Chapter 5—Extending the Resource Model

Extending the Resource Model

Web sites consist of both pure HTML and resources. Resources are everything not directly embedded into the page’s source, such as images, JavaScript, ActiveX controls, Flash, and Silverlight code. Resources can also be used more widely. You can structure a page’s content by moving parts of it into resources, thus changing its content dynamically based on different resource sources. This technique is primarily used to create localized applications able to display different languages and formats for a global market.

In this chapter, I’ll explain how to extend the resource model in ASP.NET. In particular, I’ll look at:

\* How the resource model works

\* Extending the provider-based resource management

\* Creating custom resource providers

\* Editing resources online at runtime

# Principles of Resource Management

Resource files are usually stored in XML format and located in special folders called either *App\_GlobalResources* for global resources or *App\_LocalResources* for page or control specific ones. *App\_LocalResources* is created on a per folder basis. This means that, within your application, there will normally be several such folders available.

Each resource file has a defined file name format:

PageName.Culture.resx

However, for fallback reasons, a common base resource is required. The first resource file has the default name *Pagename.resx*. Whenever ASP.NET isn’t able to find a particular resource file, it will load the default resource file. It’s a good idea to define all your resources there in order to avoid users seeing strange results.

At compile time, the resource compiler takes all these files and compiles them into a binary format. This improves the speed of reading the content and reduces the file size.

### The Fallback Strategy

If you think in terms of cultures, you’ll see a very straightforward model. A culture is a combination of a language and a country. In several countries around the world, people speak more than one language. You define the country to receive information regarding currency format or calendar, for instance, and you define the language to obtain the correct format of numbers, and names of months and weekdays. Even if several countries only define one such combination, others are much more complex. In some cases, you’ll also find information about the script; for example, certain languages can be written in either Cyrillic or Latin letters. Well-globalized pages will support this situation.

For most applications, it may be unwarranted to distinguish between countries—a simple language definition will suffice. A good example is the usage of German in Germany, Austria, and Switzerland. As long as you don’t deal with currency, it’s almost the same (the differences are subtle). Defining German as the final language will drastically reduce the development effort and still have a great impact on internationalization. In real life scenarios, browsers will usually transmit preferred culture information, which consists of language and country details. If one sends “German, Austria” and your systems only support German globally, the fallback mechanism must find the appropriate resource information.

### Using Global Resources

It’s much the same approach for global resources, except that there is no need to bind resources to a specific page. This opens the possibility of holding resources for any situation, with or without adding support for cultures, languages, and fallback behavior. Such resources are called shared resources. It’s easy to handle any kind of resource file embedded within an assembly by accessing the resource manager. The access is enabled via implicit definition in the page’s markup, using the expression:

<%$ Resources:ResourceKey,Tag %>

Alternatively, you can call the Page class’s GetGlobalResource method.

### Limitations of the Existing Provider

The only drawback is that the internal resource provider has a single method of accessing resources—based on *resx* files. File-based operations are not unusual, but they limit us to just one way of accessing resources at design-time and runtime. Large applications with frequently changing resources might need a different behavior, either with better runtime support or with using a different storage location for the resource information.

# Programming a Custom Resource Provider

The system of .*resx* files provided out-of-the-box has some limitations. While it’s well supported in Visual Studio at design-time and works at runtime without any coding effort, there is almost no way of parameterizing or configuring the behavior.

Imagine that you have different themes and want to manage different resources for each theme. Even if the resource management fits your needs (apart from wishing to store the data in a database instead of in XML files), you can’t use the default resource provider. Developers often discard the whole default resource handling approach and implement their own solution. However, resource management is based on providers and, as shown in the previous chapter, the model is extensible. It is not easy, but it can be done, and this chapter will explain how.

## Extending the Provider Model

Whenever you depart from the default resource model, you’ll need to manage some specific requirements. Leaving the default resource model and creating an independent solution has serious drawbacks. Not only do you lose the ability to assign a control’s property a resource-based value, but you also lose the whole world of design-time support, runtime behavior, and certain optimizations. Therefore, instead of implementing a completely isolated solution, the extension of the provider model is a highly recommended alternative.

To understand the resource model, we’ll look firstly at the classes involved. The classes support three phases:

\* The design-time step

\* The compilation phase

\* The runtime step

Usually we distinguish between runtime and design-time phases only. It might look as though the compilation step has a special meaning.

As you’ll see later in this chapter, this is a quirk of the resource management. The *.resx* files used by default to store resources are compiled into binary resource files. Even if not file-based, the resources must be embedded into each page before being delivered to the browser. The runtime then handles the resources and transforms the pre-compiled data into values read by the controls on the fly.

The design-time stage is mostly for local resources. Visual Studio provides a feature in the Tools menu to create local resources for all the controls on a page. This assigns the *meta:resourcekey* attribute with a key that points to the resource file. This file is a simple key-value list that is able to return the right data. The same approach occurs with any explicit call to the resources, using the <%$ Resource:Key %> expression.

Several .*resx* files co-exist and hold data for each culture supported. Implementing the design-time feature is essential for supplying full support to Visual Studio. At compile time, these expressions are translated into calls to the methods GetLocalResource and GetGlobalResource. At runtime, these methods return values from the compiled resources. The two tiers—the call to resources at the UI level and the access to the data layer—are separate, and the replacement of the data access is quite easy. It’s not unusual to change both the behavior and the data source.

The following example extends the default behavior by reading the current theme and managing as many resource file structures as you have themes. It is still using the .*resx* file model. Additionally, you’ll see a web service based client application which demonstrates how to handle resources at runtime and allow users to change formerly static content.

Insert ASPEXTf0501.tif

Figure 5-1 Bindable properties in the Expression editor in Visual Studio

### Prerequisites

There are good examples in the MSDN documentation of how to implement the various interfaces and abstract base classes to create a usable resource provider. The problem is that it doesn’t give you the whole picture, or explain how the isolated examples fit together and fill in the gaps. Normally, you’ll need the following:

\* A factory class that creates the provider at design-time

\* A factory class that creates the provider at runtime

\* Implementations of both providers

The providers are responsible for using specific reader and writer classes to access the resources. These implementations need:

\* A class to read the global resources

\* A class to write the global resources

\* A class to read the local resources

\* A class to write the local resources

You don’t have to completely implement all classes. For instance, if your application does not write resources because you manage all resources outside the application, you don’t need to implement the writer classes fully.

In Figure 5-2, you can see the base classes and their dependencies:

Insert ASPEXTf0502.tif

Figure 5-2 Class diagram of the base classes and interfaces

### Implementing the Custom Provider

The implementation of a custom resource provider starts with the ResourceProviderFactory class. There are two methods required in order to access both global and local resources. Each method returns the configured provider which is responsible for requesting resources from resource storage. Implementing the factory allows some customization features. In the following examples, I customize by adding support for the current theme. Storing the active theme in the factory allows the factory to pass the information to the provider via a parameter. The provider can then modify the storage using this additional information.

In the example, the CustomResourceProvider is based on IResourceProvider, and our particular interest is in ResourceReader. This is the class that implements the reader. The main method is GetObject, which returns the resource value based on the conditions obtained from the factory and the current conditions, such as the culture, passed via parameters. CustomResourceReader implements the IResourceReader interface to make the custom implementation available. That means the ResourceReader.GetObject method returns an object of type IResourceReader. The object is of the type that contains the actual implementation.

In a similar manner, to get the resource writer classes you can use the GetResourceWriter method to return a type that implements IResourceWriter. The implementation in the example code is called CustomResourceWriter. As mentioned before, the implementation is only required if you plan to support resource writing at runtime.

The whole schema must be implemented a second time to support the design-time environment. Visual Studio’s design-time support creates resource files, and reads and writes resources, if you use the designer view. Visual Studio writes *.resx* files by default. Implementing the design-time support allows your custom resource provider to work inside Visual Studio and still support different storage or schema.

To implement the custom resource provider, begin with the factory class:

Listing 5-1 The entry point is a simple factory class

using System.Web.Compilation;

public sealed class CustomResourceProviderFactory : ResourceProviderFactory

{

private string theme;

public string Theme

{

get { return theme; }

set { theme = value; }

}

public override IResourceProvider CreateGlobalResourceProvider(string classKey)

{

return new CustomResourceProvider(null, classKey, theme);

}

public override IResourceProvider CreateLocalResourceProvider(string virtualPath)

{

return new CustomResourceProvider(virtualPath, null, theme);

}

}

This code shows how simple the factory is. It returns a custom resource provider for local and for global resource access. In the example, the same provider is used for both, and it distinguishes between access methods by supplying either the virtualPath or classKey parameter.

Listing 5.2 The provider with theme-based resx support

public class CustomResourceProvider : IResourceProvider

{

class ResourceTuple

{

public ResourceTuple(IResourceReader reader, IResourceWriter writer)

{

this.Reader = reader;

this.Writer = writer;

this.Culture = ci;

}

public IResourceReader Reader

{

get;

set;

}

public IResourceWriter Writer

{

get;

set;

}

}

private string virtualPath;

private string theme;

private bool isGlobal;

private static Dictionary<string, ⮰

Dictionary<CultureInfo, ResourceTuple>> resourceCache;

public CustomResourceProvider(string virtualPath, string className, ⮰

string theme)

{

if (theme == null) theme = "Default";

if (String.IsNullOrEmpty(className))

{

this.virtualPath = virtualPath;

isGlobal = false;

}

else

{

this.virtualPath = className;

isGlobal = true;

}

this.theme = theme;

GetResources();

}

private CultureInfo GetNativeCulture(CultureInfo culture)

{

if (culture == null)

{

culture = System.Threading.Thread.CurrentThread.CurrentUICulture;

}

// switch to country neutral parent to handle languages only

if (culture.Parent != CultureInfo.InvariantCulture)

{

culture = culture.Parent;

}

return culture;

}

private void GetResources()

{

try

{

if (resourceCache == null)

{

resourceCache = new Dictionary<string, ⮰

Dictionary<CultureInfo, ResourceTuple>>();

}

string regPath;

if (isGlobal)

{

// global

regPath = "";

}

else

{

// local, XmlResourceHelper is explained in next listing

regPath = XmlResourceHelper.GetLocalResxBasePath(virtualPath, ⮰

theme);

}

// no context means that it's compile time and we don't support this

if (HttpContext.Current == null) return;

// read all resources for all cultures for this file

string filter = Path.GetFileName(virtualPath) + "\*.resx";

if (!resourceCache.ContainsKey(virtualPath))

{

resourceCache[virtualPath] = new Dictionary<CultureInfo, ⮰

ResourceTuple>();

}

foreach (string file in Directory.GetFiles( ⮰

Path.GetDirectoryName(regPath), filter))

{

string[] pathParts = Path.GetFileName(file).Split( ⮰

".".ToCharArray());

CultureInfo ci = CultureInfo.InvariantCulture;

if (pathParts.Length == 4)

{

// Example: In Default.aspx.de.resx => [2] is "de"

ci = new CultureInfo(pathParts[2]);

}

// per path and per culture we store a reader and a writer

resourceCache[virtualPath].Add(ci, ⮰

new ResourceTuple(new CustomResourceReader(file), ⮰

new CustomResourceWriter(file)));

}

}

catch (Exception ex)

{

throw new ApplicationException(ex.Message, ex);

}

}

object IResourceProvider.GetObject(string resourceKey, CultureInfo culture)

{

IResourceReader reader = null;

if (culture == null)

{

// if not explicitly defined take over page's UI culture

culture = CultureInfo.CurrentUICulture;

}

do

{

if (resourceCache[virtualPath].ContainsKey(culture))

{

// found it

break;

}

else

{

// try native culture

culture = GetNativeCulture(culture);

}

// end of fallback path

if (culture == CultureInfo.InvariantCulture) break;

} while (reader == null);

reader = resourceCache[virtualPath][culture].Reader;

if (reader != null)

{

object value = ((CustomResourceReader)reader).GetObject(resourceKey);

MissHandler(ref value, resourceKey);

return value;

}

return null;

}

// get object from key and culture

public object GetResxObject(string resourceKey, CultureInfo culture)

{

culture = GetNativeCulture(culture);

string uniqueKey = virtualPath;

ResXResourceReader rr = null;

object value = null;

// get the cached object or create it

if (HttpContext.Current.Cache[uniqueKey] == null)

{

uniqueKey = XmlResourceHelper.GetResxPath(uniqueKey, theme, ⮰

virtualPath == null, culture);

if (String.IsNullOrEmpty(uniqueKey)) return "RESX";

rr = new ResXResourceReader(uniqueKey);

HttpContext.Current.Cache[uniqueKey] = rr;

}

// Use resource reader to retrieve the value

rr = (ResXResourceReader)HttpContext.Current.Cache[uniqueKey];

IDictionaryEnumerator id = rr.GetEnumerator();

while (id.MoveNext())

{

if (String.Compare(id.Key.ToString(), resourceKey, true) == 0)

{

value = id.Value;

break;

}

}

// check whether nothing has been found

MissHandler(ref value, resourceKey);

// return

return value;

}

private void MissHandler(ref object value, string resourceKey)

{

// Create a generic value if resource object is missing

if (value == null)

{

if (!resourceKey.Contains("."))

{

value = String.Format("Missing:{0}", resourceKey);

}

else

{

switch (resourceKey.Substring(resourceKey.LastIndexOf(".")))

{

case ".Visible":

value = true;

break;

case ".Tooltip":

case ".ToolTip":

value = "";

break;

default:

case ".Text":

value = String.Format("Missing:{0}", resourceKey);

break;

}

}

}

if (value == null) throw new ArgumentException("Unexpectadly found ⮰

an unresolvable missing resource: " + resourceKey);

}

IResourceReader IResourceProvider.ResourceReader

{

get

{

if (resourceCache.ContainsKey(virtualPath))

{

return resourceCache[virtualPath] ⮰

[CultureInfo.InvariantCulture].Reader;

}

return null;

}

}

public IResourceWriter ResourceWriter

{

get

{

if (resourceCache.ContainsKey(virtualPath))

{

return resourceCache[virtualPath] ⮰

[CultureInfo.InvariantCulture].Writer;

}

return null;

}

}

}

The provider itself is not so simple. Firstly, we need to define the storage location and storage handling. The solution used in this example is based on .*resx* files. To improve performance, the provider retains a copy of the resources in an internal cache. This is global and therefore can be defined as static:

static Dictionary<string, Dictionary<CultureInfo, ResourceTuple>> resourceCache;

This is a nested Dictionary. The key to the outer dictionary is the resource key, which is either the virtualPath to a local resource or the className of a global one. For each resource, we need to handle multiple resource containers. There is one container per culture.

The inner Dictionary contains these containers and employs the CultureInfo as the index to a particular container. The Dictionary value (the container) holds the appropriate reader and writer objects, which are of type CustomResourceReader and CustomResourceWriter. The ResourceTuple class is an inner helper class which stores the reader and writer objects and allows type-safe access to them.

The provider is instantiated on a per file basis. Each local resource (*aspx* as well as *ascx* files) creates one provider. The same happens for each global resource. In the GetResources method, the provider immediately constructs the cache. The HttpContext.Current == null check ensures that the provider is not active at both design and compile time. The provider searches a designated directory from the GetLocalResxBasePath. This is a helper method specific to the required behavior. It creates the path to the *.resx* file based on both theme and local resource. In the example, the full path conforms to the pattern:

App\_Data/Resx/<Theme>/virtualPath/ResourceFile.<Culture>.ResourceType

The App\_Data folder is defined as the storage location. The *Resx* subfolder ensures that there is no conflict with other data stored there. The *<Theme>* part is a placeholder for the named theme. At a minimum, the default theme named “Default” must exist. The *virtualPath* is the path to the file that uses the resource within your project. *ResourceFile* is the name of the *aspx* or *ascx* file, and these extensions define the *ResourceType*. A full path could look like this:

App\_Data/Resx/MyTheme/storeFront/Basket.en-us.aspx

App\_Data/Resx/Default/storeFront/Basket.en-us.aspx

App\_Data/Resx/MyTheme/storeFront/Basket.de.aspx

App\_Data/Resx/Default/storeFront/Basket.de.aspx

These examples show how to store different resources for one file (Basket.aspx) for different themes and cultures. The fallback resource file—the one without any culture information—is handled with the CultureInfo.InvariantCulture key.

Once the cache is populated and the provider is loaded, the runtime requests the provider for a specific resource object using the GetObject method. This method is passed a key and the culture. Because each resource store has its own provider (based on the virtualPath member), this is sufficient information to retrieve the correct value. The goal of this method is to implement the fallback strategy. A do-while loop ascends the path of cultures to obtain an existing definition. Either it finds the invariant culture, or, as a final fallback, the default definition is used. You will need to create at least a default resource definition to avoid an exception being thrown if no resource is found.

If the stored resources have been located, the GetObject method of the CustomResourceReader type is called to obtain the resource reader for the current culture. Remember that such an object exists per culture. The resource reader does not know anything about cultures; it only returns the value based on the resource key.

During the process of resource creation, translation, and storing, some values may get lost. In this case, the resource reader would return null. The MissHandler method checks the value to assure valid content. Depending on the property retrieved (“.Text”, “.Tooltip” etc.), the method adds the prefix “Missing:”. This leads to a page that is shown without any exceptions, but with several text portions such as “Missing:Label1Text”. Authors can recognize the missing values and the name of the corresponding control, including the property that’s not properly set.

So far there is nothing unusual about this code. The final step is to implement the resource reader.

Listing 5-3 The resource reader used to retrieve a specific resource object

public class CustomResourceReader : IResourceReader

{

private Hashtable resources;

public CustomResourceReader(string fileName)

{

using (ResXResourceReader rr = new ResXResourceReader(fileName))

{

resources = new Hashtable();

IEnumerator enu = rr.GetEnumerator();

enu.Reset();

// Cache it in a Hashtable

while (enu.MoveNext())

{

resources.Add(((DictionaryEntry)enu.Current).Key.ToString(), ⮰

((DictionaryEntry)enu.Current).Value);

}

}

}

IDictionaryEnumerator IResourceReader.GetEnumerator()

{

return resources.GetEnumerator();

}

void IResourceReader.Close() { }

IEnumerator IEnumerable.GetEnumerator()

{

return resources.GetEnumerator();

}

void IDisposable.Dispose()

{

}

internal object GetObject(string resourceKey)

{

return resources[resourceKey];

}

}

The resource reader is easy to implement because it still uses .*resx* files. The ResXResourceReader class from the System.Resource namespace provides almost everything we need. The only addition is another internal cache, to provide rapid read-write access to the underlying resources at runtime. This means that the resource reader must have read and write access to the *.resx* files. In the code, the ResXResourceReader loads each XML file, reads all the content, and copies it into the Hashtable. (A Hashtable is appropriate because the values are of type object and a generic List would not have any advantages.) The class returns an enumerator used by the runtime to find keys. It also implements a GetObject method which retrieves the correct resource value using the Hashtable indexer.

At this point, the resource provider could start working if any required resource is present in the appropriate files. The following sections describe the usage scenarios. Refer to the end of the chapter for more information about additional tasks, such as:

\* Extending the provider by implementing a writer

\* Adding support for the design-time experience

\* Adding support for handling resources at both design-time and runtime

\* Replacing the provider to access a database that stores the resources

However, firstly we’ll put this into operation by starting the configuration step.

### Configure the Resource Provider

Because the provider is instantiated and called from a factory, registering the factory is a necessary step. This is a setting in the *web.config* file. In the <system.web> section, the following tag registers the custom factory:

<globalization resourceProviderFactoryType="CustomResourceProviderFactory"/>

Whether to use just the class name or the fully-qualified assembly name depends on the implementation. If you write the class in a separate Visual Studio project, the fully-qualified name is required.

### Using the Custom Resource Provider

Now that everything is implemented and configured, you can start using the resources at runtime. As you can see in the implementation, classes distinguish between local and global resources. For local resources, the key is the name of either the *.aspx* or *.ascx* file. Internally, the virtualPath parameter retains this data. For global resources, the classKey parameter is used. To refer to these resources in your web pages, use the expression format:

<%$ classKey:Resource %>

## Implementing Design-time Support

Implementing design-time support is required for one feature only. In Visual Studio, you can add a resource’s meta attributes through the following:

\* Open an *aspx* page or *ascx* control in the designer view

\* In the “Tools” menu, click the “Generate Local Resource” item

This will create a meta:resourcekey attribute for each localizable element. The provider is responsible for creating the corresponding entry in the resource file and writing the current values there. However, our current provider supports runtime operations only, and therefore we need an additional provider for design-time support.

Note: If you’re sure that the Visual Studio design-time support option will never be used, you should not bother to implement the design-time support classes. But it won’t harm your code if you do build it.

### Register the Design-time Support

Firstly, you’ll need to register the design-time support. There is no such option in the *web.config* file; instead, an attribute is required. You can register your design-time classes by decorating the factory class like this:

[DesignTimeResourceProviderFactory(⮰

typeof(CustomDesignTimeResourceProviderFactory))]

public sealed class CustomResourceProviderFactory : ResourceProviderFactory

The CustomDesignTimeResourceProviderFactory type defines another factory used at design-time. This factory is very similar to the one shown above, the difference being that the design-time support requires several specific actions:

\* You don’t have a current theme at design-time; only one predefined theme can be supported (the default theme in the example code)

\* You don’t have access to HttpContext to resolve paths

\* You don’t have a current culture, so anything created at design-time is written into the fallback resource file

The factory derives from the DesignTimeResourceProviderFactory base class. This abstract base class is defined as:

public abstract class DesignTimeResourceProviderFactory

{

protected DesignTimeResourceProviderFactory();

public abstract IResourceProvider CreateDesignTimeGlobalResourceProvider( ⮰

IServiceProvider serviceProvider, string classKey);

public abstract IResourceProvider CreateDesignTimeLocalResourceProvider( ⮰

IServiceProvider serviceProvider);

public abstract IDesignTimeResourceWriter CreateDesignTimeLocalResourceWriter( ⮰

IServiceProvider serviceProvider);

}

As you can see, there are three methods and a constructor only. Two methods are similar to the runtime factory. They create and return the appropriate provider. The third method returns a resource writer. This writer explicitly supports the Visual Studio resource creation option available through the Tools menu.

All these methods obtain the serviceProvider parameter. The design-time environment is also highly extensible. Visual Studio doesn’t know anything about the various controls handled within the designer. If it were to know, then every new or third party control would have to be installed and registered in Visual Studio. To avoid such a dependency, the control or the provider itself supplies all features required by Visual Studio to operate well. To determine whether a specific feature is supported, a service provider pattern is used. The IServiceProvider interface, implemented by Visual Studio, delivers a collection of services using its GetService method. The attached module or plug-in requiring these services requests the GetService method for a specific implementation.

Finally, we’ll need access to the currently designed document. Visual Studio acts as a Designer Host, represented by the IDesignerHost interface. To access it, request the service in this manner:

IDesignerHost host;  
host = (IDesignerHost) \_serviceProvider.GetService(typeof(IDesignerHost));

The idea behind this technique is that there is no direct dependency between the provider and the host. If you use another host—not Visual Studio—that is well implemented, your provider should still work. These methods also break the dependency on specific assemblies deployed with Visual Studio. Otherwise, when your code runs on a server, it would require Visual Studio to be installed there—a completely unreasonable prospect.

Once you have the designer host, you can ask for the current root document.

WebFormsRootDesigner rootDesigner;  
rootDesigner = host.GetDesigner(host.RootComponent) as WebFormsRootDesigner;

This type is from System.Web.UI.Design and does not depend on Visual Studio. Using the DocumentUrl property, you have the full name of the file currently open in the designer view without any direct access to Visual Studio.

With all these techniques in mind, it’s time to create the design-time factory:

Listing 5-4 The design-time factory implementation

public sealed class CustomDesignTimeResourceProviderFactory : ⮰

DesignTimeResourceProviderFactory

{

private DesignTimeResourceProvider globalResourceProvider;

private DesignTimeResourceProvider localResourceProvider;

private CustomDesignTimeResourceWriter localResourceWriter;

private string \_rootDocument;

public override IResourceProvider CreateDesignTimeGlobalResourceProvider( ⮰

IServiceProvider serviceProvider, string classKey)

{

// Return an IResourceProvider.

if (globalResourceProvider == null)

{

globalResourceProvider = new DesignTimeResourceProvider(classKey, ⮰

serviceProvider);

}

\_rootDocument = globalResourceProvider.RootDocument;

return globalResourceProvider;

}

public override IResourceProvider CreateDesignTimeLocalResourceProvider( ⮰

IServiceProvider serviceProvider)

{

// Return an IResourceProvider

if (localResourceProvider == null)

{

localResourceProvider = new DesignTimeResourceProvider(serviceProvider);

}

return localResourceProvider;

}

public override IDesignTimeResourceWriter CreateDesignTimeLocalResourceWriter( ⮰

IServiceProvider serviceProvider)

{

if (localResourceWriter == null)

{

// Get the host, usually Visual Studio

IDesignerHost host = (IDesignerHost) ⮰  
 serviceProvider.GetService(typeof(IDesignerHost));

// Get the designer currently represents the design view

WebFormsRootDesigner rootDesigner = host.GetDesigner(host.RootComponent) ⮰

as WebFormsRootDesigner;

// Retrieve the local URL of the file currently opened in the designer

\_rootDocument = rootDesigner.DocumentUrl;

if (\_rootDocument != null)

{

// Create the resource writer for this file

localResourceWriter = ⮰

new CustomDesignTimeResourceWriter(\_rootDocument);

}

}

return localResourceWriter;

}

}

The factory returns both a global and a local design-time resource provider. Because we can’t access global resources in the Visual Studio environment, it’s the same provider class in both cases. Future versions or other applications might behave differently and the class design is ready for this. However, the CreateDesignTimeLocalResourceWriter method is of more interest to us. This method has a serviceProvider parameter containing the current designer host. As shown before, retrieving the URL of the current document is easy with the right services.

If you’d like to extend the behavior, it’s a good idea to investigate the options by looking into the provider objects.

System.ComponentModel.Design.DesignerHost provides the following properties out-of-the-box:

\* TransactionDescription: This returns the transaction invoked from the user’s action, which is usually the “Generate Local Resource” string. That means that you can determine exactly what led to the call.

\* RootComponent: The roots for the current transaction, whether it’s Page or Control or any derived type currently open in the designer surface. You might modify your action after recognizing the type.

\* CurrentCulture: The culture run by the design-time host, which is usually the culture of the operating system. You might decide not to write a default fallback resource but instead write the specified culture. Visual Studio 2008 currently has no option to set the culture, but later versions might support this.

\* IsDesignerViewLocked: Returns either true or false depending on if the design-time surface is actually writeable.

\* IsLoading. Must return false, otherwise the control is not completely loaded.

\* ReferenceManager: Returns the Microsoft.Web.Design.ReferenceManager type. This type allows access to the @Register directives that the reference user controls. Use this to handle user controls directly instead of on a file-by-file basis.

There are several methods to modify a document file by accessing the controls in it. However, these are beyond the scope of this chapter and left for you to investigate further.

Another access method is more valuable to us. Besides the IServiceProvider interface, the parameter object also implements System.Web.UI.Design.IWebFormsDocumentService:

public interface IWebFormsDocumentService

{

string DocumentUrl { get; }

bool IsLoading { get; }

event EventHandler LoadComplete;

object CreateDiscardableUndoUnit();

void DiscardUndoUnit(object discardableUndoUnit);

void EnableUndo(bool enable);

void UpdateSelection();

}

Direct access to the DocumentUrl is the most important feature. In fact, this property does not return the URL but the full local path at design-time. This is exactly what we need. The interface also allows some interaction with undo management.

Caution: The IWebFormsDocumentService interface is marked as obsolete. Presumably Microsoft is going to transform the interface into its designated successor WebFormsRootDesigner. Actually, there is no information available as to when and with what consequences this will happen, but keep an eye on it.

The design-time resource provider has support for both reading the current resource at design-time as well as writing from the designer surface using the appropriate command. It therefore implements two interfaces, IResourceProvider and IDesignTimeResourceWriter. The following code listing shows the whole implementation.

Listing 5-5 The complete implementation of the design-time resource provider

internal sealed class DesignTimeResourceProvider : ⮰

IResourceProvider, IDesignTimeResourceWriter

{

private ResXResourceWriter \_writer = null;

private ResXResourceReader \_reader = null;

private string \_rootDocument;

internal string RootDocument

{

get { return \_rootDocument; }

}

public DesignTimeResourceProvider(IServiceProvider serviceProvider) : ⮰

this(null, serviceProvider)

{

}

public DesignTimeResourceProvider(string classKey, ⮰

IServiceProvider serviceProvider)

{

// Get the forms designer provided by Visual Studio

IWebFormsDocumentService formsDesigner = ⮰

serviceProvider.GetService(typeof(IWebFormsDocumentService)) ⮰

as IWebFormsDocumentService;

if (formsDesigner == null)

{

throw new NullReferenceException("IWebFormsDocumentService is null");

}

\_rootDocument = formsDesigner.DocumentUrl;

}

object IResourceProvider.GetObject(string resourceKey, CultureInfo culture)

{

if (\_reader == null)

{

throw ⮰

new NullReferenceException("IResourceProvider::ResourceReader::NULL");

}

object o = null;

IDictionaryEnumerator ide = \_reader.GetEnumerator();

while (ide.MoveNext())

{

if (ide.Key.ToString().ToLowerInvariant().Equals( ⮰

resourceKey.ToLowerInvariant()))

{

o = ide.Value;

break;

}

}

return o;

}

IResourceReader IResourceProvider.ResourceReader

{

get

{

if (\_reader == null)

{

\_reader = new ResXResourceReader(⮰

XmlResourceHelper.GetInvariantResxPathAtDesignTime(⮰

\_rootDocument));

// prepare same for writing

\_writer = new ResXResourceWriter(⮰

XmlResourceHelper.GetInvariantResxPathAtDesignTime(⮰

\_rootDocument));

}

if (\_reader != null)

{

return \_reader;

}

else

{

throw new Exception("IResourceProvider::ResourceReader::NULL");

}

}

}

string IDesignTimeResourceWriter.CreateResourceKey(string resourceName, ⮰

object obj)

{

((IResourceWriter)this).AddResource(resourceName, obj);

return resourceName;

}

void IResourceWriter.AddResource(string name, byte[] value)

{

\_writer.AddResource(name, value);

}

void IResourceWriter.AddResource(string name, object value)

{

\_writer.AddResource(name, value);

}

void IResourceWriter.AddResource(string name, string value)

{

\_writer.AddResource(name, value);

}

void IResourceWriter.Generate()

{

\_writer.Generate();

}

void IResourceWriter.Close()

{

\_writer.Close();

}

void IDisposable.Dispose()

{

\_writer.Close();

\_writer.Dispose();

}

}

This implementation must support both the writing of resources when the user generates the local resource file as well as the reading of current resources when the file is opened for the first time in the designer surface. The full path to the currently loaded file is retrieved in the constructor by obtaining the IWebFormsDocumentService interface:

serviceProvider.GetService(typeof(IWebFormsDocumentService))

The DocumentUrl property will return what you need. The design-time environment will use the IResourceProvider.ResourceReader property first to retrieve the current values. In the code above, I created the reader as well as the writer because the function’s intention is to retrieve and write the values back. The helper class XmlResourceHelper is a private implementation which collects several methods needed in both the design-time and runtime provider. I’ve called GetInvariantResxPathAtDesignTime to obtain the full path of the *.resx* file. As mentioned above, the provider only supports the Default theme. The method points to the paths shown at the beginning of the section in a hard coded manner. You can find the full implementation of the class in the next code listing:

Listing 5-6 Helper class to simplify developers’ life

internal static class XmlResourceHelper

{

private const string DESIGNTIME\_THEME = "Default";

private static readonly char[] TRIMCHARS = "/".ToCharArray();

// The base path according to the example’s requirements

private static string BasePath

{

get

{

if (HttpContext.Current == null)

{

return "/App\_Data/Resx/";

}

else

{

return HttpContext.Current.Server.MapPath("~/App\_Data/Resx") + ⮰

Path.DirectorySeparatorChar;

}

}

}

// full path at design-time and runtime

public static string GetFullPath(string regularPath, string theme, bool global)

{

if (String.IsNullOrEmpty(theme)) throw new ArgumentNullException("theme");

string path = "";

if (HttpContext.Current == null)

{

// assume compile time

path = GetInvariantResxPathAtDesignTime(regularPath);

}

else

{

try

{

// Trim the path

regularPath = Regex.Replace(regularPath, ⮰

HttpContext.Current.Request. ⮰

ApplicationPath.TrimStart(TRIMCHARS), ⮰

"", ⮰

RegexOptions.IgnoreCase);

if (global)

{

path = String.Format("{0}{1}\_Global", BasePath, theme, ⮰

regularPath);

}

else

{

path = String.Format("{0}{1}{2}", BasePath, theme, regularPath);

}

}

catch (Exception ex)

{

throw new Exception(ex.Message + ex.StackTrace);

}

}

return path;

}

// Get the local path to file for themed page resources

public static string GetLocalResxBasePath(string pageId, string theme)

{

if (String.IsNullOrEmpty(pageId))

{

pageId = "";

}

else

{

pageId = Regex.Replace(pageId, ⮰

HttpRuntime.AppDomainAppVirtualPath.TrimStart(TRIMCHARS), "", ⮰

RegexOptions.IgnoreCase);

}

pageId = pageId.Replace("..", ".").TrimStart(TRIMCHARS).Replace('/', ⮰

Path.DirectorySeparatorChar);

// machine name dependant absolute path

string file = String.Concat(BasePath, String.Format("{0}{2}{1}", theme, ⮰

pageId, Path.DirectorySeparatorChar));

return file;

}

// Scaffold file name to resx file from given information

public static string GetResxPath(string pageId, string theme, bool global)

{

pageId = Regex.Replace(pageId, ⮰

HttpRuntime.AppDomainAppVirtualPath.TrimStart(TRIMCHARS), "", ⮰

RegexOptions.IgnoreCase);

string file = Path.Combine(BasePath, String.Format("{0}{2}{3}{1}.resx", ⮰

theme, pageId, (global) ? "\_Global" : "", ⮰

Path.DirectorySeparatorChar));

if (!File.Exists(file))

{

throw new FileNotFoundException("The resx file does not ⮰

exists [GetInvariantResxPath]", file);

}

return file;

}

public static string GetResxPath(string pageId, string theme, bool global, ⮰

CultureInfo ci)

{

pageId = Regex.Replace(pageId, ⮰

HttpRuntime.AppDomainAppVirtualPath.TrimStart(TRIMCHARS), "", ⮰

RegexOptions.IgnoreCase);

string file = Path.Combine(BasePath, String.Format("{0}{2}{3}{1}.{4}.resx",

theme, ⮰

pageId, ⮰

(global) ? "\_Global" : "", ⮰

Path.DirectorySeparatorChar, ⮰

ci.Name));

if (!File.Exists(file))

{

throw new FileNotFoundException("The resx file does not ⮰

exists [GetInvariantResxPath]", file);

//CreateFileIfNotExists(file);

}

return file;

}

// Scaffold the fallback resource file name locally at design-time

public static string GetInvariantResxPathAtDesignTime(string pagePath)

{

// full path to file, need to get resx to invariant from this

Uri uri = new Uri(pagePath);

string path = uri.LocalPath;

string file = String.Format("{0}{2}{3}{4}{2}{1}.resx",

Path.GetDirectoryName(path),

Path.GetFileName(path),

Path.DirectorySeparatorChar,

BasePath,

DESIGNTIME\_THEME); // at design-time App\_Data/Resx/Default

if (!File.Exists(file))

{

CreateFileIfNotExists(file);

}

return file;

}

// Create file if one not exists

private static void CreateFileIfNotExists(string path)

{

if (!Directory.Exists(Path.GetDirectoryName(path)))

{

Directory.CreateDirectory(Path.GetDirectoryName(path));

}

// Use ResXResourcerWriter to create the default resx format

System.Resources.ResXResourceWriter RwX = ⮰

new System.Resources.ResXResourceWriter(path);

RwX.Generate();

RwX.Close();

}

#endregion

}

The various path operations are a drawback of the file-based solution. However, a database driven project might need additional work to support database access at design-time. For the design-time portion, the GetInvariantResxPathAtDesignTime method is required. Because the Uri is built like any web Uri, it must be resolved and converted into a local file path. A bit of string concatenation will do the trick. If the file is not present, it will be created on the fly; this is the default behavior of Visual Studio.

The creation of resource files is simple. The ResXResourceWriter class provides a Generate method that creates a file with unpleasant comments and schema codes. Calling Close flushes the content back to disk.

Insert ASPEXTf0503.tif

Figure 5-3 Resx files contain a comment (closed) and a full schema description

The GetResxPath method is called at compile time. The compiler needs to resolve the implicit expressions that define resources. To determine what values are required to be placed in the page, use the resource provider again. Since neither the HttpContext nor the design-time host exist at this point, we instead use the HttpRuntime.AppDomainAppVirtualPath property. The virtual path is defined at this stage and enables file paths to be resolved. The helper class is used to change the format slightly to meet our needs.

#### Debugging Design-time Extensions

Design-time support should be part of any extension model. However, the difficulty of debugging these extensions is probably the reason why developers avoid adding design-time support to their applications. At design-time, there is no debugger attached. Therefore, you cannot easily add breakpoints or step into the code to see the internal processes. Using smart exceptions and try/catch blocks might help, but this is not a good debugging experience.

To debug within the Visual Studio design-time environment, you must open another instance of Visual Studio and debug the code there. To do this, open the property pages of the Web project and activate the option *Start external program*.

Insert ASPEXTf0407.tif

Figure 5-4 Start another instance of Visual Studio to debug at design-time

As the external program, choose the Visual Studio executable, *devenv.exe*. When you start debugging the project by hitting F5, another instance of Visual Studio appears. Load the same project in this instance and launch the design-time environment. You can now set breakpoints in the first instance and debug as you would do at runtime. To get to specific code sections just start using the code within the second instance.

## Edit Resources at Runtime

Now that we have flexible access to the resources, it is easy to implement additional options. Imagine a complex site that contains hundreds or thousands of resources. Several authors add content, change content, and add new languages. Doing so in XML files is very inefficient. Consider also the way translators usually work. Good translators might handle the content well, but dealing with XML is not their area of proficiency. Deploying a separate application to support their translation work introduces additional complications. Text portions often require a different translation depending on the context in which they appear. This context is lost when translating fragments of a sentence or single words.

Editing content online would be a better option. However, handling resources the standard way (in files) and compiling them into binary resources precludes the saving of new content into the resources. In this section, I demonstrate how to use an extended resource provider model to access resource files online—anytime.

### How it Works

In the previous sections, we looked at a different storage model. The provider still relays the files, but the file structure and availability is much more flexible. Most importantly, we have total control over it. To have access at runtime, a service must be able to open each .*resx* file, write new or changed content into it, and close the file.

The only disadvantage with the provider reading the data directly from the .*resx* files is: how can we continue to use caching to improve performance?

There are several options. For example, you could recognize the editing mode and suppress caching while editing. Alternatively, you could add a refresh button or link that forces the cache to be destroyed. Both options are easy to implement, but beyond the scope of this chapter.

To access the resources from the client, we must find a way of communicating between the browser and the server. A web service and JavaScript could achieve this. As you only need JavaScript when editing pages—not when merely viewing pages—it’s a good idea to have a special master page containing all the JavaScript functions.

From the perspective of the application, you’ll need to create the following functions:

\* Retrieve supported languages and deliver as a web service

\* Retrieve resources for the current page and language from the web service

\* Write data back to the web service to store in the appropriate *.resx* file

\* Extend the page to support the editing of the elements on a control by control basis

\* Create an editor to allow editing the control’s content on the client

The client portion is written in JavaScript, which has several functions:

\* Create an editor to edit parts of a page

\* Load current resource data from the server

\* Save changed content back to the server

\* Change the currently edited language

In this example, I store the supported languages in another XML file, called *Cultures.xml*.

Listing 5-7 The supported cultures stored as XML

<?xml version="1.0" encoding="utf-8" ?>

<Cultures>

<CultureInfo name="English" id="en-Us" />

<CultureInfo name="Deutsch" id="de-De" />

<CultureInfo name="Italiano" id="it-It" />

</Cultures>

The web service method exposes this content to the client. To cache this information, I store the data during the application start.

Listing 5-8 Retrieve supported cultures on application start

void Application\_Start(object sender, EventArgs e)

{

XDocument xmlDoc = XDocument.Load(Path.Combine(Server.MapPath("~/App\_Data"), ⮰

"Cultures.xml"));

var cultures = from c in xmlDoc.Descendants("CultureInfo") ⮰

select new CultureInfo(c.Attribute("id").Value);

Application["cultures"] = cultures.ToList<CultureInfo>();

}

The service’s method transforms this into a serializable object that is easy to read on the client. Firstly, we need a web service class with the following definition:

Listing 5-9 The web service’s class head

[ScriptService()]

[WebService(Namespace = "http://www.apress.com/ws")]

[GenerateScriptType(typeof(Cultures))]

[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1\_1)]

public class ResourceService : System.Web.Services.WebService

{

CustomResourceProviderFactory rf;

public ResourceService()

{

rf = new CustomResourceProviderFactory();

}

…

}

This definition makes the code available for JavaScript pages and defines a private data type, Cultures, as shown in the next code listing. Additionally, the constructor prepares the factory we created earlier in this chapter to obtain access to the resources. This means that the majority of code is in the existing classes, and leaves us with only a few tasks:

Listing 5-10 The supported cultures exposed via a service

[WebMethod()]

public Cultures GetAllLanguages()

{

List<CultureInfo> cList = (List<CultureInfo>)Application["cultures"];

Cultures c = new Cultures(cList.Count);

for (int i = 0; i < cList.Count; i++)

{

c.cultureName[i] = cList[i].DisplayName;

c.cultureID[i] = cList[i].Name;

}

return c;

}

[Serializable()]

public class Cultures

{

public Cultures() : this(2)

{

}

public Cultures(int len)

{

this.len = len;

cultureName = new string[len];

cultureID = new string[len];

}

public string[] cultureName;

public string[] cultureID;

public int len;

}

The remaining tasks required of the web service are coded in two methods—LoadResource and SaveResource. Both receive four parameters:

\* pageId: The name of the resource file, which names the *aspx* or *ascx* file

\* ctrlId: The Id of the control requesting the resource. This includes the property name, such as *Label1Resource1.Text*.

\* culture: The id of the culture, such as “en-us” or “de-de”.

\* theme: The current theme that the resource is written to or read from. This is the special extension, as explained at the beginning of the chapter. If themed resources are not a requirement, you can safely remove this parameter.

The next code listing illustrates the methods. The SaveResource method has one additional parameter called content that contains the data to be written into the resource file. The LoadResource method returns exactly that value using the same parameters.

Listing 5-11 Web methods to load and save resources directly from the client

[WebMethod()]

public string LoadResource(string pageId, string ctrlId, ⮰

string culture, string theme)

{

pageId = pageId.Replace("~", "");

rf.Theme = theme;

IResourceProvider rp = rf.CreateLocalResourceProvider(pageId);

object value = ((CustomResourceProvider)rp).GetResxObject( ⮰

ctrlId, new CultureInfo(culture));

return (value == null) ? String.Empty : value.ToString();

}

[WebMethod()]

public void SaveResource(string pageId, string ctrlId, ⮰

string content, string culture, string theme)

{

pageId = pageId.Replace("~", "");

rf.Theme = theme;

IResourceProvider rp = rf.CreateLocalResourceProvider(pageId);

CultureInfo ci = new CultureInfo(culture);

((CustomResourceProvider)rp).GetResourceWriter(ci).AddResource(ctrlId, ⮰

content);

}

There are some additional steps required. To display and enable the editor, you’ll need to prepare the page. To avoid including all the JavaScript in the page, another master page is used. With two different master pages, you can decide which one to use, depending on whether the page is displayed in edit mode or not. Changing the master page dynamically requires access to the page’s PreInit event.

Listing 5-12 Changing the master page and theme programmatically

public partial class \_Default : BasePage

protected override void OnPreInit(EventArgs e)

{

base.OnPreInit(e);

// Set Editor (special handling beyound the domain scope)

if (Request.QueryString["e"] != null && Request.QueryString["e"] == "on")

{

Session["Editor"] = true;

}

if (Request.QueryString["e"] != null && Request.QueryString["e"] == "off")

{

Session["Editor"] = null;

}

string domain = Request.Url.Host;

// Set Dynamic Theming and Master Pages

if (Session["Editor"] != null)

{

Master.MasterPageFile = "editor.master";

}

else

{

Master.MasterPageFile = "default.master";

}

if (Session["theme"] != null)

{

Theme = Session["theme"] as string;

}

}

As you can see from the code, a QueryString parameter is used to switch between the master pages with editing capabilities enabled or disabled. Just append ?e=on to the URL, and the editor will appear. Once enabled, a session variable retains the state. To switch the editor off on the fly add ?e=off. This is only a suggestion—you may find smarter ways of achieving this. Now we need to build the master page to support the necessary features.

## Creating an Online Editor

For the online editor to function, the JavaScript on the page must handle all controls and call the web service’s methods with the right parameters. The first step is preparation. Each control on the page must be tested for the presence of the meta:resourcekey attribute. If this exists, then a connected resource is available, and the editor should handle this control. To achieve this, the ReadControls method loops through all the controls. The Init event is the right place in the pipeline for this method call, as the controls are ready to go at that stage. Later, in the PreRender event, the code uses the data collected here to add the JavaScript calls.

To test for the attribute, a regular expression is used. This may seem strange, since we have an object model which we could access directly. However, although the object model is present, the implicit resource expressions have already been replaced by code snippets at the page parser stage. The meta:resourcekey attributes do not exist in the object model. Your only option is to parse the source code of the page directly and to virtually add another parser cycle after the internal one.

Note: The page must exist at runtime with full markup available. Pre-compiled sites with hidden source will not work with this example.

The next code listing shows the methods that parse the page, collect the controls, and add the required JavaScript. Because the scripting is a little tricky, it’s explained afterwards in more depth.

Listing 5-13 The master page with the default setting for regular expression

public partial class EditorMasterPage : System.Web.UI.MasterPage

{

private static Regex rx;

private Dictionary<string, Dictionary<string, string>> pages;

static EditorMasterPage()

{

rx = new Regex(@"\<(?<tagName>\w+(:\w+){0,1}) (?<attr>[^>]\*) ?>", ⮰

RegexOptions.ExplicitCapture | ⮰

RegexOptions.Multiline | ⮰

RegexOptions.IgnoreCase | ⮰

RegexOptions.IgnorePatternWhitespace);

}

The regular expression simply addresses a tag with several attributes and returns the matching element as a collection of named arrays. This array contains all tags on the page captured with the key <tagName> and the attributes block—the remaining part of the element—with the key <attr>. An element like <asp:button runat=”server” meta:resourcekey=”ButtonRes1” /> would be divided into the “asp:button” part and the “runat=”server” meta:resourcekey=”ButtonRes1” part in the first step.

Regular expressions are beyond the scope of this book; please refer to the documentation if you have difficulties reading this expression. For the moment, it’s sufficient to know that the string fragments recognized by the expression are collected in named arrays, whereas the names are defined in <angle brackets>. The attributes are splitted into single ones by using the space as a divider.

Listing 5-14 Analysing the controls regarding the meta:resourcekey attribute

protected override void OnInit(EventArgs e)

{

pages = new Dictionary<string, Dictionary<string, string>>();

ReadControls(this.Page.AppRelativeVirtualPath);

base.OnInit(e);

}

privat void ReadControls(string path) {

if (!pages.ContainsKey(path))

{

StreamReader sr = new StreamReader(Server.MapPath(path));

string s = sr.ReadToEnd();

sr.Close();

pages.Add(path, new Dictionary<string, string>());

MatchCollection mc = rx.Matches(s);

foreach (Match m in mc)

{

Group g = m.Groups["attr"];

if (g != null)

{

string[] attributes = g.Value.Split(" ".ToCharArray(), ⮰

StringSplitOptions.RemoveEmptyEntries);

string key = null, id = null;

foreach (string attr in attributes)

{

if (attr.StartsWith("meta:resourcekey"))

{

string param = attr.Split("=".ToCharArray())[1].Trim();

key = param.Substring(1, param.Length - 2);

}

if (attr.StartsWith("id", StringComparison.InvariantCultureIgnoreCase))

{

string[] fragments = attr.Split("=".ToCharArray());

if (fragments.Length == 2) {

string param = fragments[1].Trim();

id = param.Substring(1, param.Length - 2);

}

}

}

if (!String.IsNullOrEmpty(id) && !(String.IsNullOrEmpty(key)))

{

// need to support server transfer

if (!pages[path].ContainsKey(id))

{

pages[path].Add(id, key);

}

}

}

}

}

}

The collection pages contain data from all the files scanned so far. Remember that a page can contain several user controls, and that these controls must become available at the same time. For each page, we store the control’s id and resource key. This makes it possible to access the control, based on its id, and the resource, based on its key. Once all the data has been collected, the page is processed normally. In PreRender, the information is used to add the JavaScript calls.

Listing 5-15 Add editing capability to the collected controls

protected override void OnPreRender(EventArgs e)

{

foreach (string name in this.ContentPlaceHolders)

{

ContentPlaceHolder cph = (ContentPlaceHolder)FindControl(name);

ControlCollection cc = cph.Controls;

this.NavigateControls(cc);

}

base.OnPreRender(e);

}

This is only the first part—the entry point. Assuming we’re operating on a page that uses a master page—a prerequisite for this solution—we’ll need to handle the ContentPlaceHolders. Each such control is processed as its own container. The NavigateControl method works recursively.

Listing 5-16 Navigating all controls and adding the required script (the next listing)

private void NavigateControls(ControlCollection cc)

{

string path = String.Empty;

for (int i = 0; i < cc.Count; i++)

{

Control ctl = cc[i];

// updatepanel container might have no id, but children

if (ctl != null && (ctl.ID != null || ctl.HasControls()))

{

if (ctl.HasControls())

{

if (ctl is UserControl && ⮰

!(ctl.GetType().Name.Contains("collapseliteral")) && ⮰

!(ctl.GetType().Name.Contains("pageheader")))

{

path = ctl.TemplateControl.AppRelativeVirtualPath;

}

else

{

path = ctl.Page.AppRelativeVirtualPath;

}

ReadControls(path);

this.NavigateControls(ctl.Controls);

if (!pages.ContainsKey(path)) continue;

if (ctl.ID == null) continue;

if (!pages[path].ContainsKey(ctl.ID)) continue;

string ctlKey = pages[path][ctl.ID];

if (ctlKey != null)

{

if (ctl is UserControl)

{

AddEditorControl(ctl, path, ctlKey, i);

}

}

}

else

{

path = ctl.TemplateControl.AppRelativeVirtualPath;

if (!pages.ContainsKey(path)) continue;

if (!pages[path].ContainsKey(ctl.ID)) continue;

string ctlKey = pages[path][ctl.ID];

if (ctlKey != null)

{

switch (ctl.GetType().Name)

{

case "ValidationSummary":

continue;

case "WebPartZone":

// access the webparts, which are not part

// of the control collection

foreach (Control part in ((WebPartZone)ctl).WebParts)

{

path = part.Controls[0].TemplateControl. ⮰

AppRelativeVirtualPath;

AddEditorControl(part, path, ctlKey, i);

}

break;

default:

AddEditorControl(ctl, path, ctlKey, i);

break;

}

cc = ctl.Parent.Controls;

}

}

}

}

}

The final step for each valid control located is to call the AddEditorControl method. The methods add a small image (8x8 pixel) which appears near the control, allowing the user to click there and open the editor. Different images express different states:

\* *validator.gif*: The control is a validator. The control itself might be invisible depending on the page’s validation state. However, the editor control is present.

\* *webpart.gif*: Webparts do not handle resources, but they have additional content which might contain resources.

\* *resx.gif*: Regular control with previously added resources

\* *edit.gif*: Regular control with no attached resources

Additionally, a tooltip is created in order to inform the user about the control beneath the mouse pointer. This is helpful as, on complex pages, it isn’t always possible to align the image exactly.

Listing 5-17 Adding the script code

private void AddEditorControl(Control ctl, string path, string ctlKey, int i)

{

bool HasRes = false;

string[] attributeList = GetPossibleResourceList(ctl);

HasRes = this.GetLocalResource(path, ctlKey, attributeList);

StringBuilder sb = new StringBuilder();

using (StringWriter sw = new StringWriter(sb))

{

HtmlTextWriter ht = new HtmlTextWriter(sw);

ht.AddStyleAttribute(HtmlTextWriterStyle.Display, "inline");

ht.AddStyleAttribute(HtmlTextWriterStyle.Cursor, "hand");

ht.AddStyleAttribute(HtmlTextWriterStyle.Position, "relative");

ht.AddStyleAttribute(HtmlTextWriterStyle.Left, "-8");

ht.AddStyleAttribute(HtmlTextWriterStyle.Top, "-8");

ht.AddStyleAttribute(HtmlTextWriterStyle.ZIndex, "100000");

if (HasRes)

{

if (ctl is IValidator)

{

ht.AddAttribute("src", Request.ApplicationPath + ⮰

"/images/Resxeditor/validator.gif");

}

else if (ctl is IWebPart)

{

ht.AddAttribute("src", Request.ApplicationPath + ⮰

"/images/Resxeditor/webpart.gif");

}

else

{

ht.AddAttribute("src", Request.ApplicationPath + ⮰

"/images/Resxeditor/resx.gif");

}

}

else

{

ht.AddAttribute("src", Request.ApplicationPath + ⮰

"/images/Resxeditor/Edit.png");

}

ht.AddAttribute("width", Unit.Pixel(8).ToString());

ht.AddAttribute("height", Unit.Pixel(8).ToString());

// Add all required data as attributes to let loading them from JScript

bool allowHtml = (ctl is Literal) || (ctl is LiteralControl) || ⮰

(ctl is HtmlGenericControl) || ⮰

(ctl.GetType().Name.Contains("collapseliteral"));

ht.AddAttribute("allowHtml", (allowHtml) ? "true" : "false");

ht.AddAttribute("pageID", path);

ht.AddAttribute("ctrlClientID", ctl.ClientID);

ht.AddAttribute("ctrlID", ctlKey);

ht.AddAttribute("attributes", String.Join(",", attributeList));

ht.AddAttribute("alt", String.Format("Type (What?):{0}\n ⮰

Name (Who?):{1}\n⮰

Content (Why?):{2}\n⮰

Container (Where?):{3}\n⮰

State (How?):{4}",⮰

ctl.GetType().Name, ⮰

ctlKey, ⮰

(allowHtml) ? "HTML" : "Text", ⮰

path, ⮰

(ctl.Visible) ? "Visible" : "Hidden by Default"));

// make the icon clickable

ht.AddAttribute("onclick", ⮰

String.Format("showEditor({0}, this); return false;", (allowHtml) ? 1 : 0));

ht.RenderBeginTag(HtmlTextWriterTag.Img);

ht.RenderEndTag();

LiteralControl lb = new LiteralControl(sb.ToString());

ctl.Parent.Controls.AddAt(i + 1, lb);

}

}

// Check for existence of a local resource to modify icon

private bool GetLocalResource(string path, string ctrlId, string[] reslist)

{

bool hasRes = false;

for (int i = 0; i < reslist.Length; i++)

{

try

{

if (reslist[i] != null)

{

string resId = String.Format("{0}.{1}", ctrlId, reslist[i]);

object oValue = HttpContext.GetLocalResourceObject(path, resId, ⮰

Thread.CurrentThread.CurrentCulture);

if (oValue != null)

{

hasRes = true;

break;

}

}

else

{

break;

}

}

catch { }

}

return hasRes;

}

// Return a list of supported properties by this editor (extend at your will)

private string[] GetPossibleResourceList(Control \_Control)

{

List<string> res = new List<string>();

switch (\_Control.GetType().ToString())

{

case "System.Web.UI.WebControls.Literal":

res.Add("Text");

break;

case "System.Web.UI.WebControls.DropDownList":

res.Add("ToolTip");

break;

case "System.Web.UI.WebControls.Menu":

res.Add("ScrollDownText");

res.Add("ToolTip");

res.Add("ScrollUpText");

res.Add("SkipLinkText");

break;

case "System.Web.UI.WebControls.TextBox":

res.Add("ToolTip");

res.Add("Text");

break;

default:

if (Control is IWebPart)

{

// internally set title

res.Add("Text");

}

else if (\_Control is IValidator)

{

res.Add("Text");

res.Add("ToolTip");

res.Add("ErrorMessage");

} else {

res.Add("Text");

res.Add("ToolTip");

}

break;

}

return res.ToArray();

}

// Resolve place holders in case of master page driven web site

private Control GetAndFindInPlaceHolder(string controlName)

{

MasterPage mp = (MasterPage)this.Page.Master;

PropertyInfo pi = mp.GetType().GetProperty("ContentPlaceHolders", ⮰

System.Reflection.BindingFlags.Instance | ⮰

System.Reflection.BindingFlags.NonPublic);

IList names = (IList) pi.GetValue(mp, null);

foreach (string name in names)

{

ContentPlaceHolder cph = (ContentPlaceHolder)MainContent.FindControl(name);

if (cph != null)

{

Control c = cph.FindControl(controlName);

if (c != null)

{

return c;

}

}

}

return null;

}

The GetPossibleResourceList method contains a hard coded sequence of supported attributes. This solution, therefore, does not support all the possible properties of a control. Extending the method is trivial. It takes much more effort to create the JavaScript-based editor to add the content. As long as it’s text, simple controls will achieve this. However, editing image paths or colors would require more work.

To understand the entire code, a glance at the result might help:

Listing 5-18 A typical JavaScript sequence created with the above code

<img src="/DemoApplication/images/Resxeditor/resx.gif"

width="8px" height="8px"

allowHtml="false"

pageID="~/Default.aspx"

ctrlClientID="ctl00\_ctl00\_MainContent\_MainContent\_LinkButtonTheme1"

ctrlID="LinkButtonTheme1Resource1"

attributes="Text,ToolTip" alt="Tooltip"

onclick="showEditor(0, this); return false;"

style="display:inline;cursor:hand;position:relative;left:-8;top:-8;z-index:100000;" />

Before defining the JavaScript editor itself, here is the working editor:

Figure 5-5 The page in edit mode

After clicking on one of the small icons, which represents a resource, the editor appears.

Insert ASPEXTf0505.tif

Figure 5-6 The editor in action

The editor needs access to the web service method we defined earlier. To achieve this, a ScriptManager control is helpful. It references the web service definition.

Listing 5.19 The supported cultures exposed via a service

<asp:ScriptManager ID="ScriptManager1" runat="server"

EnablePartialRendering="False">

<services>

<asp:ServiceReference path="~/ResxEditor/ResourceService.asmx" />

</services>

</asp:ScriptManager>

More code is required for the editor itself. This is accomplished via a user control on the editor master page. The first block of code is sourced from CodeLifter.com—it creates a draggable popup window. As it’s helpful to view the resources in the context of the page, let the user move the window around and make hidden parts of the surface accessible.

Listing 5-20 The JavaScript editor defined in the edit.ascx control, part 1: the drag window

<%@ Control Language="C#" AutoEventWireup="false" CodeFile="edit.ascx.cs" Inherits="edit" %>

<script language="JavaScript1.2" type="text/javascript">

// Script Source: CodeLifter.com

// Copyright 2003

// Do not remove this header

isIE = document.all;

isNN = !document.all && document.getElementById;

isHot = false;

function ddInit(e) {

topDog = isIE ? "BODY" : "HTML";

hotDog = isIE ? event.srcElement : e.target;

while (hotDog.id != "titleBar" && hotDog.tagName != topDog) {

hotDog = isIE ? hotDog.parentElement : hotDog.parentNode;

if (hotDog == null || hotDog.id == null || hotDog.tagName == null)

return;

}

if (hotDog.id == "titleBar") {

offsetx = isIE ? event.clientX : e.clientX;

offsety = isIE ? event.clientY : e.clientY;

nowX = parseInt(whichDog.style.left);

nowY = parseInt(whichDog.style.top);

ddEnabled = true;

document.onmousemove = dd;

}

}

function dd(e) {

if (!ddEnabled) return;

whichDog.style.left = isIE ? ⮰

nowX + event.clientX - offsetx : nowX + e.clientX - offsetx;

whichDog.style.top = isIE ? ⮰

nowY + event.clientY - offsety : nowY + e.clientY - offsety;

return false;

}

function hideMe() {

whichDog.style.visibility = "hidden";

}

function showMe() {

whichDog.style.visibility = "visible";

}

document.onmousedown = ddInit;

document.onmouseup = Function("ddEnabled=false");

</script>

This code portion is shown for completeness. It is used to move the editor window around. When editing the page, it’s helpful to see the page which might be partly hidden by the overlaid window.

Listing 5-21 The JavaScript editor defined in the edit.ascx control, part 2: the resource access

<script language="javascript" type="text/javascript">

var parentObj;

function showMessage(msg) {

var label = document.getElementById('MessageLabel')

label.innerText = msg;

label.style.display = (msg.length == 0) ? 'none' : 'block';

label.style.color = "#00FF33";

}

function showError(msg) {

var label = document.getElementById('MessageLabel')

label.innerText = msg;

label.style.display = (msg.length == 0) ? 'none' : 'block';

label.style.color = "#FF0000";

}

// These functions call the Web Service method.

function initLanguages() {

showMessage("Calling Server...");

ResxEditor.ResourceService.GetAllLanguages(initLanguagesCallback, ⮰

initLanguagesError);

}

function initLanguagesCallback(result) {

var selLoad = document.getElementById('ResxLanguageSelector');

var selCopy = document.getElementById('ResxCopySelector');

selLoad.options.length = 0;

selCopy.options.length = 0;

for (i = 0; i < result.len; i++) {

selLoad.options[i] = new Option(result.cultureName[i], ⮰

result.cultureID[i]);

selCopy.options[i] = new Option(result.cultureName[i], ⮰

result.cultureID[i]);

}

showMessage("");

document.getElementById('theContentContainer').setAttribute("disabled", ⮰

false);

}

function initLanguagesError(error) {

showError(error.get\_message());

}

function saveMe() {

showMessage("Save the data...");

var pageId = parentObj.getAttribute("pageID");

var ctrlId = parentObj.getAttribute("ctrlID");

var theme = '<% = Theme %>';

var tb = document.getElementById('<% = EditTextBox.ClientID %>').innerText;

var sel = document.getElementById('ResxLanguageSelector');

if (sel.options.length > 0) {

var culture = sel.options[sel.selectedIndex].value;

// Text

ResxEditor.ResourceService.SaveResource(pageId, ctrlId + ".Text", ⮰

tb, culture, theme, ⮰

saveMeCallbackText, ⮰

saveMeError);

}

// close after save is done

if (document.getElementById('saveMeCloseCheckbox').checked == true) {

hideEditor();

}

}

// This is the callback function that processes the Web Service return value

function saveMeCallbackText(result, eventArgs) {

if (parentObj.getAttribute("attributes").search(/ToolTip/) != -1) {

showMessage("Save the data....");

// Tooltip

var pageId = parentObj.getAttribute("pageID");

var ctrlId = parentObj.getAttribute("ctrlID");

var theme = '<% = Theme %>';

var sel = document.getElementById('ResxLanguageSelector');

if (sel.options.length > 0) {

var culture = sel.options[sel.selectedIndex].value;

var tb = ⮰

document.getElementById('<% = TextBoxTooltip.ClientID %>').value;

ResxEditor.ResourceService.SaveResource(pageId, ⮰

ctrlId + ".ToolTip", ⮰

tb, culture, theme, ⮰

saveMeCallbackTooltip, ⮰

saveMeError);

}

} else {

saveMeCallbackTooltip(null, null);

}

}

function saveMeCallbackTooltip(result, eventArgs) {

if (parentObj.getAttribute("attributes").search(/ErrorMessage/) != -1) {

showMessage("Save the data.....");

// Tooltip

var pageId = parentObj.getAttribute("pageID");

var ctrlId = parentObj.getAttribute("ctrlID");

var theme = '<% = Theme %>';

var sel = document.getElementById('ResxLanguageSelector');

if (sel.options.length > 0) {

var culture = sel.options[sel.selectedIndex].value;

var tb = ⮰

document.getElementById('<% = TextBoxErrorMessage.ClientID %>').value;

ResxEditor.ResourceService.SaveResource(pageId, ⮰

ctrlId + ".ErrorMessage",⮰

tb, culture, theme, ⮰

saveMeCallbackVisibility, ⮰

saveMeError);

}

} else {

saveMeCallbackVisibility(null, null);

}

}

function saveMeCallbackVisibility(result, eventArgs) {

// Properties::Visible

var pageId = parentObj.getAttribute("pageID");

var ctrlId = parentObj.getAttribute("ctrlID");

var theme = '<% = Theme %>';

var sel = document.getElementById('ResxLanguageSelector');

if (sel.options.length > 0) {

var culture = sel.options[sel.selectedIndex].value;

var cb = document.getElementById('<% = ChkBoxVisible.ClientID %>');

if (cb.checked == false) {

ResxEditor.ResourceService.SaveResource(pageId, ⮰

ctrlId + ".Visible",⮰

"False", culture, theme, ⮰

saveMeCallbackErrorMessage,⮰

saveMeError);

} else {

ResxEditor.ResourceService.SaveResource(pageId, ⮰

ctrlId + ".Visible", ⮰

"True", culture, theme, ⮰

saveMeCallbackErrorMessage,⮰

saveMeError);

}

}

}

function saveMeCallbackErrorMessage(result, eventArgs) {

showMessage("");

}

function saveMeError(err) {

showError(err.get\_message());

}

function loadMe() {

showMessage("Load Resources from Server...");

var pageId = parentObj.getAttribute("pageID");

var ctrlId = parentObj.getAttribute("ctrlID");

var theme = '<% = Theme %>';

var sel = document.getElementById('ResxLanguageSelector');

if (sel.options.length > 0) {

var culture = sel.options[sel.selectedIndex].value;

document.getElementById('LabelControlId').innerText = ctrlId;

document.getElementById('LabelPageId').innerText = pageId;

// Text

ResxEditor.ResourceService.LoadResource(pageId, ⮰

ctrlId + ".Text", ⮰

culture, theme, ⮰

loadMeCallbackText, ⮰

loadMeError);

}

}

function loadMeCallbackText(result, eventArgs) {

document.getElementById('<% = EditTextBox.ClientID %>').innerText = result;

if (parentObj.getAttribute("attributes").search(/ToolTip/) != -1) {

showMessage("Load Resources from Server....");

// Tooltip

var pageId = parentObj.getAttribute("pageID");

var ctrlId = parentObj.getAttribute("ctrlID");

var theme = '<% = Theme %>';

var sel = document.getElementById('ResxLanguageSelector');

if (sel.options.length > 0) {

var culture = sel.options[sel.selectedIndex].value;

ResxEditor.ResourceService.LoadResource(pageId, ⮰

ctrlId + ".Tooltip", ⮰

culture, theme, ⮰

loadMeCallbackTooltip, ⮰

loadMeError);

document.getElementById('<% = TextBoxTooltip.ClientID %>'). ⮰

disabled = false;

document.getElementById('<% = TextBoxTooltip.ClientID %>'). ⮰

style.backgroundColor = "white";

}

} else {

document.getElementById('<% = TextBoxTooltip.ClientID %>'). ⮰

disabled = true;

document.getElementById('<% = TextBoxTooltip.ClientID %>'). ⮰

style.backgroundColor = "silver";

getErrorMessage();

}

}

function loadMeCallbackTooltip(result, eventArgs) {

document.getElementById('<% = TextBoxTooltip.ClientID %>'). ⮰

innerText = result;

getErrorMessage();

}

function getErrorMessage() {

if (parentObj.getAttribute("attributes").search(/ErrorMessage/) != -1) {

showMessage("Load Resources from Server.....");

// ErrorMessage

var pageId = parentObj.getAttribute("pageID");

var ctrlId = parentObj.getAttribute("ctrlID");

var theme = '<% = Theme %>';

var sel = document.getElementById('ResxLanguageSelector');

if (sel.options.length > 0) {

var culture = sel.options[sel.selectedIndex].value;

ResxEditor.ResourceService.LoadResource(pageId, ⮰

ctrlId + ".ErrorMessage", ⮰

culture, theme, ⮰

loadMeCallbackErrorMessage,⮰

loadMeError);

}

} else {

// Proceed with next

getVisibility();

}

}

function loadMeCallbackErrorMessage(result, eventArgs) {

// Save result

document.getElementById('<% = TextBoxErrorMessage.ClientID %>'). ⮰

innerText = result;

// Proceed with next

getVisibility();

}

function getVisibility() {

var pageId = parentObj.getAttribute("pageID");

var ctrlId = parentObj.getAttribute("ctrlID");

var sel = document.getElementById('ResxLanguageSelector');

var theme = '<% = Theme %>';

if (sel.options.length > 0) {

var culture = sel.options[sel.selectedIndex].value;

ResxEditor.ResourceService.LoadResource(pageId, ⮰

ctrlId + ".Visible", ⮰

culture, theme, ⮰

loadMeCallbackVisible, ⮰

loadMeError);

}

}

function loadMeCallbackVisible(result, eventArgs) {

// Save result

if (result == null || result == true || result == "True") {

document.getElementById('<% = ChkBoxVisible.ClientID %>').checked = ⮰

true;

} else {

document.getElementById('<% = ChkBoxVisible.ClientID %>').checked = ⮰

false;

}

// Done, clear message

showMessage("");

}

function loadMeError(err) {

showError(err.get\_message());

}

function copyMe() {

showMessage("Load Resources from Server...");

var pageId = parentObj.getAttribute("pageID");

var ctrlId = parentObj.getAttribute("ctrlID");

var sel = document.getElementById('ResxCopySelector');

var theme = '<% = Theme %>';

if (sel.options.length > 0) {

var culture = sel.options[sel.selectedIndex].value;

// Text

ResxEditor.ResourceService.LoadResource(pageId, ⮰

ctrlId + ".Text", ⮰

culture, theme, ⮰

copyMeCallbackText, ⮰

loadMeError);

}

}

function copyMeCallbackText(result, eventArgs) {

document.getElementById('<% = EditTextBox.ClientID %>').innerText = result;

showMessage("Load Resources from Server....");

// Tooltip

var pageId = parentObj.getAttribute("pageID");

var ctrlId = parentObj.getAttribute("ctrlID");

var sel = document.getElementById('ResxCopySelector');

var theme = '<% = Theme %>';

if (sel.options.length > 0) {

var culture = sel.options[sel.selectedIndex].value;

ResxEditor.ResourceService.LoadResource(pageId, ⮰

ctrlId + ".Tooltip", ⮰

culture, theme, ⮰

copyMeCallbackTooltip, ⮰

loadMeError);

}

}

function copyMeCallbackTooltip(result, eventArgs) {

showMessage("Load Resources from Server.....");

// Tooltip

var pageId = parentObj.getAttribute("pageID");

var ctrlId = parentObj.getAttribute("ctrlID");

var sel = document.getElementById('ResxCopySelector');

var theme = '<% = Theme %>';

if (sel.options.length > 0) {

var culture = sel.options[sel.selectedIndex].value;

ResxEditor.ResourceService.LoadResource(pageId, ⮰

ctrlId + ".ErrorMessage", ⮰

culture, theme, ⮰

copyMeCallbackErrorMessage, ⮰

loadMeError);

}

}

function copyMeCallbackErrorMessage(result, eventArgs) {

document.getElementById('<% = TextBoxErrorMessage.ClientID %>').innerText =⮰

result;

showMessage("");

}

function showEditor(allowHtml, parent) {

showMessage("");

document.getElementById('<% = EditTextBox.ClientID %>').disabled = true;

document.getElementById('<% = TextBoxTooltip.ClientID %>').disabled = true;

document.getElementById('<% = EditTextBox.ClientID%>'). ⮰

style.backgroundColor= "silver";

document.getElementById('<% = TextBoxTooltip.ClientID %>'). ⮰

style.backgroundColor = "silver";

document.getElementById('ErrorMessage').style.display = "none";

parentObj = parent;

// image with attribute referencing to object data

var etb = document.getElementById('textBoxContainer');

// TODO: support more attributes

var attr = parentObj.getAttribute("attributes");

var a = attr.split(",");

for (i = 0; i < a.length; i++) {

switch (a[i]) {

case "Title":

break;

case "Text":

document.getElementById('<% = EditTextBox.ClientID %>'). ⮰

disabled = false;

document.getElementById('<% = EditTextBox.ClientID %>'). ⮰

style.backgroundColor = "";

etb.style.visibility = 'visible';

etb.style.display = 'inline';

break;

case "ToolTip":

document.getElementById('<% = TextBoxTooltip.ClientID %>'). ⮰

disabled = false;

document.getElementById('<% = TextBoxTooltip.ClientID %>'). ⮰

style.backgroundColor = "white";

break;

case "ErrorMessage":

document.getElementById('ErrorMessage').style.display = "block";

break;

}

}

loadMe();

showMe();

}

function hideEditor() {

var etb = document.getElementById('textBoxContainer');

etb.style.visibility = 'hidden';

etb.style.display = 'none';

hideMe();

}

function initEditor() {

// init

var sel = document.getElementById('ResxLanguageSelector');

if (sel.options.length == 0) {

initLanguages();

}

hideEditor();

}

</script>

This longer JavaScript code block invokes the web services. The pattern is, according to Ajax convention, asynchronous. One method calls the web service, defining callback methods as parameters. Once the web service responds, the callback method is invoked. This could be either a success or error path depending on the outcome. The result is written back into the elements of the editor window. Other methods transfer data from the editor window to the server. Due to the complexity, several methods are required:

\* SaveXX. Each Save method saves a particular control’s content

\* LoadXX. Each Load method loads the content from the server

\* CopyXX. The copy method copies from one language to another

\* XXCallback. The callback containing the data

Some callback methods invoke the subsequent step. For the Load procedure, the LoadMe function is the entry point. In the callback, the next control’s resource is retrieved from the server and read in its own callback method. This creates a chain of callbacks that are dependent on each other. This is similar for the Save procedure. After defining the access methods to the server, the editor window can be created.

Listing 5-22 Part 3 The editor popup window definition

<div id="theLayer" style="background-color: silver; position: absolute; top: 40px; left: 175px; width: 740px; border: solid 2px gray;

padding: 5px; font-family: Verdana; color: Blue; font-size: 10pt; visibility: visible; z-index: 10000">

<table border="0" width="100%" bgcolor="transparent" cellspacing="0" cellpadding="0">

<tr>

<td width="100%">

<table border="0" bgcolor="blue" width="100%" cellspacing="0" cellpadding="3"

height="32">

<tr>

<td id="titleBar" style="cursor: move" width="95%">

<ilayer width="100%" onselectstart="return false">

<layer width="100%" onMouseover="isHot=true;if (isN4) ddN4(theLayer)"

onMouseout="isHot=false">

<span style="font-family:Verdana;font-size:12pt; font-weight:bold;

color:Wheat">Online Resource Editor</span>

</layer>

</ilayer>

</td>

<td style="cursor: hand" valign="top" align="right" width="5%">

<div onclick="hideEditor();return false" style="padding: 2px; color: #ffffff;

background-color: Red; width: 8px; height: 12px; font-size: 9px;

font-family: Verdana; text-decoration: none; text-align: center;

vertical-align: top">

x

</div>

</td>

</tr>

<tr>

<td width="100%" bgcolor="#FFFFFF" style="padding: 4px" colspan="2"

disabled="true" id="theContentContainer">

<table>

<tr>

<td>

Language selection:

</td>

<td>

<select id="ResxLanguageSelector" onchange="loadMe();">

</select>

</td>

</tr>

<tr>

<td>

Copy from language:

</td>

<td>

<select id="ResxCopySelector" onchange="copyMe();">

</select>

</td>

</tr>

</table>

<hr width="100%" size="2" color="gray" />

Resource: <span style="font-weight: bold;"

id="LabelControlId"></span>&nbsp;on&nbsp;

<span style="font-weight: bold;"

id="LabelPageId">

</span>

<br />

<hr width="100%" size="2" color="gray" />

Tooltip:<br />

<asp:TextBox runat="server" ID="TextBoxTooltip"

TextMode="SingleLine" Width="720px"

Visible="true"></asp:TextBox>

<span style="display: none" id="ErrorMessage">

ErrorMessage (Message in group, Text = Message text)

<br />

<asp:TextBox runat="server" ID="TextBoxErrorMessage"

TextMode="SingleLine" Width="720px"

Visible="true"></asp:TextBox>

</span>

<hr width="100%" size="2" color="gray" />

<asp:CheckBox runat="server" ID="ChkBoxVisible"

Text="Make element visible" />

<hr width="100%" size="2" color="gray" />

<div id="textBoxContainer" style="visibility: hidden">

<asp:TextBox runat="server" ID="EditTextBox"

TextMode="MultiLine"

Height="400px" Width="720px"

Visible="true"></asp:TextBox>

</div>

<br />

<table border="0" width="100%">

<tr>

<td align="left" width="80%">

<label>

<input id="saveMeCloseCheckbox"

type="checkbox" value="Close" />

Close after Save

</label>

</td>

<td align="right" width="20%">

<input type="button"

onclick="saveMe();return false;" value="Save" />&nbsp;

<input type="button"

onclick="hideEditor();return false;" value="Close" />

</td>

</tr>

</table>

<br />

<div id="MessageLabel" style="color: Green;

font-weight: bold">

</div>

<br />

<span style="color: Red"><strong>Hint: </strong> ⮰

Refresh page by choosing another language!</span>

</td>

</tr>

</table>

</td>

</tr>

</table>

</div>

<script language="javascript" type="text/javascript">

whichDog = isIE ? document.all.theLayer : document.getElementById("theLayer");

setTimeout('initEditor()', 100);

if (typeof (Sys) !== "undefined") Sys.Application.notifyScriptLoaded();

</script>

This is simply a fragment of HTML containing the controls and the supporting script that manages the page loading cycle. The control has a few trivial lines of code behind (Listing 5.23). As written, everything relies on JavaScript in the client. The web service on the server acts as an access layer to the underlying resource provider.

Listing 5-23 The “code behind” for the control

using System;

using System.Web.UI;

public partial class edit : System.Web.UI.UserControl

{

protected override void OnInit(EventArgs e)

{

Page.EnableEventValidation = false;

base.OnInit(e);

}

protected override void OnLoad(EventArgs e)

{

base.OnLoad(e);

}

protected string Theme

{

get

{

return Session["theme"] as string;

}

}

}

The only public parameter we need is the currently selected theme. This value is included in the web service calls, where it is used to route the resource data into the right resource storage.

# Summary

In this chapter, we looked closely at how to handle resource providers. A complete solution is capable of changing internal behavior without losing any default features, and includes a design-time control. Design-time support is an important part of any custom solution, and it would be shortsighted to ignore the option of using designer tools (even if you do not intend to employ them yourself). All code shown in this chapter has full design-time support, including the steps for enabling debugging.

Furthermore, we made the resource provider accessible via a web service. We built a JavaScript-based editor on the client to read and save resources at runtime, enabling resource-based pages to be editable online. The examples demonstrate how to implement a highly customized solution, which fits well into the existing ASP.NET framework.