This post was republished to Eric Erhardt at 3:00:39 PM 11/19/2014

Creating a Connected Service Extension

The Visual Studio Connected Services feature is a great way for developers to connect their applications to services that run in the cloud, or on premises. The Connected Services feature can automate the multiple steps it takes to connect a Visual Studio project to a service.

The great thing about the new Connected Services experience is that it is fully extensible. Anyone can build a VS extension that adds a new Connected Service Provider into the dialog. This new provider could enable any Visual Studio application to connect to any service. In this article, I’m going to explain to you how to get started building your own Connected Service extension.

With a custom Connected Service extension, you will have the full power of the Visual Studio extensibility APIs to perform modifications to a developer’s project. This will allow you automate any steps a developer normally has to do manually to connect an application to the service.

### Definitions

To get started, I am going to use some terms that I want to define up front so there is no confusion.

* **Provider** – A Connected Service Provider provides the data that is shown in the Connected Services dialog. Its main responsibility is to authenticate to the service, retrieve any necessary data, and allow users to make choices on how they want to consume the service.
* **Handler** – A Connected Service Handler is responsible for taking any data and choices from the Provider and modifying the project to consume the selected service. Multiple Handlers can be associated with single Provider. For example, one Handler can be written for WinForms and WPF projects, while another is written to support Web projects.
* **Instance** – A Connected Service Instance is the data that is passed from the Provider to the Handler.

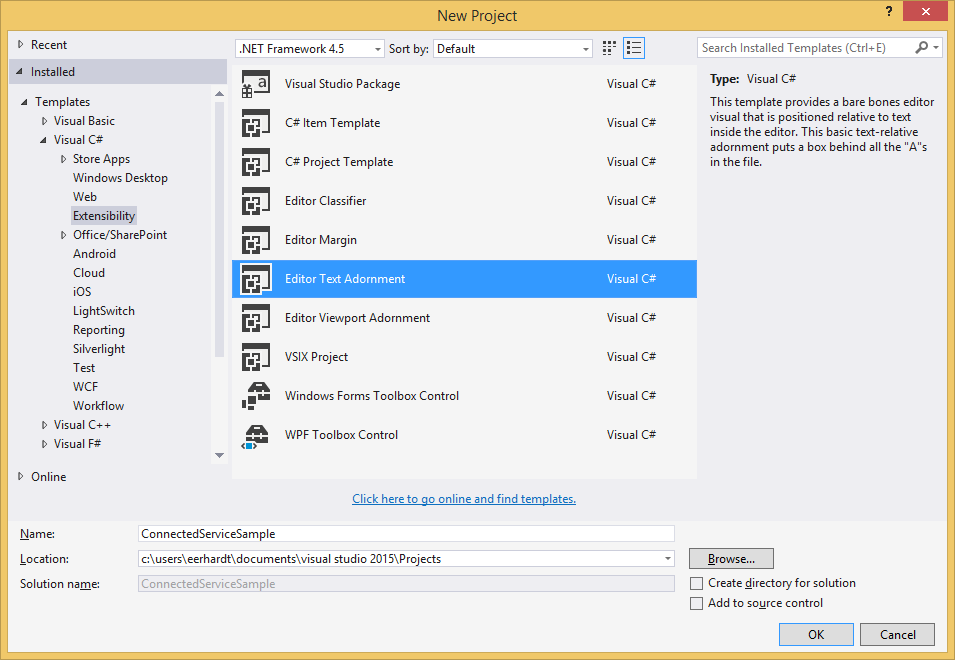
## Writing an extension

Visual Studio uses the [Microsoft Extensibility Framework](http://msdn.microsoft.com/en-us/library/dd460648) (MEF) to load extensions. Code inside Visual Studio creates an extension point by defining a contract (think .Net interface), and “Imports” the contract. Then anyone who wants to plug into that extension point just has to “Export” a component that adheres to the contract. Then at runtime, Visual Studio finds all the components that adhere to the contract, and invokes them.

The Connected Services feature in Visual Studio has done just that. It defines an extension point and a contract: IConnectedServiceProvider. To create your own extension to Connected Services, you need to provide an implementation of the IConnectedServiceProvider interface and Export it.

To do this, you first need [VS 2015 Preview](http://www.visualstudio.com/en-us/downloads/visual-studio-2015-downloads-vs) and the [VS 2015 Preview SDK](http://www.microsoft.com/en-us/download/details.aspx?id=44932) installed.

With the VS 2015 SDK installed, you should get new ‘Extensibility’ project templates:



Choose the “Editor Text Adornment” template, which will create a new project that can extend Visual Studio, and name it “ConnectedServiceSample”. (Although we are not creating an Editor Text Adornment extension, we are using this template as a starting point because it sets up almost everything we need in the project.)

You can delete the two code files “ConnectedServiceSample.cs” and “ConnectedServiceSampleFactory.cs” from the Solution Explorer. We won’t be needing them.

In the source.extension.vsixmanifest file, fill in the ‘Author’ field to remove any build errors. You can optionally change the Description of the extension in the manifest.

Now you have a Visual Studio extension that can export MEF components to add functionality in VS.

### Add References

The next thing you need to do to your project is add a reference to the Connected Services “contracts” assembly. This assembly contains the interfaces that provide the contract between the core Connected Services feature, and any extensions.

* Copy the Microsoft.VisualStudio.ConnectedServices.dll assembly to your machine.
* In VS, right click on the References node under your project and “Add Reference”.
* Browse to the Microsoft.VisualStudio.ConnectedServices.dll assembly.

### Create an IConnectedServiceProvider

We can now create a new class that implements and exports the IConnectedServiceProvider interface. I have provided a sample class below:

using Microsoft.VisualStudio.ConnectedServices;

using System;

using System.ComponentModel.Composition;

using System.Threading.Tasks;

using System.Windows.Media;

using System.Windows.Media.Imaging;

namespace ConnectedServiceSample

{

[Export(typeof(IConnectedServiceProvider))]

[ExportMetadata("ProviderId", "ConnectedServiceSample.ConnectedServiceProvider")]

internal class ConnectedServiceProvider : IConnectedServiceProvider

{

public string Name { get { return "Sample Grid Provider"; } }

public string Category { get { return "Sample"; } }

public string CreatedBy { get { return "Contoso, Inc."; } }

public string Description { get { return "Sample Provider with Grid functionality."; } }

public Version Version { get { return new Version(1, 0, 0); } }

public Uri MoreInfoUri { get { return new Uri("https://github.com/SteveLasker/ConnectedServicesCustomProviderSamples"); } }

public ImageSource Icon

{

get

{

return new BitmapImage(new Uri("pack://application:,,/" +

this.GetType().Assembly.ToString() + ";component/Image.png"));

}

}

public Task<object> CreateService(Type serviceType, IServiceProvider serviceProvider)

{

return null;

}

}

}

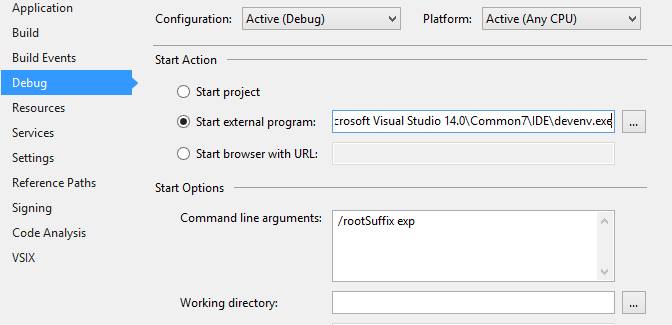
As you can see, the class implements the IConnectedServiceProvider interface and has an Export attribute for that type. We’ve listed one piece of ExportMetadata: the ProviderId. This string uniquely identifies the provider and is used to reference the provider. When we create a Handler, you will see why this is important.

The class fills out the interface with sample strings and a sample image that is included in the project as a “Resource”. You can return ‘null’ from the Icon property if you don’t have an image for testing, but it is recommended to have an image when you actually release your extension.

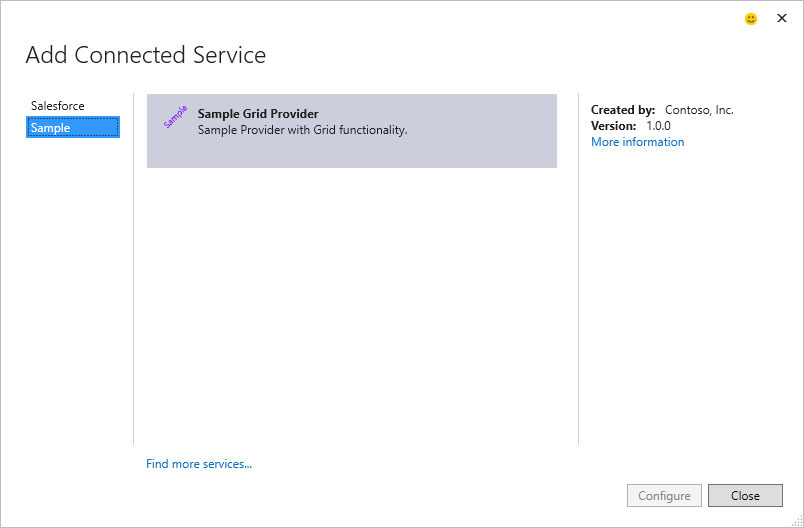
We will discuss the CreateService method later when we need to fill it out. For now, we will just return ‘null’.

This is the bare minimum information that Visual Studio needs to show your extension.

If you F5 your project, it will launch an Experimental Instance of Visual Studio. The Experimental Instance is a “sandboxed” Visual Studio where you can try out your extensions while they are in development, without affecting your real Visual Studio installation. If you are having problems with getting F5 to work, make sure the Debug settings in your startup project properties are set to the following:



In the Experimental Instance of Visual Studio, you can create a new Console Application project, and right-click the project in the solution “Add -> Connected Service…” and you will see your new provider in the dialog:



### Create an IConnectedServiceInstanceHandler

Now that you have a provider showing up in the Connected Service dialog, you have probably noticed that the “Configure” button is not enabled. This is because there is no available “Handler” that can apply this service to the current project. Since Visual Studio couldn’t find an available Handler, it decided that the current project is not supported and disabled the Configure button.

To enable the button, we will need to create a new Handler. Stop debugging the Experimental Instance, and add a new class to your VS extension project with the following code:

using Microsoft.VisualStudio.ConnectedServices;

using System.ComponentModel.Composition;

using System.Threading;

using System.Threading.Tasks;

namespace ConnectedServiceSample

{

[Export(typeof(IConnectedServiceInstanceHandler))]

[ExportMetadata("ProviderId", "ConnectedServiceSample.ConnectedServiceProvider")]

[ExportMetadata("AppliesTo", "CSharp+Web")]

internal class ConnectedServiceHandler : IConnectedServiceInstanceHandler

{

public Task AddServiceInstanceAsync(IConnectedServiceInstanceContext context, CancellationToken ct)

{

return Task.FromResult(true);

}

}

}

As you can see, the way Visual Studio finds Handlers is the same as how it finds Providers: it uses MEF. This class exports the IConnectedServiceInstanceHandler contract, allowing VS to find it when it searches for available Handlers. The first ExportMetadata attribute matches the ExportMetadata attribute from the Provider. This is how Providers and Handlers are matched up – a Handler specifies which Provider it corresponds to using the ProviderId attribute with the exact same value.

The second ExportMetadata attribute specifies which type of Visual Studio projects this handler supports. The value is a query string that specifies the project’s capabilities that are supported. A full query syntax is available. See the [AppliesTo](http://msdn.microsoft.com/en-us/library/dn497698.aspx) documentation on how to compose these query strings. The one used in the example says any C# Web project is supported. If you have a Console Application, or any VB project, the Configure button will still be disabled. But for C# Web projects, the Configure button will be enabled.

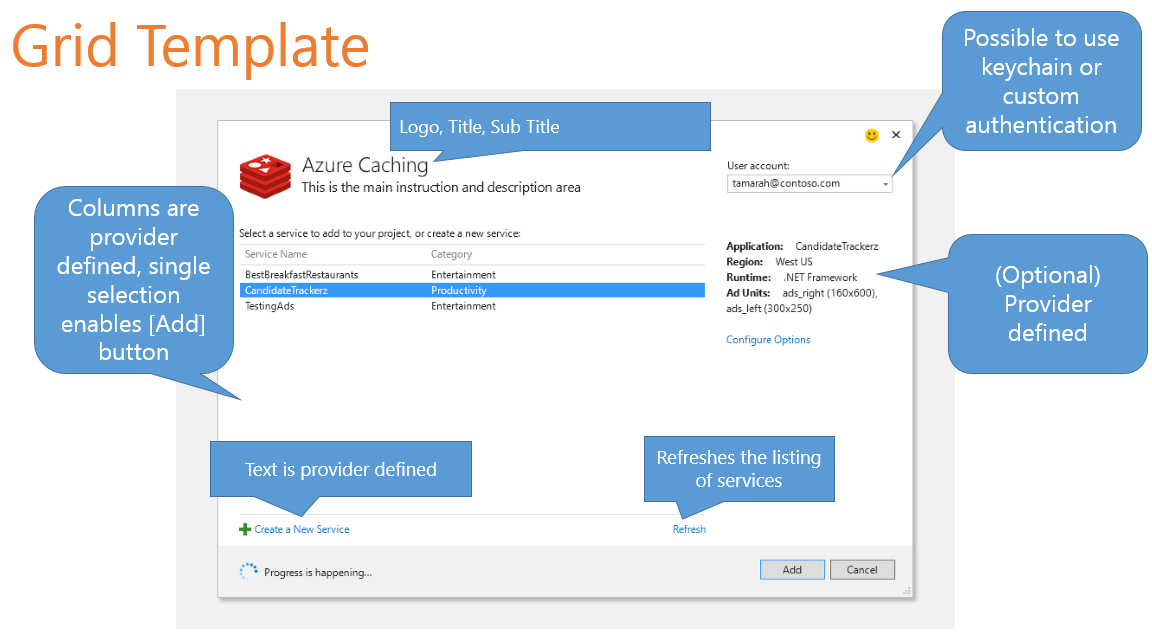
F5 again, create a C# Web project, open the Connected Service dialog, and you will see the Sample provider can now be configured.

### Filling out the Provider

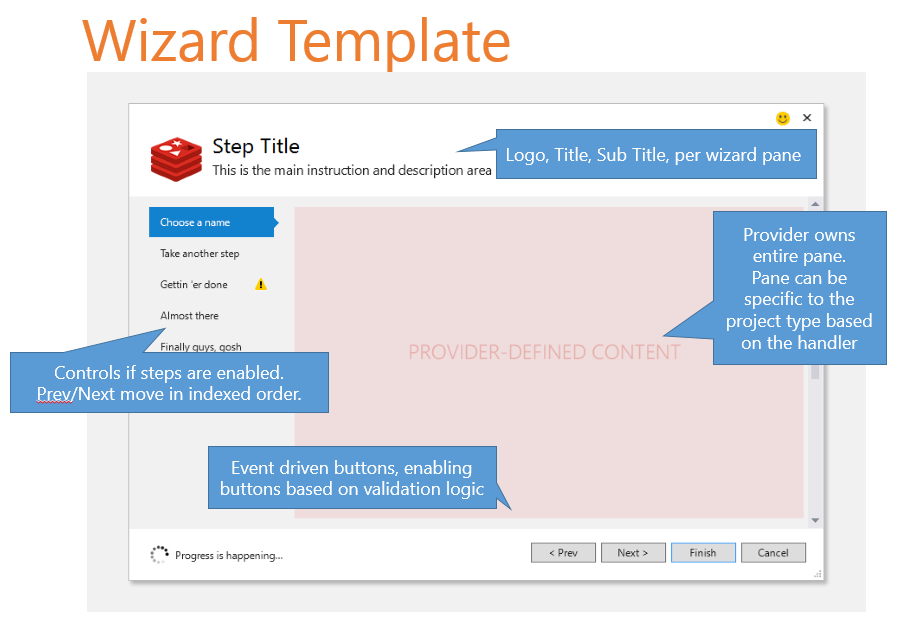
Now that the Configure button is enabled, you are probably going to want to click it. However, if you click it, you get an exception message:

The Connected Service Provider 'ConnectedServiceSample.ConnectedServiceProvider' returned an invalid IConnectedServiceProviderUI service from the CreateService method. A valid service should implement either IConnectedServiceProviderGridUI or IConnectedServiceProviderWizardUI interfaces.

This is where the “CreateService” method fits in. There are choices you can make about how you want your provider to be shown in the dialog. The first choice you have to make is whether you want to use the “Grid” or the “Wizard” template.



vs.



The mechanism for you to make these choices is by implementing the CreateService method. Visual Studio will call your CreateService method, passing in the Type of service it is looking for. The CreateService will either return null if the Provider opts out of making the choice, or it will return an object that implements the service type being requested. These services are cached for the lifetime of the dialog and, if the object implements IDisposable, they will get disposed when the dialog is closed.

As the message above indicates, the Provider must return a valid service for the Type: IConnectedServiceProviderUI. So let’s pick the Grid template and fill out the CreateService method to return an object that implements the IConnectedServiceProviderGridUI interface.

internal class ConnectedServiceProvider : IConnectedServiceProvider

{

...

public Task<object> CreateService(Type serviceType, IServiceProvider serviceProvider)

{

object service = null;

if (serviceType == typeof(IConnectedServiceProviderUI))

{

service = new ConnectedServiceProviderGrid();

}

return Task.FromResult(service);

}

}

internal class ConnectedServiceProviderGrid : IConnectedServiceProviderGridUI

{

public string EnumeratingServiceInstancesText { get { return null; } }

public string GridHeaderText { get { return null; } }

public string NoServiceInstancesText { get { return null; } }

public string ServiceInstanceNameLabelText { get { return null; } }

public string ShortDescription { get { return "A sample Connected Service"; } }

public IEnumerable<Tuple<string, string>> ColumnMetadata

{

get

{

yield return Tuple.Create("Column1", "Column1 Display");

}

}

public IEnumerable<Tuple<string, string>> DetailMetadata

{

get

{

yield return Tuple.Create("Detail1", "Detail1 Display");

yield return Tuple.Create("Detail2", "Detail2 Display");

}

}

public Task<IEnumerable<IConnectedServiceInstance>> EnumerateServiceInstancesAsync(CancellationToken ct)

{

return Task.FromResult<IEnumerable<IConnectedServiceInstance>>(

new[] {

new ConnectedServiceInstance("first", "1st column1", "1st detail1", "1st detail2"),

new ConnectedServiceInstance("second", "2nd column1", "2nd detail1", "2nd detail2"),

});

}

}

internal class ConnectedServiceInstance : IConnectedServiceInstance

{

private Dictionary<string, object> metadata = new Dictionary<string, object>();

public ConnectedServiceInstance(string name, string column1, string detail1, string detail2)

{

this.Name = name;

this.metadata.Add("Column1", column1);

this.metadata.Add("Detail1", detail1);

this.metadata.Add("Detail2", detail2);

}

public string Name { get; set; }

public string InstanceId { get { return this.Name; } }

public string ProviderId { get { return "ConnectedServiceSample.ConnectedServiceProvider"; } }

public IReadOnlyDictionary<string, object> Metadata { get { return this.metadata; } }

}

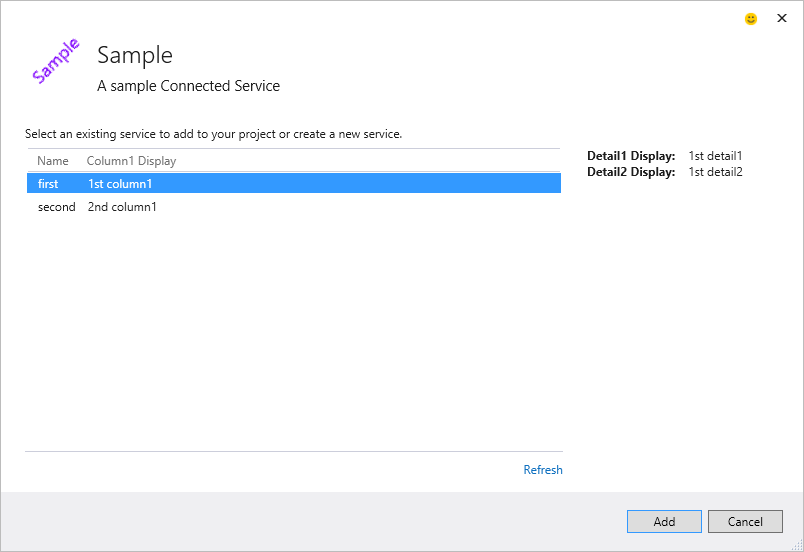
There are a few things happening in this code snippet. First, the CreateService method was implemented to check for the “UI” serviceType, and returns a new instance of the ConnectedServiceProviderGrid class. This class contains all of the logic necessary to populate the Grid template’s UI. There are some optional strings you can provide, if you want to override the default strings. Or you can return null if you want to use the defaults.

The next two properties give you the option to add custom columns to the grid control, and what detailed information you want shown to the right of the grid. You can opt out of either or both of these by just returning ‘null’. The first string in the Tuple is the “code name” for the column/property. The second string is what the property will be displayed as to the user, and should change based on the current culture.

The last method is the crux of the grid interface: EnumerateServiceInstancesAsync. This is the method that fills up the grid with the available rows of data. The method returns a Task, and takes a CancellationToken, so you can easily invoke HTTP requests, or other asynchronous operations, and use the “async” and “await” keywords in your code. For simplicity, I just instantiate new ConnectedServiceInstance objects with some hard-coded strings.

The last class is just a simple implementation of the IConnectedServiceInstance interface. One thing that needs an explanation is the Metadata property. This is where the data for the custom columns and the right-side detailed information comes from. The ConnectedServiceProviderGrid defines which properties should be shown, and gives the display names. The IConnectedServiceInstance’s Metadata dictionary contains the values of those properties. The properties are indexed using the first string in the ColumnMetadata and DetailMetadata Tuple collections from the IConnectedServiceProviderGridUI.

If you F5 your extension now, load a Web project, open the Connected Services dialog, and click the Configure button for the Sample provider you will see a grid filled out with the two rows.



A user can select one of the service instances, and click Add. When the Add button is clicked, the selected IConnectedServiceInstance object will get passed into the Handler that corresponds to the current project. The Handler is then responsible for modifying the project to be connected to the selected service.

This is the bare minimum needed in order to get a Provider shown in the Connected Service dialog. But there are other options you can choose by returning service objects from the CreateService method.

* If you use the Grid template, you can opt into:
  + Allowing the developer to authenticate to the service – IConnectedServiceAuthenticator
    - There are currently 2 sub-options:
      * IConnectedServiceHyperlinkAuthenticator
      * IConnectedServiceCustomAuthenticator
  + Allowing creation of a new service instance – IConnectedServiceInstanceCreator
  + Allowing the whole service to be configured - IConnectedServiceConfigurer
  + Allowing the selected service instance to be configured – IConnectedServiceInstanceConfigurer
* Or you can use your own UI by implementing the Wizard template - IConnectedServiceProviderWizardUI

### Filling out the Handler

Now that we have the UI filled out, and the user can select the service instance they want to connect to, it is time to fill out the Handler by modifying the project to actually connect to the service.

The Handler interface only contains one method: Task AddServiceInstanceAsync(IConnectedServiceInstanceContext context, CancellationToken ct). The method returns a Task, so if you need to write asynchronous code, you easily can with the “async” and “await” keywords. The context parameter contains all the input information provided to the Handler.

public interface IConnectedServiceInstanceContext

{

/// <summary>

/// Gets the current project context where the service should be added.

/// </summary>

IVsHierarchy ProjectHierarchy { get; }

/// <summary>

/// Gets the instance of the service to add to the project.

/// </summary>

IConnectedServiceInstance ServiceInstance { get; }

/// <summary>

/// Gets a logger that can be used to write any messages to the caller. This will include

/// warnings of steps that did not succeed but can be fixed by the end user or errors

/// that caused the operation to fail.

/// </summary>

ILogger Logger { get; }

}

The first two properties are the most important. The ProjectHierarchy represents the VS Project that should be modified. At this point, anything that is possible to do with Visual Studio extension APIs can be invoked to manipulate the project. The ServiceInstance contains the information that was configured by the Provider. It contains the name of the service, and any other information the Provider wants to give to the Handler. Connection information is a common piece of data to pass along in the ServiceInstance.

The Logger property is useful for writing debugging information and other messages you wish the user to see.

At this point, the code you need to write completely depends on your service’s requirements. Some common things that Handlers implement are:

* Installing a NuGet package. See the “[Invoking NuGet Services from inside Visual Studio](http://docs.nuget.org/docs/reference/invoking-nuget-services-from-inside-visual-studio)” article for more information on automating NuGet tasks.
  + One option that is recommended from a performance and stability standpoint is to embed the NuGet package(s) you will install in your Visual Studio extension instead of downloading the package from a remote repository. See the [IVsPackageInstaller.InstallPackagesFromVSExtensionRepository](http://docs.nuget.org/docs/reference/Extensibility-APIs) method for more information.
* Modifying the application’s configuration (app.config, web.config, config.json, etc.)
* Adding a reference
* Injecting code into the project
* Adding non-code files
* Opening a “Getting Started” documentation page

The last 5 tasks can be accomplished with APIs exposed by the Microsoft.VisualStudio.ConnectedServices.Framework.dll assembly.

## Summary

Now you have the complete Visual Studio extensibility APIs at your disposal to build up your Connected Service extension. To leave us feedback, please start a thread in the [Connected Services Preview Yammer group](https://www.yammer.com/azureadvisors/groups/connectedservicespreview?m=2878465852&m=2878465852&nid=869046&nid=869046).