an evaluation version of Code Editor. Once you activate a paid license using LicenseActivation tool, nag screen will no longer appear.

When the evaluation period expires, you will still be able to compile and run your application from within Visual Studio, however applications created with expired license will not be run in standalone mode.