

## Building Web Parts for Microsoft SharePoint Products and Technologies Part III - Connectable Web Parts

Patrick Tisseghem  
Company: U2U

### Applies to:

- Windows SharePoint Services
- Visual Studio .NET 2003

**Summary:** Part III of the SharePoint series discusses the basic steps of creating connectable Web Parts. Windows SharePoint Services provides a very good framework allowing users of these sites to drop Web parts onto their pages. We have discussed in [part I](#) how to create and deploy these Web Parts. [Part II](#) introduced a technique to boost the performance of developing Web parts by making use of ASP.NET user controls to deal with the design of the body of your Web part. In this article, we are going through the basic steps of creating two Web parts that are able to interchange data with each other. Developers of Web parts can implement standard WSS interfaces (provider and/or consumer) that are recognized by the WSS Web part framework and as a result allow users of the Web parts to make a connection between the two.

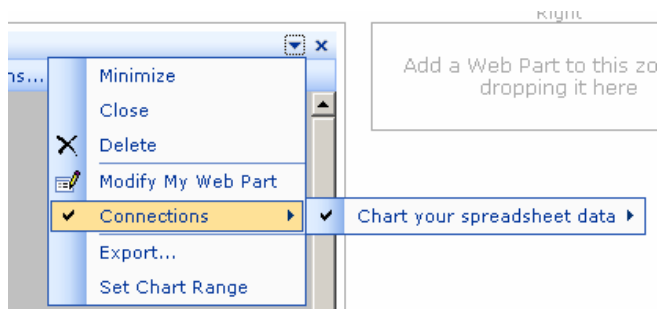
Download code: U2U-HR.zip

### Introduction

An exciting feature of the 'smart' pages in a SharePoint site is that users can drag and drop Web parts on a page and connect these Web parts with each other. These connections allow our Web parts to send and receive basic types of information. For those of you who have been doing these kinds of things with the previous version of SharePoint, you will experience that the framework provided now makes it much more easier, productive and richer to do your work.

Web part connections can be demonstrated very quickly with the Office Web parts that are available in your virtual server library.

Add an Office Spreadsheet Web part and a Pivot Chart Web part on the page. Connect to some data in the spreadsheet and put your page into design mode. Using the control menu of your Web part, you can connect the spreadsheet with the pivotchart in a very user-friendly manner.



The question of course is, can we add the same functionality to the Web parts we create? I hope to provide you the necessary information in this article.

## The Connections Framework

As mentioned, SharePoint (be it on a portal or on a team site) provides your Web parts with a connection framework that makes it possible for these Web parts to send and receive basic types of data. Examples of data are cell data, a complete row from a list or even a list itself.

The only thing that Web parts have to do in order to hook into this framework is to expose a standardized set of interfaces.



CategoryName	CategorySales
Beverages	102074.31
Condiments	55277.6
Confections	90994.14
Dairy Products	114749.78
Grains/Cereals	55948.82
Meat/Poultry	81338.06
Produce	53019.98
Seafood	65544.18

Provider exposes  
Connection Interfaces

Consumer exposes  
Connection Interfaces



The good thing about the whole connections framework in SharePoint is that we can have completely independent Web parts talking to each other and interchanging data with each other. It does not matter whether these parts are created by Microsoft, third party vendors, you or your co-workers. Once they expose the interfaces and once they are dropped on a page, the SharePoint Web part framework will automatically detect the connectability between them. The framework will even auto-detect the different combinations that are possible for doing the data interchange.

Important to note is that the connections can and will be made by end users and not directly by the developers themselves. Developers will create Web parts that expose the interfaces. Users will either in the browser itself or in FrontPage connect the parts with each other.

The interfaces to implement are well documented in the SDK you can download from the MSDN downloads. The basic set of interfaces we can use for connecting Web parts in the browser are

- the ICellProvider and ICellConsumer interface
- the IRowProvider and IRowConsumer interface
- the IListProvider and IListConsumer interface
- the IFilterProvider and IFilterConsumer interface

Each of these interfaces provide a different level of granularity with regard to the data that is transmitted from one Web part to the other.

## Example: Employee List and Employee Details

As always, I like to demonstrate the concept with an example. This time we will make use of two Web parts, one displays the list of employees (out of Northwind) visualized in a datagrid and a second one displays the details of an employee. The goal is to implement the ICellProvider interface in the first one and the ICellConsumer in the second one so that we can get the id of the employee transmitted to the second one.

Left

	First Name	Last Name	Hired On
Details	Nancy	Davolio	5/1/1992
Details	Andrew	Fuller	8/14/1992
Details	Janet	Leverling	4/1/1992
Details	Margaret	Peacock	5/3/1993
Details	Steven	Buchanan	10/17/1993
Details	Michael	Suyama	10/17/1993
Details	Robert	King	1/2/1994
Details	Laura	Callahan	3/5/1994
Details	Anne	Dodsworth	11/15/1994

Selected employee is: 3  
*Connected*

Right

Employee Details	
Details of Employee - Employee ID	
<b>Name:</b>	Janet, Leverling
<b>BirthDate:</b>	8/30/1963 12:00:00 AM
<b>Address:</b>	722 Moss Bay Blvd.
<b>Postal Code:</b>	98033
<b>City:</b>	Kirkland
<b>Country:</b>	USA
<b>Reports to:</b>	2

### Testing the User Controls

The download contains the two ASP.NET user controls to be wrapped within the Web parts together with an ASP.NET Web form you can use to test the functionality of them. Simply copy the folder containing the ASP.NET project (HR\_UserControls) under your local inetpub/wwwroot and mark the directory in IIS as an application directory.

Opening the ControlTester.aspx and reviewing the code, you will see how the user controls first get loaded using the Page.LoadControl and then added to the Controls collection of the Placeholder server control. This will also be the procedure to follow when we wrap these user controls in Web parts.

```
private void Page_Load(object sender, System.EventArgs e)
{
    Placeholder1.Controls.Add(Page.LoadControl("EmployeeList.ascx"));
    Placeholder1.Controls.Add(Page.LoadControl("EmployeeDetails.ascx"));
}
```

**Note:** It is possible that you have to modify the connection string with your specific SQL Server parameters.

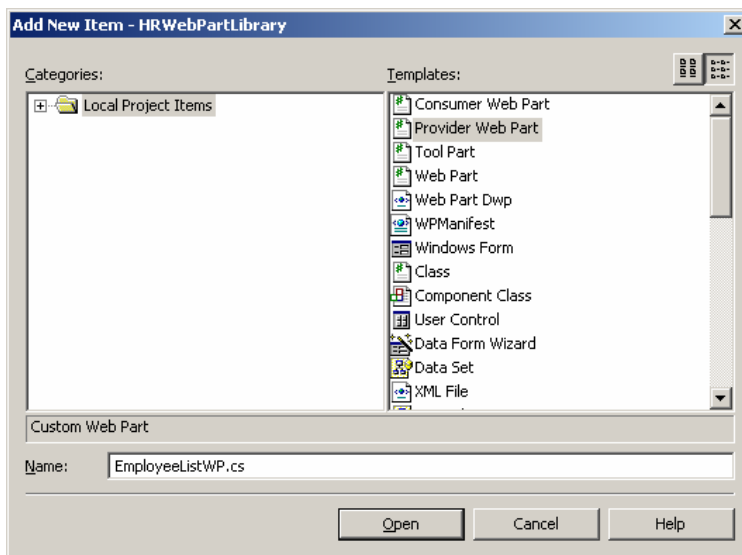
## Preparing your SharePoint folder

The virtual server hosting your SharePoint sites has a physical folder associated with it. For some of you this will be the Default Web Site (physically represented by inetpub\wwwroot). You could also have opted for extending a second virtual server with the SharePoint Services and host the SharePoint sites on that virtual server.

Any approach you go for, the ASP.NET user controls will have to be hosted under a new virtual directory (e.g. called UserControls) on that virtual server. You can create this directory in IIS. Last thing is to define, by using the SharePoint Central Administration (to be found in your Administrative Tools) this virtual directory as an excluded path. All these steps are explained in the article II you can read [here](#).

## Connectable Web Parts in VS.NET

You are ready now to create the two Web parts within a new Web part library project. The Visual Studio.NET Web Part project template (see [article I](#) for a full explanation) delivers two classes that already implement a provider (ICellProvider) or a consumer (ICellConsumer) interface. Note that there is no class template for the other interfaces. You will have to manually implement the other interfaces yourself.



Add a Web part named EmployeeListWP based on the provider Web part template and one EmployeeDetailsWP based on the consumer Web part template.

## Wrapping the ASP.NET User Controls

The Web part library project needs a reference to the assembly containing the code-behind of our ASP.NET user controls before we can load the user controls in our Web parts. Each of the Web parts loads an ASP.NET user control in three steps (see also [part two](#) of this series for more information):

1. Declaring a reference variable for the user control
2. Use the LoadControl() method of the Page object to load the control in the CreateChildControls you override from your base class.
3. Ask the loaded control to render itself in the RenderWebPart method.

Here are the snippets:

```
//-----
// Code to be inserted in EmployeeListWP
//-----

private U2U.Samples.WebParts.EmployeeList uc = null;

protected override void CreateChildControls()
{
    uc = (U2U.Samples.WebParts.EmployeeList)Page.LoadControl
        ("/UserControls/EmployeeList.ascx");
    this.Controls.Add(uc);
}

protected override void RenderWebPart(HtmlTextWriter output)
{
    this.EnsureChildControls();
    uc.RenderControl(output);
}

//-----
// Code to be inserted in in EmployeeDetailsWP
//-----

private U2U.Samples.WebParts.EmployeeDetails uc = null;

protected override void CreateChildControls()
{
    uc = (U2U.Samples.WebParts.EmployeeDetails)Page.LoadControl
        ("/UserControls/EmployeeDetails.ascx");
    this.Controls.Add(uc);
}

protected override void RenderWebPart(HtmlTextWriter output)
{
    this.EnsureChildControls();
    uc.RenderControl(output);
}
```

## ***Deploying the Web Parts and the User Controls***

The deployment of the Web parts follows the same steps as discussed in my previous articles. Instead of working with the Global Assembly Cache (GAC) as discussed in the previous articles, I will deploy everything in this example as private assemblies.

Here are the steps:

1. Add the DWP files to the project (one for each of your Web parts). Point to the correct assembly and set the type name correctly. This is the one for the details Web part.

```
<?xml version="1.0" encoding="utf-8"?>
<WebPart xmlns="http://schemas.microsoft.com/WebPart/v2" >
  <Title>Employee Details</Title>
  <Description></Description>
  <Assembly>HR-Parts</Assembly>
  <TypeName>U2U.Samples.WebParts.EmployeeDetailsWP</TypeName>
</WebPart>
```

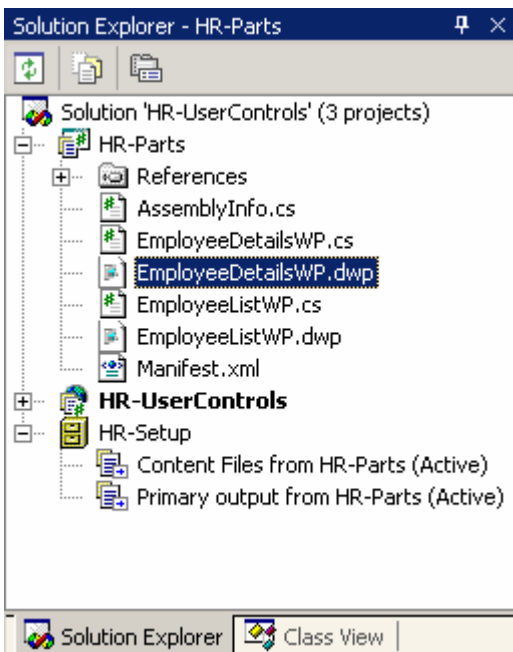
2. Modify the manifest.xml file. Set the namespace correctly and list the two DWP files.

```

...
<SafeControls>
  <SafeControl
    Namespace="U2U.Samples.WebParts"
    TypeName="*"
  />
...
<DwpFiles>
  <DwpFile FileName="EmployeeListWP.dwp" />
  <DwpFile FileName="EmployeeDetailsWP.dwp" />
</DwpFiles>

```

3. Create a CAB file project and include the project output and the content files. Note that in C#, the DWP files are not by default set to type content as is the case in VB.NET. You will have to do this manually via the properties box.



4. Use the STSADM.EXE tool to add the Web part package to the virtual server of your choice.

```

"C:\program Files\Common Files\Microsoft Shared\Web Server Extensions\60\bin\stsadm.exe" -force -o
addwppack -url http://limassol:300 -f HR-Setup.CAB

```

5. Make sure that both the Web part assembly (HR-Parts.dll) and the User Controls assembly (HR-UserControls.dll) are deployed in the \bin folder of your SharePoint virtual server folder.

6. Copy the two user control files (EmployeeList.ascx and EmployeeDetails.ascx) to the UserControls folder created before.

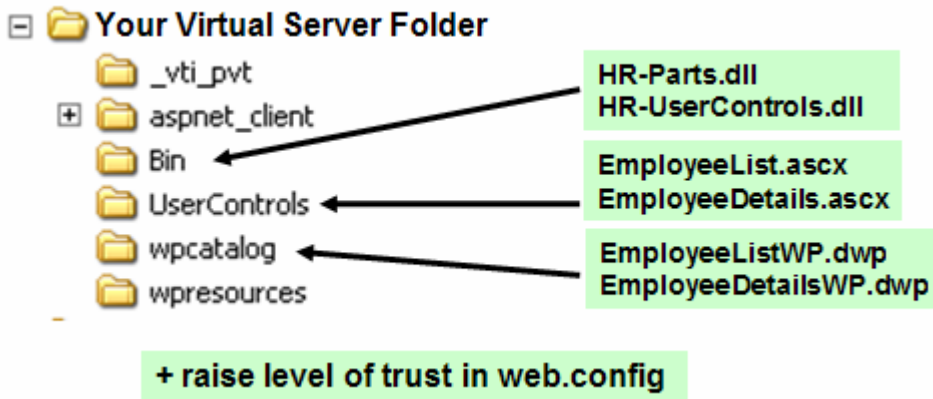
7. Modify the web.config of your virtual server and elevate the trust-level to WSS-Medium.

```

<trust level="WSS_Medium" originUrl="" />

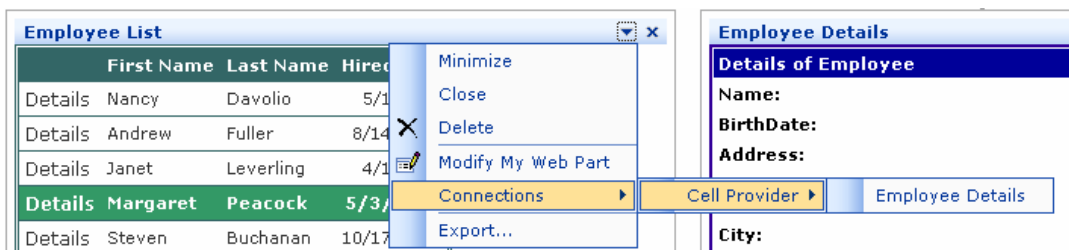
```

This picture summarizes everything. Note that you can also give a strong name and deploy your assemblies in the GAC. This procedure is discussed in the previous articles.



## Testing the Web Parts

At this moment, no interaction has been established for the two Web parts. However, the basic connection framework does already exist since we have one part implementing a provider interface and a second part implementing a consumer interface.



## Setting up the Provider and Consumer Interface

Let us now have a look at the implementation of the interfaces..

### The Provider Interface

Implementing the provider interface implicates that 3 members are to be added to your class: two events and one event-handler. The class template has already generated this code.

```
public event CellProviderInitEventHandler CellProviderInit;
public event CellReadyEventHandler CellReady;
```

The CellConsumerInit event-handler allows our provider to be notified of the type of the cell data the consumer is interested in.

```
public void CellConsumerInit
(object sender, CellConsumerInitEventArgs cellConsumerInitArgs)
{ }
```

## The Consumer Interface

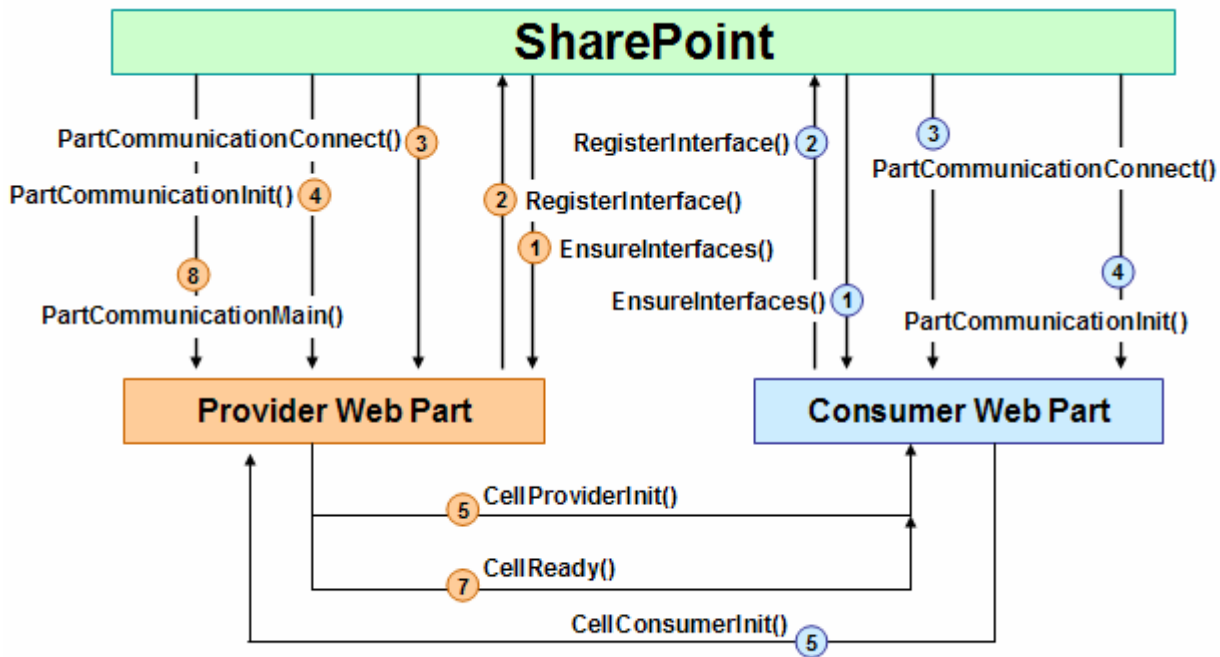
The same is true for the implementation of the consumer interface in our example. We also have 3 members: one event and two event-handlers. Again the class template has already generated this code for you.

## Overriding the Base Methods

Actually that is all you need to implement for the ICellProvider and the ICellConsumer interface. But it is not enough. The WebPart base class has 5 methods that need to be overridden in order to be hooked up in the Web Part Framework as a connectable Web Part.

- EnsureInterfaces()
- CanRunAt()
- PartCommunicationConnect()
- PartCommunicationInit()
- PartCommunicationMain()

In my [SharePoint course](#), I like to explain these methods using a small diagram on the black board. Looking at the whole picture, we have 3 major players when setting up the connection.



U2U – Connection Framework

## Registering your interfaces

After adding the Web part to the page, SharePoint will call the `EnsureInterfaces()` method. You have to use this method to register the interfaces you implement. In our example, we only have one interface. It is possible to implement multiple interfaces – e.g. both a provider and a consumer interface. The registration of your interface is done by calling the `RegisterInterface()` method with 8 parameters to be supplied:

- ***InterfaceName***  
Each interface implemented by the Web Part should get a unique name. Just like in the old days, you can prefix the name of your interface with a `_WPQ_` token that is recognized by the framework and replaced at runtime with a generated name for your Web Part.
- ***InterfaceType***  
The type of interface you are exposing. A number of constants are defined via the `InterfaceTypes` enum.
- ***MaxConnections***  
Developers can limit the amount of connections that can be made using this interface. This is most of the time used by consumer interfaces to protect the consumer Web Part for an overload of input. You have two options: one or unlimited.
- ***RunAtOptions***  
Connections can be activated and implemented on the client, on the server, or both. For this example we will concentrate on connections that are doing their work on the server.
- ***InterfaceObject***  
Simply a reference to the object implementing the interface.
- ***InterfaceClientReference***  
This parameter is only to be given a value when you go for client-side connections. In that case, you need to provide the identifier for the client-side object implementing the interface. Again use the `_WPQ_` token to generate a unique ID. Use `String.Empty` for server-side connections.
- ***MenuLabel***  
This is the label that the user will see when they click on the connections submenu.
- ***Description***  
This is a more detailed explanation only displayed in specific authoring environments such as FrontPage 2003.

We need to do this in both our provider and our consumer.

```
// EmployeeListWP - The Provider

public override void EnsureInterfaces()
{
    try
    {
        RegisterInterface
            (
                "EmployeeIDProvider_WPQ_",
                InterfaceTypes.ICellProvider,
                WebPart.UnlimitedConnections,
                ConnectionRunAt.Server,
                this,
                String.Empty,
                "Communicates Employee ID to",
                "Passes the primary key of the list"
            );
    }
    catch(SecurityException e)
    { }
}

// EmployeeDetailWP - The Consumer

public override void EnsureInterfaces()
{
    try
    {
        RegisterInterface
            (
                "EmployeeDetail_WPQ_",
                InterfaceTypes.ICellConsumer,
                WebPart.LimitOneConnection,
                ConnectionRunAt.Server,
                this,
                String.Empty,
                "Display detail for employee coming from",
                "Connect to this part to have details of employee listed"
            );
    }
    catch(SecurityException e)
    { }
}
```

## ***Type of Connection***

Another base method that you have to override is the `CanRunAt()`. During the registration of your interface you have already communicated this to SharePoint. The `CanRunAt()` method is called by SharePoint to request your connection mode when needed. This provides you with a mechanism to change the connection mode at run time maybe based on certain conditions. In our example, we do not really need it. Both the provider and consumer do not have to be changed.

```
public override ConnectionRunAt CanRunAt()
{
    return ConnectionRunAt.Server;
}
```

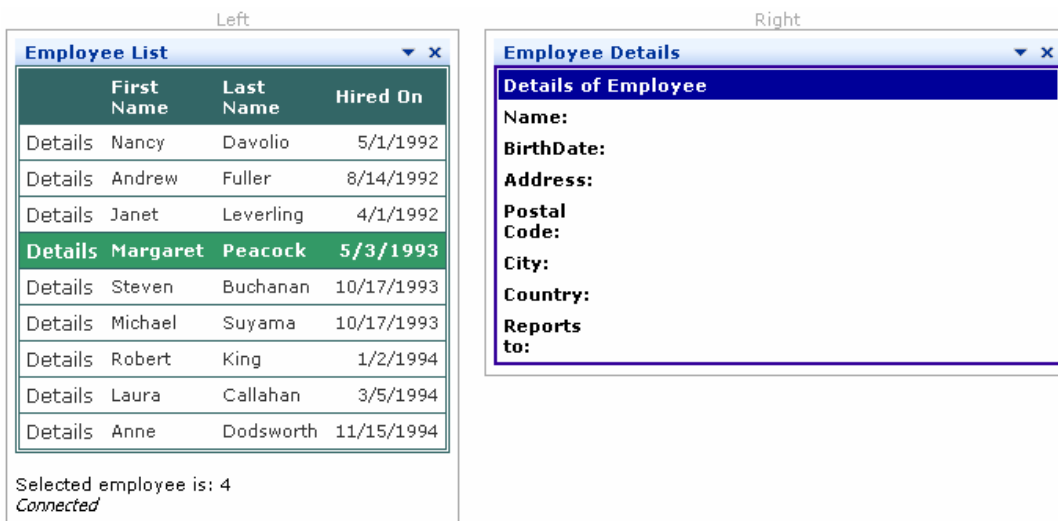
## Adapting to the Connected State

Visitors of the Web part page are able to make use of the context menu (after having switched the page in design mode) to connect the two Web parts with each other. The moment the connection is made with success, SharePoint will call the `PartCommunicationConnect()` method of both the provider and the consumer. The two Web parts then internally know that a connection has been established. Web part developers can use this to adapt their Web part and for example visualize to the user that there is a connection now.

A number of arguments can be used providing some context information regarding the connection made:

- ***interfaceName***  
the friendly name of the provider interface that has been connected
- ***connectedPart***  
a reference to the consumer Web part itself
- ***connectedInterfaceName***  
the friendly name of the consumer interface that has been connected
- ***runAt***  
whether this was done on the server or on the client

For example, the user control displaying the list of employees has a label displaying the status of the Web part. Default it is set to 'not connected'. Using `PartCommunicationConnect()` procedure, we can change this label.



```
// EmployeeListWP only - EmployeeDetailsWP remains like it is
public override void PartCommunicationConnect
    (string interfaceName, WebPart connectedPart,
     string connectedInterfaceName, ConnectionRunAt runAt)
{
    EnsureChildControls();
    uc.labelStatus.Text = "Connected";
}
```

## Sending Notifications when Connected

The two Web parts that are connected can notify each other by triggering each an event. The provider is triggering the `CellProviderInit()` and the consumer the `CellConsumerInit()`. The place where you do this is the `PartCommunicationInit()` method you override from your base class. This method is called right after SharePoint has called your `PartCommunicationConnect()` method.

You will notice that there is already some code added to this procedure. In our example, the provider sends the consumer the type of information (employee id) it is going to send. The consumer can do the same thing. You can for example think of a generic consumer Web part able to display the details of an employee, a customer or any other entity.

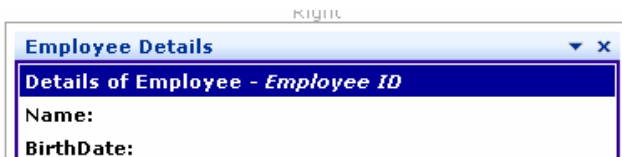
```
// EmployeeListWP only

public override void PartCommunicationInit()
{
    //If there is a listener, send init event
    if (CellProviderInit != null)
    {
        //Need to create the args for the CellProviderInit event
        CellProviderInitEventArgs cellProviderInitArgs =
            new CellProviderInitEventArgs();

        //Set the FieldName
        cellProviderInitArgs.FieldName = "Employee ID";

        //Fire the CellProviderInit event.
        //This basically tells the Consumer Web Part what type of
        //cell it will be receiving when CellReady is fired later.
        CellProviderInit(this, cellProviderInitArgs);
    }
}
```

On the other side, the consumer will handle this event in its `CellProviderInit()` event-handler. In our example, the provider is sending the field it is going to use for the data. We will make use of it and display it in a label in the employee details user control.



```
public void CellProviderInit
    (object sender, CellProviderInitEventArgs cellProviderInitArgs)
{
    this.uc.labelField.Text = cellProviderInitArgs.FieldName;
}
```

## Sending the data

Time now to send the data! SharePoint will make a call to the PartCommunicationMain() method whenever a postback is done – that is, whenever the user makes a selection in our data grid. It is here that we package the data and send it to the consumer via the CellReady() event we trigger.

Again, the template gives us the starting code. We modify the code a bit and assign the data we want to send to the consumer. In our case, this is the selected employee id exposed by the user control via the SelectedEmployeeID property.

```
// EmployeeListWP !!

public override void PartCommunicationMain()
{
    //If there is a listener, send CellReady event
    if (CellReady != null)
    {
        //Need to create the args for the CellProviderInit event
        CellReadyEventArgs cellReadyArgs = new CellReadyEventArgs();

        //Set the Cell to the value of the TextBox text
        //This is the value that will be sent to the Consumer
        cellReadyArgs.Cell = this.uc.SelectedEmployeeID;

        //Fire the CellReady event.
        //The Consumer will then receive the Cell value
        CellReady(this, cellReadyArgs);
    }
}
```

The consumer must create an event-handler to process the data sent along with the event. In our example, the user control showing the details of the employee has a method we can call with the employee id as the parameter. This refreshes the contents of the Web part and the result are the details of the selected employee in the list.

Left

Employee List			
	First Name	Last Name	Hired On
Details	Nancy	Davolio	5/1/1992
Details	Andrew	Fuller	8/14/1992
Details	Janet	Leverling	4/1/1992
Details	Margaret	Peacock	5/3/1993
Details	Steven	Buchanan	10/17/1993
Details	Michael	Suyama	10/17/1993
Details	Robert	King	1/2/1994
Details	Laura	Callahan	3/5/1994
Details	Anne	Dodsworth	11/15/1994

Selected employee is: 3  
*Connected*

Right

Employee Details	
Details of Employee - Employee ID	
<b>Name:</b>	Janet, Leverling
<b>BirthDate:</b>	8/30/1963 12:00:00 AM
<b>Address:</b>	722 Moss Bay Blvd.
<b>Postal Code:</b>	98033
<b>City:</b>	Kirkland
<b>Country:</b>	USA
<b>Reports to:</b>	2

```
public void CellReady(object sender, CellReadyEventArgs cellReadyArgs)
{
    if(cellReadyArgs.Cell != null)
    {
        this.uc.SetDetails(int.Parse(cellReadyArgs.Cell.ToString()));
    }
}
```

If you redeploy nicely and test the Web parts you will see the effects of the previous modifications.

## Conclusion

Creating connectable Web parts is not difficult if you understand the flow of registering your interface, the Web part base methods getting called by SharePoint and triggering the events. This flow is common (with some minor changes) for the different types of interfaces supported in SharePoint. In this article, I have discussed a small example where two user controls have been wrapped into Web parts, and then we have added the necessary coding to connect them with each other.

## About the author



Patrick Tisseghem is managing partner of U2U nv/sa. He has been working with and teaching Microsoft .NET since its early alpha releases and always has had an interest in the concept of document management and collaboration. He has created his own 5-day course on the development aspects of SharePoint and is a frequent speaker for Microsoft on this topic.

You can contact him at [patrick@u2u.be](mailto:patrick@u2u.be) or read his blog on <http://radio.weblogs.com/0126624/>. U2U is Microsoft CTEC based in Brussels, hosts the Microsoft Regional Directorship and is specialized in .NET development.