Chapter 8— Site Management and Navigation

Site Management

In this chapter, I’ll look at site-specific extensibility features such as the site map provider, which is based on the extensible provider model that we’ve already learned about. A site map defines the navigation structure of a web site. By default, the data store is the *web.sitemap* XML file. Although it’s easy to change the default setting, you’ll need a custom site map provider in order to store data anywhere other than in an XML file.

The navigation depends usually on the web pages on disc. If you wish to hide the internal physical structure of your site or to serve pages that do not originate from disk files, the VirtualPathProvider can accomplish this. Using this you can create a virtual site structure.

In this chapter, you’ll learn how to implement and modify the behavior of the SiteMapProvider and VirtualPathProvider.

## Site Map Providers

ASP.NET features a data driven navigation system that uses hierarchical data sources and associated controls. It’s much easier to create a site navigational system and track the current position of the user if you use the Menu control, TreeView control or SiteMapPath control.

As is common in ASP.NET, these controls use a provider to obtain data from a specific data source.

### Internal Site Map Provider

Before you start writing your own provider, let’s look at how the internal one is implemented and how it works. The XmlSiteMapProvider provider reads an XML file containing the site structure as a hierarchy of SiteMapNode elements. The file itself looks like this:

<siteMap>

<siteMapNode title="Home" description="Home" url="~/default.aspx" >

<siteMapNode title="Services" description="Services we offer" ⮰

url="~/Services.aspx">

<siteMapNode title="Training" description="Training classes" ⮰

url="~/Training.aspx" />

<siteMapNode title="Consulting" description="Consulting services" ⮰

url="~/Consulting.aspx" />

</siteMapNode>

</siteMapNode>

</siteMap>

The file is named *web.sitemap* by default. However, you can configure the provider to accept any filename. The best way is to use the SiteMapDataSource control. A page using these controls could look like this:

<body>

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

<div>

<h2>Using SiteMapPath</h2>

<asp:SiteMapPath ID="SiteMapPath1" Runat="server"></asp:SiteMapPath>

<asp:SiteMapDataSource ID="SiteMapDataSource1" Runat="server" />

<h2>Using TreeView</h2>

<asp:TreeView ID="TreeView1" Runat="Server" DataSourceID="SiteMapDataSource1">

</asp:TreeView>

<h2>Using Menu</h2>

<asp:Menu ID="Menu2" Runat="server" DataSourceID="SiteMapDataSource1">

</asp:Menu>

<h2>Using a Horizontal Menu</h2>

<asp:Menu ID="Menu1" Runat="server" DataSourceID="SiteMapDataSource1"

Orientation="Horizontal"

StaticDisplayLevels="2" >

</asp:Menu>

</div>

</form>

</body>

This web page already has a complete navigation structure. The SiteMapPath control indicates your current position in the site structure via a “breadcrumb” style display. The TreeView and Menu controls create a tree of all pages. The Menu control provides additionally a JavaScript-driven drop-down menu. Both include plenty of features and assistance for creating sophisticated menus. As shown in the example, the controls can appear several times to create top level, side level, or other kinds of menu. You can even use several SiteMapDataSource controls to define different entry points in the site structure or prepare submenus and partial menus.

Tip: If you have not yet used these controls intensively, the following web address is one entry point into the documentation: http://msdn.microsoft.com/en-us/library/ms178423.aspx .

This model has several advantages—as well as disadvantages. On the positive side, it’s easy and fast to use, and the structure of the content relates directly to the navigation. Navigation is usually structured hierarchically, and XML represents this perfectly.

There are also some negatives. Firstly, the URL is used internally as a key, which means that you cannot use the same URL twice to point to a page from different locations in the hierarchy. It’s possible to work around this—you could add a dummy querystring parameter, or use the UrlMappings defined in the *web.config*, but this isn’t ideal. Furthermore, the default provider reads the definition file once the application starts, and holds the whole content in memory. Large structures operating on several levels for submenus consume a lot of memory.

The internal provider is defined in the *web.config* file like any other:

<system.web>

<siteMap defaultProvider="XmlSiteMapProvider" enabled="true">

<providers>

<add name="XmlSiteMapProvider"

description="Default SiteMap provider."

type="System.Web.XmlSiteMapProvider"

siteMapFile="Web.sitemap" />

</providers>

</siteMap>

</system.web>

The configuration, the sitemap file, and the appropriate controls are everything you need for complete site navigation.

### Localization

The embedded sitemap provider supports full localization. The resource provider (demonstrated in chapter 4) is aware of the encoded resource keys and can obtain the content from a resource file. When providing different resource files for languages or cultures, you only need one sitemap definition for each different language. It is, however, not possible to have different site structures for each culture.

### Security Issues

The sitemap provider model does support a security concept. If you define the roles for each node, the provider will display only those nodes the user is permitted to view. Security trimming can be switched on and off. Microsoft states that using more than 150 nodes will have a serious performance impact. When using the sitemap programmatically, the provider will return null if the current user has no access rights, and an exception will occur if the user is not permitted to read the root node.

### Reasons to Write a Custom Sitemap Provider

There are several reasons why you might consider creating a custom provider:

\* You support localization and need a different site structure for each culture

\* You have hundreds of nodes and need security trimming

\* You have limited memory but a large structure

\* You want to support the same nodes several times in the hierarchy

\* You want to allow users to add/remove nodes

\* You want to persist the structure in an alternative data store

There are two ways of customizing the provider’s behavior: either extending the default provider, or writing your own. Extending the default provider lets you keep the XML file, but allows different behavior when retrieving nodes or adding security features. Using your own provider gives you the ability to use a different data source.

Tip: You can use XmlDocument or XDocument classes to retrieve and change nodes directly in the *web.sitemap* file in order to edit the site structure on the fly. This does not require a new provider, but extends the behavior of the site navigation universe. Consider this before writing a lot of code just to get read-write access.

## Writing a Custom Sitemap Provider

The purpose of the provider shown here is to access another data source. For web farm or web garden scenarios, or flexible access to the site structure definition, a SQL Server database is the best choice. The amount of data is relatively small, and SQL Server retains it in memory, resulting in fast access.

### Prerequisites

A custom provider inherits from a base class (as we’ve seen many times). For the sitemap provider, you have a choice between the System.Web.StaticSiteMapProvider class and its base class, System.Web.SiteMapProvider. These greatly simplify things, because they come with several basic methods already implemented.

You can use the StaticSiteMapProvider if:

\* You read nodes either once or rarely

\* All information is cached through the lifetime of the provider

The lifetime of the provider is usually the lifetime of the application. Changing the sitemap requires a restart.

If any one of the following conditions is met, the SiteMapProvider type is a better choice:

\* The data is stored in a database

\* The data changes frequently throughout the application’s lifetime

\* The user can change the structure and personalize the menu

### Learning About the Base Classes

Before you start implementing, you’ll need to know which methods you’ll be dealing with. Firstly, take a look at the SiteMapProvider class:

public abstract class SiteMapProvider : ProviderBase

{

protected SiteMapProvider();

public virtual SiteMapNode CurrentNode { get; }

public bool EnableLocalization { get; set; }

public virtual SiteMapProvider ParentProvider { get; set; }

public string ResourceKey { get; set; }

public virtual SiteMapNode RootNode { get; }

public virtual SiteMapProvider RootProvider { get; }

public bool SecurityTrimmingEnabled { get; }

public event SiteMapResolveEventHandler SiteMapResolve;

protected virtual void AddNode(SiteMapNode node);

protected internal virtual void AddNode(SiteMapNode node, ⮰

SiteMapNode parentNode);

public virtual SiteMapNode FindSiteMapNode(HttpContext context);

public abstract SiteMapNode FindSiteMapNode(string rawUrl);

public virtual SiteMapNode FindSiteMapNodeFromKey(string key);

public abstract SiteMapNodeCollection GetChildNodes(SiteMapNode node);

public virtual SiteMapNode GetCurrentNodeAndHintAncestorNodes(int upLevel);

public virtual SiteMapNode GetCurrentNodeAndHintNeighborhoodNodes(int upLevel, ⮰

int downLevel);

public abstract SiteMapNode GetParentNode(SiteMapNode node);

public virtual SiteMapNode ⮰

GetParentNodeRelativeToCurrentNodeAndHintDownFromParent(⮰

int walkupLevels, ⮰

int relativeDepthFromWalkup);

public virtual SiteMapNode GetParentNodeRelativeToNodeAndHintDownFromParent(⮰

SiteMapNode node, int walkupLevels, int relativeDepthFromWalkup);

protected internal abstract SiteMapNode GetRootNodeCore();

protected static SiteMapNode GetRootNodeCoreFromProvider(⮰

SiteMapProvider provider);

public virtual void HintAncestorNodes(SiteMapNode node, int upLevel);

public virtual void HintNeighborhoodNodes(SiteMapNode node, int upLevel, ⮰

int downLevel);

public override void Initialize(string name, NameValueCollection attributes);

public virtual bool IsAccessibleToUser(HttpContext context, SiteMapNode node);

protected internal virtual void RemoveNode(SiteMapNode node);

protected SiteMapNode ResolveSiteMapNode(HttpContext context);

}

The following tables explain the methods, events and properties, as we need to know what to implement and the intended behavior. The only unusual thing is the length of some method names, but this clarifies their usage.

Because the provider is transparent, controls such as Menu or TreeView must work with a custom implementation, too. Firstly, the properties defined by the provider:

Table 8-1. Properties defined in SiteMapProvider

|  |  |
| --- | --- |
| Property | Description |
| CurrentNode | Reference to the current node (and therefore the related page) |
| EnableLocalization | Indicates whether the provider supports localization using resources |
| ParentProvider | The parent provider, if the provider supports a hierarchy of providers |
| ResourceKey | The root name of the resources used to localize content |
| RootNode | The root node |
| RootProvider | The root provider, if a hierarchy of providers is used |
| SecurityTrimmingEnabled | Indicates whether or not the provider supports security settings. If supported, the provider doesn’t return nodes that don’t match the security level. |

Secondly, an event that the provider should support:

Table 8-2. Events defined in SiteMapProvider

|  |  |
| --- | --- |
| Event | Description |
| SiteMapResolve | Event raised when data requested from data sources is ready |

Thirdly, the methods that build most features:

Table 8-3. Methods defined in SiteMapProvider

|  |  |
| --- | --- |
| Method | Description |
| AddNode | Adds either the root node or another node |
| FindSiteMapNode | Finds a node |
| FindSiteMapNodeFromKey | Finds a node using a key |
| GetChildNodes | Gets all child nodes of the specified node |
| GetCurrentNodeAndHintAncestorNodes | Gets the current node and pre-loads the ancestors |
| GetCurrentNodeAndHintNeighborhoodNodes | Gets the current node and pre-loads the neighbors |
| GetParentNode | Gets the parent node |
| GetParentNodeRelativeToCurrentNodeAndHintDownFromParent | Gets the current node’s parent node and pre-loads all ancestors for the provided number of levels |
| GetParentNodeRelativeToNodeAndHintDownFromParent | Gets the given node’s parent and pre-loads all ancestors for the provided number of levels |
| GetRootNodeCore | Gets the parent regardless of the provider if a hierarchy of providers is used |
| GetRootNodeCoreFromProvider | Gets the parent of the specified provider if a hierarchy of providers is used |
| HintNeighborhoodNodes | Pre-loads a number of levels down or up the hierarchy from the given node |
| Initialize | Initializes the provider (inherited from base class) |
| RemoveNode | Removes a node |
| ResolveSiteMapNode | Fires the SiteMapResolved event |

Several methods include “Hint” in their name. Use these methods to handle data in huge hierarchies, as they allow you to deal with large amounts of data and they avoid holding the whole tree in memory. Consider implementations such as SharePoint, which could create deep hierarchies of site collections, sites, pages, and subpages that might have their own navigation systems. Using the “Hint” methods, you can define the depth that is populated from the tree. This limits the data handled by the provider to an amount that makes sense for the specific action taken by the user. It also assumes that nobody wants to see the whole hierarchy at once.

#### Sitemap Nodes in Code

The provider transforms the data from the data source into a hierarchy of SiteMapNode objects. The base class implements three interfaces, ICloneable, IHierarchyData, and INavigateUIData. This indicates the true nature of the object hierarchy, as well as how to deal with the nodes.

Table 8-4. Members defined in SiteMapNode

|  |  |
| --- | --- |
| Method | Description |
| GetAllNodes | Gets a read-only list of all nodes beginning from the current node |
| GetDataSourceView | Gets the SiteMapDataSourceView object that represents the underlying data source |
| GetHierarchicalDataSourceView | Gets the SiteMapHierarchicalDataSourceView object that represents the underlying hierarchical data source |
| IsAccessibleToUser | Determines whether the node is accessible according to current security context |
| IsDescendantOf | Determines whether the current node is a descendant of the given node |
| GetExplicitResourceString | Gets the localized string to retrieve the explicit resource definition |
| GetImplicitResourceString | Gets the localized string to retrieve the implicit resource definition |
| GetChildren | Gets all child nodes |
| GetParent | Gets the parent node |
| HasChildren | Indicates whether a node has children |
| Item | Gets the current node |
| Path | Gets the path that describes the node |
| Type | Gets the node’s type |
| Description | Gets the node’s description |
| Name | Gets the node’s name |
| NavigateUrl | Gets the node’s URL |
| Value | Gets the node’s title |

#### Implementing Security

The SecurityTrimmingEnabled property indicates if the nodes are limited for the current user. It can be set in *web.config*:

<siteMap defaultProvider="XmlSiteMapProvider" enabled="true">

<providers>

<add name="XmlSiteMapProvider"

description="Default SiteMap provider."

type="System.Web.XmlSiteMapProvider "

siteMapFile="Web.sitemap"

securityTrimmingEnabled="true" />

</providers>

</siteMap>

The default implementation used in XmlSiteMapProvider resolves the roles attributes defined in the siteMapNode elements in the *web.sitemap* file. If the current user is in the role, and the RoleProvider is activated, the XmlSiteMapProvider limits visibility to the correct nodes. If used programmatically, the IsAccessibleToUser property possessed by each SiteMapNode instance indicates the same thing. The list of available roles is also available in the Roles property. A query can use either the IList type directly or “\*” for all. The programmatic access uses a simple fall back strategy:

\* If the role was found or the query consists of “\*”, return the node

\* If the role was not found, attempt to authenticate the URL defined by the node

\* If Windows authentication is used, attempt to authenticate again using the credentials of the current user and the ACL (access control list) of the target file for the URL

If everything fails, the provider does not return the node and assumes that the user does not have sufficient authority. Note that this is not an inheritance strategy. If a user has the right to read a specific node, it does not follow that he or she has the right to read any subsequent node. This allows different security settings for each level. However, if the right to access a specific node is denied, the strategy stops and all child nodes are blocked. This ensures that there are no isolated child nodes accessible from any places other than the sitemap hierarchy.

### Implementing a SQL Server Based Navigation

SQL Server 2008 includes a new data type, HierarchyId. This can be used to implement a custom database-driven sitemap provider.

#### Preparing the Database

Firstly, a database is required. In this example, I use the common *aspnetdb.mdf* database created for the role and membership provider, if default providers are used. Listing 8-1 defines an additional table named *aspnet\_Navigation*:

Listing 8-1. The table definition using the HierarchyId data type

CREATE TABLE aspnet\_Navigation

(

SiteMapNode hierarchyid NOT NULL,

Title varchar(100) NOT NULL,

Description varchar(200) NULL,

Url varchar(200) NULL,

Roles varchar(200) NULL

)

Let’s proceed by filling in some data, as shown in the next image:

Insert 19835f0800.tif

Figure 8-1. Table with some test values in Visual Studio Professional

The HierarchyId column contains the node definition as a path beginning with a slash. The internal format is binary and must be transformed using the ToString method SQL provides for this type. This is what Visual Studio does when presenting the table data in edit mode.

#### Creating the Provider

Now that we have everything, we can create the custom provider. Listing 8-2 shows the final result. You can create it in the current Web project or in a separate project. Using it in a Web project will change the configuration slightly. The big advantage of building a custom sitemap provider in its own project is to be able to re-use it with other web applications.

Listing 8-2. The SqlSiteMapProvider using the HierarchyId data type

using System;

using System.Collections.Generic;

using System.Collections.Specialized;

using System.Configuration.Provider;

using System.Data;

using System.Data.Common;

using System.Data.SqlClient;

using System.Linq;

using System.Security.Permissions;

using System.Web;

using System.Web.Configuration;

namespace Apress.Extensibility.SqlSiteMap

{

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

public class SqlSiteMapProvider : StaticSiteMapProvider

{

private string \_connect;

private int \_indexNode;

private int \_indexTitle;

private int \_indexUrl;

private int \_indexDesc;

private int \_indexRoles;

private int \_indexParent;

private int \_indexRoot;

private SiteMapNode \_root;

private Dictionary<string, SiteMapNode> \_nodes = ⮰

new Dictionary<string, SiteMapNode>(16);

public override void Initialize(string name, NameValueCollection config)

{

if (config == null)

throw new ArgumentNullException("config");

if (String.IsNullOrEmpty(name))

name = "SqlSiteMapProvider";

base.Initialize(name, config);

string connect = "siteMap";

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

throw new ProviderException("No connection");

\_connect = WebConfigurationManager.ConnectionStrings[connect]. ⮰

ConnectionString;

if (String.IsNullOrEmpty(\_connect))

throw new ProviderException("no connection string");

}

public override SiteMapNode BuildSiteMap()

{

lock (this)

{

if (\_root != null)

return \_root;

SqlConnection connection = new SqlConnection(\_connect);

try

{

connection.Open();

SqlCommand command;

command = new SqlCommand(

@"SELECT \*, SiteMapNode.ToString() AS SiteMapNodeString,

SiteMapNode.GetAncestor(1).ToString() AS Parent,

hierarchyid::GetRoot().ToString() AS Root

FROM aspnet\_Navigation", connection);

command.CommandType = CommandType.Text;

SqlDataReader reader = command.ExecuteReader();

\_indexNode = reader.GetOrdinal("SiteMapNodeString");

\_indexUrl = reader.GetOrdinal("Url");

\_indexTitle = reader.GetOrdinal("Title");

\_indexDesc = reader.GetOrdinal("Description");

\_indexRoles = reader.GetOrdinal("Roles");

\_indexParent = reader.GetOrdinal("Parent");

\_indexRoot = reader.GetOrdinal("Root");

string parentKey;

if (reader.Read())

{

\_root = CreateSiteMapNode(reader, true, out parentKey);

AddNode(\_root, null);

while (reader.Read())

{

SiteMapNode node = CreateSiteMapNode(reader, false, ⮰

out parentKey);

SiteMapNode parent = GetParentSiteMapNode(parentKey);

AddNode(node, parent);

}

}

}

finally

{

connection.Close();

}

return \_root;

}

}

protected override SiteMapNode GetRootNodeCore()

{

BuildSiteMap();

return \_root;

}

private SiteMapNode CreateSiteMapNode(DbDataReader reader, bool forRoot, ⮰

out string parentKey)

{

string nodeString = reader.GetString(forRoot ? \_indexRoot : \_indexNode);

string title = reader.IsDBNull(\_indexTitle) ? null : ⮰

reader.GetString(\_indexTitle).Trim();

string url = reader.IsDBNull(\_indexUrl) ? null : ⮰

reader.GetString(\_indexUrl).Trim();

string description = reader.IsDBNull(\_indexDesc) ? null : ⮰

reader.GetString(\_indexDesc).Trim();

string roles = reader.IsDBNull(\_indexRoles) ? null : ⮰

reader.GetString(\_indexRoles).Trim();

string[] rolelist = null;

if (!String.IsNullOrEmpty(roles))

rolelist = roles.Split(new char[] { ',', ';' }, 512);

SiteMapNode node = new SiteMapNode(this, nodeString,

url,

title,

description,

rolelist, null, null, null);

parentKey = reader.IsDBNull(\_indexParent) ? null : ⮰

reader.GetString(\_indexParent);

\_nodes.Add(nodeString, node);

return node;

}

private SiteMapNode GetParentSiteMapNode(string parentKey)

{

var parent = from n in \_nodes ⮰

where n.Key.Equals(parentKey) ⮰

select n.Value;

return parent.FirstOrDefault<SiteMapNode>();

}

}

}

The implementation is intentionally brief. The HierarchyId datatype makes it much easier to fetch a hierarchy. The heart is this SQL query, which retrieves all the needed data in one query:

SELECT \*, SiteMapNode.ToString() AS SiteMapNodeString,

SiteMapNode.GetAncestor(1).ToString() AS Parent,

hierarchyid::GetRoot().ToString() AS Root

FROM aspnet\_Navigation

A great attribute of SQL Server is the ability to provide additional methods for data types. Here I use the GetAncestor and GetRoot methods. GetAncestor(1) fetches the parent element, while GetRoot() obtains the hierarchy’s root. Bear in mind that some methods of these T-SQL functions are members, such as GetAncestor, while other methods are static, like GetRoot. The double colon, “::” indicates access to static methods in SQL Server syntax.

A DataReader object provides fast forward-only access to the SQL query results, from which we assemble SiteMapNode objects. The node type is provided so that the same method can be re-used. The value is retrieved using reader.GetString(<index>). The index is the number of the column returned by the SQL query. The member fields’ \_indexRoot, \_indexNode or \_indexParent contain the ordinal number for the column. In the SQL statement, the result is cast using ToString. This requires the GetString method when returning results.

The AddNode method from the StaticSiteMapProvider base class is responsible for building the hierarchy. To obtain the necessary information, provide a parent for each node. In the example, I assume that the parent is already defined as a node. This requires the table data to be ordered so that parents appear before their children. The GetParentSiteMapNode method is used to retrieve the parent from the private \_nodes collection when building the map. This collection exists for that purpose and is no longer used when the nodes are retrieved from the sitemap controls. The parentKey used here is simply the value returned from SQL with the GetAncestor method, as mentioned before.

#### Configuring the Custom Provider

The configuration consists of two parts. The first step is to define the connection string in order to access the data source:

<connectionStrings>

<add connectionString="Data Source=.\SQLEXPRESS;Initial ⮰

Catalog=aspnetdb;Integrated Security=True" ⮰

name="siteMap"/>

</connectionStrings>

The provider itself needs the basic format, as shown before:

<system.web>

<siteMap enabled="true" defaultProvider="MySqlSiteMapProvider">

<providers>

<add name="MySqlSiteMapProvider"

type="Apress.Extensibility.SqlSiteMap.SqlSiteMapProvider"

description="My SqlSiteMapProvider"

securityTrimmingEnabled="false"

connectionStringName="siteMap"

/>

</providers>

</siteMap>

</system.web>

Finally, the data source used on the web pages needs to point to the provider. If you have several providers, each data source can use its own provider.

<asp:SiteMapDataSource ID="SiteMapDataSource1" runat="server" ⮰

SiteMapProvider="MySqlSiteMapProvider" />

Although there are no changes to the various controls using this data source, some features such as security trimming are not yet implemented. However, you’ve learned how to extend the default provider model regarding site maps, and how to benefit from the new features SQL Server 2008 provides.

#### Testing the Provider

To test the provider, write a simple page that contains at least a Menu and a SiteMapPath control bound to the modified SiteMapDataSource control, above. A page such as this one will suffice:

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

<div>

<asp:SiteMapPath ID="SiteMapPath1" runat="server"

SiteMapProvider="MySqlSiteMapProvider">

</asp:SiteMapPath>

<br />

<br />

Menu:<br />

<asp:Menu ID="Menu1" runat="server" ⮰

StaticDisplayLevels="1" DataSourceID="SiteMapDataSource1" >

</asp:Menu>

<br />

<br />

</div>

<asp:SiteMapDataSource ID="SiteMapDataSource1" runat="server"

SiteMapProvider="MySqlSiteMapProvider" />

</form>

The data source is only required for the Menu control. Breadcrumbs are displayed using a SiteMapPath control, which can connect directly to the custom provider using its name. The following figure shows the output:

Insert 19835f0801.tif

Figure 8-2. Menu and SiteMapPath controls with data from SQL Server

Remember that you need to add the page names to the table data. Otherwise, you’ll encounter page load errors when you choose the menu items.

### Suggestions for Extending the Example

The example is as simple as possible, containing nothing more than the code required to get it working. It lacks error checking and proper exception handling. Consider making the following changes before using the code in a production environment:

\* Allow duplicate nodes

\* Dispose of the dictionary that stores the parents

\* Order the SQL query to accept unordered node definitions

\* Add a primary key to the table in order to improve speed for large node sets

There are further ways of extending the behavior.

## Extending the VirtualPathProvider

Complex sites with hundreds of pages are difficult to maintain. For some sections, a file structure with static data seems to be more productive, whereas other sections are composed dynamically from databases. This difference should be indiscernible to regular users and search engines alike. Search engines follow each link on a page and find the navigation paths through a site, indexing the content of each page on the way. Users bookmark pages and return directly to them later. However, neither of these behaviors are what developers are looking for in creating pages.

The VirtualPathProvider is designed to separate the internal and external structures. Like any other provider, it works transparently, using a pluggable approach. The difference is that the virtual path provider does not access a database and internally is different from all providers described so far. However, to implement a virtual path provider, you’ll have to inherit and implement an abstract base class—VirtualPathProvider.

### Using the VirtualPathProvider

In previous chapters, I started with a brief overview of the default provider before looking at a custom implementation. Here it’s a bit different. There is no default provider, which means that each page is handled at its physical location and dynamically-generated content appears as is, within the one and only page possessing that functionality.

Implementing a virtual path provider is all about changing the default behavior of the path resolution for any regular web site. Some examples of what it can accomplish might help you decide if it’s worth the effort:

\* Storing all content in a database

\* Intercepting the parsing of pages in scenarios where pages come from different sources

\* Customizing the path resolution for the root of the application using “~”

\* Modifying the compilation process

The parsing aspect especially is not one that you’ll be using on a daily basis. You might have heard about large ASP.NET-based projects such as SharePoint. SharePoint is an application providing a content management system, among several other features. Users can create content and add active parts—called webparts—to these pages. The content is stored in a database and pages are built dynamically. ASP.NET does not support this out-of-the-box, as the compiler would not be able to locate any pages to compile. A powerful virtual path provider is used to resolve pages found in the database, mix them with those still stored as files, and create a final structure.

While SharePoint is a technically admirable example of a virtual path provider, in our example we’ll consider a more modest scenario. Given a website with a “MySite” section, let’s expose it with a structure like this:

Site  
 MySiteJoerg

Controls

MySiteJohn

Controls

MySiteKathryn

Controls

Users will think that they have a private site. They can upload and modify pages there, and you could even allow them to store files. However, if users follow convention and use paths such as *~/controls/mycontrol.ascx* to link their web pages to user controls, it will fail. Consider this example:

<%@ Register Src="~/Controls/my.ascx" TagName="my" TagPrefix="john" %>

User John is allowed to upload active content to his private space under the path *MySiteJohn*. He assumes—wrongly—that this is the root for his private space. John expects it to resolve to */MySiteJohn/Controls/my.ascx*. It actually resolves to */Site/Controls/my.ascx*. Of course, this causes a compilation exception.

### Register the VirtualPathProvider

In the previous description, I noted that the virtual path provider is unlike other providers. In fact, “provider” is probably not the best word to describe it, as it’s different from other providers in the way it works, the way it accesses the data, and the way it’s configured. To correctly resolve paths, even during compilation, the provider must be initialized earlier in the ASP.NET framework. Thus, the *web.config* file is not the correct place for defining it. Instead, it has to be defined using a special static method, AppInitialize, or in the *global.asax* file.

The Nature of AppInitialize

You can use any class in either the code path of your web site or in the App\_Code folder of your web application project to define a public, static method called AppInitialize:

public class SomeInitClass

{

public static void AppInitialize()

{

// Action required to initialize the application

}

}

The method is called during the initialization cycle of the application. This is exactly the same as if you’d put the code in the Application\_Start event handler defined in *global.asax*. There is little information available about why two ways exist to achieve the same thing. Perhaps it’s because some applications lack a *global.asax* file, and creating and maintaining one for a single event would be overkill. In this case, the AppInitialize method is a better option, as it moves the code portion to one location. You can use any class to define this method—even the one we created here to define a VirtualPathProvider. This puts the two parts—configuration and definition—tightly together.

With either method of registering the provider, the following code is required:

namespace Apress.Extensibility.PathProvider

{

public static class AppStart

{

public static void AppInitialize()

{

CustomPathProvider customProvider = new CustomPathProvider ();

HostingEnvironment.RegisterVirtualPathProvider(customProvider);

}

}

}

This code assumes that the custom provider is named CustomPathProvider, as shown in the example below. The code also uses other types, and those that are useful regarding virtual path providers are explained below.

### Prerequisites for a VirtualPathProvider

A custom VirtualPathProvider requires specific permissions, according to MSDN. If you look into the code using Reflector, you’ll find:

[AspNetHostingPermission(SecurityAction.Demand, ⮰

Level=AspNetHostingPermissionLevel.High)]

public static void RegisterVirtualPathProvider(VirtualPathProvider virtualPathProvider)

{

if (\_theHostingEnvironment == null)

{

throw new InvalidOperationException();

}

if (!BuildManager.IsPrecompiledApp)

{

RegisterVirtualPathProviderInternal(virtualPathProvider);

}

}

The AspNetHostingPermission attribute means that you’ll need to run in full trust in order to register a provider. If you don’t have permission, a SecurityException is thrown. This is the same reason as for the check regarding pre-compilation. Both tests ensure that a VirtualPathProvider cannot be used without permissions or within a running environment. The third check searches for an existing hosting environment. Without this, a path provider cannot function.

While the missing hosting environment throws an exception, the pre-compilation check allows the method to fail silently. To understand why, let’s examine the Initialize method in RegisterVirtualPathProviderInternal. It’s not the public method, but the internal one that looks like this:

internal virtual void Initialize(VirtualPathProvider previous)

{

this.\_previous = previous;

this.Initialize();

}

If a VirtualPathProvider supports a fallback scenario, this setting ensures that the “fallback provider” is available. The overridden Initialize method that you may have changed is called after this.

### Helpful Classes for Path and File Operations

HostingEnvironment is a class defined in System.Web.Hosting that defines the very basic types used to support the ASP.NET hosting environment. The same namespace contains the VirtualPathProvider base class. The HostingEnvironment class contains several static methods. Some are useful when dealing with paths and path resolving issues.

#### The HostingEnvironment Class

Let’s take a look at the path handling capabilities provided by ASP.NET before we start writing a path provider. The following table shows a selection of methods in the class, which explicitly support path operations.

Table 8-4. Members of the HostingEnvironment class pertaining to path operations

Method or Property Description

MapPath Maps a virtual path to its physical counterpart on the server

RegisterVirtualPathProvider Registers a virtual path provider

ApplicationPhysicalPath Gets the physical path of the application

ApplicationVirtualPath Gets the virtual path of the application

VirtualPathProvider Gets the currently registered path provider

SiteName The name of the site

All properties and methods mentioned above are static and don’t require an instance of the class. The other methods and properties concern application domains, object registering, caching, and impersonation. But that’s beyond the scope of this chapter. If you’d like to learn more, take a look at the following MSDN page:

http://msdn.microsoft.com/en-us/library/system.web.hosting.hostingenvironment.aspx

#### VirtualFile and VirtualDirectory

VirtualFile allows direct access to virtual files and reads the file contents as a stream:

public abstract class VirtualFile : VirtualFileBase

{

protected VirtualFile(string virtualPath);

public override bool IsDirectory { get; }

public abstract Stream Open();

}

IsDirectory always returns false. The Open method must be overridden and return a read-only stream to the virtual resource.

The VirtualDirectory class is similar and manages a directory:

public abstract class VirtualDirectory : VirtualFileBase

{

protected VirtualDirectory(string virtualPath);

public abstract IEnumerable Children { get; }

public abstract IEnumerable Directories { get; }

public abstract IEnumerable Files { get; }

public override bool IsDirectory { get; }

}

Directories lists the subdirectories in this virtual directory. Files returns the files only. Children returns both the files and the folders. IsDirectory always returns true.

#### VirtualPathUtility

This class is defined in the System.Web namespace and provides further useful methods:

Table 8-5. Members of the VirtualPathUtility class

|  |  |
| --- | --- |
| Method or Property | Description |
| AppendTrailingSlash | Adds a slash at the end of the path if there isn’t one |
| Combine | Concatenates a base path and a relative path |
| GetDirectory | Gets the directory path of a virtual path |
| GetExtension | Gets the extension of a file with a virtual path |
| GetFileName | Gets the file name of a file with a virtual path |
| IsAbsolute | Indicates whether a path is absolute |
| IsAppRelative | Indicates whether a virtual path is relative against the application |
| MakeRelative | Transforms an application relative path with stem operator “~” into a relative virtual path |
| RemoveTrailingSlash | Removes a trailing slash from the end; does nothing if there is no slash present |
| ToAbsolute | Converts a virtual path into an absolute one |
| ToAppRelative | Converts an absolute path into a relative one |

All these methods are static, and thus don’t require an instance of the class to be created.

### Creating a Virtual Path Provider to Get Themes from Database

Now that we have all the prerequisites, let’s write a provider. In this example, I want to change the behavior of the App\_Themes folder. A theme is usually a static conglomeration of skins, style sheet files and images. You can specify a theme either in *web.config* or within each web page. Let’s define individual themes in another folder and relate them to pages. This is possible using a custom VirtualPathProvider. Its path is defined in *web.config*, and you can either set it dynamically or manipulate it. The provider shown in this section gives you the basic framework for handling file access and file resolving.

#### Prepare the Project

The themes are still in a folder in your project. The database only defines a reference to the theme’s name. To avoid confusing Visual Studio with a regular theme, all themes are kept in a different folder named *Path\_Themes*. You can use any valid name except for App\_Themes, which is reserved.

#### Create the VirtualPathProvider

The path provider should function as a transparent layer for ASP.NET. Implementing it completely requires more steps than other providers. For the sake of brevity, I’ve created a simplified but functional path provider that consists of three classes:

\* VirtualThemeFile

\* VirtualThemeDirectory

\* VirtualThemePathProvider

This provider lacks caching, which causes it to work slowly, but it’s fully functional. However, the cache implementation is not particularly difficult and can be added easily. I’ve excluded it as it’s out of this section’s scope.

Listing 8-3. Handling virtual files

public class VirtualThemeFile : System.Web.Hosting.VirtualFile

{

private string \_themeAbsolutePath = String.Empty;

private string \_globalAbsolutePath = String.Empty;

private VirtualThemeDirectory \_parent = null;

public VirtualThemeFile(String virtualPath,

string themeAbsolutePath,

string globalAbsolutePath,

VirtualThemeDirectory parent)

: base(virtualPath)

{

\_themeAbsolutePath = themeAbsolutePath;

\_globalAbsolutePath = globalAbsolutePath;

\_parent = parent;

}

public override Stream Open()

{

return File.Open(AbsolutePath, FileMode.Open);

}

public VirtualThemeDirectory Parent

{

get

{

return \_parent;

}

}

private String AbsolutePath

{

get

{

// Get the current set value

String currentSet = ThemePathProvider.Current.CurrentSet;

if ((!String.IsNullOrEmpty(\_themeAbsolutePath))

&& (Parent.FileIsIncluded(Name, currentSet, true)))

return \_themeAbsolutePath;

else if ((!String.IsNullOrEmpty(\_globalAbsolutePath))

&& (Parent.FileIsIncluded(Name, currentSet, false)))

return \_globalAbsolutePath;

return String.Empty;

}

}

internal Boolean ExistsInThemeDirectory

{

get

{

return (!String.IsNullOrEmpty(\_themeAbsolutePath));

}

}

internal Boolean ExistsInGlobalDirectory

{

get

{

return (!String.IsNullOrEmpty(\_globalAbsolutePath));

}

}

}

The VirtualThemeFile class holds a reference to a specific file from the *Path\_Themes* folder. This is usually a skin file, a style sheet, or a resource (such as an image) used with a theme. The file’s complete location is stored in the AbsolutePath property and used in the Open method. (The Open method is called to read the file’s contents.) The Parent property points to the containing directory.

The main focus of the implementation is the directory handling. Hence, the VirtualThemeDirectory class contains much more code.

Listing 8-4. Handling virtual directories

public class VirtualThemeDirectory : VirtualDirectory

{

struct ItemSearchInfo

{

public String Name;

public String VirtualPath;

public String ThemeAbsolutePath;

public String GlobalAbsolutePath;

}

private VirtualThemeDirectory \_parent = null;

private String \_themeAbsolutePath = String.Empty;

private String \_globalAbsolutePath = String.Empty;

private Dictionary<String, VirtualThemeDirectory> \_directories = null;

private Dictionary<String, VirtualThemeFile> \_files = null;

private Dictionary<String, VirtualFileBase> \_children = null;

public VirtualThemeDirectory(String virtualPath)

: this(virtualPath, String.Empty, String.Empty, null)

{

}

public VirtualThemeDirectory(String virtualPath,

String themeAbsolutePath,

String globalAbsolutePath)

: this(virtualPath, themeAbsolutePath, globalAbsolutePath, null)

{

}

public VirtualThemeDirectory(String virtualPath,

String themeAbsolutePath,

String globalAbsolutePath,

VirtualThemeDirectory parent)

: base(virtualPath)

{

if (String.IsNullOrEmpty(themeAbsolutePath))

{

String sThemeRelativePath = ⮰

ThemePathProvider.Current.ConvertToThemeRelativePath(virtualPath);

themeAbsolutePath = ⮰

HttpContext.Current.Server.MapPath(sThemeRelativePath);

}

if (!Directory.Exists(themeAbsolutePath))

{

themeAbsolutePath = String.Empty;

}

if (String.IsNullOrEmpty(globalAbsolutePath))

{

String sGlobalRelativePath = ⮰

ThemePathProvider.Current.ConvertToGlobalRelativePath(VirtualPath);

globalAbsolutePath = ⮰

HttpContext.Current.Server.MapPath(sGlobalRelativePath);

}

if (!Directory.Exists(globalAbsolutePath))

globalAbsolutePath = String.Empty;

\_themeAbsolutePath = themeAbsolutePath;

\_globalAbsolutePath = globalAbsolutePath;

\_parent = parent;

// Create the collections to hold the virtual items

\_files = new Dictionary<string, VirtualThemeFile>();

\_directories = new Dictionary<string, VirtualThemeDirectory>();

\_children = new Dictionary<string, VirtualFileBase>();

FindFiles();

FindSubDirectories();

FindChildren();

}

private void FindFiles()

{

Dictionary<String, ItemSearchInfo> fileList = ⮰

new Dictionary<string, ItemSearchInfo>();

if (Directory.Exists(ThemeAbsolutePath))

{

var files = from f in Directory.GetFiles(ThemeAbsolutePath)

select new ItemSearchInfo

{

Name = Path.GetFileName(f),

VirtualPath = VirtualPathUtility.Combine(VirtualPath, ⮰

Path.GetFileName(f)),

ThemeAbsolutePath = f

};

foreach (ItemSearchInfo fileInfo in files)

{

fileList.Add(fileInfo.Name, fileInfo);

}

}

if (Directory.Exists(GlobalAbsolutePath))

{

var files = from f in Directory.GetFiles(GlobalAbsolutePath)

select new ItemSearchInfo

{

Name = Path.GetFileName(f),

VirtualPath = VirtualPathUtility.Combine(VirtualPath, ⮰

Path.GetFileName(f)),

GlobalAbsolutePath = f

};

foreach (ItemSearchInfo fileInfo in files)

{

if (fileList.ContainsKey(fileInfo.Name))

{

ItemSearchInfo themeFileInfo = fileList[fileInfo.Name];

fileList.Remove(themeFileInfo.Name);

fileList.Add(themeFileInfo.Name, themeFileInfo);

}

else

{

fileList.Add(fileInfo.Name, fileInfo);

}

}

}

// Loop through each file found

foreach (ItemSearchInfo fileInfo in fileList.Values)

{

// Add each file to the files dictionary using the

// information stored for the file

\_files.Add(fileInfo.Name, new VirtualThemeFile(fileInfo.VirtualPath, ⮰

fileInfo.ThemeAbsolutePath, ⮰

fileInfo.GlobalAbsolutePath, ⮰

this));

}

}

private void FindSubDirectories()

{

Dictionary<String, ItemSearchInfo> directoryList = ⮰

new Dictionary<string, ItemSearchInfo>();

Func<string, string, string> MakePath = delegate(string b, string v)

{

return VirtualPathUtility.AppendTrailingSlash(⮰

VirtualPathUtility.Combine(b, v));

};

if (Directory.Exists(ThemeAbsolutePath))

{

var themeDirectories = ⮰

from t in Directory.GetDirectories(ThemeAbsolutePath)

select new ItemSearchInfo

{

Name = Path.GetFileName(t),

VirtualPath = MakePath(VirtualPath, ⮰

Path.GetFileName(t)),

ThemeAbsolutePath = t

};

foreach (ItemSearchInfo directoryInfo in themeDirectories)

{

directoryList.Add(directoryInfo.Name, directoryInfo);

}

}

if (Directory.Exists(GlobalAbsolutePath))

{

var themeDirectories = ⮰

from t in Directory.GetDirectories(GlobalAbsolutePath)

select new ItemSearchInfo

{

Name = Path.GetFileName(t),

VirtualPath = MakePath(VirtualPath, ⮰

Path.GetFileName(t)),

GlobalAbsolutePath = t

};

foreach (ItemSearchInfo directoryInfo in themeDirectories)

{

if (directoryList.ContainsKey(directoryInfo.Name))

{

ItemSearchInfo themeDirectoryInfo = ⮰

directoryList[directoryInfo.Name];

directoryList.Remove(themeDirectoryInfo.Name);

directoryList.Add(themeDirectoryInfo.Name, themeDirectoryInfo);

}

else

{

directoryList.Add(directoryInfo.Name, directoryInfo);

}

}

}

foreach (ItemSearchInfo directoryInfo in directoryList.Values)

{

VirtualThemeDirectory directory = new VirtualThemeDirectory(⮰

directoryInfo.VirtualPath, ⮰

directoryInfo.ThemeAbsolutePath, ⮰

directoryInfo.GlobalAbsolutePath, ⮰

this);

\_directories.Add(directory.Name, directory);

}

}

private void FindChildren()

{

foreach (VirtualThemeDirectory directory in Directories)

{

\_children.Add(directory.Name, directory);

}

foreach (VirtualThemeFile file in Files)

{

\_children.Add(file.Name, file);

}

}

public Boolean GetFileIsIncluded(String fileName)

{

String currentSet = ThemePathProvider.Current.CurrentSet;

if (FileIsIncluded(fileName, currentSet, true))

{

return true;

}

else

{

return FileIsIncluded(fileName, currentSet, false);

}

}

public Boolean GetDirectoryIsIncluded(String directoryName)

{

String currentSet = ThemePathProvider.Current.CurrentSet;

if (DirectoryIsIncluded(directoryName, currentSet, true))

{

return true;

}

else

{

return DirectoryIsIncluded(directoryName, currentSet, false);

}

}

internal Boolean FileIsIncluded(string fileName, string currentSet, ⮰

boolean checkAgainstTheme)

{

if (!\_files.ContainsKey(fileName))

return false;

VirtualThemeFile file = \_files[fileName];

if ((checkAgainstTheme)

&& (!file.ExistsInThemeDirectory)

{

return false;

}

else if ((!checkAgainstTheme)

&& (!file.ExistsInGlobalDirectory))

{

return false;

}

if (String.IsNullOrEmpty(currentSet))

return true;

String fileExtension = Path.GetExtension(fileName);

if (fileExtension.ToUpper() == ".SKIN"

||

fileExtension.ToUpper() == ".CSS"

||

fileExtension.ToUpper() == ".JPG")

{

return true;

}

return false;

}

internal Boolean DirectoryIsIncluded(string directoryName, ⮰

string currentSet, ⮰

boolean checkAgainstTheme)

{

if (!\_directories.ContainsKey(directoryName))

return false;

VirtualThemeDirectory directory = \_directories[directoryName];

if ((checkAgainstTheme)

&& (!directory.ExistsInThemeDirectory))

{

return false;

}

else if ((!checkAgainstTheme)

&& (!directory.ExistsInGlobalDirectory))

{

return false;

}

return true;

}

internal VirtualThemeFile GetFile(String fileName)

{

return \_files[fileName];

}

internal VirtualThemeDirectory GetDirectory(String virtualDir)

{

if (\_directories.Count == 0)

return null;

if (virtualDir.StartsWith(VirtualPath, ⮰

!StringComparison.InvariantCultureIgnoreCase))

return null;

String relativeVirtualPath = virtualDir.Substring(VirtualPath.Length);

String directoryName = relativeVirtualPath.Substring(0, ⮰

relativeVirtualPath.IndexOf("/"));

VirtualThemeDirectory childDirectory = \_directories[directoryName];

if (childDirectory.VirtualPath == virtualDir)

return childDirectory;

else

return childDirectory.GetDirectory(virtualDir);

}

public VirtualThemeDirectory Parent

{

get

{

return \_parent;

}

}

public Boolean Exists

{

get

{

return ((Directory.Exists(\_themeAbsolutePath))

|| (Directory.Exists(\_globalAbsolutePath)));

}

}

private Boolean ExistsInThemeDirectory

{

get

{

return (!String.IsNullOrEmpty(\_themeAbsolutePath));

}

}

private Boolean ExistsInGlobalDirectory

{

get

{

return (String.IsNullOrEmpty(\_globalAbsolutePath));

}

}

public override IEnumerable Directories

{

get

{

return \_directories.Value;

}

}

public override IEnumerable Files

{

get

{

return \_files.Value;

}

}

public override IEnumerable Children

{

get

{

return \_children.Value;

}

}

private String ThemeAbsolutePath

{

get

{

return \_themeAbsolutePath;

}

}

private String GlobalAbsolutePath

{

get

{

return \_globalAbsolutePath;

}

}

}

The main purpose of this class is to build a copy of the directory and file structure found in the physical theme folder (*Path\_Themes*) and to store references to the files. When the provider retrieves the files, it accesses this virtual structure.

Three internal methods are called in the constructor: FindFiles, FindSubDirectories and FindChildren. The Findfiles method reads files in the current base folder. FindSubDirectories loads directories recursively. In each directory located, the FindFiles method is called to load the files for that folder. The FindChildren method creates a single combined list of both directories and files.

The provider calls the GetDirectory method to retrieve a specific directory. A simple filter, FileIsIncluded, limits the allowed file types. This is exclusive to the handling of themes; other VirtualPathProvider implementations might require different types. The example ignores all files except for \*.skin, \*.css, and \*.jpg, but you can extend the list to support any file type used in your themes.

The GetFile method allows access to a file once it has been located in a directory. The resolving of files and their paths occurs in the compiler module. A simple file list handles all files in all directories. The key contains the full path, so files with the same filename but located in different folders still have a unique full name.

Listing 8-5. Implementation of a custom VirtualPathProvider

public sealed class ThemePathProvider : VirtualPathProvider

{

private static ThemePathProvider \_currentProvider = null;

private const string ASPNetThemeBasePath = "/App\_Themes/";

private string \_themeRelativePath = String.Empty;

private string \_currentThemeSet = String.Empty;

private string \_globalThemeName = String.Empty;

private ThemePathProvider()

{

\_themeRelativePath = ⮰

WebConfigurationManager.AppSettings["CustomThemeBasePath"];

}

public override System.Web.Caching.CacheDependency GetCacheDependency(⮰

string virtualPath, ⮰

System.Collections.IEnumerable virtualPathDependencies, ⮰

DateTime utcStart)

{

return null;

}

public override bool DirectoryExists(string virtualDir)

{

if (virtualDir.IndexOf(ASPNetThemeBasePath) == -1)

return base.DirectoryExists(virtualDir);

VirtualThemeDirectory directory = GetDirectory(virtualDir) as ⮰

VirtualThemeDirectory;

return directory.Exists;

}

public override bool FileExists(string virtualPath)

{

if (virtualPath.IndexOf(ASPNetThemeBasePath) == -1)

return base.FileExists(virtualPath);

string fileName = System.Web.VirtualPathUtility.GetFileName(virtualPath);

string virtualDirectoryPath = ⮰

System.Web.VirtualPathUtility.GetDirectory(virtualPath);

VirtualThemeDirectory directory = ⮰

GetDirectory(virtualDirectoryPath) as VirtualThemeDirectory;

return directory.GetFileIsIncluded(fileName);

}

public override VirtualDirectory GetDirectory(string virtualDir)

{

if (virtualDir.IndexOf(ASPNetThemeBasePath) == -1)

return base.GetDirectory(virtualDir);

if (IsThemeDirectoryVirtualPath(virtualDir))

{

return new VirtualThemeDirectory(virtualDir);

}

else

{

String themeVirtualPath = GetThemeDirectoryVirtualPath(virtualDir);

VirtualThemeDirectory directory = new VirtualThemeDirectory(virtualDir);

return directory.GetDirectory(virtualDir);

}

}

public override VirtualFile GetFile(string virtualPath)

{

if (virtualPath.IndexOf(ASPNetThemeBasePath) == -1)

return base.GetFile(virtualPath);

String virtualDirectoryPath = ⮰

System.Web.VirtualPathUtility.GetDirectory(virtualPath);

VirtualThemeDirectory directory = ⮰

GetDirectory(virtualDirectoryPath) as VirtualThemeDirectory;

String fileName = System.Web.VirtualPathUtility.GetFileName(virtualPath);

return directory.GetFile(fileName);

}

private StringCollection GetDependentDirectories(

String parentDirectoryPath,

StringCollection dependentPaths)

{

String[] directories = Directory.GetDirectories(parentDirectoryPath);

for (int loopIndex = 0; loopIndex < directories.Length; loopIndex++)

{

dependentPaths.Add(directories[loopIndex]);

GetDependentDirectories(directories[loopIndex], dependentPaths);

}

return dependentPaths;

}

private Boolean IsThemeDirectoryVirtualPath(String virtualPath)

{

String parentVirtualPath = ⮰

System.Web.VirtualPathUtility.GetDirectory(virtualPath);

return parentVirtualPath.EndsWith(ASPNetThemeBasePath, ⮰

StringComparison.InvariantCultureIgnoreCase);

}

private String GetThemeDirectoryVirtualPath(String virtualPath)

{

String parentVirtualPath = ⮰

System.Web.VirtualPathUtility.GetDirectory(virtualPath);

while (!IsThemeDirectoryVirtualPath(parentVirtualPath))

{

parentVirtualPath = ⮰

System.Web.VirtualPathUtility.GetDirectory(parentVirtualPath);

}

return parentVirtualPath;

}

internal String ConvertToThemeRelativePath(String relativePath)

{

return ConvertToThemeNameRelativePath(relativePath, false);

}

internal String ConvertToGlobalRelativePath(String relativePath)

{

return ConvertToThemeNameRelativePath(relativePath, true);

}

private String ConvertToThemeNameRelativePath(String relativePath, ⮰

Boolean replaceThemeNameWithGlobal)

{

String themeNameRelativePath = String.Empty;

if ((!relativePath.StartsWith(ASPNetThemeBasePath))

&& (!\_themeRelativePath.StartsWith("/")))

{

themeNameRelativePath = relativePath.Substring(0, ⮰

relativePath.IndexOf(ASPNetThemeBasePath));

}

if ((!themeNameRelativePath.EndsWith("/"))

&& (!\_themeRelativePath.StartsWith("/")))

{

themeNameRelativePath = ⮰

System.Web.VirtualPathUtility. ⮰

AppendTrailingSlash(themeNameRelativePath);

}

themeNameRelativePath += \_themeRelativePath;

String remainderPath = relativePath.Substring(⮰

relativePath.IndexOf(ASPNetThemeBasePath) + ⮰

ASPNetThemeBasePath.Length);

if (replaceThemeNameWithGlobal)

{

remainderPath = remainderPath.Substring(remainderPath.IndexOf("/"));

}

themeNameRelativePath += remainderPath;

return themeNameRelativePath;

}

public static ThemePathProvider Current

{

get

{

if (\_currentProvider != null)

return \_currentProvider;

\_currentProvider = new ThemePathProvider();

return \_currentProvider;

}

}

public String CurrentSet

{

get

{

return \_currentThemeSet;

}

set

{

\_currentThemeSet = value;

}

}

}

The GetCacheDependency returns null in order to suppress any internal caching. You must either return null or fully implement the caching. Otherwise, the compiler will try to resolve the default path (*App\_Themes*) from the cache and this will fail.

Two methods are called when the compiler tries to resolve an internal path: GetDirectory and GetFile. If a file is referenced in a page that points to a theme, the compiler asks the provider to retrieve the same. Using the GetFile method, we first check whether the theme’s folder is being used. This is the folder for which the provider in the example is responsible. From the file’s name, the containing directory is built. The directory is used to obtain the VirtualThemeDirectory implementation, which in turn returns the file held there.

At the core of this implementation is a path manipulation algorithm. It could be far more sophisticated than the simple code in the example.

Before a folder or file is retrieved, the DirectoryExists and FileExists methods are called. This allows the provider to programmatically “hide” parts of the structure or to check for the physical presence of the requested resource.

The basic approach is always the same. The calling instance “asks” the provider for a physical file. The provider returns a VirtualFile instance, which allows direct access to the file’s contents via a Stream object. The remarkable thing is that the file might not necessarily exist anywhere on the file system. If, for example, the stream is obtained from a database call, the VirtualPathProvider is simulating a physical file system. The “Exists” methods could always return true in order to simulate a system where the page developers can use any value, and the system returns a set of globally predefined resources.

#### Configuring the Path Provider

The provider requires two final configuration steps. Firstly, it must have a setting in *web.config* to show that it’s configurable. Secondly, the provider must be registered, which follows the pattern previously explained. To demonstrate, I’ve used the Application\_Start event handler defined in the *global.asax* file:

Listing 8-6. Configuration settings in web.config

<appSettings>

<add key="CustomThemeBasePath" value="/Path\_Themes/"/>

</appSettings>

In the example, the key is used to configure the physical path.

Listing 8-7. Registering the path provider using the Application\_Start event

protected void Application\_Start(object sender, EventArgs e)

{

HostingEnvironment.RegisterVirtualPathProvider(ThemePathProvider.Current);

}

The registration uses a singleton instance of the provider instead of the constructor. This results in slightly shorter code. A singleton instance works well because the provider works globally and exists only once in the application.

### Limitations of the VirtualPathProvider Approach

The VirtualPathProvider is deeply integrated in the ASP.NET engine. Although it’s very powerful, certain tasks can be problematic. In this section, I explain two typical issues found when working with custom path providers.

#### Working with LoadControl

The VirtualPathProvider allows you to override the way in which tilde-based paths are resolved in page directives:

<%@ Register TagPrefix="Test" TagName="MyControl" Src="~/userctrl.ascx" %>

However, the VirtualPathProvider cannot affect the way in which tilde-based paths are resolved for dynamically loaded user controls:

MyPlaceHolder.Controls.Add(LoadControl("~/userctrl.ascx"));

For the LoadControl method to resolve tilde-based paths in the same way as Register directives, the page code-behind needs to override this method:

public new Control LoadControl(string relativePath)

{

string newPath = relativePath;

string site = this.Request.QueryString["site"];

if (!String.IsNullOrEmpty(site))

{

newPath = VirtualPathUtility.ToAppRelative(newPath);

newPath = newPath.Substring(1);

newPath = "~/MySites/" + site + newPath;

}

return base.LoadControl(newPath);

}

This workaround will function as long as LoadControl is called directly on your derived class, since it’s not virtual. A similar override in a UserControl base class might be also required, since LoadControl lives on TemplateControl.

#### Working with Precompiled Sites

Unfortunately, VirtualPathProvider is not supported in precompiled web sites at the moment. As shown in the code exhibit decompiled using Reflector at the beginning of the VirtualPathProvider section, the precompiled state is explicitly detected. Consequently, MSDN notes that, “if a Web site is precompiled for deployment, content provided by a VirtualPathProvider instance is not compiled, and no VirtualPathProvider instances are used by the precompiled site.”

One way to avoid deploying source code is to use the Web Application Project model, in which all classes and code-behind files are compiled. Although pre-compilation will give our site a performance boost, the difference in speed will only be noticeable during the first request to each folder. A more significant benefit is the new deployment option made available by pre-compilation—the option to deploy a site without copying any of the original source code to the server. This includes the code and markup in *aspx*, *ascx*, and *master* files.

There is also a hack available to force the path provider to work with pre-compiled pages. This is not intended for production code, but if you’re under pressure to solve a similar issue, this tip might help. The solution is DynamicMethod. Just call a Microsoft internal method to register VirtualPathProvider directly. This method is internal and therefore only reflection can obtain it. The definition in HostingEnvironment looks like this:

internal static void RegisterVirtualPathProviderInternal( ⮰

VirtualPathProvider virtualPathProvider)

{

VirtualPathProvider previous = \_theHostingEnvironment.\_virtualPathProvider;

\_theHostingEnvironment.\_virtualPathProvider = virtualPathProvider;

virtualPathProvider.Initialize(previous);

}

This is, again, a simple hack that may fail in future versions due to changes to the internal code by Microsoft.

## Summary

In this chapter, we looked at site management. Providing an existing structure to the navigation is the primary task of the SiteMapProvider. This provider allows the management of larger sites based on any data store you decide to implement. It’s transparent to the controls used to create navigational elements.

The VirtualPathProvider gives you more control over internal (physical) and external (virtual) paths for directories and files. Replacing the provider allows you to “fake” the paths and set them to virtual ones, which makes it easy to manage complex file structures. The VirtualPathProvider is not as straightforward as the other ones, but it makes sense to use it if your embedded features, such as themes that can be extended or customized, are based on special folders.