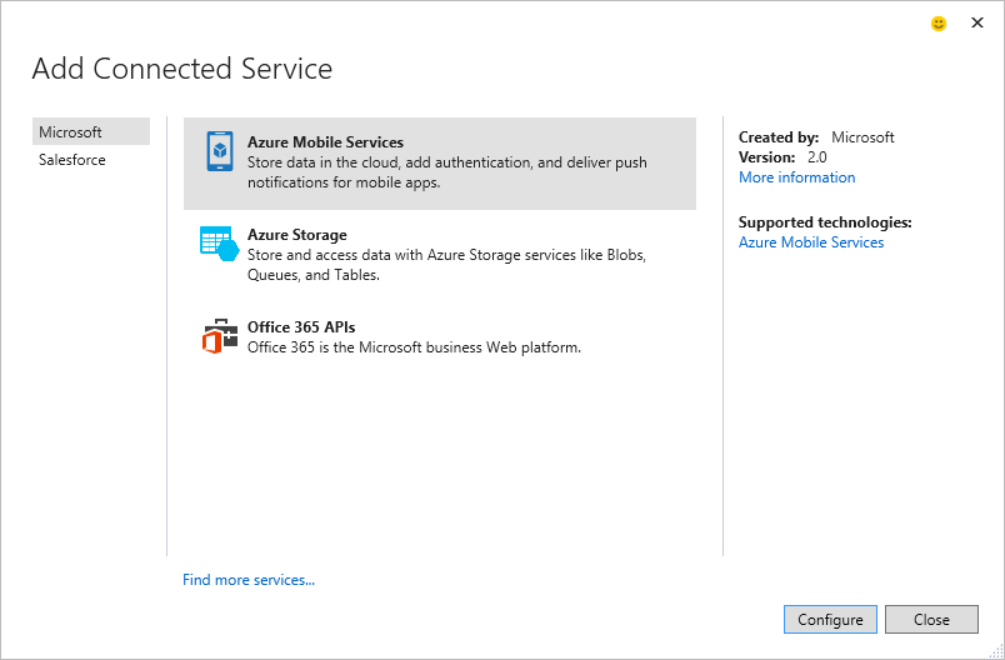
# Connected Services Extensibility API Contracts

This document describes the extensibility APIs that are available to extend the Connected Services experience in Microsoft Visual Studio 2015 RC.

The Connected Services experience is made up of two dialogs. The first dialog allows developers to discover which services are available, and to select a service.



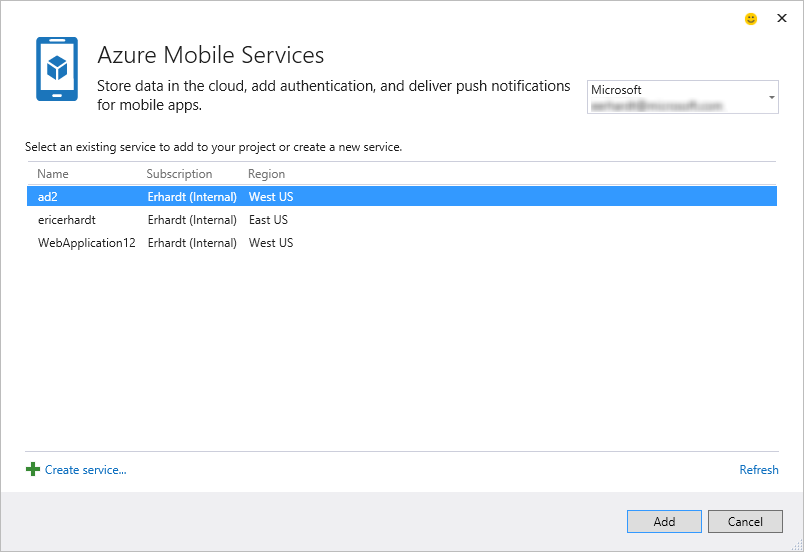
Each extension can designate:

* which category on the left it should be included in
* an icon, title, description for the center pane
* created by and version strings, a “More information” link, and optionally a list of “Supported technologies” on the right

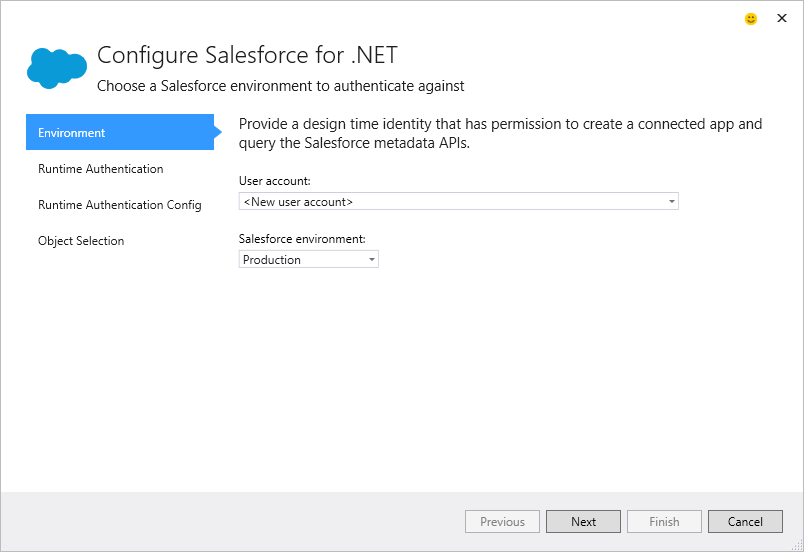
Once a service is selected in the first dialog and the “Configure” button is clicked, a second dialog is opened to configure the service. This dialog allows for more customizability than the first dialog.

At a high-level there are 3 UI templates to pick from:

1. A “Grid” template that allows for custom grid columns, a “Details” pane on the right with custom properties, and a few optional, custom hyperlinks at various places in the dialog. You should use the grid template if you want a user experience where users either select an existing service or provision a new one and fit within a common UI paradigm.



1. A “SinglePage” template that allows for a similar user experience as the Grid template, but you get full control of the UI elements that are shown in the middle of the dialog. You can use your own grid, list, buttons, etc. If your service only requires a single step of configuration, but the Grid template doesn’t provide you enough flexibility, then picking the SinglePage template is the right choice.
2. A “Wizard” template that allows for a custom number of steps, and a fully custom UI pane to display the UI needed to configure the service. If your service requires multiple steps of configuration, then picking the wizard template is the right choice.



## ConnectedServiceProvider

The main class you need to inherit from to show a new entry in the dialogs is the ConnectedServiceProvider class.  This is a type that is imported through MEF.

A class that inherits from ConnectedServiceProvider must have a ConnectedServiceProviderExport attribute, with a “ProviderId” value that uniquely identifies the Provider.

[ConnectedServiceProviderExport("ConnectedServiceSample.Provider")]

internal class Provider : ConnectedServiceProvider

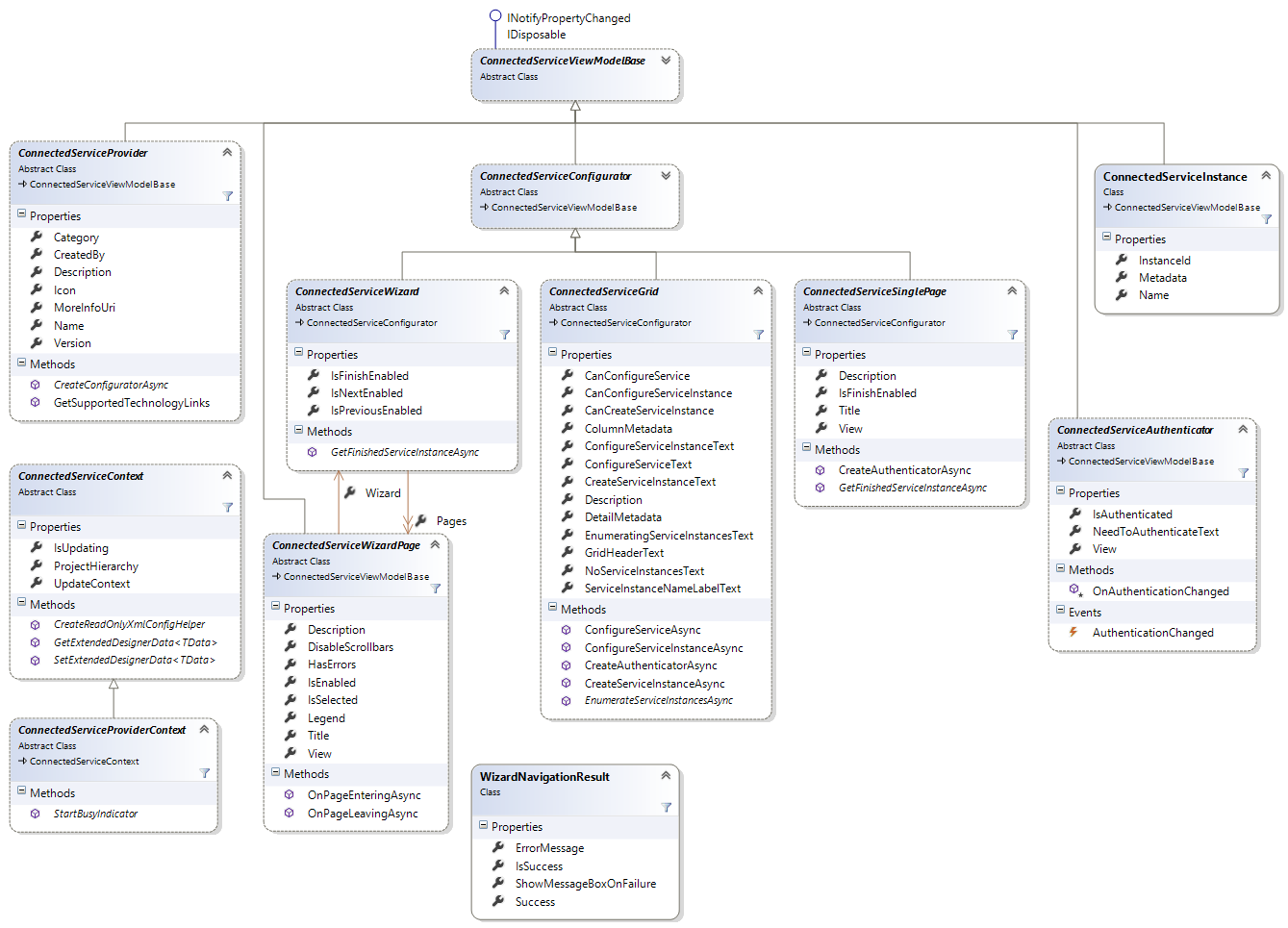
{

…

}

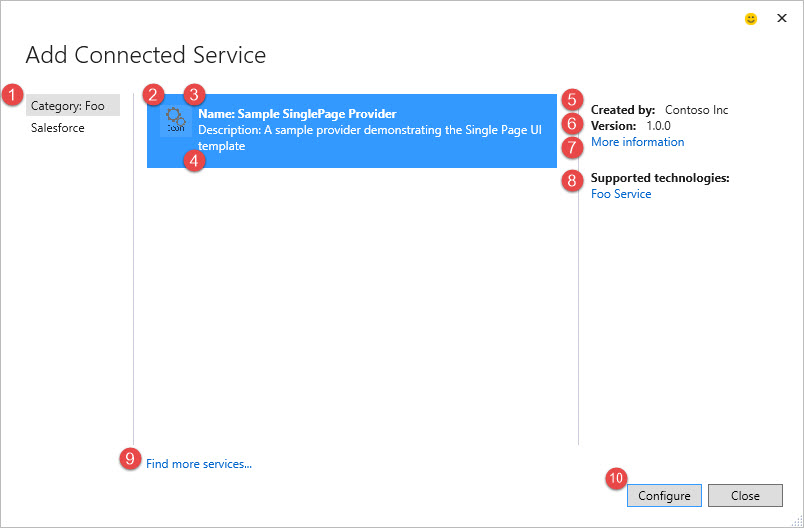
The ConnectedServiceProviderExport attribute takes an optional Boolean value “SupportsUpdate”, which indicates whether your Provider and Handler(s) allow a developer to update a connected service that was previously added to a project. When SupportsUpdate is true, the “update” command is available on the service folder that is generated in the project after a service is added.

The following diagram show the contracts that are exposed to Connected Service Providers. This diagram gives a good overview of what is available to extend/customize in the Connected Service experience.



### Add Connected Service Dialog

Each ConnectedServiceProvider is displayed in the Add Connected Service dialog based on the category and name they provide. The image below shows where each of the ConnectedServiceProvider properties show up in the dialog.



The Categories (1) on the left are meant to describe the service platform that is targeted. For example, if a 3rd party like RSSBus is building a connected service for Salesforce, then it should be under the “Salesforce” category.

The Icon (2), Name (3), and Description (4) for each provider in the category are shown in the center and come from properties on the ConnectedServiceProvider type.

On the right hand side, the Created by (5), Version (6), and More information (7) values are populated by the CreatedBy, Version, and MoreInfoUri properties on the ConnectedServiceProvider type.

The Supported technologies (8) links aren’t populated by a property. Instead, they are an optional feature that can be specified by overriding and returning a collection of link information objects from the GetSupportedTechnologiesLinks() method. The supported technologies are a way to provide more information about the service itself. As opposed to the ‘More information’ link, which can be used to provide information about the connected service extension you are building. For example, the ‘More information’ link could point to the [Visual Studio Gallery](https://visualstudiogallery.msdn.microsoft.com/) page for your extension.

The Find more services… (9) link at the bottom opens the Visual Studio Extensions and Updates dialog to the Connected Services extensions category. This allows developers to find more connected services to install.

The Configure (10) button is enabled if the current Provider is supported on the current project. An extension supports a Visual Studio project by exporting a [ConnectedServiceHandler](#_ConnectedServiceHandler) that has the same “ProviderId” as the selected ConnectedServiceProvider and it specifies that it supports the current Visual Studio project.

### Configure Connected Service Dialog

Once the developer selects a service and clicks Configure, the Add Connected Service dialog is dismissed and a new Configure Connected Service dialog is displayed. As stated previously, this dialog is more customizable by the extension author than the selection dialog. The extension can choose between the UI templates available.

#### CreateConfiguratorAsync

The way you specify which UI template you are choosing is by overriding the CreateConfiguratorAsync method and returning an object that inherits from either ConnectedServiceGrid, ConnectedServiceSinglePage, or ConnectedServiceWizard. Each UI template has its own extensibility points.

The CreateConfiguratorAsync method takes in a ConnectedServiceProviderContext object, which provides information and services to your extension. It inherits from a base ConnectedServiceContext class which is shared between Provider and Handler classes. The base class has the following members:

* **ProjectHierarchy** – Gets the project's IVsHierarchy that the current Connected Service dialog is operating on.
* **IsUpdating** - Gets a value that indicates whether a connected service is being updated.
* **UpdateContext** - Gets the information that is provided when a connected service is being updated, or null if the connected service is not being updated.
* **CreateReadOnlyXmlConfigHelper()** - Creates an XmlConfigHelper that can be used to read xml configuration files.
* **GetExtendedDesignerData()** - Gets the designer data for the connected service that is stored in the ConnectedServices.json file for the service.
* **SetExtendedDesignerData()** - Sets the designer data for the connected service that is stored in the ConnectedServices.json file for the service upon successfully adding/updating the service.

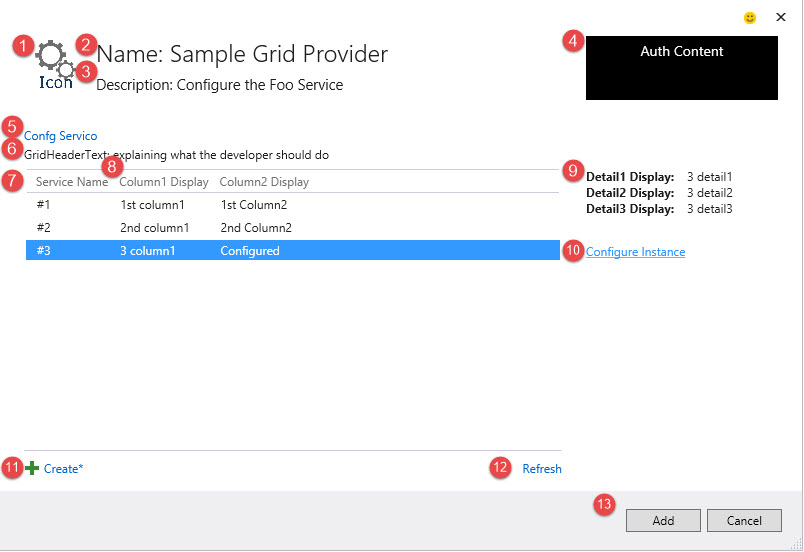
And the ConnectedServiceProviderContext exposes the following members that are only available to Provider classes and don’t make sense on Handlers:

* **StartBusyIndicator(message)** - Shows a busy indicator and message to the user to indicate that Visual Studio is actively working on processing the current request. When the busy indicator should be dismissed, simply call Dispose() on the object returned.

The ConnectedServiceConfigurator object, and objects it creates, are cached for the lifetime of the dialog. So Visual Studio will only ask to create these objects once per session of the dialog. However, some of these objects might be holding on to resources that need to be cleaned up when the dialog is dismissed. In order to accommodate these scenarios, when the dialog is dismissed Visual Studio calls Dispose() on each object, which can be overridden to perform the cleanup.

\* Note that the ConnectedServiceProvider and ConnectedServiceHandler objects are instantiated through MEF. Once these objects are initialized, they are maintained throughout the lifetime of the devenv.exe process. It is highly recommended to not maintain state on these objects, as these resources will not be cleaned up until the process exits.

### Grid Template



When the provider wants to use the Grid template, it returns an object that inherits from ConnectedServiceGrid. The Icon (1) and Name (2) are reused from the ConnectedServiceProvider properties. The other UI elements are taken from the following members:

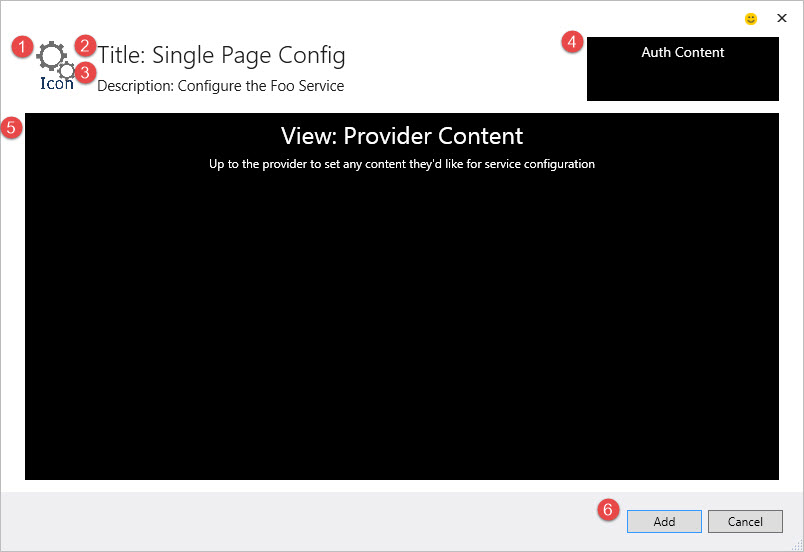
* **Description** (3) – Gets or sets a value to display under the title of the dialog.
* **CreateAuthenticatorAsync**() (4) - Creates the ConnectedServiceAuthenticator that allows users to log into the service. See [ConnectedServiceAuthenticator](#_ConnectedServiceAuthenticator) for more information.
* **CanConfigureService** (5) - Gets or sets a value indicating whether this service can be configured through the connected service provider.
* **ConfigureServiceText** (5) - Gets or sets a value to use as the text of the control for the users to configure the service.
* **ConfigureServiceAsync**() (5) - Configure a service either through a dialog or linking to the configure web page.
* **GridHeaderText** (6) - Gets or sets a value to display directly above the grid.
* **ServiceInstanceNameLabelText** (7) - Gets or sets a value to display as the column header or label for the service instance name.
* **ColumnMetadata** (8) - Gets the keys and display strings for metadata to show as columns in the service instance list.
  + The Item1 field in the enumerable's Tuple is used as a key into the ConnectedServiceInstance.Metadata dictionary. The Item2 field is a localized display string to use as the column header.
  + The columns will be added in the order given after the column that shows the service instance's name. To change the display text for the service instance name, use the ServiceInstanceNameLabelText property.
* **DetailMetadata** (9) - Gets the keys and display strings for metadata to show in the property pane (to the right) when an instance is selected from the service instance list.
  + The Item1 field in the enumerable's Tuple is used as a key into the ConnectedServiceInstance.Metadata dictionary. The Item2 field is a localized display string to use as the field label.
  + The labels and fields will be displayed in the order given.
* **CanConfigureServiceInstance** (10) - Gets or sets a value indicating whether instances of this service can be configured through the connected service provider.
* **ConfigureServiceInstanceText** (10) – Gets or sets a value to use as the text of the control for the users to configure an instance.
* **ConfigureServiceInstanceAsync**() (10) - Configure a service instance either through a dialog or linking to the configure web page.
* **CanCreateServiceInstance** (11) - Gets or sets a value indicating whether the provider can currently create a service instance.
* **CreateServiceInstanceText** (11) - Gets or sets a value to use as the text of the control for the users to create a new instance.
* **CreateServiceInstanceAsync**() (11) - Create a new instance of the service.
* **EnumerateServiceInstancesAsync**() (12) - Get a list of all of the service instances for the currently authenticated user, if any.
  + This method is called to populate the grid initially, and when the Refresh button is clicked.
* **EnumeratingServiceInstancesText** - Gets or sets a value to display while Connected Services is attempting to retrieve the service instances.
* **NoServiceInstancesText** - Gets or sets a value to display when no instances are available because the enumeration returned no instances.

Since the Grid template has all of the UI defined by the core Connected Services feature, it has a lot of optional extensibility points to make it possible for extension authors to customize the experience.

You can change the display name of the first column in the grid by setting a value for the ServiceInstanceNameLabelText property. This first column is always shown and is bound to the ConnectedServiceInstance.Name value for each service instance shown in the grid.

When there are no instances to show, the grid is removed and a watermark is displayed. There are three different strings that can control this text, depending on the current state of the dialog. If there are no services available, the NoServiceInstancesText is used. If the instances are currently being enumerated, the EnumeratingServiceInstancesText is used. And if the Provider implements an authenticator and the user isn’t logged in, the ConnectedServiceAuthenticator.NeedToAuthenticateText is used.

### Single Page Template



When the provider wants to use the Single Page template, it returns an object that inherits from ConnectedServiceSinglePage. The Icon (1) is reused from the ConnectedServiceProvider.Icon property. The other UI elements are taken from the following members of the ConnectedServiceSinglePage class:

* **Title** (2) – Gets or sets the title of the page that is shown at the top of the dialog.
* **Description** (3) - Gets or sets the description of the page that is shown at the top of the dialog.
* **CreateAuthenticatorAsync**() (4) - Creates the ConnectedServiceAuthenticator that allows users to log into the service. See ConnectedServiceAuthenticator for more information.
* **View** (5) - Gets or sets the WPF UI Element that describes the content of the page.
* **IsFinishEnabled** (6) - Gets or sets a value indicating whether the ConnectedServiceSinglePage.GetFinishedServiceInstanceAsync() can be invoked.
* **GetFinishedServiceInstanceAsync**() (6) - The method that is invoked when the user gestures that they are finished configuring the service and want their project updated. This method will return a “finished” service instance that will be passed off to the Handler.

The Single Page template allows you to fully customize the UI content in the middle of the dialog, and allows you to have your own authentication UI at the top right. This template is useful if your service can be configured in a single step.

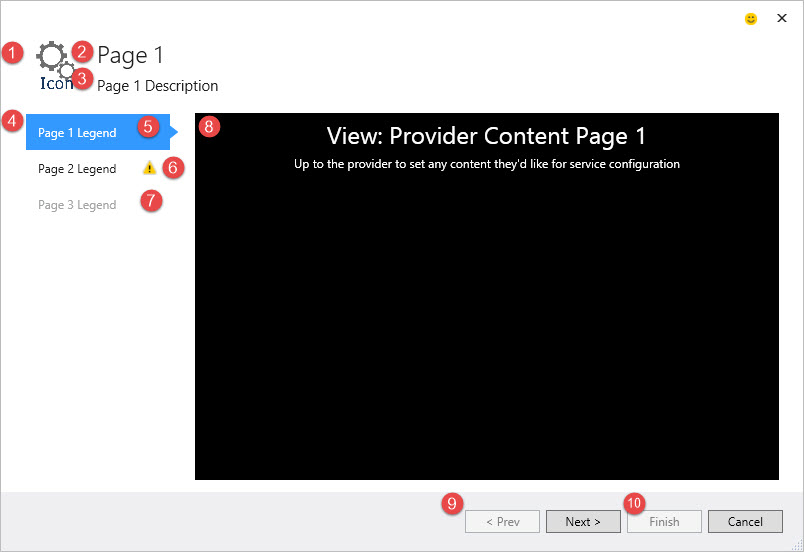
### ConnectedServiceAuthenticator

A very common option to implement with the Grid and Single Page templates is authentication. This allows the user to authenticate to the service from Visual Studio. The extension is given a space in the dialog at the top right where it can implement its own UI to authenticate the user. You can choose this option by returning an object that inherits from ConnectedServiceAuthenticator from the CreateAuthenticatorAsync() method. The class exposes the following members:

* **IsAuthenticated** – Gets or sets a value indicating whether the user is already logged in or otherwise authenticated.
* **NeedToAuthenticateText** - Gets or sets a value to display when no instances are available because IsAuthenticated is 'false'.
* **AuthenticationChanged** – An event that signals a change to the current authentication state such as the authenticated user has changed.
* **OnAuthenticationChanged**() – Raises the AuthenticationChanged event.
* **View** - Gets or sets the Framework Element that should be displayed in the UI to allow the user to authenticate to the service.
  + The space available for this view is 192px width X 64px height.

Note that the Wizard template doesn’t allow this option, because it is expected that the first page of the wizard does the authentication, and from that point forward, the design time authentication won’t change.

### Wizard Template



When the provider wants to use the Wizard template, it returns an object that inherits from ConnectedServiceWizard. The Icon (1) is reused from the ConnectedServiceProvider.Icon property. The other UI elements are taken from the members on the following two classes:

#### ConnectedServiceWizard

* ObservableCollection<ConnectedServiceWizardPage> **Pages** (4) – The collection of pages in the wizard. Each page is a “step” in the wizard.
* **IsPreviousEnabled** (9) - Gets or sets a value indicating whether the Previous button is enabled.
* **IsNextEnabled** (9) - Gets or sets a value indicating whether the Next button is enabled.
* **IsFinishEnabled** (10) - Gets or sets a value indicating whether the Finish button is enabled.
* **GetFinishedServiceInstanceAsync**() (10) – The method that is invoked when the user clicks the Finish button. This method will return a “finished” service instance that will be passed off to the Handler.

#### ConnectedServiceWizardPage

The ConnectedServiceWizardPage class controls most of the UI in the Wizard dialog. It contains the following members:

* **Title** (2) – Gets or sets the title of the page that is shown at the top of the wizard when the page is active.
* **Description** (3) – Gets or sets the description of the page that is shown at the top of the wizard when the page is active.
* **Legend** (4) – Gets or sets the name of the page shown on the left-hand side of the wizard.
* **IsSelected** (5) – Gets or sets a value indicating whether this page is the currently shown page in the wizard.
* **HasErrors** (6) – Gets or sets a value indicating whether this page has errors that the user needs to resolve.
* **IsEnabled** (7) – Gets or sets a value indicating whether this page is enabled, and can be navigated to using the left-hand size page picker.
* **View** (8) – Gets or sets the WPF UI Element that describes the content of the page.
* **Wizard** – Gets the parent ConnectedServiceWizard view model that contains this page.
* **DisableScrollBars** – Gets or sets a value indicating whether or not the default scroll bar functionality for the page should be disabled.
* **OnPageEnteringAsync**() – OnPageEnteringAsync is called on a ConnectedServiceWizardPage when the wizard is navigating to the page.
* **OnPageLeavingAsync**() - OnPageLeavingAsync is called on a ConnectedServiceWizardPage when the wizard is navigating away from the page.

#### WizardNavigationResult

The WizardNavigationResult class is used to block a user from navigating away from the current page. This is useful if the user can’t proceed to a different page unless some action is taken on the current page, for example a validation error. To block navigation, set IsSuccess to false when returning from the OnPageLeavingAsync method. Optionally, you can choose to show a message box to the user by setting ShowMessageBoxOnFailure to false and setting the text you want to display on the ErrorMessage property.

There is a static WizardNavigationResult.Success object that can be returned from OnPageLeavingAsync to easily allow navigation to proceed.

## ConnectedServiceInstance

The ConnectedServiceInstance class represents a single instance of a connected service. You can think of it as the information that gets passed from a Provider into the appropriate Handler. This class is the only way for the Provider and Handler to communicate, since they don’t have references to each other. It contains the following members:

* **InstanceId** – Gets or sets a unique ID for the service instance.
* **Name** – Gets or sets the human readable display name of the service instance.
* **Metadata** - Gets metadata for the service instance used by the service provider and the service consumer.

## And you can inherit your own class from ConnectedServiceInstance to add additional properties.

## ConnectedServiceHandler

A ConnectedServiceHandler is responsible for taking any data 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.

There are a number of ways for a Handler to describe which type of VS projects it supports. The first is an “AppliesTo” query that matches the current Project’s capabilities.

[ConnectedServiceHandlerExport(

"ConnectedServiceSample.ConnectedServiceProvider",

AppliesTo = "CSharp+Web")]

internal class FooHander : ConnectedServiceHandler

{

…

}

The AppliesTo value is a query string that specifies the project type that the Handler is valid for. An example can be found under the **SDKManifest.xml** heading in the [Creating a Software Development Kit](http://msdn.microsoft.com/en-us/library/hh768146.aspx) documentation.

The example above says this Handler applies to any C# Web project.

The other ways to specify which project types are supported are other ExportMetadata attributes with the following names:

* **TargetPlatform** – matches the TargetPlatform property of the Visual Studio project. For example, “Windows” or “WindowsPhoneApp”.
* **TargetPlatformVersion** – matches the version of the TargetPlatform. The [syntax for specifying a version range](http://msdn.microsoft.com/en-us/library/ee943167.aspx) is the same syntax as used in the .vsixmanifest InstallationTarget Version. For example, “[8.1, 10.0]”.
* **SupportedOutputType** – matches the [OutputType property](http://msdn.microsoft.com/en-us/library/microsoft.visualstudio.shell.interop.__vsprojoutputtype.aspx) of the project. For example, “WinExe”, “Exe”, or “Library”.
* **SupportedProjectTypes** – matches the [ProjectType Guids](http://msdn.microsoft.com/en-us/library/microsoft.visualstudio.shell.interop.ivsaggregatableproject.getaggregateprojecttypeguids.aspx) of the project.
* **CustomEvaluator** – If the above functionality doesn’t meet your needs, we have an escape hatch that allows a handler to write its own custom code to evaluate whether it supports a project. To use this escape hatch, set the CustomEvaluator ExportMetadata to true:

[ExportMetadata("CustomEvaluator ", true)]

And then implement the IVsHierarchyCustomEvaluator interface on the class that inherits from ConnectedServiceHandler. The bool Evaluate(IVsHierarchy hierarchy) method will be invoked and you can return true or false to indicate whether the Handler supports the project or not.

Note that you can use any combination of the ExportMetadata attributes on your Handler. If you use more than one, all of the specified attributes have to match the project for the Handler to support the project. If you don’t specify an attribute, it matches by default. If you don’t specify any attributes on your Handler, all VS projects will be considered supported, so it is recommended to specify at least one attribute.

The following class diagram shows all the contracts exposed for Handlers:

### 

### ConnectedServiceHandlerContext

The ConnectedServiceHandlerContext class provides information and services to your Handler’s AddServiceInstanceAsync and UpdateServiceInstanceAsync methods. It inherits from the ConnectedServiceContext class, just like ConnectedServiceProviderContext. The Provider and Handler context objects share all of the members on ConnectedServiceContext. The Handler context object adds the following members:

* **ServiceInstance** – Gets the instance of the service to add to the project.
* **HandlerHelper** – Gets a ConnectedServiceHandlerHelper that can be used to perform project configuration tasks such as adding references, manipulating configuration files, and adding files to the project.
* **Logger** – 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.
* **CreateEditableXmlConfigHelper()** - Creates an EditableXmlConfigHelper that can be used to read or write xml configuration files.

### ConnectedServiceHandlerHelper

The ConnectedServiceHandlerHelper class contains helper methods that ConnectedServiceHandler objects commonly use. The ConnectedServiceHandlerHelper operates on the same Visual Studio project as the corresponding ConnectedServiceInstanceContext that it was retrieved from. It contains the following members:

* **TokenReplacementValues** – Gets a dictionary of token replacement key and values that will be used to replace tokens in AddFilesAsync and PerformTokenReplacement.
* **AddAssemblyReference()** – Adds a reference to the specified assembly to the project.
* **AddFileAsync()** – Add the specified file to the project after doing token replacement.
* **GetServiceArtifactsRootFolder()** – Gets the name of the root folder to place the service related artifacts in. Typcially each provider instance will create its own subfolder under this root folder.
* **PerformTokenReplacement()** – Given an input, replace any tokens found in TokenReplacementValues or the specified dictionary with the specified values.

### ConnectedServiceLogger

The ConnectedServiceLogger class allows for your Handler to write messages to give the end user status on what it is doing. Since manipulating a Visual Studio project (adding NuGet packages, adding files, etc.) takes a non-trivial amount of time, users like getting updates on what is happening.

While your handler is running, a progress dialog is shown and any message you write under the Information, Warning, or Error category is shown in the progress dialog. You can also write Debug category messages, which only get written to the Visual Studio Output window.

### AddServiceInstanceResult

The AddServiceInstanceResult object is returned from the ConnectedServiceHandler.AddServiceInstanceAsync() method. It allows you to tell Visual Studio the name of the folder to store your service artifacts. For .NET projects, this folder gets created under the “Service References” folder. For JavaScript projects, this folder gets created under the “Services” folder. You can also specify a “Getting Started” URL, which is a document that gets launched to inform the developer how to use the service and what modifications were made in the project.

### UpdateServiceInstanceResult

The UpdateServiceInstanceResult object is returned from the ConnectedServiceHandler.UpdateServiceInstanceAsync() method. This method can be invoked on a Handler if the Provider says that it supports update, and the developer updates a previously added service. The UpdateServiceInstanceResult allows you to modify the “Getting Started” URL that was previously set during AddServiceInstanceAsync.