Scripting Introduction Walkthrough

One of the key features Roslyn adds for C# Interactive is the ability to execute snippets of C# code in a host-created execution environment. This walkthrough shows a C# application that uses the Roslyn Scripting APIs to let the application’s users define command handlers and affect the object model of the host application. The application is a simple WPF window with a colored background and two buttons whose command handlers are supplied by hosted C# code. The object model of the application has a background color property and an AddCommand method.

This walkthrough shows you how to:

* Use the Roslyn Scripting APIs to create a C# script engine.
* Create a Session that holds the cumulative execution context for script code.
* Execute C# code in a host-supplied scope.
* Push top-level function definitions as command implementations.
* Add default references and namespace 'using's for C# script code allowing script code to access the host’s extensibility DLL and namespaces.

Open the ScriptingIntro project in Visual Studio, which installed in your Documents folder as "Microsoft Roslyn CTP - September 2012\csharp\ScriptingIntro\ScriptingIntro.csproj".

Looking at MainWindow.xaml.cs (which is under MainWindow.xaml in the Solution Explorer), you can see the code has namespace 'using's to be able to access Roslyn Scripting APIs:

using Roslyn.Compilers;

using Roslyn.Scripting.CSharp;

using Roslyn.Scripting;

This project defines a WPF window with two buttons on it. HostObjectModel.cs defines a host object model to expose functionality to script code:

// HostObjectModel is the type of object we put in the Interactive Session

// for script code. Script code binds free identifiers to public members

// on this type.

//

public class HostObjectModel

{

// Need to create the host object model instance with the WPF main window

// so that we can call back on it from script code.

//

private MainWindow mainWindow = null;

public HostObjectModel(MainWindow window)

{

this.mainWindow = window;

}

// BackgroundColor is an example of host functionality exposed to script

// code. The host could just expose mainwin and let script code party

// all over the main window, or the host can expose functionality

// selectively.

//

public Color BackgroundColor

{

get { return ((SolidColorBrush)this.mainWindow.Background).Color; }

set { mainwin.Background = new SolidColorBrush(value); }

}

// Commands holds a dictionary of named commands. Script code can add

// commands to the hot application. The host can invoke commands

// attached to buttons via name lookup.

//

internal Dictionary<string, Action> Commands =

new Dictionary<string, Action>();

public void AddCommand(string name, Action commandImplementation)

{

Commands[name] = commandImplementation;

}

} // class HostObjectModel

Navigate to MainWindow.xaml.cs. Notice the constructor of MainWindow adds a common Button.Click handler to each button. The common handler invokes a delegate from a dictionary depending on the button/command name:

this.button1.Click += new RoutedEventHandler(this.ButtonClick);

this.button2.Click += new RoutedEventHandler(this.ButtonClick);

The ButtonClick handler defined in the MainWindow class uses the host's object model to look up commands in the dictionary:

private void ButtonClick(object sender, RoutedEventArgs e)

{

var name = ((Button)sender).Name;

if (hostObj.Commands.ContainsKey(name))

{

hostObj.Commands[name]();

}

else

{

MessageBox.Show("No command implementation for " + name + ".");

}

}

The MainWindow constructor creates a C# script engine and a Session. ScriptEngines represent the language and execution semantics, plus hold onto some general context such as references for compilation or seeding new Sessions. Sessions represent cumulative execution context for a group of definitions (variables, functions, types, etc.) that can all work together, but you can add the definitions to the Session incrementally or all at once.

A Session also supports redefining variables, functions, types, etc., by simply executing new definitions. If you redefine a name, then executing new code snippets that reference the name bind to the new definition of the name. A Session can also hold onto an instance of the host object model so that free identifiers in hosted script code can bind to accessible members on the host object to interact with the host. The sample code below adds references to the engine. Then the sample code creates the session, which inherits the engine’s references, and passes an instance of the host’s object:

var engine = new ScriptEngine();

// Add references

engine.AddReference("System");

engine.AddReference("PresentationCore");

engine.AddReference(this.GetType().Assembly.Location);

// Create the host object model and seed the Session with it.

hostObj = new HostObjectModel(this);

var session = engine.CreateSession(hostObj);

The MainWindow constructor defines a couple of baked-in command handlers, but you can imagine that these came from a user's script file or a text box in the UI:

// Imagine host gets file names from options and executes code

// from those files to customize the app.

const string button1Implementation = @"

using System.Windows.Media;

void b1 () { BackgroundColor = Colors.Red; }

AddCommand(""button1"", b1);";

engine.Execute (button1Implementation, session);

// Note, we do not need the 'using' here due to cumulative

// execution context.

const string button2Implementation = @"

void b2 () { BackgroundColor = Colors.Chartreuse; }

AddCommand(""button2"", b2);",

engine.Execute(button2Implementation, session);

You can hit F5 to launch the application and verify clicking buttons now changes the background color via interactive code using host's object model.

That's it! You just saw a simple C# application that uses the Roslyn Scripting APIs to let the application’s users define command handlers and affect the object model of the host application. Please play around with the Roslyn Scripting APIs and provide feedback. See the [Roslyn Scripting API document](http://www.codeplex.com/Download?ProjectName=dlr&DownloadId=294856) for features we're planning but have not yet implemented, but there is a lot more work to do on the scripting APIs that is not yet reflected in the document.