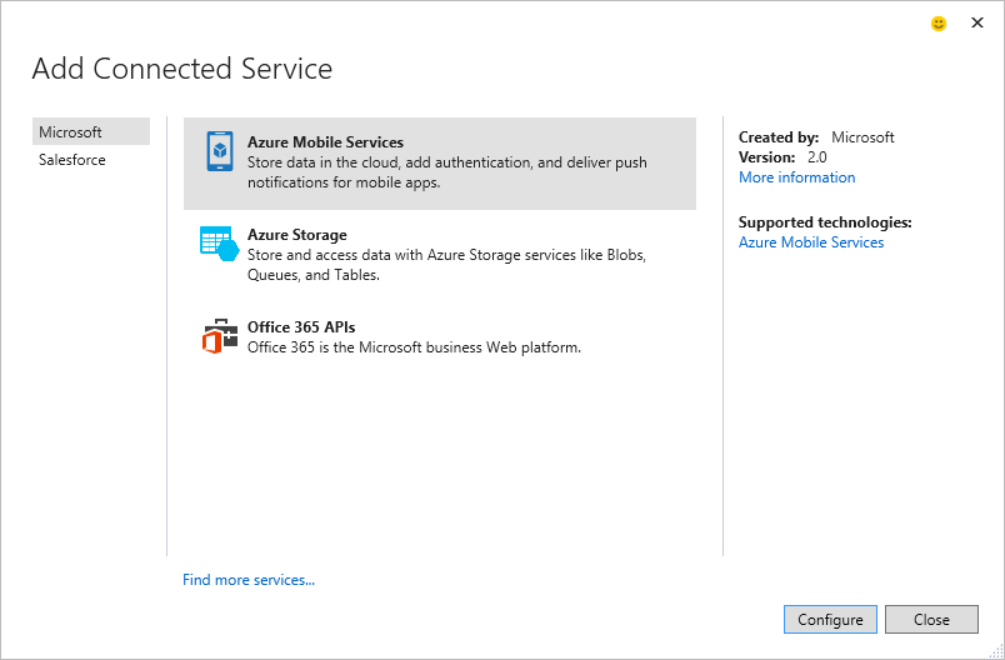
# Connected Services Extensibility API Contracts

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

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.



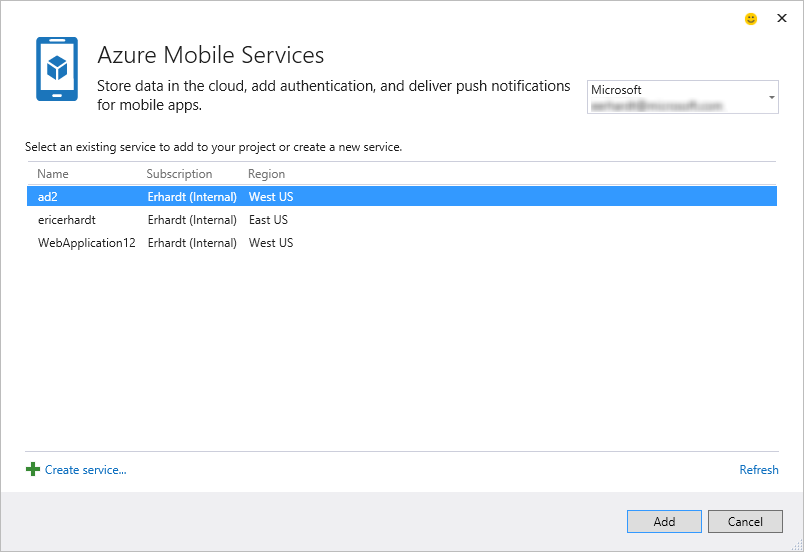
Each extension can designate:

* 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

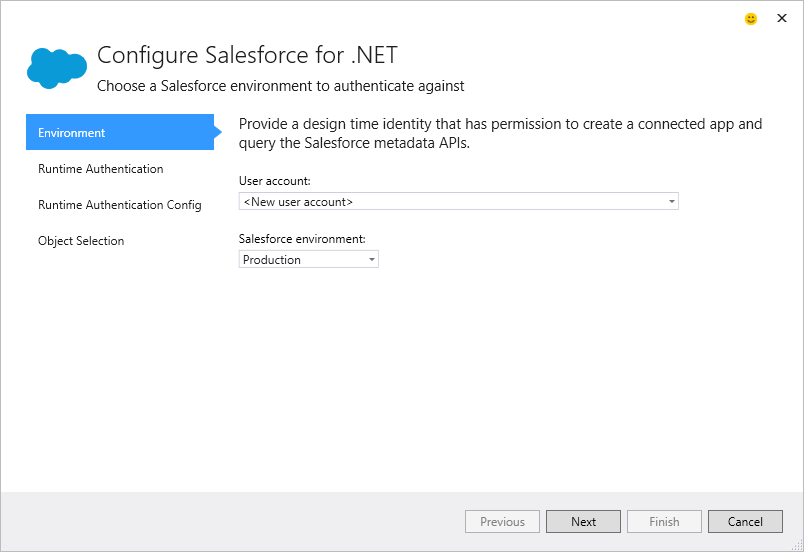
Once a service is selected in the first dialog and the “Configure” button is clicked, a second dialog is opened to configure the service. This dialog allows for more customizability than the first dialog.

At a high-level there are 2 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.



1. 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.



## Class Diagram

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



The main interface that a provider has to implement is the IConnectedServiceProvider.  This is the interface that is imported through MEF.

The class that implements IConnectedServiceProvider must have an [Export(typeof(IConnectedServiceProvider))] attribute, with at least a “ProviderId” metadata attribute that uniquely identifies the Provider.

[Export(typeof(IConnectedServiceProvider))]

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

internal class ConnectedServiceProvider : IConnectedServiceProvider

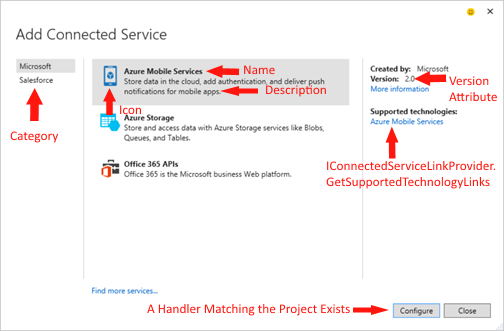
{

…

}

## Add Connected Service Dialog

Every IConnectedServiceProvider imported through MEF, that returns a non-empty “Category” string, is shown in the Add Connected Service dialog. The image below shows where each of the IConnectedServiceProvider properties show up in the dialog.



The Categories 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.

On the right hand side, the ‘Created by’ and ‘More information’ values are populated by the CreatedBy and MoreInfoUri properties respectively.

Note that the Version value on the right isn’t populated by a property on IConnectedServiceProvider. Instead, this value is retrieved from a MEF metadata attribute on the class that exports IConnectedServiceProvider:

[Export(typeof(IConnectedServiceProvider))]

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

[ExportMetadata("Version", "1.0")]

internal class ConnectedServiceProvider : IConnectedServiceProvider

The Supported technologies links aren’t populated by a property either. Instead, they are an optional feature that can be specified by returning an object from the CreateService method that implements IConnectedServiceLinkProvider. (More on the [CreateService](#_CreateService) method later.) 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 Configure button is enabled if an IConnectedServiceHandler can be found that has the same “ProviderId” as the selected IConnectedServiceProvider and it has an “AppliesTo” query that matches the current Project’s capabilities.

[Export(typeof(IConnectedServiceInstanceHandler))]

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

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

internal class ConnectedServiceHandler : IConnectedServiceInstanceHandler

A Connected Service Handler 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.

The AppliesTo value is a query string that specifies the project type that the Handler is valid for. There is a full query syntax described in the following links:

<http://msdn.microsoft.com/en-us/library/dn497698.aspx>

<http://msdn.microsoft.com/en-us/library/hh768146.aspx> (under SDKManifest.xml heading)

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

### CreateService

The way any choice is made by a provider in Connected Services is through the Task<object> CreateService(Type serviceType, IServiceProvider serviceProvider) method. This method is invoked by Visual Studio whenever it wants to ask the extension to make a decision. The method is weakly-typed so we can add more extension points in the future. It takes in a Type that specifies which extensibility point is being queried, and an IServiceProvider that can be used to get Visual Studio services, like EnvDTE objects. The Provider object can choose to return an object that implements the Type being passed in, or it can choose to return ‘null’ to opt out of the extensibility point. Note that the method actually returns a Task<object>, which means it can easily call asynchronous methods, if needed.

One of the Visual Studio services that can be retrieved from the IServiceProvider is the IConnectedServiceProviderHost service. This host object can be used to get the IVsHierarchy, which represents the current VS project being configured. The host object can also be used to start a “busy” indicator in the bottom left corner of the configure dialog to tell the user it is working on satisfying the current request. When the busy indicator should be dismissed, simply call Dispose() on the object returned from the IConnectedServiceProviderHost.StartBusyIndicator method.

The objects that are returned by the CreateService method are cached for the lifetime of the dialog. So Visual Studio will only ask for a certain service Type once, and an extension doesn’t need to worry about caching it.

Some of these objects returned by CreateService 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 checks if each service object implements IDisposable. If it does, it gets Dispose() called on it.

### INotifyPropertyChanged

None of the IConnectedService\* interfaces inherit from INotifyPropertyChanged. It is optional to implement this interface. This is a convenience for providers whose properties don’t change dynamically. However, if your provider’s properties do change dynamically and you want the UI to be updated when your properties change, ensure you are implementing INotifyPropertyChanged on any objects whose properties are changing.

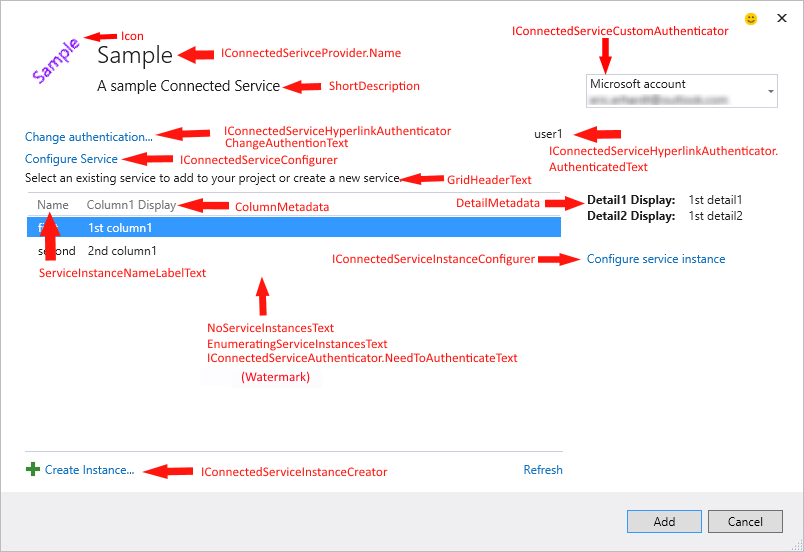
### Grid Template

The only service that must be implemented by an IConnectedServiceProvider is which UI template to show in the Configure dialog. Visual Studio calls CreateService(typeof(IConnectedServiceProviderUI)) to get the UI template. When the provider wants to use the Grid template, it returns an object that implements IConnectedServiceProviderGridUI. This interface contains the following members:

* ShortDescription - Gets a value to display under the title of the dialog.
* GridHeaderText - Gets a value to display directly above the grid.
* EnumeratingServiceInstancesText - Gets a value to display while Connected Services is attempting to retrieve the service instances.
* NoServiceInstancesText - Gets a value to display when no instances are available because the enumeration returned no instances.
* ServiceInstanceNameLabelText - Gets a value to display as the column header or label for the service instance name.
* ColumnMetadata - 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 IConnectedServiceInstance.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 - 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 IConnectedServiceInstance.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.
* EnumerateServiceInstancesAsync - Get a list of all of the service instances for the currently authenticated user, if any.

Since the Grid template has all of the UI defined by the core Connected Services feature, it has a lot of optionally extensibility points to make it possible for extension authors to customize the experience. The first two are listed above with “ColumnMetadata” and “DetailMetadata”. These properties control the columns in the Grid, and the detail pane on the right hand side.

The image below shows each of the extensibility points in the UI and what controls each point.



You can change the display name of the first column in the grid by returning a value for the ServiceInstanceNameLabelText property. This first column is always shown and is bound to the IConnectedServiceInstance.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 IConnectedServiceAuthenticator.NeedToAuthenticateText is used.

#### IConnectedServiceAuthenticator

A very common option to implement with the Grid template is authentication. This allows the user to authenticate to the service from Visual Studio. There are two types of authenticators that the Grid template supports:

* A hyperlink authenticator
  + Allows the user to sign in and out by clicking a hyperlink.
  + Choose this option by returning an object that implements IConnectedServiceHyperlinkAuthenticator.
  + This is the approach the Office 365 provider uses.
* A custom UI authenticator
  + The extension is given a space in the dialog at the top right where it can implement its own UI to authenticate the user.
  + Choose this option by returning an object that implements IConnectedServiceCustomAuthenticator.
  + This is the approach Azure Mobile Services and Storage use.

*IConnectedServiceConfigurer, IConnectedServiceInstanceConfigurer, and IConnectedServiceInstanceCreator*

An optional hyperlink can be displayed above the grid by implementing the IConnectedServiceConfigurer interface. A common usage of this hyperlink is if the whole service needs to be configured. The hyperlink can be hidden by returning false from the CanConfigureService property. A Provider can change the text displayed in the hyperlink by returning a value for the ConfigureServiceText property. When the hyperlink is clicked, the ConfigureService method is invoked.

An optional hyperlink can be displayed below the grid by implementing the IConnectedServiceInstanceCreator interface. This link is normally used to allow the user to create a new service instance. The same pattern applies as above: hide the link by returning false from CanCreateServiceInstance; change the text by returning a value from CreateServiceInstanceText; and when the link is clicked, the CreateServiceInstance method is invoked.

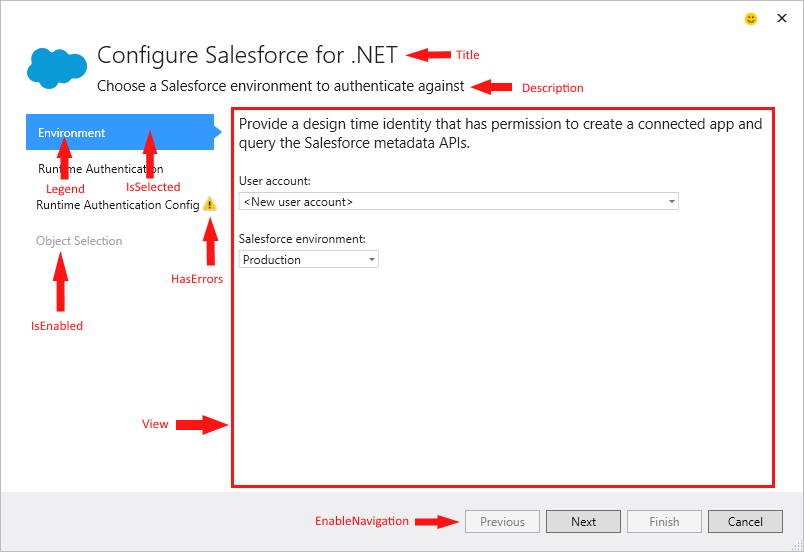
### An optional hyperlink can be displayed to the right of the grid by implementing the IConnectedServiceInstanceConfigurer interface. This link is normally used to allow the user to configure the currently selected service instance. The same pattern applies as above: hide the link by returning false from CanConfigureServiceInstance; change the text by returning a value from ConfigureServiceInstanceText; and when the link is clicked, the ConfigureServiceInstance method is invoked with the selected service instance passed as an argument.

### Wizard Template

When the provider wants to use the Wizard template, it returns an object that implements IConnectedServiceProviderWizardUI. This interface contains the following members:

* ObservableCollection<IConnectedServiceWizardPage> Pages – The collection of pages in the wizard. Each page is a “step” in the wizard.
* EnableNavigation - Event for enabling the navigation buttons (Previous, Next, or Finish).
* GetFinishedServiceInstance – 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.

The image below shows each of the extensibility points in the UI and what controls each point.



#### IConnectedServiceWizardPage

The IConnectedServiceWizardPage interface controls most of the UI in the Wizard dialog. It contains the following members:

* Title – Gets the title of the page that is shown at the top of the wizard when the page is active.
* Description – Gets the description of the page that is shown at the top of the wizard when the page is active.
* Legend – Gets the name of the page shown on the left-hand side of the wizard.
* IsEnabled – Gets a value indicating whether this page is enabled, and can be navigated to using the left-hand size page picker.
* IsSelected – Gets or sets a value indicating whether this page is the currently shown page in the wizard.
* HasErrors – Gets a value indicating whether this page has errors that the user needs to resolve.
* View – Gets the WPF UI Element that describes the content of the page.
* OnPageEntering – OnPageEntering is called on a IConnectedServiceWizardPage when the wizard is navigating to the page.
* OnPageLeaving - OnPageLeaving is called on a IConnectedServiceWizardPage when the wizard is navigating away from the page.

#### NavigationEnabledState

The NavigationEnabledState class is used to toggle the enabled state of the Previous, Next, and Finish buttons in the wizard. The class contains three nullable Boolean properties corresponding to each of these buttons. If the current state of the button should be preserved, a ‘null’ value should be used for the NaviationEnabledState’s corresponding property. This is useful when you want to toggle a single button’s state, and leave the other buttons as they currently are.

#### 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 OnPageLeaving 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 OnPageLeaving to easily allow navigation to proceed.

### IConnectedServiceInstance

The IConnectedServiceInstance interface represents a single instance of a connected service. It contains the following members:

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

### IConnectedServiceInstanceContext

The IConnectedServiceInstanceContext interface is the contract for the data being passed into a Handler’s AddServiceInstanceAsync method. It contains the following members:

* ProjectHierarchy – Gets the current project context where the service should be added.
* ServiceInstance – Gets the instance of the service to add 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.
* TargetPlatformIdentifier – Gets the target platform identifier for the project hierarchy.
* TargetPlatformVersion - Gets the target platform version for the project hierarchy.