8 Advanced: Connected web parts

Until now, we have dropped the word connection here and there, but we have never explained what they are and how to use them. This is going to change in this chapter. We will first look at what connectable web parts are and how to use them. Then we will set out to build a connectable web part ourselves.

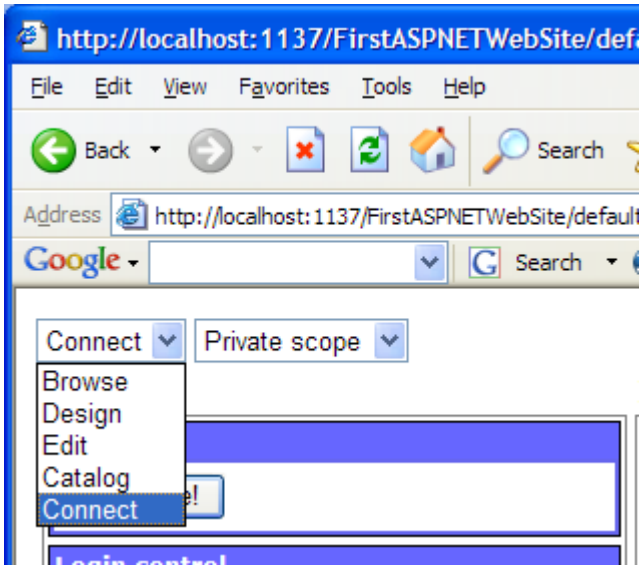
First of all: what are connectable web parts? All web parts we have seen this far where isolated blocks on the page. They could be personalized and configured by the user, but web parts are unaware of the existence of other web parts on the page and will not react to them. While this is fine for a portal-like homepage, it doesn't work for more task oriented application pages.

Imagine a page on an intranet where you can lookup information about your colleagues. On one side of the screen you would typically have a search box with a list of search results, while the center or right part of the screen would show a detail view of a selected employee. It would be nice if we could set up such a page as a web part page, as it would allow users to configure the page to their liking (filtering by default on their department) and would allow the manager of the HR department to add new functionality to the page through the shared scope. But we don't want the search box, results list and detail view to be combined in one web part.

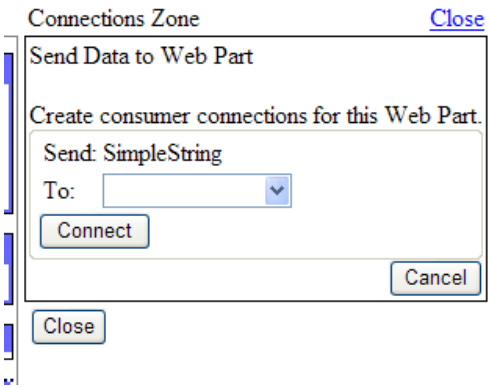
Enters: web part connections. Web part connections allow the web part author to mark the part as a connection consumer or connection provider (or both). The user can connect a provider and a consumer up in the "connect" display mode. Once the web parts are connected, the web part framework will ask the provider for information and passes this information on to the consumer. The provider and consumer do not necessarily have to "know" each other.

Connecting web parts through the UI

The key to connecting two connectable web parts is to switch to the "connect" display mode. To be able to do this, the page should contain a `ConnectionsZone` control. In the next screenshot, we have created a web page with one connection provider web part and one connection consumer web part. Then we set the display mode to "connect" (using our mode switch, see chapter 10):



On all web parts that are marked as a connection provider or as a connection consumer, a Connect verb appears. When this verb is selected, the Connection Zone appears:



The Connections Zone show a list of active connections for the web part (in this case, none). If it is possible to use the web part to create a new connection, the Connections Zone displays these actions as links (for our provider web part: "Create a connection to a Consumer"). If we choose this action, the zone shows us a list of all compatible consumers (in the list,

the title of the part is displayed). By selecting the right consumer and clicking the Connect button, the connection is established. From now on, every time the web part page is rendered, the provider will get the chance of passing information to the consumer.

Connecting web parts in code

It is also possible to configure a connection between two web parts on a web part page through mark-up code in your .ASPX page. The connections are managed by the `WebPartManager`, so the connections must be configured inside the declaration of the `WebPartManager`:

```
<asp:WebPartManager ID="Webpartmanager1" Runat="server">
  <StaticConnections >
    <asp:WebPartConnection ID="staticConn"
       ConsumerID="StaticInteropCons"
       ProviderID="StaticInteropProv" />
  </StaticConnections>
</ asp:WebPartManager >
```

Obviously, these static connections can only be created between static web parts (web parts that are configured into the ASPX page and not through personalization). The connections can however be removed (disconnected) through the web UI by users with the appropriate privileges.

Creating a connectable web part

Creating connectable web parts is not difficult in a technical coding sense. What can be hard though is designing the information that should be passed from one part to another. First we'll have a look at the simplest conceivable case: passing a string. We'll build two web parts: one that allows users to select a term from a dropdown list and passes the selected term through to a connected part. The other part can receive the selected term and will display it in a large font. Not a very useful set of parts, but it will do for the demonstration. Note: this first implementation is too naïve for real use. You should use the setup using interfaces. We will see implementations with interfaces in later sections. The merit of this naïve implementation is to show why you need to work with interfaces.

Naive implementation

First, we create the provider part as a simple user control. On it we place a `ListBox` containing a number of hard coded list items for our user to select.

```
<asp:ListBox ID="ListBox1" runat="server" AutoPostBack=true >
```

```
  <asp:ListItem Value="B" Text="Belgium"></asp:ListItem>
  <asp:ListItem Value="DK" Text="Denmark"></asp:ListItem>
  <asp:ListItem Value="F" Text="France"></asp:ListItem>
  <asp:ListItem Value="D" Text="Germany"></asp:ListItem>
  <asp:ListItem Value="NL" Text="Netherlands"></asp:ListItem>
  <asp:ListItem Value="S" Text="Sweden"></asp:ListItem>
</asp:ListBox>
```

Note that we have set the `AutoPostback` property of the `ListBox` to true. Passing information from providers to consumers of a connection only happens server-side, so we must force a postback to see the connections work.
Now we will make this simple control into a connection provider by adding a method to the class that returns the selected value from the dropdown list and mark it with an attribute as a `ConnectionProvider`:

```
[ConnectionProvider("SimpleString")]
public string GetSelectedValue() {
  return ListBox1.SelectedValue;
}
```

The web part framework will now treat this part as a connection provider. We will see the meaning of the string 'SimpleString' we passed in later on. Still we cannot use the provider without a compatible consumer. Like the provider, a connection consumer is a normal web part with a special method on it. Check out the code of our connection consumer:

```
<asp:Label ID="Label1" runat="server" Text="Label"></asp:Label>
```

and this is the special method:

```
    [ConnectionConsumer("SimpleString")]
    public void SetSelectedValue(string value)
    {
      Label1.Text = value;
    }
```

The `SetSelectedValue` method is marked as a connection consumer using the `ConnectionConsumer` attribute. To be compatible with the provider, the parameter the method takes must be of the same type as the return value of the provider method (in this case: `string`). If the types don't match the consumer will not show up in the list of available consumers in the browser. The string value passed to the attribute ('SimpleString') will show up in the user interface to indicate what kind of

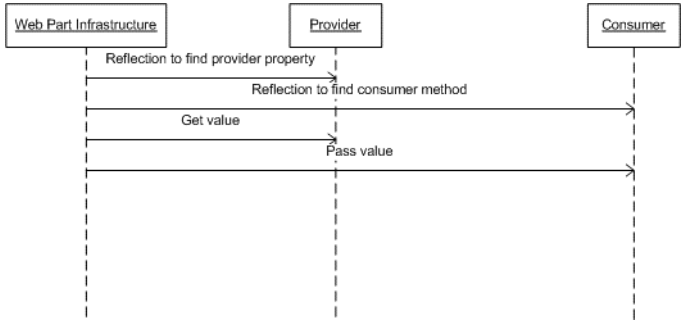
information will be passed through the connection (or to distinguish the connection points when a part has more than one).

So if your provider part allows the user to select a US zip code, the property could be of type string, but you could explain to the user creating the connection that the method will return a zip code by specifying the attribute like this:

```
    [ConnectionProvider("US Zip code")]
```

What happens now is that in every request, before the `PreRender` phase of the page, the `WebPartManager` will call the provider method (on the provider part) and pass the returned value to the consumer method (on the consumer part). This is actually all we need for our very simple connection; it looks like this:

In a UML sequence diagram, this would be represented like this:



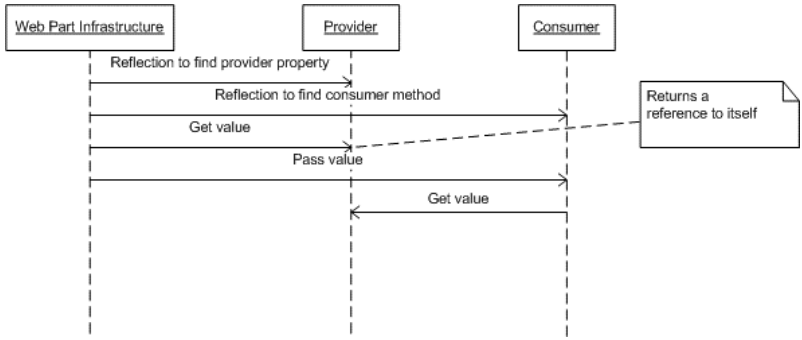
This implementation has a number of limitations. First, it is very dependent on timing. In the whole page lifetime, provider and consumer are connected for one short moment to pass information from provider to consumer. This may not be the most convenient moment for your application: suppose that you still have to do some calculations in the provider part before you can pass the right value to your subscriber. Second, you can only pass one value. In many cases, you may need to pass both a string and a number or more complex combinations of data. Or maybe you even want to call a method on the provider. Third, a string can be many different things. Consumers that can perform interesting operations when fed a zip code can probably not do much with any other string. Therefore, we shouldn't even be able to connect a zip code consumer to a completely different provider that happens to produce strings as well. We want more strongly typed contracts.

These three limitations are solved by using a slightly more complex, but very elegant way of specifying connection endpoints using interfaces. This way is also the suggested pattern in all documentation from Microsoft. It goes like this:

1. The provider returns not the value it wants to pass, but a reference to itself. This allows the consumer to get the required information at a later, more convenient, time.
2. The provider implements an interface that contains properties for all values it wants to pass to the consumer. The return type of the provider method is this interface.
3. The interoperability of consumers and providers is defined with the interfaces you use.

Interface based implementation

Before we delve into the implementation, we have a look at the sequence diagram again. Note that the framework does not do anything different from the naïve implementation; it is the provider and consumer parts that behave differently.



So we create a special interface for returning the selected value. We don't expect consumers to have any special knowledge of the passed data, except that it is of type string. This is the code for the interface:

```
public interface IStringValue {
    string Value { get;}
}
```

The provider part will now implement this interface and pass a reference to itself in the provider method:

```
public partial class BetterProvider : System.Web.UI.UserControl,
                                      IStringValue
{
  [ConnectionProvider("SimpleString")]
  public IStringValue GetSelectedValue()
  {
    return this;
  }

  string IStringValue.Value
  {
    get { return this.ListBox1.SelectedValue; }
  }
}
```

Note how we moved the code that really retrieves the selected value from the `ListBox` to the `IStringValue.Value` property. The consumer may now choose the moment for calling this (the value might change after the moment of connection). In our example though, we still retrieve the selected value immediately.

```
public partial class BetterConsumer : System.Web.UI.UserControl
{
  IStringValue _stringProvider = null;
```

```
  [ConnectionConsumer("SimpleString")]
  public void SetSelectedValue(IStringValue value)
  {
    _stringProvider = value;
    // We could keep this reference for later use,
    // but in this case we will get the data now
    this.Label1.Text = _stringProvider.Value;
  }
}
```

If you want your part to be usable with many other parts, you want to use a very common interface, if you want to be sure that provider and consumer truly understand each other, you would prefer a very specific interface. As always, defining the interfaces is the hard task. You can compare this to specifying the required interfaces for a web service. When you define a specific interface, it is often wise to specify this interface as a subclass of a more common interface. Let's walk through an imaginary situation:

In our website, we have a web part that allows users to pick a zip code by clicking on a map of the city (`PickZip`). A second part can calculate the best itinerary from any zip code to our office (`CalcItin`). A third part can provide a list of articles for any keyword by full text search (`ShowArtc`). `CalcItin` is a connection consumer that can only handle zip codes. Therefore, we create a specific interface type `IZipCode` to return the string value of the zip code. `PickZip`'s functionality is also very specific for zip codes, so it is a connection provider of type `IZipCode`. `ShowArtc` is a connection consumer that consumes a much more general type of data: arbitrary keywords. We create the interface `IStringValue` for use in `ShowArtc`.
`CalcItin` can now only be connected to `PickZip` parts. This is a good thing, because it would only mess things up when non-zip code data would be passed to `CalcItin`. But `PickZip` can also *only* be connected to `CalcItin`. That is a bit of a pity, as we might use the zip codes provided by `PickZip` in other ways as well. We could for example use them to search for articles containing the zip code. We can achieve this situation by subclassing `IZipCode` from `IStringValue` like this:

```
public interface IZipCode : IStringValue {}
```

The inheritance gives the two interfaces a one-way 'is' relation. `IZipCode` 'is' `IStringValue`, but `IStringValue` 'is' not necessarily `IZipCode`. It is

the consumer that specifies the required contract, so an `IStringValue` consumer can be connected to an `IZipCode` provider, but not vice versa.

Standard connection interfaces

When you produce web parts that should be able to connect to web parts from other authors, you'll have to find a common interface for exchanging your information. This is actually very difficult, as you do not know this other author. To make these matches a little bit easier, Microsoft has shipped a number of standard interfaces with the web part infrastructure. As long as you use these interfaces, your parts will be able to cooperate with those from other authors using the same interface.
However, in the effort to make these interfaces as broadly applicable as possible, the Microsoft engineers came up with a design that is rather meta and not very self-explicatory. I'll first list the different available interfaces and specify their intended use and then we will show how to use one of them.

Interface name	Used for
`IWebPartField`	Passing a single value (of type `object`) to the consumer
`IWebPartRow`	Passing an object exposing a number of values (as properties), as in a strong typed data row
`IWebPartTable`	Passing a collection of rows
`IWebPartParameters`	Like `IWebPartRow`, but this interface allows the consumer to specify which parameters it expects, the provider returns a Dictionary of values

There are two remarkable aspects to these interfaces that make them very versatile, but also rather complex:

1. To allow for maximum interoperability, the interfaces do not specify the types of the objects passed between provider and consumer. Instead, each interface contains a `Schema` property, which returns a `PropertyDescriptor` (or a collection of descriptors), which can be used by the consumer to know how to handle the passed data and decide if it can somehow convert the passed values to the required type.

2. The interface specifies no simple property to retrieve the value (as our custom `IStringValue` interface did), but a method that can be used to pass in a delegate to a method on the consumer that can be used to set the value to be passed. Still there?

We'll explain the rationale about these two remarkable features before getting into implementation details.

The Schema property

By querying the `Schema` property, the consumer can gain knowledge about the value that will eventually be passed by the provider part. The `PropertyDescriptor` that is passed, can contain a lot of information on this value, like the type (e.g. `System.String`), the name (e.g. "ZipCode"), whether the value is read-only, etc. A consumer may use this information to decide on its actions: some types may be easy to convert to the expected type, while others are not.

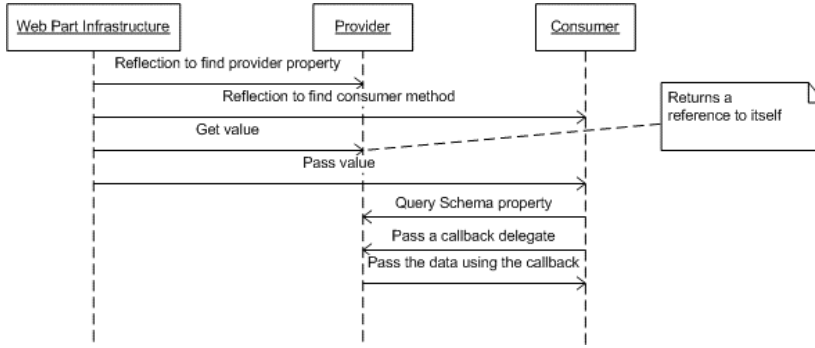
A provider should always provide a correct value for this property, while many consumers will ignore it. The framework uses the `Schema` property for the standard connection transformers (See chapter 9). These query the `Schema` property and try to figure out which provider properties are fit to connect to which consumer properties. But other web part authors may also expect you to inform them about the types you may return and act upon it.

Passing a delegate to the provider

So why all these indirections? It is really all about timing. The web part framework connects the two parts by calling a method on the provider and immediately passing this value to a method on the consumer. But the framework has no idea what the provider wants to pass and when it's ready for action. So, instead of the value itself, a reference to the web part itself is passed, so the consumer can decide the appropriate time for retrieving the value (this is our `IStringValue` implementation). But in cases where the consumer doesn't really know the provider, how is the consumer supposed to know the "right time"? It doesn't. So it instead passes a delegate to a method on the consumer, effectively saying to the provider: "I'm ready when you are: pass me the info here". Then, it is up to the provider to decide when to pass the info. Obviously, after the `PreRender` stage, there is little point in passing the value anymore, as the

consumer has no opportunity anymore to act upon it, but the consumer should allow the value to be passed at any time from the moment of passing the delegate up to the `Render` stage. And of course, a provider may also decide not to pass any info after all.

In a sequence diagram, the behavior of this pattern is represented like this:



An interoperable implementation

So let's make an implementation of the connectable web parts from the start of this chapter using the standard interfaces from Microsoft. To begin with, we have to choose an interface for the provider. In our case, as we are only passing a simple string value, the `IWebPartField` interface is most applicable. First, we use the `IWebPartField` in the `ConnectionProvider` property:

```
public partial class InteroperableProvider : UserControl,
IWebPartField
{
  [ConnectionProvider("SimpleString")]
  public IWebPartField GetSelectedValue()
  {
    return this;
  }
}
```

To implement this interface, we have to create the `GetFieldValue` method and the `Schema` property.

```
FieldCallback _consumerCallback = null;
void IWebPartField.GetFieldValue(FieldCallback callback)
{
```

```
  _consumerCallback = callback;
  // We save this reference for later, when we are ready to provide
  // the value. In this case, we will do this at PreRender
}

PropertyDescriptor IWebPartField.Schema
{
  get
  {
    return TypeDescriptor.GetProperties(
              this.ListBox1)["SelectedValue"];
  }
}
```

We save the callback delegate that is passed for later use. The `Schema` property is more interesting: note that we use the `TypeDescriptor.GetProperties` static method to create a `PropertyDescriptor` for the `SelectedValue` property on the `ListBox` on the control. This is the actual property that we will eventually pass to the delegate method. Any consumer reading the `Schema` property will now know that we will pass a read-only string value named "SelectedValue". As soon as we know which value to pass (for arguments sake, we will do this in the `PreRender` stage), we call the delegate and pass in the correct value

```
void PreRenderHandler (object sender, EventArgs e)
{
  if (_consumerCallback != null)
  {
    _consumerCallback(ListBox1.SelectedValue);
  }
}
```

That's it for the provider. Now let's look at the consumer side. The consumer connection property must use the correct interface type:

```
IWebPartField _stringProvider = null;

[ConnectionConsumer("SimpleString")]
public void SetProviderPart(IWebPartField value)
{
  _stringProvider = value;
  // We could keep this reference for later use,
  // but in this case we will ask the provider to send the info now
  _stringProvider.GetFieldValue(new
           FieldCallback(SetProvidedValue));
}
```

We pass a new delegate into the `GetFieldValue` on the provider part. This means that the consumer is ready to receive the value from the connection,

but the provider may not pass the value immediately. However, when the value is passed, we know what to do: set the text of the label to the passed value:

```
public void SetProvidedValue(object value)
{
  Label1.Text = Convert.ToString(value);
}
```

Our consumer doesn't really check the `Schema` property, so it is not sure that the `Convert` class can actually convert the passed object to a string. You could argue that you should always use a `try/catch` in this situation, because you have really no way to tell what type is passed in.
In the next chapter, you will see how transformers can help you to connect parts that are not compatible and how standard transformers are available to connect for example a `IWebPartField` consumer to a `IWebPartRow` provider.

Multiple connections

It is certainly possible to have multiple connections on a page. A web part can have multiple connection end points (consumer or provider methods) and can be provider and consumer at the same time. One connection provider can provide information to many connection consumers, but a connection consumer can be connected to only one connection provider (you wouldn't want this otherwise when you think about it).
Web parts can be chained, where a web part is a consumer in one connection and a provider in another connection. This is a very common way to create master-detail scenarios with web parts. The first part shows a list of countries. If a country is selected, it provides the selected country to a second part displaying states or provinces. This part in turn provides the selected state to a third part displaying, say, the available vacancies in this area. You should beware of circular references: it is not illegal to create a circular set of parts, but when you get `StackOverflowException`, you know you should have thought it through first.
Note that in these more complex scenarios, it is imperative to use interfaces and to retrieve the data not immediately, but for example during the `PreRender` stage. The order of creation of all connections is not something you can influence or rely on. In these cases you really want to use your connections only after they have all been set up.

Cross-page connections

One of the questions often posed to Microsoft since the announcement of the ASP.NET Web Part Framework is "will you support cross-page connections?". After all, in SharePoint web parts it is possible to connect the output of a part on page 1 to the input of a page on page 2. The answer of Microsoft to this question has always been: "no, but if you really need it, you can build it yourself". The principal problem with cross-page connections is that to create connections across pages, you need to know which parts are available on other pages. In the ASP.NET architecture, the scope is very much limited to the page. The web part system is not aware of other pages, let alone of the web parts on those pages. Also, the concept of personalization makes connecting parts across pages a lot harder.
Nikhil Kothari, lead developer from the ASP.NET team wrote an article showing how it could be done (http://www.nikhilk.net/-CrossPageConnections.aspx). This solution works with proxy and stub parts that pass the information to the next page through the URL. So the provider part is connected to the proxy and passes its selected value. The proxy then causes a redirect to another page, passing its information on the URL. On the second page, a stub part retrieves the information from the URL and acts as a provider part on its page.
It is an interesting approach, but it is rather complex and setting up the parts in the browser is a lot of work and certainly not something that can be done by a business user.
The SharePoint platform also has included a modified web part manager that allows you to create cross-page connections. SharePoint is freely available on the Windows Server platform, so if you have the need for cross-page connections, introducing SharePoint in the equation may well be worth the extra complexity (See chapter 15).