Chapter 4—Providers And Configuration

Providers and Configuration

In the previous chapters, you learned how to extend ASP.NET at a low level. For daily tasks, however, you typically work at a high level. A core concept of the ASP.NET extensibility paradigm is the provider model: a software module that provides a uniform interface to a service. ASP.NET incorporates several common providers that allow developers to replace functionality with their own modules, without breaking compatibility with existing modules.

In this chapter, you’ll learn:

\* The provider model concept

\* How providers work internally and how to configure them properly

\* What the built-in providers are and how to extend them

\* How to develop your own custom provider model to allow others to extend your application

\* How to extend the configuration model

\* How to use expression syntax and expression generators to access configuration from markup

At the end of this chapter, you’ll be able to create and extend your own providers. You’ll also know how to integrate them with existing ASP.NET applications so that other developers can extend your application. The extensibility of built-in providers is covered in more detail in the remaining chapters.

# The Provider Model

Providers are software modules build on top of interfaces or abstract classes that define the façade for the application. The interfaces constitute a “seam” in the architecture, which allows you to replace providers without affecting other modules. For instance, the data access provider enables access to any kind of storage for data: including databases from different vendors. Hence, the provider model encapsulates the functionality and isolates it from the rest of the application.

Because almost all the major parts of ASP.NET are built on top of providers, there are multiple ways of modifying the internal behavior. Providers are responsible for the extensibility of ASP.NET . Creating your own providers gives you the ability to construct a sophisticated architecture that others might use—and to alter its behavior without disturbing internal processing. Consider an application such as Microsoft Office SharePoint Server 2007 (MOSS), which is an ASP.NET application and a framework that others use as a foundation for their applications. A similar extensibility concept is supplied on this level. Providers build the core technology that all of this is based on.

## Goals of the Provider Model

When you work with providers for the first time, you may find that writing or extending a provider can be a complicated task. Due to the constraints of compatibility and transparency towards other modules, there is often no other option. You may need to decide whether or not to write your own provider model. This chapter gives you the information you need for making that decision.

Recall what the provider model was designed for:

\* It makes ASP.NET both flexible and extensible

\* It’s robust and well documented

\* It provides a common and modern architecture for your application

\* It’s part of a multi-tier architecture

The provider model does not consist only of simple provider modules. At the top level of the model are services. “Services” in ASP.NET is a generic term for separate modules, such as Membership, Site Maps, or Profiles. These all are high-level components which make your life as a developer easier. Almost all of these services need some kind of data storage or at least a communication channel. From the perspective of a multi-tier application, the service should be independent of the particulars of data persistence.

Service

Provider

Data Store

Figure 4.1 Provider as part of a multi-tier architecture

The Provider sits between these two layers: the Service and the Data Store (Figure 4.1). Modifying the Provider allows the Service to use a different Data Store or communication channel without changing the Service functionality. From the perspective of the user, this architecture is transparent.

Additionally, the Provider is a readily configurable module. You can usually change the Provider by editing the *web.config* file or by setting properties in base classes.

## Default Provider

ASP.NET comes with several built-in providers. As long as they perform their function for you, they need no modification. However, circumstances change, and at some point you might find it necessary to create your own provider. Before commencing, though, it’s illuminating to learn about the existing providers and how they enhance their associated services.

Note: In this book, I do not supply a full description of all the built-in providers. Please refer to the documentation on MSDN to learn more about usage and configuration scenarios.

Before looking into the provider model, let’s differentiate between the two levels, services and providers, as mentioned before. Table 4.1 shows all the services in ASP.NET and the provider associated with each service.

Table 4.1. ASP.NET services using providers

|  |  |
| --- | --- |
| Service | Default Provider |
| Membership | System.Web.Security.SqlMembershipProvider |
| Roles | System.Web.Security.SqlRoleProvider |
| Site Map | System.Web.XmlSitemapProvider |
| Profile | System.Web.Profile.SqlProfileProvider |
| Session State | System.Web.SessionState.InProcSessionStateStore |
| Web Parts | System.Web.UI.WebControls.Webparts.SqlPersonalizationProvider |

You can deduce the storage location from the naming scheme. Membership, Role Management, Profile, and Web Parts services use SQL Server by default, while the Site Map service defaults to a XML file, the *web.sitemap* file. Session variables are held in memory within the IIS process.

Two services do not include preconfigured built-in providers. In fact, these services do have default providers, but you need to configure them explicitly before you can use them. See Table 4.2 for more information:

Table 4.2. ASP.NET services with unconfigured default providers

|  |  |
| --- | --- |
| Service | Recommended Provider (Not assigned by default) |
| Web Events | System.Web.Management.EventLogWebEventProvider |
| Protected Configuration | System.Configuration.RsaProtectedConfigurationProvider |

Web events are events that monitor ASP.NET. They are normally off and must be configured in the <healthMonitoring> section of *web.config*.

By default, configuration data is saved in plain text. The RsaProtectedConfigurationProvider enables encryption of sensitive configuration data—particularly useful if your application is hosted on external servers where unvetted administrators could read the files.

The providers shown in the preceding tables are merely recommendations—there are alternatives. Table 4.3 gives the full list of providers included with ASP.NET:

### Built-In Providers

Some of the built-in ASP.NET services have multiple provider options:

Table 4.3. Built-in providers—the complete list

|  |  |
| --- | --- |
| Service | Available Providers |
| Membership | System.Web.Security.SqlMembershipProvider |
|  | System.Web.Security.ActiveDirectoryMembershipProvider |
| Roles | System.Web.Security.SqlRoleProvider |
|  | System.Web.Security.AuthorizationStoreRoleProvider |
| System.Web.Security.WindowsTokenRoleProvider |  |
| Site Map | System.Web.XmlSitemapProvider |
| Profile | System.Web.Profile.SqlProfileProvider |
| Session State | System.Web.SessionState.InProcSessionStateStore |
|  | System.Web.SessionState.OutOfProcSessionStateStore |
|  | System.Web.SessionState.SqlSessionStateStore |
| Web Parts | System.Web.UI.WebControls.Webparts.SqlPersonalizationProvider |
| Protected Configuration | System.Configuration.RSAProtectedConfigurationProvider |
|  | System.Configuration.DPAPIProtectedConfigurationProvider |

If none of the providers available for a specific service fit your needs, you might consider writing your own provider. There are two ways to accomplish this:

\* Extend an existing provider in order to change its behavior slightly

\* Write your own provider on top of the provider’s base class

As you can imagine, extending an existing provider is the easier option. Whenever you keep the existing data store but wish to change the behavior it’s better to extend an existing provider. If you want to use a different data store, it is preferable to write your own provider. The next sections of this chapter will provide the necessary information.

### Extending Built-In Providers

Extending the built-in providers is an important feature. Several sections of this book describe extensibility methods based on extending providers. Please refer to related parts in:

\* Chapter 5: extending page persistence through session state providers

\* Chapter 6: extending resource management through resource providers

\* Chapter 7: extending the membership and role providers

\* Chapter 7: profile provider extensibility features

\* Chapter 8: extending the sitemap provider

Extensibility for all the providers is supplied by the same base class, as explained in the next section, while the section “Creating a Custom Provider” later in this chapter shows the basic techniques for custom provider implementation. Read these first before proceeding with one of the chapters mentioned above.

# The Anatomy of a Provider

Both custom and built-in providers inherit from the ProviderBase base class. The full definition looks like this:

using System;

using System.Collections.Specialized;

namespace System.Configuration.Provider

{

public abstract class ProviderBase

{

protected ProviderBase();

public virtual string Description { get; }

public virtual string Name { get; }

public virtual void Initialize(string name, NameValueCollection config);

}

}

There is nothing specific to providers here so far. The properties Name and Description are for descriptive purposes only. Name is the internal name used in configuration settings and Description supports graphical tools. If the description is not set, the property returns the name instead. The name is mandatory within the configuration collection. The only method to put the provider in operation is Initialize. The config parameter passes the configuration settings. The name parameter is required because you can name the provider in the configuration file, while the call to Initialize transfers this name for further reference.

As you can see from this code snippet, the base class does not do anything useful. This is what makes it hard to develop your own provider from scratch. If you want to modify the behavior, it’s much easier to develop one of the existing classes and override the properties or methods that don’t fit your needs. However, in this chapter I’ll explain how to write a custom provider from scratch. Modifying built-in providers will be covered in subsequent chapters.

## Making the Provider Available

To make the provider available, you’ll need a service that uses the provider. However, there are no one-to-one relationships between services and their providers. The provider model exists to enable the substitution of one provider with another. To create a custom provider, you’ll need a custom service and one or more custom providers.

The service itself does not have a base class, but it does hold several references to other types discussed in this section:

\* The collection of configured providers

\* The particular provider currently associated with the service

\* The configuration parameters assigned to the current provider

\* The code to instantiate and initialize the provider

Somewhere in your application, there will be a consumer of this service. Consider the service from the perspective of this consumer. The consumer code needs the service in order to obtain data or do something useful. The consumer is not concerned with how the service is configured or how it stores or retrieves data. Even the service itself does not care about this, but handles it using a custom definition of the provider. With both of these abstraction tiers, it’s possible to replace the provider without altering anything within either the service or its consumers.

## Configuring the provider

A provider serves as a layer between an upper level tier and a data tier. It can be simple or complex, but there’s nothing specific to providers except for the Initialize method. The uniqueness of providers is in the special way you can configure and attach them using the *web.config* file. This makes a provider accessible to those who configure and maintain a Web server. (Incidentally, the namespace for the provider’s base class indicates that providers are all about configuration.)

All provider configurations follow the following pattern:

<configuration>

<system.web>

<serviceName>

<providers>

<clear/>

<add name="myProvider" />

</providers>  
 </serviceName>

</system.web>

</configuration>

Notice is that there is not one provider but a collection of them. Usually there will be only one, but the configuration schema allows you to configure several. The configuration classes, as explained later, manage the typical pattern in the xml file based on the instructions <clear/>, <remove>, and <add>. This is necessary because the *web.config* files form a hierarchy from machine level down to the specific subfolder in the application. As long as you have permission, you can override the inherited settings, remove one or all entries, and add your own.

Because you have to handle a collection of providers even if only one is needed, a collection class is required. As for the provider’s base class, there is a base class for collections, too.

namespace System.Configuration.Provider

{

public class ProviderCollection : ICollection, IEnumerable

{

public ProviderCollection();

public bool IsSynchronized { get; }

public object SyncRoot { get; }

public ProviderBase this[string name] { get; }

public virtual void Add(ProviderBase provider);

public void CopyTo(ProviderBase[] array, int index);

public IEnumerator GetEnumerator();

public void Remove(string name);

public void SetReadOnly();

}

}

Using the ProviderBase type, it’s easy to handle the custom provider as well as any of the default built-in ones using this collection. You can also specify one of the providers to be the default provider. This links the service to a provider in the absence of any other selection criteria.

So far, we have a service supplying the consumers, a provider as a late-bound component that hides the details of the data or communication tier, and finally a storage location for a collection of provider definitions. However, the format of the configuration also requires a section definition.

Configuration sections are defined at the beginning of the *web.config* file and extend the configuration by specifying where the actual definitions are stored. Initially, I assumed that provider configurations are stored in the <configuration><system.web> hive. This is the default setting for built-in providers but not compulsory for custom providers. In fact, you can use any path within the <configuration> top-level element. It’s good practice to stay with this model and place custom providers there, too. The configuration sections are defined within the <configSections> element. Beneath this element, you’ll find a hierarchy of <sectionGroup> and <section> elements in no particular order. The hierarchy is the exact representation of the sections used below this part in the *web.config*. However, the default *web.config* has more sections than the <configSection> section does. The reason is that your current *web.config* inherits from the global, machine wide definition. You can find the configuration file that contains all possible section definitions with the following path:

%system%\Microsoft.NET\Framework\v2.0.50727\CONFIG\machine.config

To add your own configuration section to support a specific location for your custom provider definition, you’ll need to add a section definition. Firstly, a custom section definition class has to be implemented. The following base class is the final step in obtaining a complete definition for custom providers:

namespace System.Configuration

{

public abstract class ConfigurationSection : ConfigurationElement

{

protected ConfigurationSection();

public SectionInformation SectionInformation { get; }

protected internal virtual void DeserializeSection(XmlReader reader);

protected internal virtual object GetRuntimeObject();

protected internal override bool IsModified();

protected internal override void ResetModified();

protected internal virtual string SerializeSection(

ConfigurationElement parentElement,

string name,

ConfigurationSaveMode saveMode);

}

}

From this declaration, you can see that ConfigurationSection in turn inherits from the ConfigurationElement class. This is the reason why you can mix <sectionGroup> and <section> elements easily. The class contains several implemented methods and others that you must override to obtain the required behavior. The SerializationSection and DeserializationSection methods convert the settings to and from XML.

## General Considerations

All providers have certain characteristics in common. The base class already explained forces some of the usage scenarios this implies. You should know some of the best practices for creating providers that fit well into the framework.

### Initialization Procedure of a Provider

All providers derive from ProviderBase. Therefore, they inherit the Initialize method, which is declared virtual. In C#, this indicates a method that you need to override. Doing so transfers a few critical tasks over to you.

Each provider needs specific permission to run. This is obtained either by access to a SQL database or local file system. The best way to implement permission checks is to add the required attributes from the System.Security.Permissions namespace. Imagine that a provider is a pluggable module that others may re-use, even if it does not operate well. Instead of eventually failing in custom code, the initialization procedure should immediately report an exception indicating what went wrong. This is precisely the purpose of such permission attributes.

Next, the config parameter passed to the Initialize method is important. If it is null, it means that something went wrong within the configuration. In that case, throw an ArgumentNullException, even if there are no configuration parameters required. The error indicates an unexpected condition and must be reported to the calling method.

The base class’s Initialize method is not abstract. It contains code and must be called to ensure that basic requirements are met. As shown in the last section, the code validates the name and description settings. Even if you intend to change this, and you replace the name and description values in your code, call the base method. If a future version of the .NET framework adds code here to ensure required functionality, then failing to call the base class could break your provider.

Reading the configuration is an important part. However, textual definitions, even if made in XML, can be error-prone. Use the NameValueCollection class and its indexer to access the configuration values. If a required parameter is missing, throw a ProviderException. Make sure you call Remove each time you read a parameter. That clears the config objects one by one. After all required parameters are processed nothing should be left in the config object. If config.Count > 0 is still true throw a ProviderException. This ensures that you have all the required parameters and that the user has no other elements in the configuration that can’t be processed. You might wonder why we don’t just ignore these values. This is because the user’s intention was probably not to add private values but to store additional information there. They have probably mistyped an attribute. The exception helps to recognize these typos instead of searching for the error in event logs.

There are also best practices regarding provider code. Firstly, the service and the provider are two different things. Even if you develop them together in one project, with nothing else in mind, they should stay independent. Secondly, never call a method of the service from the provider. This could lead to infinite, recursive loops, or a break in modularity. Exchanging the provider would be impossible because it is tightly-bound to the service.

### Lifetime

Providers are loaded dynamically the first time the service requires them. This ensures that providers are not held in memory when they are not needed. However, loading a complex provider could be a time consuming task—clearly not desirable when you’re trying to improve the responsiveness of a site. You can force the provider to load by calling certain methods of the services in the Application\_Start event.

This assumes that the provider has a lifetime equalling the lifetime of the application and that the provider has a global state. A provider should not depend on the current session or the context. However, HttpContext and all subsequent classes can be used at any point to retrieve data. This makes it safe to use private fields to store values. It saves memory and speeds up the provider. However, you must remember to write thread-safe code.

### Thread Safety

Threads in ASP.NET are explained in depth in Chapter 2. Web applications are multi-user applications—as with mainframe computer software, you can consider web applications as massively parallel applications. Consequently, when dealing with providers, you must handle threads with care. Providers are instantiated only once and shared across all requests. This speeds them up and causes them to consume less memory, which is necessary for basic tasks. Because threads run in parallel—that’s what threads are for—one provider might be called at the same time by multiple threads. This means that you must always code a provider to be thread-safe. Otherwise you risk throwing exceptions or creating garbage on any request. There are subtle differences that might appear under heavy workload only.

The only exception is the Initialize method. Because it’s only called once, thread safety isn’t an issue. As long as the service initializes in Application\_Start, this is always true. If you cannot be sure of this, the code that calls the method in the service has to be locked, as shown in the following code snippet:

if (\_provider == null)

{

lock (\_lock)

{

if (\_provider == null)

{

This ensures that the provider is not loaded twice. The lock statement blocks other threads. However, another thread might have passed the lock and already be running inside the method. That’s why you must check the existence of the provider again after getting the lock back.

For property access, a similar technique is appropriate. Usually, you write your properties like this:

private Unit \_size;

public Unit Size  
{

get { return \_size; }

set { \_size = value; }

}

Imagine that two threads access this method. There is only one instance of this class in the memory and hence only one storage (memory place) for the value. If one thread writes the value and the other reads the value at the same time, you’ll receive an incorrect value back. The lock statement is again the solution. It’s a shortcut to the System.Threading.Monitor class of the framework the compiler creates for us. The code should look like the following snippet:

private Unit \_size;

private object \_synch

public Unit Size  
{

get { lock(\_synch) { return \_size; } }

set { lock(\_synch) { \_size = value; } }

}

The Monitor method is a very basic lock. It blocks all threads, even if they are being accessed for reading only. This could slow down an application under heavy load.

There are several other ways to optimize the behavior. The ReaderWriterLock class improves behavior by allowing shared reads but preventing overlapping write and read access to the property. This improves performance if the values are mostly read rather than written.

Another method uses the System.Runtime.CompilerServices.MethodImplAttribute. See the following code for a usage scenario:

private Unit \_size;

[MethodImpl(MethodImplOptions.Synchronized)]

public Unit GetSize()  
{

return \_size;

}

[MethodImpl(MethodImplOptions.Synchronized)]

public void SetSize(Unit size)  
{

\_size = size;

}

However, this replaces properties with methods, which isn’t the best coding style. The locking experience isn’t any better. The compiler also uses the Monitor method. As you can see from the code snippet, there is no object for storing the locking state. The compiler selects an object at the best level, like lock(this). This could be an object on either the type or application level, which is a higher level than within the method. Usually this means that the locking phase is longer, and this causes the threads to last longer. Unless locking on type level is required, this attribute is not appropriate for providers.

Which method is best will depend on the specific conditions. Generally, you have to ensure thread-safe access to all instance data, including private fields. However, when you write configuration data in the Initialize method (thread-safe because it’s only called once) and provide read-only access, no action is required to ensure thread safety.

Calls to access all stack-based data and local variables within a property or method should not be locked.

# Creating a Custom Provider-Based Service

Images are an essential part of almost all web applications. Powerful image management makes your life and your users’ lives easier. Imagine that your final storage solution isn’t clearly defined and so you want to create a flexible and extensible image management solution. Other developers in your organization should be able to adopt your code and replace parts of it to suit their needs without knowing the internal details of the image creation and delivery process. In such a situation, you could consider a custom provider-based service. Again, the multi-tier model points to a solution:

1. You need to create a service that is able to retrieve and send image data

2. You need a provider interface that makes data storage access replaceable

3. You need a specific provider to put it into operation

4. You need a configuration definition to handle the configuration in *web.config*

Following the model of the provider as described in the previous sections, you should create these classes:

\* An implementation of a configuration section definition class

\* An abstract base class that describes the provider

\* An implementation of this base class that creates the provider

\* A class that implements the service to do anything useful using the provider

\* A provider collection class that stores multiple providers

\* A provider that serves as a concrete implementation against data storage

Additionally, data storage must be provided. Because it’s common to use SQL Server, I’ll create the image service’s default provider using a SQL Server database.

You can build such a solution from the bottom up, beginning with the configuration, followed by the provider and finally the service. This is fine if you have a design and planning phase in your project and clearly defined requirements. However, for learning purposes, we’ll create a provider from the top down and start with the last step—the service. This clarifies the purpose of the provider and makes the next steps easier to understand.

## Limitations of the Code Samples

Depending on real-life requirements, creating a provider-based solution might require a little more effort. Remember that, for the sake of clarity, all the code snippets here lack error handling, unit testing and logging features. In addition, no code snippets here feature localization. The best practice is to store resources in resource files and to localize if the application is used in multiple countries. See Chapter 6 for more information about resources.

Certain samples in this chapter access databases. If you are writing more sophisticated providers, you may have multiple database operations for one action. When the database supports transactions, use transactions to ensure the atomicity of updates. Transactions ensure a rollback if one of the database operations fails. If the database does not inherently support transactions, you must ensure the atomicity within your own code. Add try/catch blocks and check conditions to the code shown here.

Some sample code regarding providers in this and the next few chapters shows only one part of the provider to explain specific techniques. Any methods or properties that are not implemented have been omitted for the sake of clarity and simplicity. However, for production code you should add a NotImplementedException to redundant methods. This verifies that users of the provider only implement the features you intended. If completeness is not required, don’t forget to explain this in the documentation as well.

## Configuring Providers

The next step is to implement the configuration support. Because many providers can serve a single service, a collection is appropriate. The next code listing shows a simple implementation based on the abstract base class, ProviderCollection.

Listing 4.1 The provider collections represents all configured providers

using System;

using System.Configuration.Provider;

namespace Apress.Extensibility.CustomProvider

{

public class ImageProviderCollection : ProviderCollection

{

public ImageProvider this[string name]

{

get

{

return base[name] as ImageProvider;

}

}

public override void Add(ProviderBase provider)

{

if (provider == null)

throw new ArgumentNullException("provider");

if (!(provider is ImageProvider))

throw new ArgumentException

("Invalid provider type", "provider");

base.Add(provider);

}

}

}

There are only two features. You can add providers of the ImageProvider type and retrieve them using an indexer. The base class handles all the other features. The method and property shown above ensure the integrity of the base class type. This means that you won’t be able to add a provider to the configuration section which does not serve the service.

The last step required in order to operate the service is to define the configuration section.

Listing 4.2 The configuration section and the supporting custom attribute

using System;

using System.Configuration;

namespace Apress.Extensibility.CustomProvider

{

[AttributeUsage(AttributeTargets.Class)]

public class SectionAttribute : Attribute

{

public SectionAttribute(string sectionName)

: base()

{

SectionName = sectionName;

}

public string SectionName

{

get;

set;

}

}

[Section("system.web/imageService")]

public class ImageProviderSection : ConfigurationSection

{

[ConfigurationProperty("providers")]

public ProviderSettingsCollection Providers

{

get { return (ProviderSettingsCollection)base["providers"]; }

}

[StringValidator(MinLength = 1)]

[ConfigurationProperty("defaultProvider", ⮰

DefaultValue = "SqlImageProvider")]

public string DefaultProvider

{

get { return (string)base["defaultProvider"]; }

set { base["defaultProvider"] = value; }

}

}

}

As suggested before, a custom attribute helps to have a single location definition of the configuration path—“system.web/imageService” in this example. The custom SectionAttribute attribute does not involve anything special. It is used to decorate the ImageProviderSection class, which contains two properties, a collection of providers, and the name of the default provider.

This results in configuration code which could look like this:

<system.web>

<imageService defaultProvider="">

<providers>

<clear/>

<add … />  
 </providers  
 </imageservice>  
</system.web>

The ConfigurationProperty attribute declares the elements in the *web.config* file. Although it follows the pattern of all provider-based definitions in the configuration, you’re free to choose different definitions. However, the best practice is to follow this pattern.

This is all you need to know in order to write your own provider configuration sections.

## Creating and Using a Service

The purpose of the sample service is to retrieve an image via the RetrieveImage method. All the other service properties and methods are required to set up the provider. The provider allows other developers using the service to replace the default provider with their own version. For instance, you could choose to use the file system on a file server to store images instead of the database. No change would be required in the application or the service implementation.

Let’s take a look at the class itself.

Listing 4.3 The service itself is able to retrieve an image

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Web.Configuration;

using System.Configuration.Provider;

using System.Drawing;

namespace Apress.Extensibility.CustomProvider

{

public class ImageService

{

private static ImageProvider \_provider = null;

private static ImageProviderCollection \_providers = null;

private static object \_lock = new object();

public ImageProvider Provider

{

get { return \_provider; }

}

public ImageProviderCollection Providers

{

get { return \_providers; }

}

public static Image RetrieveImage(int imageID)

{

LoadProviders();

return \_provider.RetrieveImage(imageID);

}

private static void LoadProviders()

{

if (\_provider == null)

{

lock (\_lock)

{

if (\_provider == null)

{

object[] attributes = ⮰

typeof(ImageProviderSection).GetCustomAttributes(⮰

typeof(SectionAttribute), false);

if (attributes.Length != 1)

throw new ConfigurationException("SectionAttribute not set");

SectionAttribute sa = (SectionAttribute)attributes[0];

ImageProviderSection section = ⮰

(ImageProviderSection) ⮰

WebConfigurationManager.GetSection(sa.SectionName);

\_providers = new ImageProviderCollection();

ProvidersHelper.InstantiateProviders(section.Providers, ⮰

\_providers, ⮰

typeof(ImageProvider));

\_provider = \_providers[section.DefaultProvider];

if (\_provider == null)

throw new ProviderException("Unable to load default ImageProvider");

}

}

}

}

}

}

The RetrieveImage method is static, because we don’t need to handle multiple instances of the service. Although the configuration allows multiple providers, only one can act as the current provider. The other parts of the code interact with this current provider. In this application, the service class is created once and starts working when the LoadProviders method is called.

Each method launches with a call to the LoadProviders method to ensure that a provider is present when retrieving data. Loading the provider is a one-time operation. To avoid claiming the lock if providers are already loaded, the first action is checking the *\_provider* variable. After claiming the lock, this test is made again to ensure the provider is still not loaded. This is required as parallel running requests might invoke the service and start a duplicate procedure.

In the next step, the name of the section is retrieved by reading a private attribute. There are several ways to do this. Using an attribute allows a single location definition. The section configuration, which provides the necessary information, is the only class where the definition is used. Using data from the SectionAttribute attribute, it’s possible to get a reference to the <imageService> section we use to define the current provider. Then the code loads all the registered providers and points *\_provider* to the default provider to make it the current one. ProviderHelper is a static class in the System.Web.Configuration namespace that simplifies the process.

If anything goes wrong, an exception is raised to inform the calling code. If everything is fine, the ImageService class will be ready to serve images.

### Creating the Provider

The provider consists of two classes. One is an abstract base class that extends the ProviderBase class with the required additional methods. The other is the implementation of that class. Again, this provider is just one implementation to bring the whole solution into operation. Other developers could replace this provider with their own creation and extend the behavior of the service at will.

Listing 4.4 The base class for the provider

using System.Configuration.Provider;

using System.Drawing;

namespace Apress.Extensibility.CustomProvider

{

public abstract class ImageProvider : ProviderBase

{

public abstract string ApplicationName { get; set; }

public abstract Image RetrieveImage(int id);

}

}

This class has two characteristics. It must inherit the ProviderBase in order to be recognized as a provider in other parts of the application. It must also define the methods that serve the service. In this case, RetrieveImage is such a method. The service knows that it can call this method to get an image, and the provider completes the process.

Now the implementation is required. Because we want to pull the images from a database, some preparation are needed. Assuming you have local database, *aspnetdb*, with integrated security, add the following table there:

Insert figure ASPEXTf0401.tif

Figure 4.2 The table that stores the images

Internally, we use LINQ to SQL to retrieve the image information from the specified table. The table does not actually contain the images. It contains paths to the image folders and allows further management of the relationship between resources requesting the images. The data storage method leads to the first requirement of the provider: it must store a connection string. It’s a good idea to use the predefined <connectionString> section in *web.config* and only handle the name that references it.

Next, look into the provider code.

Listing 4.5 The implementation of the provider

using System;

using System.Collections.Specialized;

using System.Configuration.Provider;

using System.Data.SqlClient;

using System.Drawing;

using System.Linq;

using System.Security.Permissions;

using System.Web;

using System.Web.Configuration;

namespace Apress.Extensibility.CustomProvider

{

[SqlClientPermission(SecurityAction.Demand, Unrestricted = true)]

public class SqlImageProvider : ImageProvider

{

public override string ApplicationName

{

get;

set;

}

public string ConnectionString

{

get;

set;

}

public override void Initialize(string name, NameValueCollection config)

{

if (config == null)

throw new ArgumentNullException("config");

if (String.IsNullOrEmpty(name))

name = "SqlImageProvider";

if (string.IsNullOrEmpty(config["description"]))

{

config.Remove("description");

config.Add("description",

"SQL image provider");

}

base.Initialize(name, config);

ApplicationName = config["applicationName"];

if (string.IsNullOrEmpty(ApplicationName))

ApplicationName = "/";

config.Remove("applicationName");

string connect = config["connectionStringName"];

if (String.IsNullOrEmpty(connect))

throw new ProviderException

("Empty or missing connectionStringName");

config.Remove("connectionStringName");

if (WebConfigurationManager.ConnectionStrings[connect] == null)

throw new ProviderException("Missing connectionStringName");

ConnectionString = WebConfigurationManager. ⮰

ConnectionStrings[connect].ConnectionString;

if (String.IsNullOrEmpty(ConnectionString))

throw new ProviderException("Empty connection string");

if (config.Count > 0)

{

string attr = config.GetKey(0);

if (!String.IsNullOrEmpty(attr))

throw new ProviderException

("Unrecognized attribute: " + attr);

}

}

public override Image RetrieveImage(int id)

{

ImageDataDataContext ctx = new ⮰

ImageDataDataContext(ConnectionString);

var qr = from i in ctx.aspnet\_Configurations

where i.cfg\_category == "ImageProvider" ⮰

&& i.cfg\_id == id ⮰

&& i.cfg\_type == "image"

select i.cfg\_content;

string data = qr.FirstOrDefault<string>();

Image img = Image.FromFile(HttpContext.Current.Server.MapPath(data));

return img;

}

}

}

The life of a provider begins with the call to its Initialize method. The first step is to verify that config is not null. If so, an exception is thrown. Remember that configuration is essential for the provider model. As other parts of the configuration might reference the provider, it needs a name. Assign the provider a default name if it doesn’t have one. In the example, it’s called “SqlImageProvider”. It’s the same for the description. Even if the description is optional, it should be set properly. Several graphical tools might refer to the description. If the attribute “description” doesn’t exist, the code creates one.

Secondly, the base class’s Initialize method is called. The provider retrieves the applicationName and connectionStringName attributes from the configuration file. If the applicationName doesn’t exists we assume it is the root application “/”. The connection string is mandatory and the lack of it throws an exception. Now, the provider instance should have all the information required to operate. The code checks for the remaining configuration attributes. As explained in the best practice sections, the check— if (config.Count > 0)—ensures that unrecognized attributes are reported to the user in order to avoid typos.

Next, the RetrieveImage method must be implemented. The ImageDataDataContext is a class created with the LINQ to SQL wizard. To create this class, follow these steps:

1. In Visual Studio choose *Add > New Item* in the context menu of the solution

2. In section *Data* of the *Add New Item* dialog choose *LINQ to SQL Classes* item

3. Give the item a common name, such as *ImageData.dbml*

Insert figure ASPEXTf0402.tif

Figure 4.3 Add a LINQ to SQL item to the provider project

Close the dialog and an empty designer surface will appear. Open the Server Explorer (View > Server Explorer, or press Ctrl+Alt+S instead) and add a connection to your database. Either it will already be present or you’ll have to add the database and the appropriate tables. The full script for the table is shown in Listing 4.6.

Listing 4.6 SQL table definition for the provider project

CREATE TABLE [dbo].[aspnet\_Configuration](

[cfg\_id] [int] IDENTITY(1,1) NOT NULL,

[cfg\_key] [varchar](50) NOT NULL,

[cfg\_category] [varchar](50) NOT NULL,

[cfg\_type] [varchar](10) NOT NULL,

[cfg\_content] [varchar](max) NULL,

CONSTRAINT [PK\_aspnet\_Configuration] PRIMARY KEY CLUSTERED

(

[cfg\_id] ASC

)WITH (PAD\_INDEX = OFF, STATISTICS\_NORECOMPUTE = OFF, IGNORE\_DUP\_KEY = OFF, ALLOW\_ROW\_LOCKS = ON, ALLOW\_PAGE\_LOCKS = ON) ON [PRIMARY]

) ON [PRIMARY]

This is just a simple example of a storage method. Real life projects tend to be more sophisticated. Once the table exists, and the Server Explorer shows the server, you can add a connection. Choose the icon *Data Connections* and *Add Connection* from the context menu, then, in the following dialog, choose the server. If you use a local SQL Express Edition, the name might look like ./SQLEXPRESS. Select the database where you have created the table. Test the connection and close the dialog to add the connection.

Insert figure ASPEXTf0403.tif

Figure 4.4 Adding a connection to the current project

You can now drag and drop the table onto the designer surface of the LINQ to SQL class. The result should look like the image already shown in Figure 4.2. Based on the class’s name, the designer will create a context file. If the name is *ImageData,* the data context is called *ImageDataDataContext* (DataContext is the suffix). The context contains a property called aspnet\_Configurations, which represents the table. If you have named the table differently, the property will have that name. Using the context, you can use simple LINQ statements to query the database, as shown in the RetrieveImage method.

In the example, the image is retrieved based on its Id, the category “ImageProvider”, and the type “image”. This is only a suggestion. Based on the retrieved name, the Server.MapPath method is used to obtain the full path to the image. The Image.FromFile creates the image in memory for further processing.

Imagine another provider that simply takes the image and creates a thumbnail of it. In a derived class, you could override the RetrieveImage method, call the base class, get the image, and manipulate it to create a thumbnail. The configuration is already able to change the provider. This shows once more the power of the provider architecture.

### Using the Service

This service can be used by any code in the application. In chapter 3, you saw how to use handlers to manage images. Let’s create a handler which uses our ImageService. Start with a simple *.aspx* page which uses a handler, as shown in the next code listing.

Listing 4.7 A simple page that calls the handler

<%@ Page Language="C#" AutoEventWireup="true" ⮰

CodeFile="Default.aspx.cs" Inherits="\_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" ⮰

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

<head runat="server">

<title></title>

</head>

<body>

<form id="form1" runat="server">

<div>

<asp:Image runat="server" ID="Image2" ImageUrl="~/ImageHandler.ashx?id=4" />

<br />

<asp:Image runat="server" ID="Image1" ImageUrl="~/ImageHandler.ashx?id=9" />

</div>

</form>

</body>

</html>

The *ImageHandler.ashx* file is a generic handler that doesn’t need to be assigned in IIS. The parameter id corresponds to the Id column in the image resource table. This is a simplified scenario. In real-life applications, it would be better to use readable strings to define the resources. However, for learning purposes, these code snippets are as short and easy as possible.

The handler calls the service’s static method to retrieve the image and adds the Image object to the output stream.

Listing 4.8 The image handler that uses the configurable service

<%@ WebHandler Language="C#" Class="ImageHandler" %>

using System;

using System.Web;

using System.Drawing;

using System.Drawing.Imaging;

using Apress.Extensibility.CustomProvider;

public class ImageHandler : IHttpHandler

{

public void ProcessRequest(HttpContext context)

{

int id;

if (context.Request.QueryString["id"] != null)

{

if (Int32.TryParse(context.Request.QueryString["id"], out id))

{

Image img = ImageService.RetrieveImage(id);

context.Response.ContentType = "image/jpg";

img.Save(context.Response.OutputStream, ImageFormat.Jpeg);

}

}

}

public bool IsReusable

{

get

{

return true;

}

}

}

Within the ProcessRequest method, the handler attempts to retrieve the id from the QueryString property. If the id is present and is a number, the service class ImageService retrieves the image using the RetrieveImage method. Finally, the right content type is set and the image is streamed into the OutputStream property using the Save method.

The last step—the handler—is our target usage scenario. This is why we’ve written the configuration, the service and the provider.

Any developer can use the handler to retrieve images. They only need to know how to use one specific parameter (the image id). They don’t need to know how the image is created. This is part of the service. Moreover, they don’t need to know how the image is stored. This is the role of the provider. These three parts form a multi-tier model:

\* Tier one—the user interface—is created by the handler

\* Tier two—the business logic—is contained in the service

\* Tier three—the data storage—is handled by the provider

The provider model is a distinct way of creating a multi-tier architecture within your application. It makes the application flexible and extensible, and it simplifies the writing of unit tests, logging, and supervision by splitting a monolithic block of code into smaller chunks.

## Extending the Configuration

In the previous sections, you learned how to extend the provider model and configure the provider using the *web.config* file. The extensibility of parts of the *web.config* is not limited to configuring providers. Using the base classes within the System.Configuration namespace, you can create custom sections and handle them directly. If the settings defined in <AppSettings> are too limited for your application’s needs, you can extend them.

### How to Scaffold a Configuration Section

The first step is to add a reference to the *System.Configuration.dll* assembly and the System.Configuration namespace. Creating a new project of type class library for the new configuration definition is not required, but recommended. This makes the code reusable and easier to test and deploy. Before you start creating a section like this, it’s worth examing the anatomy of a configuration section.

### Anatomy of a configuration section

The configuration section is based on the implementation of two abstract classes, ConfigurationSection and ConfigurationElement. ConfigurationSection is a successor of ConfigurationElement that makes it easy to create hierarchies of sections that contain elements on each level. The concrete ConfigurationSection is defined at the top of the *web.config* file, as shown in Listing 4.9:

Listing 4.9 Definition of private configuration sections

<configSections>

<sectionGroup name="system.web.extensions" type="…">

<sectionGroup name="scripting" type="…">

<section name="scriptResourceHandler" type="…"

requirePermission="false" allowDefinition="MachineToApplication"/>

<sectionGroup name="webServices" type="…">

<section name="jsonSerialization" type="…" requirePermission="false"

allowDefinition="Everywhere" />

<section name="profileService" type="…" requirePermission="false"

allowDefinition="MachineToApplication" />

<section name="…" requirePermission="false"

allowDefinition="MachineToApplication" />

<section name="roleService" type="…" requirePermission="false"

allowDefinition="MachineToApplication" />

</sectionGroup>

</sectionGroup>

</sectionGroup>

</configSections>

The type attributes are empty for the sake of clarity. They contain the fully qualified assembly names of the type that holds the configuration definition. The top-level element, <sectionGroup>, defines in which group the new element appears:

<sectionGroup name="system.web.extensions">

The section <system.web.extensions> is thus defined as the location for all subsequent groups, elements, or any combinations of groups and elements. You can define exactly what appears there simply by implementing the base classes mentioned above.

### The Class Model

Figure 4.5 shows the complete class model behind the configuration classes. To create a private configuration section, you’ll need to implement these abstract classes.

Insert figure ASPEXTf0404.tif

Figure 4.5 Base classes for a custom configuration

The ConfigurationSection and ConfigurationElement classes are the most important. Many elements will allow collections. Collections define tags such as <add>, <remove>, and <clear> to handle multiple elements in XML. The corresponding definition in this code is based on the ConfigurationElementCollection class.

These classes form the structure of the configuration elements. You can refine their behavior by adding attributes. Attributes not only decorate the properties that define elements, but also add features in order to:

\* Validate scalar data

\* Define custom validation methods

\* Set a default if the value is not set

\* Define a subclass’s type to refine the elements hierarchy

### Attributes to Control Elements’ Behaviors

The following table shows all the attributes available to define the behavior of properties.

Table 4.4 Validator attributes that control the elements’ behaviors

Validator Purpose of this Attribute

IntegerValidator Checks whether the element’s value is of type Int32.

LongValidator Checks whether the element’s value is of type Int64.

StringValidator Checks specific string conditions such as minimum and maximum Length. It’s even possible to define forbidden character (for paths, for instance).

RegexStringValidator Defines a regular expression the element’s value must match.

CallbackValidator Defines a callback method that is responsible for checking the value.

TimespanValidator Checks time conditions, declares usually a valid range.

PositiveTimespanValidator Checks time conditions, usually a positive range (ahead in time).

SubclassTypeValidator Defines the Type of a class this element must derive from.

The following attributes refine the configuration element properties at a Meta level. They can be used together and with the Validator attributes in any combination.

Table 4.5 Attributes to refine the configuration elements’ behavior

Attribute Description

DefaultValue The default value. If defined, this value is used if no other value is assigned.

IsDefaultCollection True, if this element represents a collection.

IsRequired True, if the element is mandatory.

IsKey True, if this is the key element of a collection. Applies only if it is a collection.

Options A combination of the IsKey, IsRequired, and IsDefaultCollection elements as a flagged enum.

### Definition of a Simple Configuration Section

Now that we have all the parts of the puzzle handy, we can start creating a real-life example which defines a customized configuration section. First, let’s take a look at the section definition. The following code defines a section with two new elements.

Listing 4.10 A section with two elements

using System;

using System.Collections;

using System.Text;

using System.Configuration;

using System.Xml;

namespace Apress.Extensibility.Configuration

{

public class PageAppearanceSection : ConfigurationSection

{

[ConfigurationProperty("remoteOnly", DefaultValue = "false", ⮰

IsRequired = false)]

public Boolean RemoteOnly

{

get

{

return (Boolean)this["remoteOnly"];

}

set

{

this["remoteOnly"] = value;

}

}

[ConfigurationProperty("font")]

public FontElement Font

{

get

{

return (FontElement)this["font"];

}

set

{ this["font"] = value; }

}

[ConfigurationProperty("color")]

public ColorElement Color

{

get

{

return (ColorElement)this["color"];

}

set

{ this["color"] = value; }

}

}

This class defines three allowed properties. The property RemoteOnly is of type Boolean and doesn’t need any additional definition, beyond declaring it as a ConfigurationProperty.

If you have a more complex class and wish to distinguish between private properties and those exposed to the configuration manager, this requires an additional attribute. The two named properties set in the attribute’s constructor (DefaultValue and IsRequired) are explained in Table 4.5. For the other elements (Font and Color), we’ll need to create our own structures. This means that an element of type <font> should contain specific attributes, just as for <color>.

The definition of the FontElement class demonstrates this.

Listing 4.11 The FontElement class defines a single element that represents a font

public class FontElement : ConfigurationElement

{

[ConfigurationProperty("name", DefaultValue = "Arial", ⮰

IsRequired = true)]

[StringValidator(InvalidCharacters = "~!@#$%^&\*()[]{}/;'\"|\\", ⮰

MinLength = 1, MaxLength = 60)]

public String Name

{

get

{

return (String)this["name"];

}

set

{

this["name"] = value;

}

}

[ConfigurationProperty("size", DefaultValue = "12", IsRequired = false)]

[IntegerValidator(ExcludeRange = false, MaxValue = 24, MinValue = 6)]

public int Size

{

get

{ return (int)this["size"]; }

set

{ this["size"] = value; }

}

}

For the font’s name, a StringValidator is used. It limits the length to between 1 and 60. The name is not allowed to contain several characters. The font’s size is limited with an IntegerValidator. ExcludeRange defines whether the values provided as MaxValue and MinValue are part of the range or not. Here, the range is between 6 and 24, including the values “6” and “24” respectively.

The access to the underlying configuration element is through an indexer. The code behind this serializes and deserializes the values to create a link between the element’s name and its attribute in code. This means that the values used as keys for the indexer (“size”, “name”) are responsible for retrieving the configuration’s XML. In the string <font size=”12”>, the name of the attribute “size” is defined by this["size"] and made accessible in the code via the Size property. Although it’s highly recommended to use the same names, there is no technical restriction on changing them.

The color element operates similarly:

Listing 4.12 The ColorElement class defines a single element that represents a color value

public class ColorElement : ConfigurationElement

{

[ConfigurationProperty("background", DefaultValue = "FFFFFF", ⮰

IsRequired = true)]

[StringValidator( ⮰

InvalidCharacters="~!@#$%^&\*()[]{}/;'\"|\\GHIJKLMNOPQRSTUVWXYZ", ⮰

MinLength = 6, MaxLength = 6)]

public String Background

{

get

{

return (String)this["background"];

}

set

{

this["background"] = value;

}

}

[ConfigurationProperty("foreground", DefaultValue = "000000", ⮰

IsRequired = true)]

[RegexStringValidator(Pattern="[0-9A-Fa-f]{6}"]

public String Foreground

{

get

{

return (String)this["foreground"];

}

set

{

this["foreground"] = value;

}

}

}

}

The StringValidator is again used to constrain the Background property. The RegexStringValidator has the same effect by using a regular expression. As you can see, the values have the same constraint. However, the regular expression is shorter, clearer, and easier to read. Using regular expressions is a good style and allows better control when searching, restricting, or replacing strings.

To understand what all this creates, see the following valid *web.config* section. Firstly, the custom configuration itself has to be registered.

Listing 4.13 The configuration definition must be registered using this code

<configSections>

<sectionGroup name="pageAppearanceGroup">

<section

name="pageAppearance"

type="Apress.Extensibility.Configuration.PageAppearanceSection"

allowLocation="true"

allowDefinition="Everywhere" />

</sectionGroup>

…

</configSections>

Secondly, the configuration section can be used. The code snippet below is an example.

Listing 4.14 The configuration definition allows the following usage scenario

<pageAppearanceGroup>

<pageAppearance remoteOnly="true">

<font name="TimesNewRoman" size="18"/>

<color background="000000" foreground="FFFFFF"/>

</pageAppearance>

</pageAppearanceGroup>

However, this does not make sense if the values are not used. Defining values is one issue; using them in custom code is another.

Recall the point of this chapter: providers and their configuration should assist other developers to easily replace parts of your application, simply by defining another provider. On the user interface side, ASP.NET uses powerful server controls as much as possible to construct the HTML markup. On the code side, anything that’s configurable in any way should be placed in either the *web.config* file, another configuration file, or a database. Custom configuration is a part of the extensibility model.

### Usage of a Custom Configuration Section

Assuming the configuration section is defined, present, and filled with data, it’s time to access its values at runtime from your own code. The following code shows how to achieve this.

Listing 4.15 Access the configuration data

<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" ⮰

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">

protected void Page\_Load(object sender, EventArgs e)

{

Apress.Extensibility.Configuration.PageAppearanceSection config = ⮰

(Apress.Extensibility.Configuration.PageAppearanceSection) ⮰

System.Configuration.ConfigurationManager.GetSection( ⮰

"pageAppearanceGroup/pageAppearance");

StringBuilder sb = new StringBuilder();

sb.Append("<h2>Settings in the PageAppearance Section:</h2>");

sb.Append(String.Format("RemoteOnly: {0}<br>", config.RemoteOnly));

sb.Append(String.Format("Font name and size: {0} {1}<br>", ⮰

config.Font.Name, config.Font.Size));

sb.Append( ⮰

String.Format("Background and foreground color: {0} {1}<br>", ⮰

config.Color.Background, config.Color.Foreground));

lblConfig.Text = sb.ToString();

}

</script>

<html >

<head id="Head1" runat="server">

<title>Custom Configuration Section Example</title>

</head>

<body>

<form id="form1" runat="server">

<div>

<asp:Label runat="server" ID="lblConfig" ></asp:Label>

</div>

</form>

</body>

</html>

The ConfigurationManager class is the entry point into the configuration. You can address any section here by giving the full path or, at least, a distinct part of the leaf path. As shown in the section about providers, there are several ways to handle these strings. Consider attributes, constant literals, and helper classes.

Once you have the section, and the section is a private type, you can cast as shown in the listing. The properties will represent the current values in the configuration. This will not only handle private data but also give fully typed access to the elements.

The configuration is based on the serialization and deserialization features that the base class provides. This could make it difficult to read or to modify the values in the *web.config* file. If you understand how the serializer works and can create similar strings by hand, your configuration class should function correctly as is.

## Accessing the Configuration Declaratively

We now have an extended configuration model allowing the storage of complex values in a clearly defined way. The programmatic access is easy. However, ASP.NET favors declarative techniques. Making the configuration accessible from markup would be ideal.

### Extending the Expression Binding Syntax

Integrated expressions play an important role in accessing data dynamically. You probably work with data binding expressions following the <%# %> pattern. A different declarative syntax is available, with this pattern:

<%$ %>

This is also an extensible model, allowing the creation of your own syntax in order to access data within the markup:

<asp:literal text="<%$ MyConfig:GetData %>" runat="server" />

Read this as an extension to the configuration extension described in the section “Extending the Configuration”. It transfers access to the markup. Developers can access data without writing explicit code. Now page designers working with *.aspx* pages can add specific formatting options without doing any coding tasks.

### Introduction to Expression Syntax

Expression builders process expressions. These build code from expressions during the page processing phase—the parsing procedure. Expressions have this distinct pattern:

<%$ [prefix]:[declaration] %>

The first part, the prefix, maps to a type that handles the expression. Any string will do for the prefix. The colon is mandatory, but you can enter whatever you like in the declaration part. Anything from the colon to the end of the expression is treated as a string and processed at once. One common usage is the accessing of the AppSettings section in *web.config*. It follows this pattern:

<%$ AppSettings:KeyName %>

The prefix is “AppSetting”. It determines the expression builder via a mapping (explained in the “Declare the Prefix” section below) to a built-in class called AppSettingsExpressionBuilder. The expression builder must be able to handle the remaining part, “KeyName” in the example.

For localized resources there is another built-in class:

<%$ Resources:ResourceCategory,Name %>

The prefix is now “Resources”. The expression builder used behind the scenes is the

ResourceExpressionBuilder. There is also another, ConnectionStringsExpressionBuilder, which makes connection strings available in markup. All expression builders derive from the abstract base class ExpressionBuilder. The extensibility concept follows the common pattern. By implementing this base class, you can build your own expression builder.

#### How it Works Internally

Internally, the expression builder creates a code snippet. This code is inserted into the page during the parsing step. The compiler treats this code as part of your custom code and creates the page object. This requires the code snippet to follow specific rules. Essentially, the code is an assignment. It assigns a value to the property of a control. Where the value comes from might require complex code, but the assignment is simple and limits the usage of such expressions. The most evident limitation is that usage without a control is not allowed. The following code will not function:

<div>

<%$ MyConfig:Value1 %>

</div>

Instead, you must use a control:

<div>

<asp:label runat="server" Text="<%$ MyConfig:Value1 %>" />

</div>

However, using the prefix is not enough to put it into operation. First, you must map the prefix to a specific type.

#### Declare the Prefix

The declaration of the prefix takes place in the *web.config* file. The path to the configuration element is <system.web><compilation><expressionBuilders>. The <expressionBuilders> element is usually absent. Add the following element to define a new mapping between a prefix and a type:

<expressionBuilders>

<add expressionPrefix="prefix" type="type,assembly"/>

</expressionBuilders>

The type follows the common schema of an assembly reference:

\* Namespace.Class, Assembly, Version, Culture, PublicKeyToken

\* Namespace.Class, Assembly

\* Namespace.Class, \_\_code

The first form defines the fully qualified name of an assembly. The last references the App\_Code folder, if the appropriate project type is used in Visual Studio. (Note the two underscores before the word “code” here.)

Now we need to create the type. As mentioned above, the abstract base class is where we start.

#### How the Expression Builder Works

The expression builder has only a few methods. Using these methods, it can parse the expression and generate the required code. Parsing is initiated using the ParseExpression method. If this step is successful, the GetCodeExpression method generates a code snippet. The only condition is that the compiler must be able to assign the snippet to a property. Therefore, the snippet must form the right hand side of an assignment:

control.Property = <this is the code expression>;

While the right hand side is merely a method call, this method can contain any amount of code (meaning that even the most complex operations are possible).

If the parsing fails, you should throw an exception. Otherwise, the GetCodeExpression method is invoked to create the code using the Code Document Object Model (CodeDOM). Although a short explanation follows, this book does not teach CodeDOM in depth. Refer to the MSDN documentation to learn more about source code generation:

http://msdn.microsoft.com/en-us/library/system.codedom.aspx

Table 4.6 lists the methods to implement to add an expression builder to an operation.

Table 4.6 Methods of an expression builder

|  |  |
| --- | --- |
| Method | Description |
| EvaluateExpression | Returns the value for non-compiled pages (see section “Accessing Settings for Non-Compiled Pages” for details) |
| GetCodeExpression | Creates the code snippet to insert into the page |
| ParseExpression | Parses the syntax of the expression |
| SupportsEvaluate | Indicates whether this expression builder supports non-compiled pages |

These methods are sufficient for creating our own expression builder.

### Creating an Expression Builder

In the first example, we need direct access to values stored in a SQL Server database. We are using the same database and table we created earlier in this chapter. Please refer to Figure 4.2. Specific criteria are used to retrieve the right values. The prefix is called “Cfg” and the syntax of the expressions looks like this:

<%$ Cfg:Key FROM category WHERE type %>

The syntax is similar to SQL, which makes it more readable. The type refers to the type of data, such as “image” or “label”. In the markup portion, this would look like:

<asp:Label ID="lblHeader" runat="server" ⮰

Text="<%$ Cfg:Header FROM PageData WHERE label %>"></asp:Label>

The following code is the same for an image:

<asp:Image ID="imgHeader" runat="server" ⮰

ImageUrl="<%$ Cfg:Header FROM PageData WHERE image %>"></asp: Image>

The created code for the image example looks like “images/header1.png”. Once defined, Visual Studio recognizes the expression syntax, as shown in the following figure:

Insert figure ASPEXTf0405.tif

Figure 4.6 The private expression builder appears in Visual Studio’s expression dialog

One solution is shown in Listing 4.16.

Listing 4.16 Using expressions for retrieving configuration data from a database

using System;

using System.Text;

using System.Web.Compilation;

using System.Data.SqlClient;

using System.CodeDom;

namespace Apress.Extensibility.Expressions

{

public class CfgExpression : ExpressionBuilder

{

private static ConfigDataDataContext ctx;

private static void EnsureCfgContext()

{

ctx = new ConfigDataDataContext();

}

public static string GetCfg(string key, string category, string type)

{

EnsureCfgContext();

var res = from row in ctx.aspnet\_Configurations ⮰

where row.cfg\_key == key ⮰

&& row.cfg\_category == category ⮰

&& row.cfg\_type == type ⮰

select row.cfg\_content;

return res.FirstOrDefault<string>();

}

public override System.CodeDom.CodeExpression GetCodeExpression( ⮰

System.Web.UI.BoundPropertyEntry entry, ⮰

object parsedData, ⮰

ExpressionBuilderContext context)

{

ExpressionValues cfgValues = parsedData as ExpressionValues;

if (cfgValues == null) throw new ArgumentException("parsedData");

CodePrimitiveExpression[] cArg = new CodePrimitiveExpression[]

{

new CodePrimitiveExpression(cfgValues.Key),

new CodePrimitiveExpression(cfgValues.Category),

new CodePrimitiveExpression(cfgValues.Type)

};

CodeTypeReferenceExpression t = new CodeTypeReferenceExpression( ⮰

typeof(CfgExpression));

CodeMethodInvokeExpression exp = new CodeMethodInvokeExpression(t, ⮰

"GetCfg", cArg);

return exp;

}

public override object ParseExpression(string expression, ⮰

Type propertyType, ⮰

ExpressionBuilderContext context)

{

return TokenParser.Parse(expression);

}

public override bool SupportsEvaluate

{

get

{

return false;

}

}

public override object EvaluateExpression(object target, ⮰

System.Web.UI.BoundPropertyEntry entry,

object parsedData, ⮰

ExpressionBuilderContext context)

{

return base.EvaluateExpression(target, entry, parsedData, context);

}

#region ConfigValues

class ExpressionValues

{

public string Key { get; set; }

public string Category { get; set; }

public string Type { get; set; }

}

#endregion

#region SimpleTokenParser

static class TokenParser

{

static ExpressionValues values;

static TokenParser()

{

values = new ExpressionValues();

}

internal static ExpressionValues Parse(string toParse)

{

if (String.IsNullOrEmpty(toParse))

throw new ArgumentNullException(toParse);

int i = 0;

string currentToken = String.Empty;

// value FROM cat WHERE type

while (true)

{

char c = toParse[i];

switch (c)

{

case 'F':

if (toParse.Substring(i, 5).Equals("FROM "))

{

values.Key = currentToken.Trim();

currentToken = String.Empty;

i += 4;

}

break;

case 'W':

if (toParse.Substring(i, 6).Equals("WHERE "))

{

values.Category = currentToken.Trim();

currentToken = String.Empty;

i += 5;

}

break;

default:

currentToken += c;

break;

}

if (++i < toParse.Length) continue;

values.Type = currentToken;

break; // end while

}

return values;

}

}

#endregion

}

}

The creation of dynamic code is the most important part. However, it doesn’t make sense to do everything dynamically. This is why the code for retrieving data is moved to the static method, GetCfg. The dynamic portion calls that method, which leads to generated code, such as:

lblHeader.Text = GetCfg(…);

The method is static because the private expression builder is not instantiated at run time. It’s a design-time tool which supports the page parser and design-time experience in Visual Studio. Only the assignment remains in the page’s code after compilation.

The procedure commences with the ParseExpression method. The object created here appears later as the parsedData parameter of the GetCodeExpression method. The TokenParser class in the example demonstrates how to parse custom strings. It analyses the string after the expression’s colon and creates three required parameters from it. The private class, ExpressionValues, stores the values, while the GetCodeExpression method builds the necessary code. It must return an object of type System.CodeDom.CodeExpression, which will contain a code fragment assignable to the right hand side of an assignment statement. Refer to the next table to view the most important CodeDOM functions. In this example, I use a simple call to a static method so that the code generation is as simple as possible. The core element is the method invocation:

new CodeMethodInvokeExpression(t, "GetCfg", cArg)

The argument

Table 4.7 Important CodeDOM methods

|  |  |
| --- | --- |
| Method | Description |
| CodeExpression | An abstract base class for all CodeDOM types. |
| CodeTypeReferenceExpression | A type reference, such as a type used in code |
| CodePrimitiveExpression | Any code expression which does not fit into other categories |
| CodeMethodInvokeExpression | A method call which requires, as parameters, the type where the method is defined, the name of the method, and an array of code fragments used as parameters for the method |

The GetCfg method follows the same pattern as used in the configuration example earlier. This draws from the DataContext class and LINQ to SQL to retrieve the values. The name of the context, ConfigDataDataContext, is based on the definition in *ConfigData.dbml*.

### Accessing Settings for Non-Compiled Pages

If pages contain mainly static content, the compilation step can be suppressed. Code expressions and expression builders will still function. Moreover, expression builders which support this offer a different way of handling the code creation for non-compiled pages.

You can suppress the compilation by setting the following attribute in the @Page directive:

<%@ Page Language="C#" CompilationMode="Never" %>

This can also be set globally in *web.config*.

<pages compilationMode="Never" />

This might sound strange, but imagine that this is a *web.config* file for a subfolder where all static pages reside. It makes sense to suppress compilation if you have a tool which is modifying the static content directly and frequently, as the compiler causes an additional load on the server when compiling pages over and over, without creating any useful code.

The point of this section is the fact that expression builders are still allowed and assigned. However, they aren’t able to create code snippets, as there is no compilation step able to support them. To indicate this usage scenario, the property SupportsEvaluate of the expression builder class must return true. This forces the page parser to call EvaluateExpression. The same will happen at runtime. Even if the page contains no code, and no compiler is invoked, the page handler serving the page will call the expression builder’s EvaluateExpression method. As shown in the first example, the code summons the appropriate configuration settings. This is a high-performance technique. It has limitations, but it is not as restricted as HTML pages would be. Beneath the configuration settings, you can also express date specifications or statistical data.

Listing 4.17 Support for non compiled pages

using System;

using System.Text;

using System.Web.Compilation;

using System.Data.SqlClient;

using System.CodeDom;

namespace Apress.Extensibility.Expressions

{

public class CfgExpression : ExpressionBuilder

{

private static ConfigDataDataContext ctx;

private static void EnsureCfgContext()

{

ctx = new ConfigDataDataContext();

}

public static string GetCfg(string key, string category, string type)

{

EnsureCfgContext();

var res = from row in ctx.aspnet\_Configurations ⮰

where row.cfg\_key == key ⮰

&& row.cfg\_category == category ⮰

&& row.cfg\_type == type ⮰

select row.cfg\_content;

return res.FirstOrDefault<string>();

}

public override bool SupportsEvaluate

{

get

{

return true;

}

}

public override object EvaluateExpression(object target, ⮰

System.Web.UI.BoundPropertyEntry entry, ⮰

object parsedData, ExpressionBuilderContext context)

{

ExpressionValues cfgValues = parsedData as ExpressionValues;

return GetCfg(cfgValues.Key, cfgValues.Category, cfgValues.Type);

}

}

Read the EvaluateExpression method as a combination of the ParseExpression and GetCodeExpression methods. The database access is the same as in the last example.

### Beyond Simple Expressions

This expression model supports several other features. However, as they are beyond the scope of this book, I’ll only give a short overview of them. Please refer to the official documention for more usage scenarios.

The ParseExpression method has another parameter named context. It’s not used in the example, as it’s intended for more advanced scenarios. This parameter is of the type ExpressionBuilderContext. Its purpose is to support the template base’s controls, and it contains the virtual path to the page or user control that contains the control. The expression builder refers to the control and is able to support different code generation strategies, depending on the requirements of the template based control.

Another parameter, propertyType, refers to the type of the property. Usually, this is String, but it can be modified to any type that suits your needs. For instance, when using a very limited set of options, an enum would be appropriate. This would create a dropdown element in Visual Studio’s expression builder for an advanced design-time experience.

The last parameter named entry is of the BoundPropertyEntry type. This allows access to the control itself, its ControlID property, and the control type.

Finally, the parameter ExpressionPrefix retrieves the currently used prefix. This assumes that you can assign the same expression builder multiple times and create complex and sophisticated frameworks on top of expression builders.

### Design-time Support

As demonstrated earlier, Visual Studio supports expressions through simple Expression dialogs. These can be invoked using the Expression section of the property browser. Simply select a control on the designer surface, open the property browser (F4) and click on the ellipses in the *Expressions* element in the *Data* section. The dialog contains a mapping tool which assigns properties to expressions. Here, the property grid uses attributes read by reflection to modify its behavior.

To support this, decorate the class of the expression builder with an ExpressionEditorAttribute, defined in the System.Web.UI.Design namespace of the System.Design assembly:

[ExpressionEditor(typeof(CfgExpressionEditor))]

The type CfgExpressionEditor implements the abstract base class ExpressionEditor. The only required step is to implement the GetExpressionEditorSheet method:

public override ExpressionEditorSheet GetExpressionEditorSheet(⮰

string expression, ⮰

IServiceProvider serviceProvider)

{

return new CfgExpressionEditorSheet(expression, serviceProvider);

}

The CfgExpressionEditorSheet type results from the implementation of the base class ExpressionEditorSheet. This type defines the properties in the property grid of the expression builder dialog that Visual Studio displays. Regarding other types, you can use the classes EditorAttribute and TypeConverterAttribute to change the data the property grid reads via reflection. You can find both types in the System.ComponentModel. In the example, it would be helpful to have three properties, such as “Key”, “Category”, and “Type”. The EditorAttribute defines the editor used in one entry of the property grid. This can be a dropdown list, or another dialog opened by clicking on the ellipses.

Note Dialogs and controls that are used in Visual Studio to extend the design-time experience must be WinForm controls. You cannot use web controls here. Some familiarity with Windows Forms Control programming is highly recommended before you begin extending Visual Studio’s design-time environment.

The next listings show the complete implementation of an expression builder with design-time support.

Listing 4.18 The complete definition of the design-time environment

using System;

using System.CodeDom;

using System.Web.Compilation;

using System.Web.UI;

using System.Web.UI.Design;

namespace Apress.Extensibility.Expressions.Time

{

[ExpressionPrefix("Time")]

[ExpressionEditor(typeof(TimeExpressionEditor))]

public class TimeExpression : ExpressionBuilder

{

public override CodeExpression GetCodeExpression(BoundPropertyEntry entry, ⮰

object parsedData, ⮰

ExpressionBuilderContext context)

{

TimeOptions to = (TimeOptions)Enum.Parse(typeof(TimeOptions), ⮰

entry.Expression, true);

switch (to)

{

case TimeOptions.Today:

// System.DateTime.Now

return new CodePropertyReferenceExpression( ⮰

new CodeTypeReferenceExpression(typeof(DateTime)), "Now");

case TimeOptions.Yesterday:

// System.DateTime.Now.AddDays(-1)

return new CodeMethodInvokeExpression( ⮰

new CodePropertyReferenceExpression( ⮰

new CodeTypeReferenceExpression(typeof(DateTime)), ⮰

"Now"), ⮰

"AddDays", ⮰

new CodePrimitiveExpression(-1));

case TimeOptions.Tomorrow:

// System.DateTime.Now.AddDays(1)

return new CodeMethodInvokeExpression( ⮰

new CodePropertyReferenceExpression( ⮰

new CodeTypeReferenceExpression(typeof(DateTime)), ⮰

"Now"), ⮰

"AddDays", ⮰

new CodePrimitiveExpression(1));

case TimeOptions.NextWeek:

// System.DateTime.Now.AddDays(7)

return new CodeMethodInvokeExpression(⮰

new CodePropertyReferenceExpression(⮰

new CodeTypeReferenceExpression(typeof(DateTime)), ⮰

"Now"), ⮰

"AddDays", ⮰

new CodePrimitiveExpression(7));

default:

throw new InvalidOperationException("The expression ⮰

value should be one of: Today, Yesterday, Tomorrow, NextWeek.");

}

}

}

}

This expression builder takes an enumeration value (see Listing 4.19) and creates code snippets that return a date value based on the current time and date. Note the attribute at the top of the class:

[ExpressionEditor(typeof(TimeExpressionEditor))]

This assigns the TimeExpressionEditor type to modify the design-time environment.

Listing 4.19 The complete definition of the design-time environment

namespace Apress.Extensibility.Expressions.Time

{

public enum TimeOptions

{

Yesterday,

Today,

Tomorrow,

NextWeek

}

}

The TimeExpressionEditor type derives from the ExpressionEditor. Its purpose is twofold. Firstly, it allows the output of values in the designer. If a user opens the designer surface editing a page or user control, the expressions can be used just like any other expression. Instead of showing the expression’s code, such as *Time:Today*, it shows the calculated value, *March 20, 2009*. This results in a more realistic design-time experience.

Listing 4.20 The complete definition of the design-time environment

using System;

using System.Web.UI.Design;

namespace Apress.Extensibility.Expressions.Time

{

public class TimeExpressionEditor : ExpressionEditor

{

// Evaluates an expression at design-time for preview purposes

public override object EvaluateExpression(string expression, ⮰

object parseTimeData, ⮰

Type propertyType, ⮰

IServiceProvider serviceProvider)

{

TimeOptions to = (TimeOptions) Enum.Parse(typeof(TimeOptions), ⮰

expression, true);

switch (to)

{

case TimeOptions.Today:

return System.DateTime.Now.ToString();

case TimeOptions.Yesterday:

return System.DateTime.Now.AddDays(-1).ToString();

case TimeOptions.Tomorrow:

return System.DateTime.Now.AddDays(1).ToString();

case TimeOptions.NextWeek:

return System.DateTime.Now.AddDays(7).ToString();

default:

throw new InvalidOperationException("The expression value⮰

should be one of: Today, Yesterday, Tomorrow, NextWeek.");

}

}

public override ExpressionEditorSheet GetExpressionEditorSheet(⮰

string expression, IServiceProvider serviceProvider)

{

return new TimeExpressionEditorSheet(expression, serviceProvider);

}

}

}

The code demonstrates how to define the editor sheet using the GetExpressionEditorSheet method. The TimeExpressionEditorSheet type used here is defined below:

Listing 4.21 The complete definition of the design time environment

using System;

using System.ComponentModel;

using System.Web.UI.Design;

namespace Computacenter.Expressions.Time

{

public sealed class TimeExpressionEditorSheet : ExpressionEditorSheet

{

private string \_day;

public TimeExpressionEditorSheet(string day, ⮰

IServiceProvider serviceProvider)

: base(serviceProvider)

{

Day = day;

}

[TypeConverter(typeof(TimeConverter))]

public string Day

{

get

{

return \_day;

}

set

{

\_day = value;

}

}

public override string GetExpression()

{

return Day;

}

// TypeConverter to provide dropdown of valid values

private sealed class TimeConverter : StringConverter

{

public override bool GetStandardValuesExclusive(⮰

ITypeDescriptorContext context)

{

return true;

}

public override bool GetStandardValuesSupported(⮰

ITypeDescriptorContext context)

{

return true;

}

public override TypeConverter.StandardValuesCollection ⮰

GetStandardValues(ITypeDescriptorContext context)

{

string[] standardValues = Enum.GetNames(typeof(TimeOptions));

return new StandardValuesCollection(standardValues);

}

}

}

}

These are the properties shown in the expression builder dialog in Visual Studio. The [TypeConverter(typeof(TimeConverter))] attribute converts values from the property grid into the enumeration. This is necessary as the property grid transforms a type internally into a string representation, which allows you to select it. In this case, the dropdown will return single strings like “Today” or “Tomorrow”. These values must be converted back and forth by the TimeOptions enumeration.

In the property grid, we handle the enumeration as a dropdown. You can force this by returning true in the GetStandardValuesSupported method. The GetStandardValues method must return the array of elements presented in the dropdown. Enum.GetNames performs this function. If the enumeration values do not fit your needs, you may decorate them with DescriptionAttribute attributes (System.Componentmodel namespace) and obtain the final strings from this attribute instead of using the enum values.

public enum TimeOptions

{

[Description("The day before today")]

Yesterday,

[Description("Now")]

Today,

[Description("The day after today")]

Tomorrow,

[Description("Some time in the future")]

NextWeek

}

This is just one of the many options you have in extending the design-time environment. However, the more code you add, the harder it will be to debug. The next section covers the basic principles of design-time debugging.

#### Debugging Design-time Extensions

Design-time support should be part of any extension model. However, it’s difficult to debug these extensions, and this is probably why developers avoid adding design-time support to their applications. There is no debugger attached to design-time code. This means that you cannot add breakpoints easily or step into the code to view its internal operations. 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 launch another instance of Visual Studio and debug the same code there. To accomplish this, open the property pages of the Web project and select the option *Start external program*.

Insert ASPEXTf0407.tif

Figure 4.7 Start another instance of Visual Studio to debug at design-time

Choose the Visual Studio executable *devenv.exe* as the external program. When you start debugging the project (by hitting F5) another instance of Visual Studio will appear. In this instance, you load the same project and start using the design-time environment. In the first instance of Visual Studio, you can now set breakpoints and debug as you would do at runtime. To reach specific code sections, just start using the code within the second instance.

# Summary

In this chapter, we saw an overview of providers and how they can be used to create modern, multi-tier architecture. Providers result in highly configurable and extensible applications which don’t require additional code when changes to basic features are made. The configuration of providers is a major part of this environment. In addition to providers, the whole configuration can be extended through *web.config*. This chapter showed the extensibility of *web.config* in defining private sections and elements. Lastly, the configuration settings were made available in the declarative part of the web pages, the markup, by supporting private expression builders. Expression builders form a final extensibility feature that improves the design-time experience.