EPiSERVER®

ASP.NET 2.0 Web Parts in EPiServer

Product version:	EPiServer 4.60
Document version:	1.0
Document creation date:	11-05-2006

Purpose

ASP.NET 2.0 introduces Web Parts as a set of controls for building portal-like pages. EPiServer makes use of this support from version 4.60 and extends it in a number of ways. This document describes the setup requirements and outlines the features that are built into the EPiServer Web Part support.

TECHNICAL NOTE

Table of Contents

INTRODUCTION 2

OVERVIEW 2

TEMPLATES 3

CONFIGURATION 5

ADMINISTRATION 7

EXAMPLE 1 – DISPLAYING EPISERVER CONTENT IN A WEB PART 8

EXAMPLE 2 – DISPLAYING EPISERVER PAGE LISTINGS IN A WEB PART 14

EXAMPLE 3 – IMPLEMENT A CONNECTION BETWEEN WEB PARTS 17

FURTHER REFERENCES 20

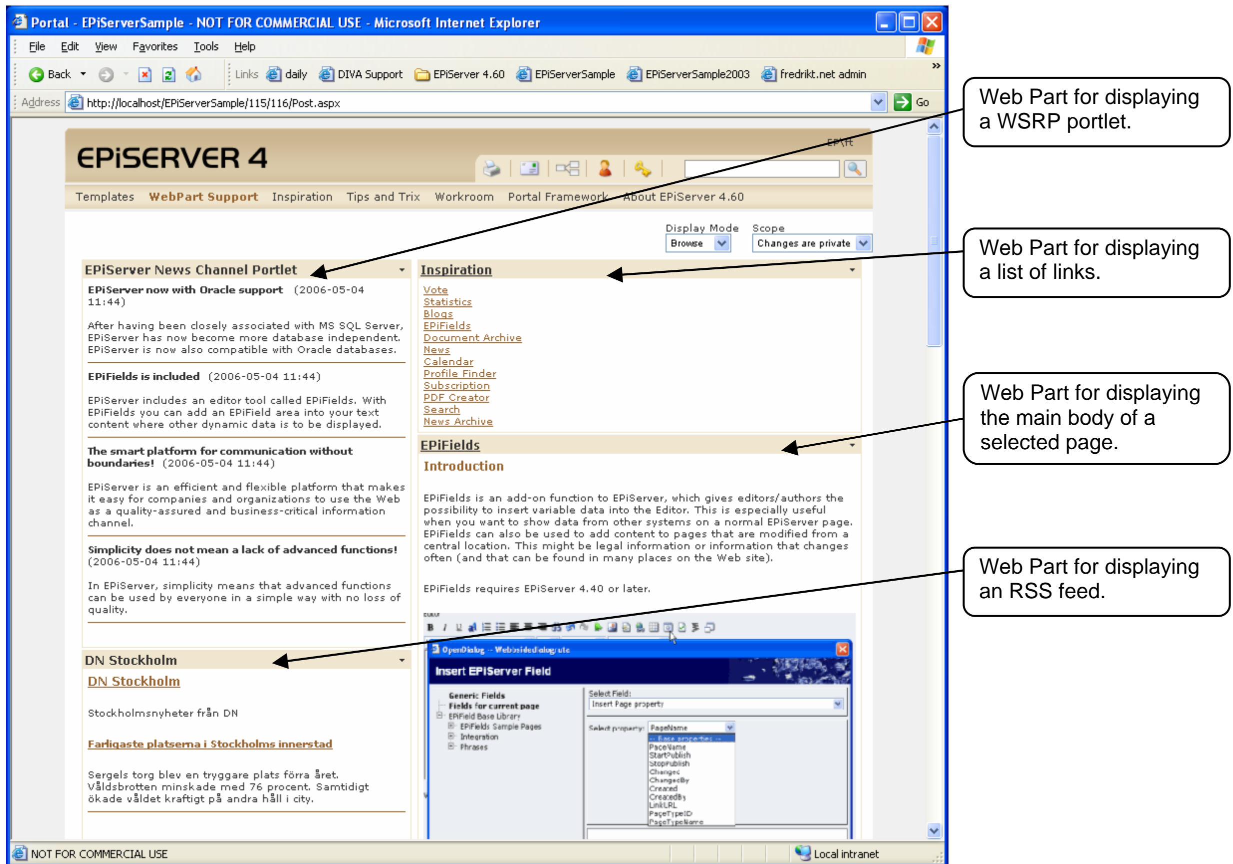
APPENDIX – API DOCUMENTATION 21

Introduction

Web Parts is a framework for building portal-like, highly personalized pages in ASP.NET 2.0. The technology is known from the Microsoft SharePoint product family, where it has been available for a number of years. The release of .NET 2.0 made this technology available to ASP.NET developers not specifically targeted to SharePoint environments. In EPiServer 4.60, which is the first release that targets ASP.NET 2.0, we use Web Parts for our portal templates.

Overview

Web Parts are typically used for building portal-like Web applications. The framework has built-in support for various types of customization of the user interface (personalization) and provides a number of controls to back this up. All controls and classes defined in the framework are designed with extensibility in mind, so there are a lot of opportunities to plug in your own code for altering things like rendering and behavior.



The image above displays what a Web Parts page looks like when a couple of Web Parts have been applied. The example shows a Web Part displaying an RSS feed and a WSRP portlet in the left zone, and two Web Parts, one for displaying a page list and one for the contents of a page in the right zone.

The Web Parts in the right zone are connected, i.e. clicking a link in the list of pages in the upper Web Part will automatically update the Web Part below to display the content for the selected page. This view can be “private”, where the user that is logged specifies what should be displayed on the page, or it can be a view that is shared for all users (including anonymous users).

The page shows the Portal.aspx template, which is included in the EPiServer installation package for the Web Part support. The installation package also includes a template for displaying a combination of traditional content and Web Part content.

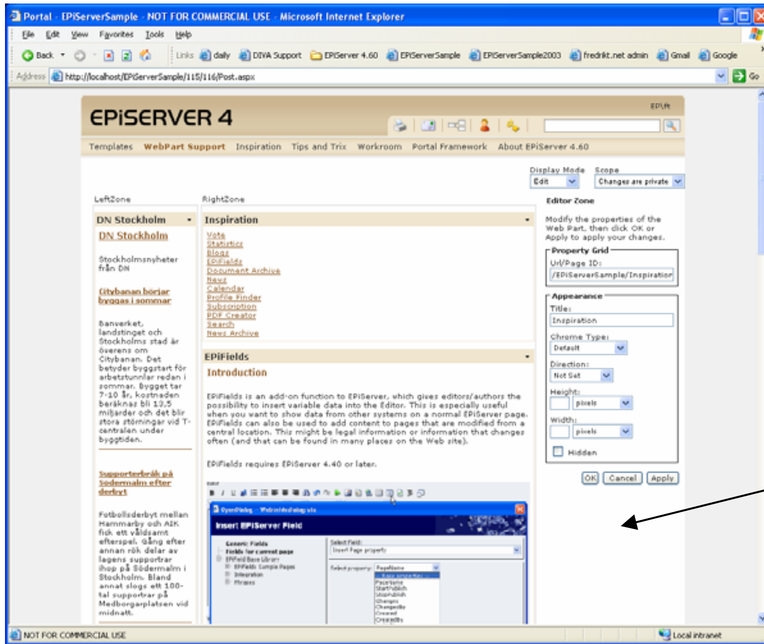
Templates

The EPiServer installation includes two templates that exemplify how to use the Web Parts features in general and the EPiServer extensions in particular.

Portal.aspx

The portal template is used for traditional portal scenarios where all content is viewed using Web Parts. This template supports most of the various display modes that are available on a Web Part page. The page contains controls for switching between the different displays modes and for switching between private and shared personalization scope.

The image below displays the portal template displaying the Web Part Edit user interface for the Page List Web Part.



Modification of content, appearance and behavior can be done from the various display modes. Additional user interface for changing settings will show up to the right.

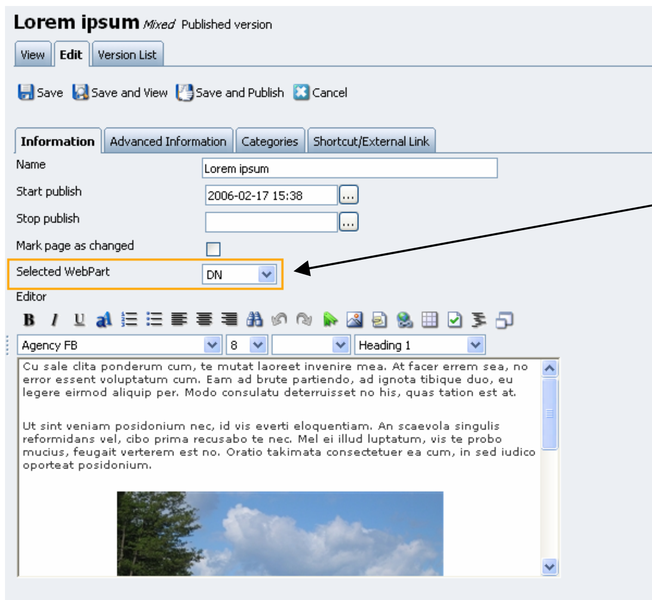
In this example the user has selected Edit from the pop-up menu of the Page List Web Part.

Mixed.aspx

In situations where you want a traditional Web page look, but want to expose a Web Part in a section of the page, you can make the **WebPartProperty** property of the **ExtendedWebPartZone** point to an EPiServer page property of type WebPart. With this configuration the editor will be able to define which Web Part should be displayed in the zone.

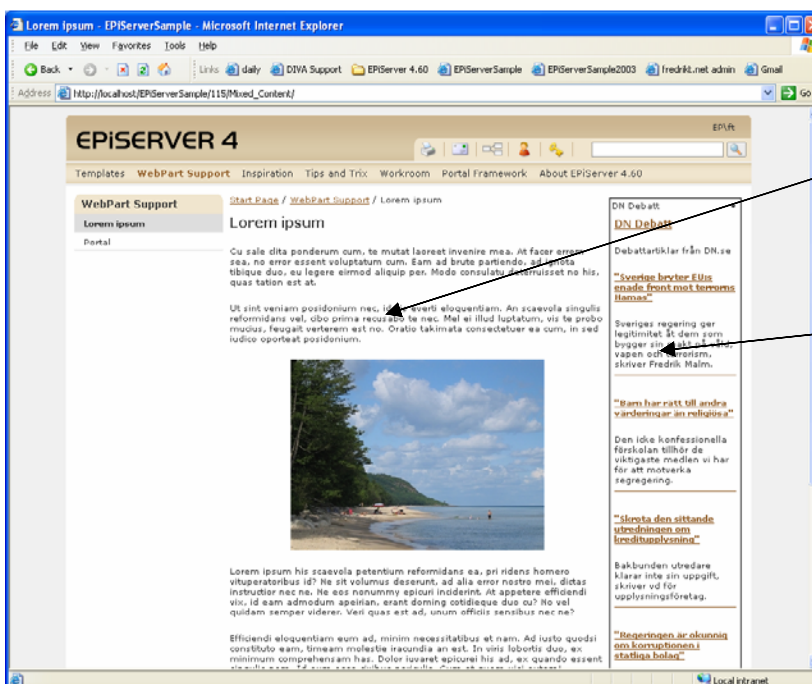
This feature is implemented in the Mixed.aspx template, which basically replaces the right content placeholder with an **ExtendedWebPartZone**. The template expects a WebPart property named “SelectedWebPart” and a XHTML property named “MainBody” to be defined for the page type.

The image below displays the Edit tab of EPiServer Edit mode for the "Mixed" content template. The SelectedWebPart property is displayed as a drop-down list of available Web Parts. The set of available Web Parts are defined in EPiServer's Admin mode.



When switching to EPiServer Edit mode, the WebPart property will show a list of available Web Parts to choose from.

The image below displays the "Mixed" content template showing the page tree in the left column, the main body in the main content area and a Web Part in the right column.



Main content and the navigation displayed in the same way as you see in the Ordinary Web Page template.

Web Part displaying an RSS feed.

Configuration

This chapter outlines the various aspects concerning the configuration of Web Parts support.

Install ASP.NET 2.0

In order to use and develop pages with Web Part support in EPiServer, the site must be running on .NET 2.0. The steps required to set this up are described in the technical note "WSRP Configuration".

Sample Package

There are two options if you want to use Web Parts in your site.

1. Install the entire template package for ASP.NET 2.0.
2. If you are upgrading an existing site, you may want to manually install the files required just for Web Parts. See a list of the files below.

The sample templates for ASP.NET 2.0 contain the following crucial files that must exist for Web Parts to work.

- /admin/webpartadmin.aspx
- /admin/webpartadminedit.aspx
- /bin/episerver.webparts.dll
- /templates/masterpages/masterpage.master
- /templates/webparts/mixed.aspx
- /templates/webparts/mixed.aspx.cs
- /templates/webparts/portal.aspx

- /templates/webparts/portal.aspx.cs
- /templates/webparts/portal.aspx.designer.cs
- /templates/webparts/scripts/rss.js
- /templates/webparts/units/rsscontrol.ascx
- /templates/webparts/units/rsscontrol.ascx.cs
- /templates/webparts/units/rsscontrol.ascx.designer.cs
- /templates/webparts/webcontrols/rsszone.cs

web.config

In order to make the database available for the Web Parts framework, the database connection string needs to be published in the ASP.NET 2.0 `connectionsStrings` element. Follow the instructions below to configure web.config.

1. Add the following element as a child element to the `configuration` element.

```
  <connectionStrings>
    <add name="EPiServerDB" connectionString="Data
Source=SERVER;Database=DATABASE;User Id=USER;Password=PASSWORD;Network
Library=DBMSSOCN;" providerName="System.Data.SqlClient" />
  </connectionStrings>
```

2. Replace `SERVER`, `DATABASE`, `USER` and `PASSWORD` with your values.

```
Make sure that the Web Parts tag prefix is registered in the pages element.
    <pages validateRequest="false" enableEventValidation="false">
      <controls>
        <add tagPrefix="EPiServer" namespace="EPiServer.WebControls"
assembly="EPiServer" />
        <add tagPrefix="WebParts" namespace="EPiServer.WebParts.WebControls"
assembly="EPiServer.WebParts" />
        <add tagPrefix="WebParts"
namespace="EPiServer.WebParts.WebControls.Wsrp" assembly="EPiServer.WebParts"
/>
      </controls>
    </pages>
```

3. The personalization default engine used for extracting and persisting personalized data is the `SqlPersonalizationProvider`. EPiServer uses an altered version of the `SqlPersonalizationProvider` named `EPiServerPersonalizationProvider` which handles the multiple pages per page template model used in EPiServer. You enable this personalization provider by adding the following element somewhere inside the `system.web` element.

```
    <webParts>
      <personalization defaultProvider="EPiServerPersonalizationProvider">
        <providers>
          <add name="EPiServerPersonalizationProvider"
type="EPiServer.WebParts.Core.EPiServerPersonalizationProvider"
connectionStringName="EPiServerDB" />
        </providers>
        <authorization>
          <allow verbs="enterSharedScope" roles="WebAdmins, WebEditors,
Administrators" />
        </authorization>
      </personalization>
    </webParts>
```

4. Make sure to point out the correct connection string in the `connectionStringName` attribute of the `add` element. The section above also sets the permission for entering shared scope to only allow WebAdmins and WebEditors.

Configuring the Database

The personalization provider relies on a specific database schema. This schema can easily be set using the `aspnet_regsql` command line tool found in `%windir%\Microsoft.NET\Framework\v2.0.50727\`. This tool can be used to set up specific parts or the entire database schema necessary for the available ASP.NET 2.0 providers (membership, roles, personalization, profiles and Web events).

The following command enables the entire ASP.NET 2.0 scheme using SQL authentication:

```
aspnet_regsql /S SERVER /d DATABASE /U USER /P PASSWORD /A all
```

The following command does the same thing using integrated security:

```
aspnet_regsql /S SERVER /d DATABASE /E /A all
```

The following command installs the minimum parts necessary for the Web Parts support, i.e. personalization and membership. The example uses integrated security:

```
aspnet_regsql /S SERVER /d DATABASE /E /A mc
```

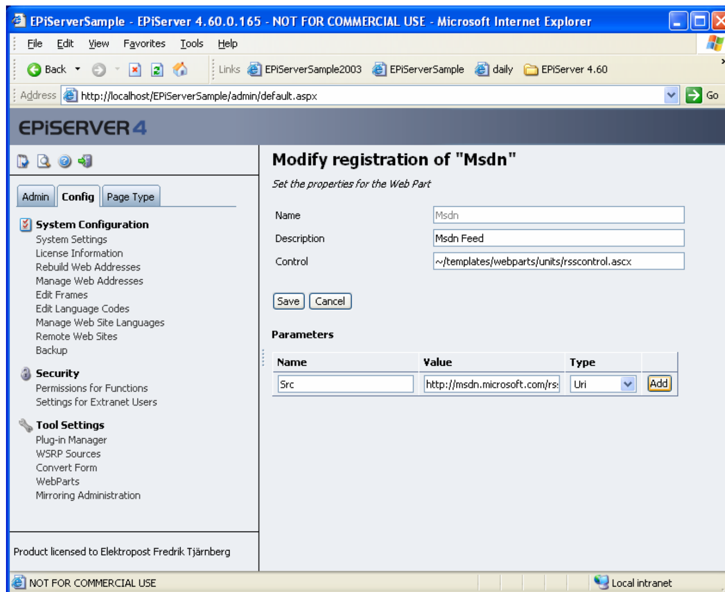
Administration

Web Part Registration

In order to make it possible to change the set of available Web Parts in a live site, there is a new administration page for registering and altering Web Parts. Web Parts that have been registered are available from the custom WebPart property that can be added to a page type. It is also available from the RegisteredWebPartsCatalog that can be added to any catalog zone in the EPiServer site. The registration process makes it possible to extend the available services of the site without having to recompile or even change any markup files.

Register a Web Part

1. From the Config tab in Admin mode, click Web Parts and **Install Web Part**.
2. Enter a name, description and specify the control to be used by entering its physical path relative to the application root. You specify the application root using the tilde character (~). Enter "~/templates/webparts/units/rsscontrol.ascx" in the Control field to point to the RSS control available in the sample package. Click **Save**.
3. You can specify properties of the Web Part by selecting its name.
4. In the properties table, enter values for the properties you need to specify.



5. The Src property of the RssControl in the above example is set to point to the MSDN feed at http://msdn.microsoft.com/rss.xml.
6. Click **Add** to define the property.
7. Remove any property by selecting **Delete** in the table.

Register a WSRP Portlet

If you need to expose WSRP portlets, you can make them available by registering them in a similar way.

1. From the WebParts page in Admin mode, select **Consume Portlet**.
2. Enter a name and a description. Select one of the producers registered in the WSRP Sources section of Admin mode in the Producer field.
3. Select one of the available portlets for the selected producer. Click **Save**.

Example 1 – Displaying EPiServer Content in a Web Part

This example shows how to build a simple Web Part for displaying EPiServer content. The idea is to display a configurable property from a configurable page.

Derive from a Web Part?

In ASP.NET 2.0 there are three main choices when selecting a base class for a Web Part control.

1. **Deriving from System.Web.UI.WebControls.WebParts.WebPart** will give you a tight integration with the Web Part framework. The (major) drawback is that you must provide the markup of your control using code.

2. **Deriving from System.Web.UI.WebControls.UserControl** gives you the possibility to separate code and layout, but you will not have all the features of Web Part deriving control. Most of the features can be “re-engineered”.

3. **Derive from any other member of the System.Web.UI.WebControls namespace**, or any other Web control. You do this to extend or customize an existing control.

The example uses markup to define the layout, so we create a new user control and name it PropertyPart.ascx.

Markup

The markup for the control is trivial.

```
<%@ Control Language="C#" AutoEventWireup="true"
   CodeBehind="PropertyPart.ascx.cs"
   Inherits="EPiServerSample.WebParts.PropertyPart" %>
<asp:Label runat="server" ID="Message" Visible="false" Font-Bold="true" />
<EPiServer:Property runat="server" PropertyName="" ID="PropertyControl" />
```

Note that the @Control declaration still contains a code-behind file, although the code is targeted for ASP.NET 2.0. This is because the control is created in a Web Application Project which is how the sample project for ASP.NET 2.0 is packaged.

In the markup we simply add two control declarations; a Label for displaying messages and a Property control for displaying the EPiServer content.

Code-Behind File

Code view will look something like this:

```
using System;
using System.Data;
using System.Configuration;
using System.Collections;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;

namespace EPiServerSample.WebParts
{
    public partial class PropertyPartEx : System.Web.UI.UserControl
    {
        private string _url;
        private EPiServer.Core.PageData _pageData;

        protected void Page_Load(object sender, EventArgs e)
        {

        }
    }
}
```

There is also a way to specify which page and property to display. There are two new properties for this: Url and PropertyName.

```
        [Personalizable]
        [WebBrowsable]
        [WebDisplayName("Url/Page ID")]
        public string Url
        {
```

```
            get { return _url; }
            set {
                _url = value;
            }
        }

        [Personalizable]
        [WebBrowsable]
        public string PropertyName
        {
            get { return PropertyControl.PropertyName; }
            set { PropertyControl.PropertyName = value; }
        }
```

In the code above, the `Personalizable` attribute is used to let the Web Part framework know that the data in this property should be persisted in the personalization for the page where the Web Part is used. The `WebBrowsable` attribute makes the property available for user editing when the page is displayed in Web Part Edit mode. The `WebDisplayName` sets the caption of the property when displayed in Edit mode.

The default behavior displays the property name. The declaration of the WebDisplayName requires a constant expression for the DisplayName argument, but what if you want to provide a translated text? To do this you need to extend the WebDisplayName by deriving from it and provide an overridden version of the DisplayName property that returns a translated text.

The `PropertyName` property just wraps the `PropertyName` property of the contained EPiServer Property control in order to hook up the personalization engine with its value. The URL is stored in a field since it needs some extra attention before we can use it.

The Property control we added in the markup expects a `PageReference` in order to display any content. For this we need a way to transform the URL provided by the user into a `PageReference` object. This is done by adding another read-only property named `PageData`.

```
public EPiServer.Core.PageData PageData
{
    get
    {
        if (_pageData == null && Url != null)
        {
            try
            {
                int id;
                if (Int32.TryParse(Url, out id))
                {
                    _pageData =
                       EPiServer.Global.EPDataFactory.GetPage(
                          new EPiServer.Core.PageReference(id));
                }
                else
                {
                    _pageData =
                       EPiServer.Global.EPDataFactory.GetPageByURL(
                         _url);
                }
            }
            catch (EPiServer.Core.PageNotFoundException x)
            {
                    /* Error processing */
            }
```

```
            }
            return _pageData;
        }
    }
```

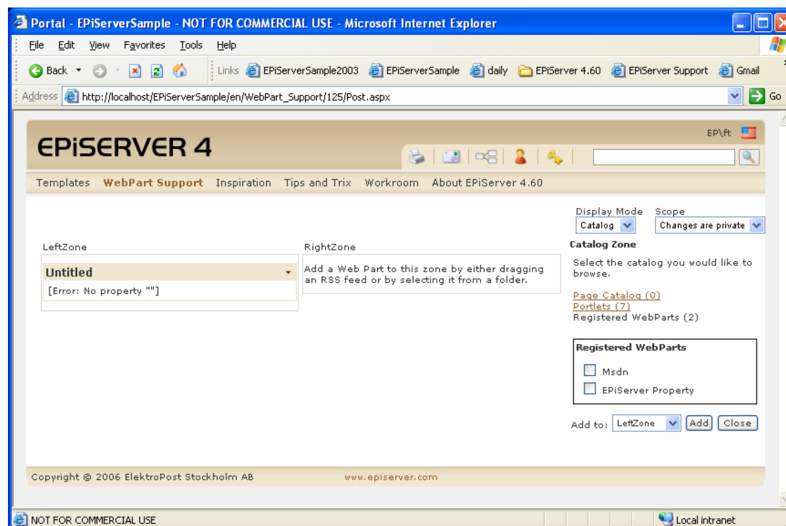
This property uses a field, `_pageData`, to cache its calculated value, so if the field is set to anything other than null, the field value is returned. If however the field is null, we need to create an instance based on the value given by Url. In our case we accept the value to be a relative URL to a page in the site or an integer representing the page ID. So when calculating the `_pageData` value, we first make an attempt to parse it as an integer using the new convenient `TryParse` method. If this succeeds, we simply retrieve a reference to the page using the `GetPage` API. If the URL was not in numeric format, we use `GetPageByUrl` to look up the page using its relative URL. We skip the error handling code that is needed to handle lookup failures.

Now everything is set up to feed the Property control with a page reference. We do this by overriding the `OnPreRender` method.

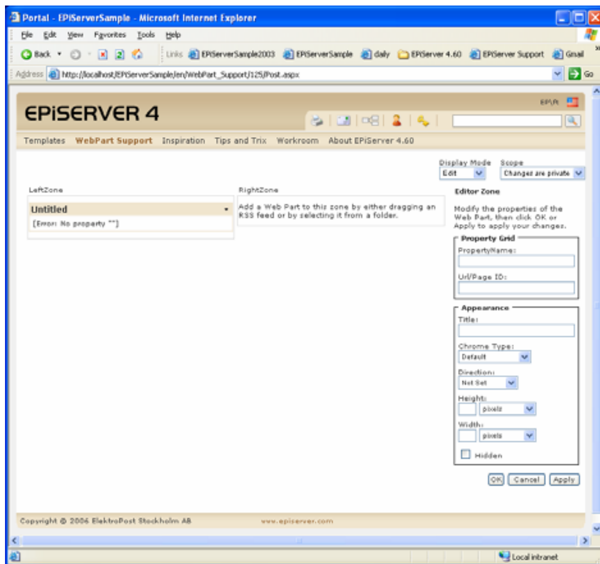
```
protected override void OnPreRender(EventArgs e)
{
    if (PageData != null)
    {
        PropertyControl.PageLink = PageData.PageLink;
    }

    base.OnPreRender(e);
}
```

The simplest way to expose this Web Part in your site is to register it in Admin mode. By doing this, the Web Part will be available from the Registered Web Parts catalog in the Portal template. Registration is done by entering "~/WebParts/PropertyPart.ascx" in the Control field in the Install Web Part window in Admin mode.

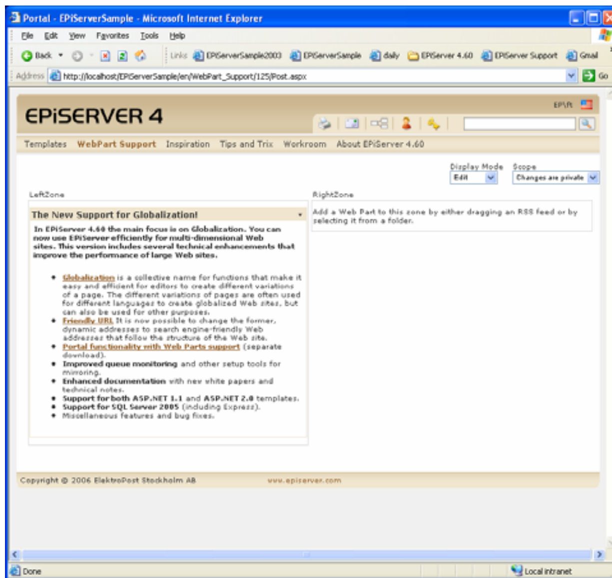
When you browse a page that uses the Portal template and switch display mode to Catalog mode, you will see the EPiServer Property under the Registered Web Parts as follows. When it is added to the LeftZone, it will look something like this:



If you switch to Edit mode and select Edit from the pop-up control menu of the Web Part, you will see the user interface for modifying the Web Part.



Enter a property name (MainBody in this case), the path to a page in your site, e.g. /EPiServerSample/Inspiration/News/The_New_Support_for_Globalisation/, click OK and the Web Part will render the main body of the specified page.



You might notice that the result you get differs slightly from the page shown above. Instead of getting the name of the page in the Web Part title bar, you get the less informative “Untitled”. This is because in the example above the control implements the `IWebPart` interface, which gives the Web Part framework some additional information on how to render standard elements as icons and titles. The implementation is very simple.

In your class add the following to the class declaration.

```
…
public partial class PropertyPart : System.Web.UI.UserControl, IWebPart
{
…
```

If this is done in Visual Studio 2005, you will receive a smart tag giving you the option to implement the interface you just added.

A typical implementation will look something like this.

```
// IWebPart backing variables
private string _title;
private string _catalogIconImageUrl;
```

```
private string _description;
private string _subtitle;
private string _titleIconImageUrl;
private string _titleUrl;

#region IWebPart Members

public string CatalogIconImageUrl
{
    get { return _catalogIconImageUrl; }
    set { _catalogIconImageUrl = value; }
}

public string Description
{
    get { return _description; }
    set { _description = value; }
}

public string Subtitle
{
    get { return _subtitle; }
    set { _subtitle = value; }
}

[Personalizable]
public string Title
{
    get
    {
        if (_title == null && PageData != null)
            return PageData.PageName;
        else
            return _title;
    }
    set
    {
        if (value == string.Empty)
            _title = null;
        else if (PageData != null && value != PageData.PageName)
            _title = value;
    }
}

public string TitleIconImageUrl
{
    get { return _titleIconImageUrl; }
    set { _titleIconImageUrl = value; }
}

public string TitleUrl
{
            get
```

```
            {
                if (PageData == null) return _titleUrl;
                return PageData.LinkURL;
            }
            set { _titleUrl = value; }
        }

        #endregion
```

The only thing that we treat differently than just getting and setting the value of a field variable is the implementation of `Title` and `TitleUrl`. `Title` checks its corresponding field variable for null and if that is the case attempts to retrieve the page name from the `PageData` property. Likewise, `TitleUrl` is simply the `LinkUrl` property of the `PageData` property.

Example 2 – Displaying EPiServer Page Listings in a Web Part

In this example we will build a Web Part that lists pages from a specified location of the site tree. Each page will be represented with a link to the page.

First add a new Web User Control to your project named `PageListPart.ascx`.

Markup

In the markup, add a `PageList` control that puts out link buttons as follows.

```
<EPiServer:PageList runat="server" ID="PageList">
    <ItemTemplate>
        <asp:LinkButton runat="server"
            ID="LinkButton"
            OnCommand="PageClick_Command"
            CommandArgument="<%# Container.CurrentPage.PageLink.ID %>">
                <%# Container.CurrentPage.PageName %>
        </asp:LinkButton><br />
    </ItemTemplate>
</EPiServer:PageList>
```

In this code we could have replaced the `LinkButton` with a standard HTML anchor element, but in the next example we will add extra processing in the event handler. This means that we are dependent on a postback, so we use the slightly more complicated `LinkButton` server control.

Code-Behind

The markup will require a collection of pages as a data source and it will also need an event handler for the `PageClick_Command`.

In the code-behind we add the following members.

```
private string _url;
private EPiServer.Core.PageData _pageData;
private string _pageDataUrl;
private string _boundUrl;

[Personalizable]
[WebBrowsable]
```

```
[WebDisplayName("Url/Page ID")]
public string Url
{
    get { return _url; }
    set { _url = value; }
}

public EPiServer.Core.PageData PageData
{
    get
    {
        if (IsDirty && Url != null)
        {
            try
            {
                int id;
                if (Int32.TryParse(Url, out id)) {
                    _pageData = EPiServer.Global.EPDataFactory.GetPage(
                                new EPiServer.Core.PageReference(id));
                }
                else {
                    _pageData =
                        EPiServer.Global.EPDataFactory.GetPageByURL(Url);
                }
                _pageDataUrl = Url;
            }
            catch (EPiServer.Core.PageNotFoundException x)
            {
                /* Handle the error! */
            }
        }
        return _pageData;
    }
}
```

As you can see, we use (almost) the same properties as in the previous example for Url and `PageData`. The only difference is in `PageData`, where we need “dirty”-handling of the provided Url. In the previous example it was ok to bind the data to the property control as late as in the `OnPreRender` override, where we could be certain that the data available was the correct data.

In this case we must bind the data earlier since the rendered control will contain server controls that should cause event handlers to execute. In order for these postbacks to work, the controls must be recreated prior to where the postback handling occurs. The thing is that when a user changes the Url from the edit user interface, the change happens after the data is bound. In case this happens we need to rebind before rendering the content.

Below is the `IsDirty` property, the override of `OnLoad`, where the initial data binding occurs, and the `OnPreRender` override, where we make a conditional rebind.

```
private bool IsDirty
{
    get { return _pageDataUrl != Url; }
}
protected override void OnLoad(EventArgs e)
{
    // need to bind data before any event handling takes place
    // ... or it will never take place.
    BindData();
```

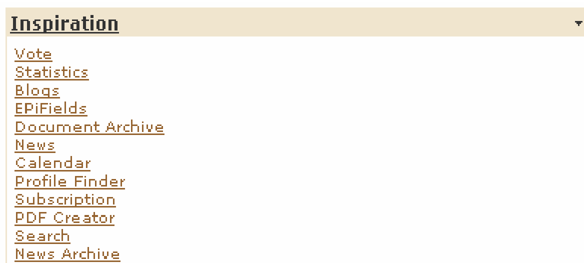
```
    base.OnLoad(e);
}

protected override void OnPreRender(EventArgs e)
{
    // if the url has changed after the control
    // was initially bound we need to rebind.
    if (_boundUrl != Url)
    {
        BindData();
    }

    base.OnPreRender(e);
}

private void BindData()
{
    if (PageData != null)
    {
        PageList.DataSource =
            EPiServer.Global.EPDataFactory.GetChildren(PageData.PageLink);
        PageList.DataBind();
        _boundUrl = Url;
    }
}
```

Finally we have the event handler for the `LinkButton` postback. In this version we just construct a PageData object to retrieve its `LinkUrl` property. We use this URL to redirect the client.

```
protected void PageClick_Command(object sender, CommandEventArgs e)
{
    EPiServer.Core.PageData child =
        EPiServer.Global.EPDataFactory.GetPage(new
            EPiServer.Core.PageReference(_selectedPageId));
    Response.Redirect(child.LinkURL, true);
}
```

The Result

Compile the code and add and register the Web Part as described in the previous example. When added to the page and given a URL using the Edit mode, the Web Part should look something like this:



Example 3 – Implement a Connection between Web Parts

In a sense the previous examples show master/detail view of EPiServer page information. It would be useful if these two Web Parts could interact so that selecting a link in the page list Web Part will update the property Web Part with a new page reference. There is a mechanism called Web Part Connections that can help you accomplish this functionality.

This example shows you how to implement a connection between the Web Parts created in the previous examples. The idea is to let the content Web Part created in the first example be fed with its page context by the page list, so that when a user selects a link in the page list Web Part, the content Web Part is updated to display new content.

Connection Interface

Connections are implemented by defining a provider and a consumer of a common connection interface. The connection interface defines the methods that the consumer can call in order to retrieve data from the connection. In our case the interface is very simple, since its only purpose is to serve a page ID of the page that was selected in the page list.

```
public interface IPageProvider
{
    int GetPageId();
}
```

The implementation of `PageProvider` is very simple and only acts as an information carrier.

```
public class PageProvider : IPageProvider
{
    private int _pageId;

    public PageProvider(int id)
    {
        _pageId = id;
    }

    public int GetPageId()
    {
        return _pageId;
    }
}
```

Connection Provider

In the page list we need to add a connection provider method that looks like this.

```
[ConnectionProvider("Page Connection")]
public IPageProvider GetProvider()
{
    return new PageProvider(_selectedPageId);
}
```

The purpose of the provider method is to serve an instance of an object that implements the connection interface whenever the Web Part framework asks for it. This method must be tagged with the

`ConnectionProviderAttibute` attribute for the framework to know which method to call. The display name argument of the attribute is only used for rendering the user interface.

We also need to add code that sets the `_selectedPageId` that is used to instantiate the `PageProvider` instance and we do that by changing the event handler for the `LinkButton` control to look like this.

```
protected void PageClick_Command(object sender, CommandEventArgs e)
{
    _selectedPageId = Int32.Parse((string)e.CommandArgument);
    if (! IsConnected)
    {
        EPiServer.Core.PageData child =
            EPiServer.Global.EPDataFactory.GetPage(new
                EPiServer.Core.PageReference(_selectedPageId));
        Response.Redirect(child.LinkURL, true);
    }
}
```

We simply store the selected ID in a field for use in the connection provider method. We also need to put a condition on the redirect statement, so that it only occurs if the Web Part not is connected. The `IsConnected` property iterates through all existing connections to see if the current Web Part is part of any connection by checking the ID of the provider.

```
private bool IsConnected
{
    get {
        WebPartManager manager =
            WebPartManager.GetCurrentWebPartManager(this.Page);
        foreach (WebPartConnection connection in manager.Connections)
        {
            if (connection.ProviderID == this.ID) return true;
        }
        return false;
    }
}
```

That is all we need to do on the provider side.

Connection Consumer

In the consumer Web Part we need to add a connection consumer method in order to be fed with data from a connection. The consumer method must be tagged with the `ConnectionConsumerAttribute` attribute to make it visible for the Web Part framework. This is the implementation.

```
[ConnectionConsumer("Page Connection")]
public void SetProvider(IPageProvider provider)
{
    _pageProvider = provider;
}
```

We simply store the passed page provider for later use.

In order to make the control render the correct information, we need to update the `PageData` property to also handle the case, where we receive information from a connection. The changed `PageData` property looks like this.

```
public EPiServer.Core.PageData PageData
{
    get {
        if (IsDirty) {
```

```
            if (IsConnected &&
                _pageProvider.GetPageId() != 0 &&
                (_pageData == null ||
                 _pageData.PageLink.ID != _pageProvider.GetPageId()))
            {
                _pageData =
                    EPiServer.Global.EPDataFactory.GetPage(
                        new EPiServer.Core.PageReference(
                            _pageProvider.GetPageId()));
                _pageDataUrl = _pageProvider.GetPageId().ToString();
                Url = _pageDataUrl;
            }
            else if (Url != null) {
                try {
                    int id;
                    if (Int32.TryParse(Url, out id)) {
                        _pageData =
                            EPiServer.Global.EPDataFactory.GetPage(
                                new EPiServer.Core.PageReference(id));
                    }
                    else {
                        _pageData =
                            EPiServer.Global.EPDataFactory.GetPageByURL(
                                _url);
                    }
                    _pageDataUrl = Url;
                }
                catch (EPiServer.Core.PageNotFoundException x) {
                   /* Handle the error! */
                }
            }
        }
        return _pageData;
    }
}
```

In the above code, we firstly do a dirty check similar to the one we have in the page list Web Part. If the page data is dirty, we need to update it.

First of all we check to see if there is an active connection to get page information from. If that is the case, we retrieve a `PageData` object from EPiServer given the page ID provided by the `PageProvider` implementation. If the Web Part is unconnected, the processing proceeds as with the code that we had before, where we look up the page using `GetPage` or `GetPageByUrl`. The dirty handling requires an update of the `_pageDataUrl` field that is supposed to be synchronized with the currently available page data object.

The `IsDirty` property looks like this.

```
private bool IsDirty
{
    get { return
              _pageData == null ||
              (_pageDataUrl != Url) ||
              (IsConnected &&
                 _pageProvider.GetPageId() != _pageData.PageLink.ID);
    }
}
```

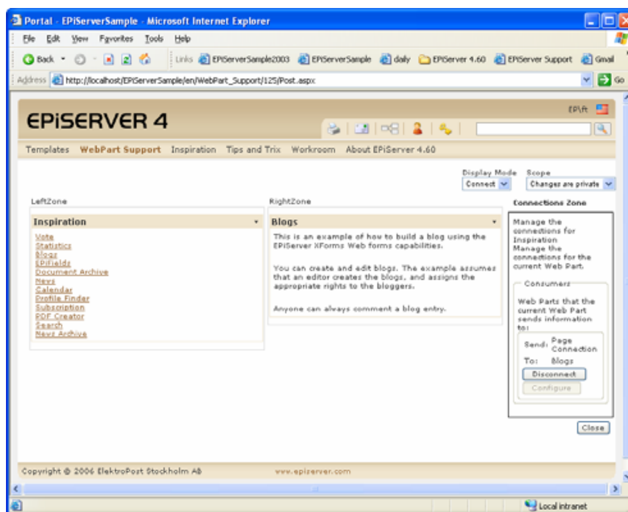
Here we consider the data to be dirty if the data is missing, if the URL for the current data is different than the provided URL, or if there is an active connection and the page ID given by that connection is different from the page ID of the current page data.

That is all we need to change on the consumer side.

The Web Parts in Action

Compile this code and browse the portal page. If you have done the two first examples you will not need to do any new registration.

In the portal page you can now select **Connect** in the display mode drop-down list and then **Connect** from the control menu in the page list Web Part. (A connection can be initiated from the consumer or the provider side.) The connection zone will appear giving you the option to create a connection for the Web Part by selecting a link. After this a select box will appear that lists all Web Parts that can provide data for you or all Web Parts that can consume data that you provide. In this case the list will only contain the property part represented by its title caption. Select the Web Part and then select **Connect** and the connection should be activated. Test it by selecting a link in the page list.



The above image shows how the page will appear after the connection has been initiated and the “Blogs” link has been selected.

Further References

Further reading on Web Parts

- WebParts: From SharePoint to ASP.NET 2.0, Dino Esposito
  http://www.theserverside.net/articles/showarticle.tss?id=WebParts1
- Personalize Your Portal with User Controls and Custom Web Parts
  http://msdn.microsoft.com/msdnmag/issues/05/09/WebParts/default.aspx
- Web Parts Control Set Overview
  http://msdn2.microsoft.com/en-us/library/k3w2y2tf.aspx
- ASP.NET Web Parts Overview
  http://msdn2.microsoft.com/en-us/library/hhy9ewf1.aspx
- ASP.NET Web Parts Pages
  http://msdn2.microsoft.com/en-us/library/e0s9t4ck.aspx

Further reading on WSRP

- WSRP Primer
  http://www.oasis-open.org/committees/download.php/10539/wsrp-primer-1.0.html

Appendix – API Documentation

EPiServer WebPart Controls

This section will move to EPiServer SDK as soon as the documentation system supports .NET 2.0 assemblies.

EPiServer.WebParts.WebControls.ExtendedWebPartManager

`ExtendedWebPartManager` replaces the `WebPartManager` control from the `System.Web.UI.WebControls.WebParts` namespace on pages where the `MaximizedWebPartZone` is to be used.

In addition to members of the `WebPartManager`, this control defines the following public properties.

Property / Method	Description
`MaximizedZoneId`	Gets or sets the ID of the zone that acts as the maximized Web Part zone.
`SoloZoneId`	Gets or sets the ID of the zone that acts as the solo Web Part zone
`MaximizedZone`	Gets the control instance of the zone that acts as the maximized zone.
`SoloZone`	Gets the control instance of the zone that acts as the solo zone.

EPiServer.WebParts.WebControls.ExtendedWebPartZone

`ExtendedWebPartZone` derives from the `WebPartZone` control from the `System.Web.UI.WebControls.WebParts` namespace and adds support for Maximize and Solo by defining two custom verbs. This control also adds support for the EPiServer WebPart property. `ExtendedWebPartZone` can only be placed on pages that has `ExtendedWebPartManager`.

The following public or protected properties are defined or overridden by the ExtendedWebPartZone.

Property	Description
`ExtendedWebPartManager`	Gets `ExtendedWebPartManager` on the current page. This property is protected.
`MaximizeVerb`	Gets a reference to a `WebPartVerb` object that enables end users to maximize Web Part controls in a zone.
`SoloVerb`	Gets a reference to a `WebPartVerb` object that enables end users to display a Web Part control in Solo mode.
`WebPartProperty`	Gets or sets the property name of a `WebPartProperty` to be used for rendering the current zone content. If this property is set, any personalized content will be replaced with the Web Part that is selected for the `WebPartProperty`.

The following public events are defined by `ExtendedWebPartZone.`

Event	Description
`WebPartMaximize`	Occurs when a Web Part is about to be maximized.
`WebPartRestore`	Occurs when a Web Part is about to be restored.

WebPartMinimize	Occurs when a Web Part is about to be minimized.
WebPartEdit	Occurs when a Web Part is about to be edited.
WebPartDelete	Occurs when a Web Part is about to be deleted.
WebPartClose	Occurs when a Web Part is about tot be closed.
WebPartConnect	Occurs when a Web Part is about to be connected.

The following protected methods have been overridden by `ExtendedWebPartZone`.

Method	**Description**
OnCreateVerbs	Overridden to provide custom verbs for Maximize and Solo.
CreateInitialWebParts	This method handles the `WebPartProperty` configuration. If `WebPartProperty` is set, this control attempts to initialize an instance of the configured Web Part.
CloseWebPart ConnectWebPart DeleteWebPart EditWebPart MinimizeWebPart RestoreWebPart	Overridden to expose the corresponding event.

EPiServer.WebParts.WebControls.MaximizedWebPartZone

This is a zone that can act as a Maximize or Solo zone on a Web Part page. A page with support for maximize must contain an `ExtendedWebPartManager`, at least one `ExtendedWebPartZone` and a `MaximizedWebPartZone`. When this configuration is set up, Web Parts placed in the `ExtendedWebPartZone` will have a Maximize verb present in its list of verbs.

The `MaximizedWebPartZone` control extends `System.Web.UI.WebControls.WebParts` with the following public properties:

Property	**Description**
RestoreMaximizedVerb	Gets a reference to a `WebPartVerb` object that enables end users to restore a maximized Web Part control.
RestoreZoneIndex	Gets the order index that the currently maximized Web Part has in its normal (restored) state.
RestoreZone	Gets the zone that the currently maximized Web Part belongs to in its normal (restored) state.
RestoreHeight	Gets the height that the currently maximized Web Part has in its normal (restored) state.
RestoreWidth	Gets the width that the currently maximized Web Part has in its normal (restored) state.

This control defines the following public event::

Event	**Description**
RestoreMaximized	Occurs when a Web Part is about to be restored to its normal state.

The following protected methods are defined or overridden:

Method	**Description**

OnCreateVerbs	Overridden to provide a custom restore verb.
OnRestoreMaximized	Fires the RestoreMaximized event.
OnLoad	Overridden.
MaximizeWebPart	Maximizes a Web Part in the current page.
SetRestoreState	Stores information about a Web Part to be retrieved when the Web Part is restored to its normal state.
RestoreWebPart	Overridden to restore the currently maximized Web Part to its normal state.

EPiServer.WebPart.WebControls.RegisteredWebPartCatalog

This Web Part catalog lists all registered Web Parts in the application. Place this control in a catalog zone to let users add instances of registered Web Parts.

The following public methods are overridden in this control:

Method	Description
GetAvailableWebPartDescriptions	Returns WebPartDescriptionCollection containing information about all registered Web Parts.
GetWebPart	Returns a Web Part instance of specified by a WebPartDescription.

EPiServer.WebParts.WebControls.RssControl

A data-bound control that allows rendering of an RSS feed using templates for the channel element and for the feed entries. In order to use this control you must provide markup templates since the control has no built-in layout.

The following example renders the channel title and the top five item titles:

```
<WebControls:RssControl runat="server" ID="MyRss" ItemCount="5"
EnableViewState="false">
    <ChannelTemplate>
        <h2><%# Container.Title %></h2>
    </ChannelTemplate>
    <ItemTemplate>
        <a href='<%# Container.Link %>'><%# Container.Title %></a><br />
    </ItemTemplate>
</WebControls:RssControl>
```

The following public properties are defined in RssControl:

Properties	Description
Doc	Gets XmlDocument for the currently specified feed.
NamespaceManager	Gets XmlNamespaceManager that is used for querying the XML document for the current feed.
ItemCount	Gets or sets the number of items that should be displayed from the current feed.
Src	Gets or sets the URL to the feed that should be displayed.
ChannelTemplate	Gets or sets System.Web.UI.ITemplate that defines how the channel element of the feed is displayed.
ItemTemplate	Gets or sets System.Web.UI.ITemplate that defines how item

	elements of the feed are displayed.
Title	Gets the title of the feed.

The following methods are defined or overridden in this control:

Methods	Description
CheckSrc	Checks the response returned from a URL and makes sure that it is of type “text/xml”. If not, this method returns false; otherwise true.
OnDataBinding	Overridden to create child controls.
CreateChildControls	Overridden to instantiate ChannelTemplate and ItemTemplate for the bound data source.
OnPreRender	Overridden to bind data prior to the rendering phase.

EPiServer.WebParts.WebControls.DisplayModeDropDown

This is a drop-down control that lets a user switch Web Part display mode on the current page.

The following public properties are defined in this class:

Property	Description
Manager	Gets an object reference to the current WebPartManager instance.

This control overrides the following protected methods:

Method	Description
OnLoad	Overridden to populate available display mode items.
OnSelectedIndexChanged	Overridden to change display mode when selected index changes.

EPiServer.WebParts.WebControls.ScopeDropDown

This control lets a user change personalization scope. The available values are User and Shared personalization scope. In order to change the personalization scope to shared scope the user must have that specific permission. This is can be configured for specific users or roles in web.config under `configuration/system.web/webParts/authorization.`

The following public properties are defined in this class:

Property	Description
SelectedScope	Gets a PersonalizationScope representing the currently selected scope.
WebPartManager	Gets an object reference to the current WebPartManager instance.

The following protected methods are overridden by this class:

Method	Description
OnLoad	Overridden to populate available scope items.
OnSelectedIndexChanged	Overridden to set the personalization scope when the selected index changes.

EPiServer.WebParts.WebControls.ScopeCheckBox

This control lets a user switch between User and Shared personalization scope by selecting and unselecting a check box.

The following public properties are defined in this class:

Property	Description
WebPartManager	Gets an object reference to the current WebPartManager instance.

The following protected methods are overridden by this class:

Method	Description
OnCheckedChanged	Overridden to set the personalization scope when the checked state changes.

EPiServer.WebParts.WebControls.Wsrp.PortletCatalogPart

Provides a catalog that contains portlets from all registered WSRP producers in the application. The catalog displays a drop-down control that lists all registered WSRP producers. All portlets available from the currently selected producer are listed below.

PortletCatalogPart defines the following public properties.

Property	Description
PortletManager	Gets an object reference to the current PortletManager instance.

The following public and protected methods are overridden by this class:

Method	Description
GetAvailableWebPartDescriptions	Returns WebPartDescriptionCollection containing information about all portlets available from the currently selected producer.
GetWebPart	Returns a Web Part instance specified by WebPartDescription.
CreateChildControls	Overridden to provide a custom user interface for this catalog.

EPiServer.WebParts.WebControls.Wsrp.PortletManager

This class serves as the central control for pages exposing WSRP portlets, managing all interactions between WsrpWebParts and the underlying infrastructure.

The PortletManager control defines the following public properties.

Property	Description
ConsumerContextInitialized	Gets a value indicating if the ConsumerContext instance has been initialized.
GlobalParameters	Gets a dictionary that can be used to pass variables that should be available across blocking interaction roundtrips.
HasPageId	Gets a value indicating if the current request has a PageId parameter present on the query string.
PageId	Gets the PageId parameter from the query string.

The following public and protected methods are overridden by this class:

Method	Description
AddPortlet (2 overloads)	Adds a portlet instance defined by a producerId and a portletHandle to the current page. The second overload makes it possible to pass a specific windowId to be used for the portlet instance.

`ChangePortletWindowMode (2 overloads)`	Changes the window mode on a specific portlet defined by `WindowId` or a `PortletWindowSession` object. Available modes are: • `"wsrp:view"` • `"wsrp:edit"` • `"wsrp:help"` • `"wsrp:preview"`
`ChangePortletWindowState (2 overloads)`	Changes the window state on a specific portlet defined by a `windowId` or a `PortletWindowSession` object. Available states are: • `"wsrp:normal"` • `"wsrp:minimized"` • `"wsrp:maximized"` • `"wsrp:solo"`
`ContainsWindowId`	Returns a value indicating if the provided `windowId` is present in the current page.
`CreateClienPage`	Creates a new `ClientPage` instance as a child of a specified parent with a specific name.
`GetClientPage`	Returns a `ClientPage` instance based on the passed page name. If there is no page matching the name a new instance is created using the `CreateClientPage` method.
`GetClientPageName`	Returns a normalized page name of the name passed. Override this method to provide your own normalization.
`GetCurrentPortletManager`	Static method that returns an object reference to the current `PortletManager` object. This method can only be called in a page context.
`GetMarkupType`	Returns the HTML `MarkupType` for the provided `windowId`.
`GetPortlet`	Returns an `IWsrpPortlet` for the provided `windowId`.
`GetPortletDriver`	Returns an `IPortletDriver` for the provided `windowId`.
`GetPortletKey`	Returns an `IPortletKey` for the provided `windowId`.
`GetPortletWindowSession`	Returns an `IPortletWindowSession` for the provided `windowId`.
`GetSupportedModes`	Returns an array containing supported window modes for the provided `windowId`.
`GetSupportedWindowStates`	Returns an array containing supported window states for the provided `windowId`.
`GetWindowId`	Gets the `windowId` for the first matching portlet having the provided `producerId` and `portletHandle`.
`GetWsrpPortlet`	Returns an `IWsrpPortlet` for the provided `windowId`.
`HandleWsrpClick`	This method handles all WSRP link clicks and dispatches to different APIs of the underlying infrastructure based on the query string variables. WSRP form posts are handled by the `PortletDataControl`.
`OnBeforePortletInteraction`	Raises the `BeforePortletInteraction` event.
`OnInit`	Overridden to initialize the consumer context object.

OnLoad	Overridden to initialize Error handler and to handle user interactions.
OnUnload	Overridden to persist any unsaved data.
PerformPortletInteraction (2 overloads)	Requests a portlet interaction on the portlet specified by windowId or a windowSession object. The query and form parameters are passed as interaction parameters.
RemovePortlet	Removes the portlet specified by windowId from the current page.

The PortletManager class defines the following public events:

Event	Description
BeforePortetlInteraction	Occurs before a portlet interaction roundtrip is about to be performed.

EPiServer.WebParts.WebControls.Wsrp.WsrpWebPart

This is a Web Part class that wraps a WSRP portlet by rendering the portlet using PortletDataControl. This control can be instantiated dynamically using PortletCatalogPart or RegisteredWebPartCatalog.

This control defines the following public properties:

Property	Description
CustomVerbs	Gets a WebPartVerbCollection containing the custom verbs available for WSRP portlets.
IsEdit	Gets a value indicating if the portlet is in edit mode.
IsHelp	Gets a value indicating if the portlet is in help mode.
IsMaximized	Gets a value indicating if the portlet is in maximized state.
IsMinimized	Gets a value indicating if the portlet is in minimized state.
IsNormal	Gets a value indicating if the portlet is in normal state.
IsPreview	Gets a value indicating if the portlet is in preview mode.
IsSolo	Gets a value indicating if the portlet is in solo state
IsView	Gets a value indicating if the portlet is in view mode.
Portlet	Gets an IWsrpPortlet instance for the current portlet.
PortletData	Gets the PortletData control used for rendering the portlet content.
PortletManager	Gets an object reference to the current PortletManager object.
PortletMode	Gets the window mode for the current portlet. Supported modes are: • Edit • View • Help • Preview
SupportsEdit	Gets a value indicating if the portlet supports edit mode.
SupportsHelp	Gets a value indicating if the portlet supports help mode.
SupportsMaximized	Gets a value indicating if the portlet supports maximized state.
SupportsMinimized	Gets a value indicating if the portlet supports minimized state.

SupportsNormal	Gets a value indicating if the portlet supports normal state.
SupportsPreview	Gets a value indicating if the portlet supports preview mode.
SupportsSolo	Gets a value indicating if the portlet supports solo state.
SupportsView	Gets a value indicating if the portlet supports view mode.
Title	Overridden to provide the title of the portlet..
Verbs	Overridden to provide additional custom verbs for Edit Portlet and View.
WindowId	Gets the WindowId for the currently portlet.
WindowSession	Gets the WindowSession object for the current portlet.
WindowState	Gets the window state of the current portlet Possible states are: • Normal • Minimized • Maximized • Solo
WsrpEditVerb	Gets a reference to a WebPartVerb object (that enables users to change the portlet to edit mode.
WsrpViewVerb	Gets a reference to a WebPartVerb object that enables users to change portlet to view mode.

The following public or protected methods are defined or overridden by the WsrpWebPart class:

Method	Description
CreateChildControls	Overridden to populate the child control collection with PortletDataControl that is used for rendering the portlet content.
OnDeleting	Overridden to call RemovePortlet on the ClientPage object associated with the current page.
OnLoad	Overridden to attach event handlers for the various standard verbs.
OnPreRender	Overridden to update PortletDataControl with the current windowId and setting the chrome state according to the current window state.
SupportsMode	Returns a true if the portlet supports the provided mode.
SupportsWindowState	Returns a true if the portlet supports the provided window state.

EPiServer.WebParts.Core.EPiServerPersonalizationProvider

This class extends the built-in SqlPersonalizationProvider to handle the template – page model of EPiServer. SqlPersonalizationProvider stores personalization data based on the query string excluding query variables. With this behavior all EPiServer pages using the same .aspx file would share personalization data.

A site is configured to use this provider by adding or modifying the webParts section in web.config. See the “web.config” chapter for configuration details.

This class overrides the following public methods:

Method	Description
SavePersonalizationBlob	Saves personalization data to the underlying data store under an

	altered version of the provided path.
`LoadPersonalizationBlobs`	Loads raw personalization data from the underlying data store based on an altered version of the provided path.
`ResetPersonalizationBlob`	Deletes raw personalization data from the underlying data store based on an altered version of the path.

PropertyWebPart

This is a custom EPiServer property for storing Web Part registration references. In EPiServer Edit mode a PropertyWebPart is rendered as a drop-down list where all registered Web Parts are listed. The selected Web Part is available in the property value when queried. This property can be used to populate `ExtendedWebPartsZone` using its WebPartProperty property.

This example illustrates how to hook up `ExtendedWebPartZone` with a name of a property, in this case "SelectedWebPart".

```
<WebParts:ExtendedWebPartManager runat="server" ID="WebPartManager" />
<WebParts:PortletManager runat="server" ID="PortletManager" />
<WebParts:ExtendedWebPartZone runat="server" ID="theZone"
    WebPartProperty="SelectedWebPart" />
```

Copyright © ElektroPost Stockholm AB. ElektroPost and EPiServer® are registered trademarks of ElektroPost Stockholm AB. Other product and company names mentioned in this document may be the trademarks of their respective owners.

The document may be freely distributed in its entirety, either digitally or in printed format, to all EPiServer users. Changes to the content or partial copying of the content may not be carried out without permission from ElektroPost Stockholm AB:

ElektroPost Stockholm AB
Finlandsgatan 38
SE-164 74 Kista
Sweden

Changes are periodically made to the document and these will be published in new editions of the document. ElektroPost reserves the right to improve or change the products or programs included in this document at any time.