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.

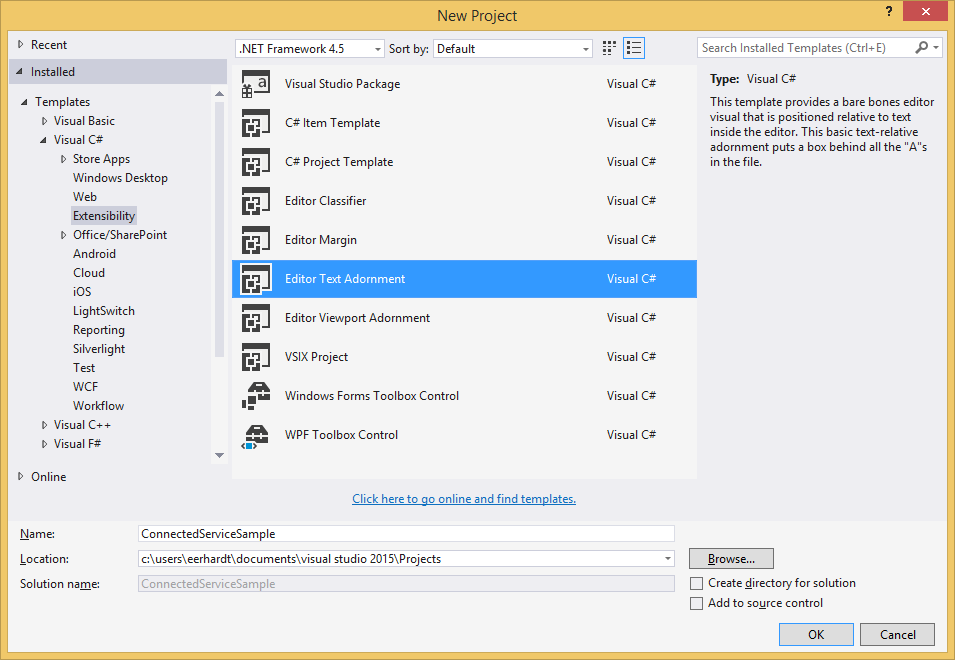
## Writing an extension

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: ConnectedServiceProvider. To create your own extension to Connected Services, you need to implement a class that inherits from the ConnectedServiceProvider class 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.)

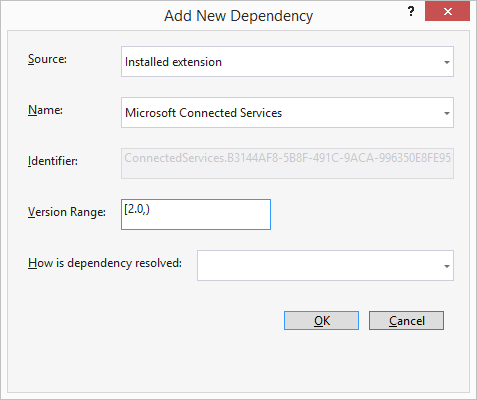
You can delete the two code files “ConnectedServiceSample.cs” and “ConnectedServiceSampleFactory.cs” from the Solution Explorer. We won’t be needing them.

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.

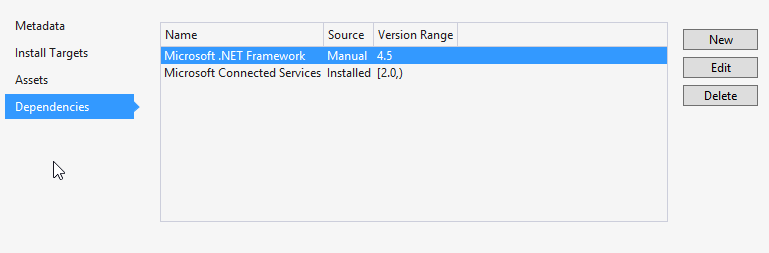
Now you have a Visual Studio extension that can export MEF components to add functionality in VS.

### Setting Dependencies

Since the Connected Services feature in Visual Studio is built as an extension, you need to also include that your extension is dependent on the Connected Services extension. So in the source.extension.vsixmanifest, switch to the Dependencies tab and add a New Dependency on the “Installed extension” with Name “Microsoft Connected Services” with Version Range [2.0,), which means version 2.0 or higher.



So when you are done, it should look like this:



### 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 types that provide the contract between the core Connected Services feature, and any extensions.

* Copy the Microsoft.VisualStudio.ConnectedServices.dll assembly and .xml file from [GitHub](https://github.com/SteveLaskerOrg/ConnectedServicesCustomProviderSamples/tree/master/Externals) 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.
* Under Properties, be sure to toggle CopyLocal to false, as we want the implementation from the Connected Services VSIX

### Create a ConnectedServiceProvider

We can now create a new class that inherits from and exports the ConnectedServiceProvider class. 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.Imaging;

namespace ConnectedServiceSample

{

[Export(typeof(ConnectedServiceProvider))]

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

internal class Provider : ConnectedServiceProvider

{

public Provider()

{

this.Category = "Sample";

this.Name = "Sample Grid Provider";

this.Description = "Sample Provider with Grid functionality.";

this.Icon = new BitmapImage(Utilities.GetResourceUri("Icon.png"));

this.CreatedBy = "Contoso, Inc.";

this.Version = new Version(1, 0, 0);

this.MoreInfoUri = new Uri("https://github.com/SteveLasker/ConnectedServicesCustomProviderSamples");

}

public override Task<ConnectedServiceConfigurator> CreateConfiguratorAsync(ConnectedServiceProviderHost host)

{

return null;

}

}

}

As you can see, the class inherits from the ConnectedServiceProvider base class 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.

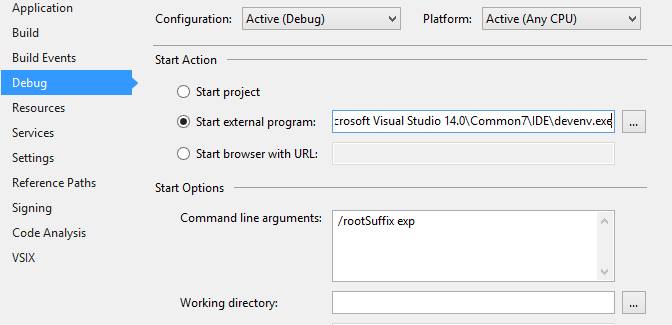
The class fills out the required properties with sample strings and a sample image that is included in the project as a “Resource”. You can choose to set the Icon property to ‘null’ if you don’t have an image for testing, but it is recommended to have an image when you actually release your extension.

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

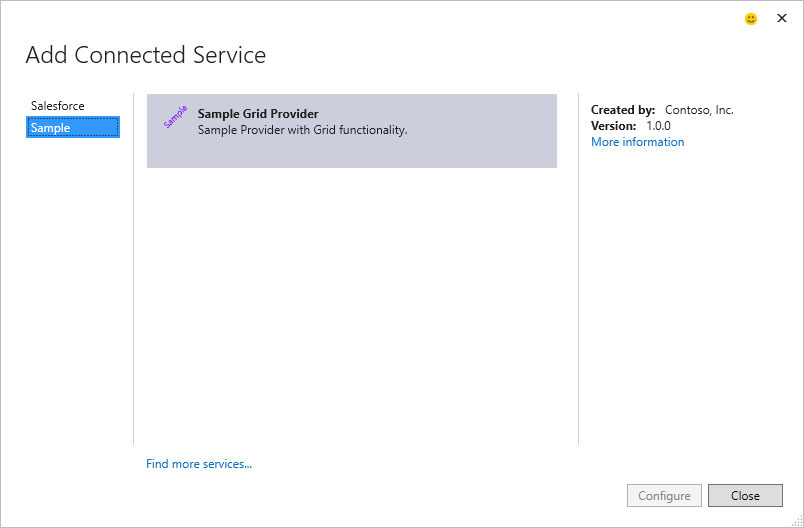
This is the bare minimum information that Visual Studio needs to show your extension in the Add Connected Services dialog.

### Debugging Your Provider

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 a ConnectedServiceHandler

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;

using System.ComponentModel.Composition;

using System.Threading;

using System.Threading.Tasks;

namespace ConnectedServiceSample

{

[Export(typeof(ConnectedServiceHandler))]

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

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

internal class Handler : ConnectedServiceHandler

{

public override Task AddServiceInstanceAsync(ConnectedServiceInstanceContext 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 ConnectedServiceHandler 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.

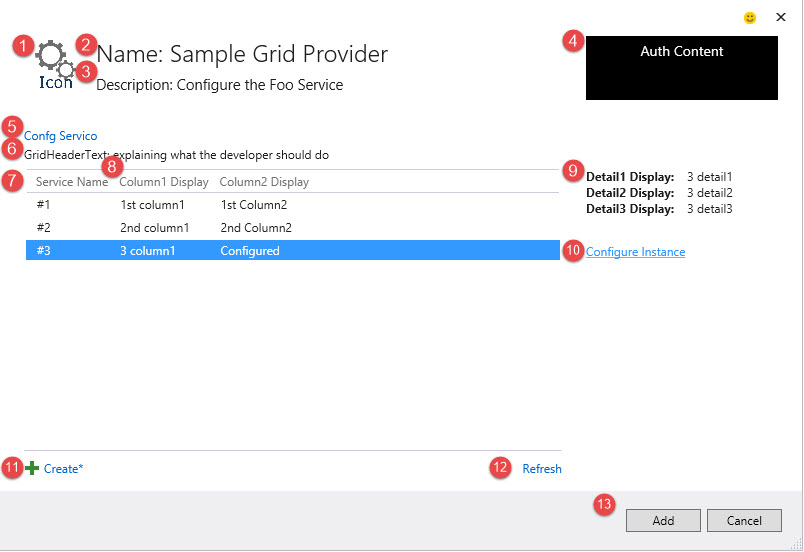
F5 again, create a C# Web project, open the Connected Service dialog, and you will see the Sample provider can now be configured.

### Filling out the Provider

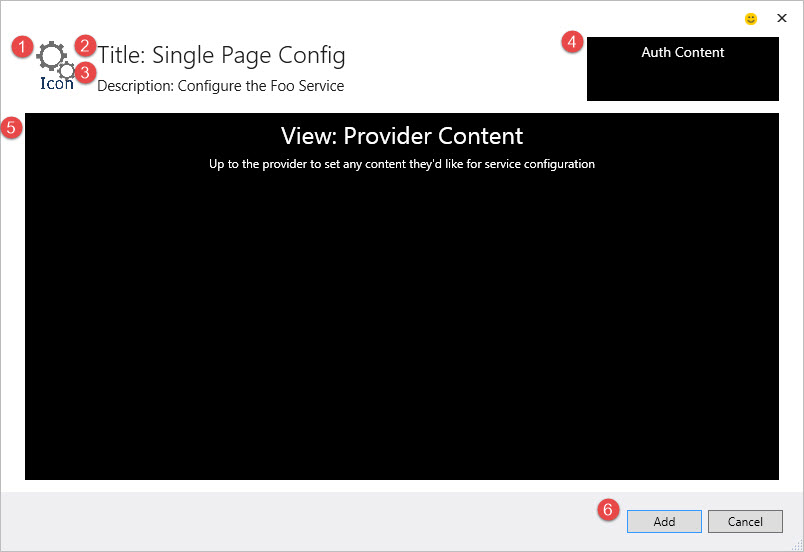
Now that the Configure button is enabled, you are probably going to want to click it. However, if you click it, you get an exception message:

The Connected Services component 'Sample Grid Provider' failed: (HRESULT:0x80131509) The Connected Service Provider 'ConnectedServiceSample.Provider' returned an invalid ConnectedServiceConfigurator from the CreateConfiguratorAsync method. A valid object should inherit from ConnectedServiceGrid, ConnectedServiceSinglePage, or ConnectedServiceWizard.

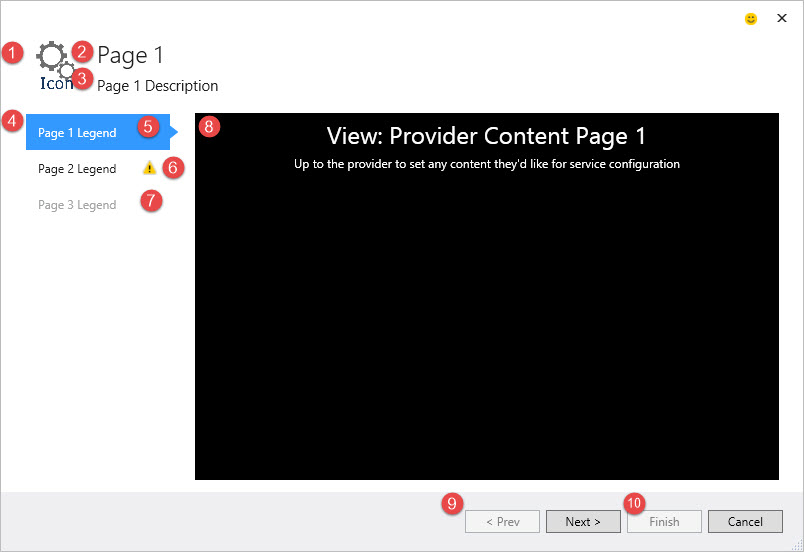
This is where the “CreateConfiguratorAsync” method fits in. You have a choice to make about how you want your provider to be shown in the “configure” dialog. You can choose whether you want to use the “Grid”, “SinglePage” or the “Wizard” template.



vs.



vs.



As the error message above indicates, the Provider must return a valid object from the CreateConfiguratorAsync() method. So let’s pick the simple Grid template and fill out the method to return an object that inherits from the ConnectedServiceGrid base class.

internal class Provider : ConnectedServiceProvider

{

...

public override Task<ConnectedServiceConfigurator> CreateConfiguratorAsync(ConnectedServiceProviderHost host)

{

ConnectedServiceConfigurator configurator = new GridViewModel();

return Task.FromResult(configurator);

}

}

internal class GridViewModel : ConnectedServiceGrid

{

public GridViewModel()

{

this.Description = "A sample Connected Service";

}

public override Task<IEnumerable<ConnectedServiceInstance>> EnumerateServiceInstancesAsync(CancellationToken ct)

{

IEnumerable<ConnectedServiceInstance> instances = new[]

{

new ConnectedServiceInstance() { Name = "first", InstanceId = "first" },

new ConnectedServiceInstance() { Name = "second", InstanceId = "second" },

};

return Task.FromResult(instances);

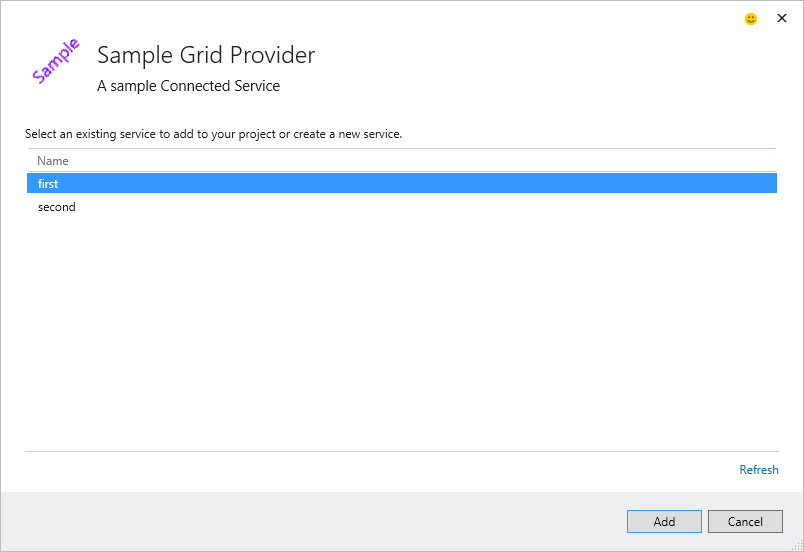
}

}

There are a couple things happening in this code snippet. First, the CreateConfiguratorAsync method was overridden to return a new instance of the GridViewModel class. This class contains all of the logic necessary to populate the Grid template’s UI. There are some optional strings you can set in your constructor, if you want to override the default strings. Or you can leave them ‘null’ if you want to use the defaults.

The last method is the crux of the grid class: 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.

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 ConnectedServiceInstance 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 using the Grid template. But there are other customization options you can choose.

* If you use the Grid template, you can opt into:
  + Adding custom columns with the ColumnMetadata property.
  + Adding detailed information on the right side with the DetailMetadata property.
  + Renaming the first column with the ServiceInstanceNameLabelText property.
  + Allowing the developer to authenticate to the service by overriding the CreateAuthenticatorAsync method.
    - This allows you to put a WPF control in the top right of the dialog that can sign users into your service. For example, you could use a hyperlink to pop open a sign in dialog, or integrate with Visual Studio’s Account Picker control to get an account from the KeyChain.
  + Allowing creation of a new service instance by overriding CreateServiceInstanceAsync and setting CanCreateServiceInstance to true.
  + Allowing the whole service to be configured by overriding ConfigureServiceAsync and setting CanConfigureService to true.
  + Allowing the selected service instance to be configured by overriding ConfigureServiceInstanceAsync and setting CanConfigureServiceInstance to true.
* Or you can use your own UI by:
  + Returning an object that inherits from ConnectedServiceSinglePage class from the CreateConfiguratorAsync method.
  + Returning an object that inherits from ConnectedServiceWizard class from the CreateConfiguratorAsync method.

For more examples of the above, please see our [Connected Services Provider samples on GitHub](https://github.com/Microsoft/ConnectedServices-ProviderAuthorSamples)

### 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’s time to fill out the Handler by modifying the project to actually connect to the service.

The Handler class only contains one method: abstract Task AddServiceInstanceAsync(ConnectedServiceInstanceContext 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.

The ProjectHierarchy property 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 property 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. This information will show up in the progress dialog, and it is logged to the Output window in VS.

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.

We’ll make additional samples for the above scenarios over time. For a complete reference implementation, please see the open source [Connected Service Salesforce Provider](https://github.com/developerforce/visual-studio-tools).

## 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).