CHAPTER 3

Programming with Features and Solutions

IN THIS CHAPTER

Features and Solutions are two new aspects of Windows SharePoint Services (WSS) and Microsoft Office SharePoint Server (MOSS) that make it dramatically easier to customize sites and site templates. This chapter focuses on illustrating how to work with site Features, Feature definitions, and Solutions using the SharePoint object model.

For an introduction to the concepts behind Features and Solutions as well as information on how to build your own Features and Solutions, refer to the *Microsoft SharePoint 2007 Unleashed* book (ISBN: 0672329476) that contains the best administrator's reference available for SharePoint.

Overview of Features and Solutions

Features provide the ability for sites to reuse functionality that exists in other sites without requiring the tedious task of copying and pasting complex Extensible Markup Language (XML) code from one template to another.

By installing Feature definitions at the farm level, Features can then be activated at any site within the farm. This allows reusable pieces of functionality to be created and deployed without modifying site templates, and it allows site templates to be far less complex than they used to be by referring to Features instead of directly embedding mountains of complex XML.

Using Features, you can do everything from adding a link to the Site Settings page to creating a complete, fully functioning Project Management suite that can be added to any SharePoint site.

Solutions allow you to package Features in a cabinet (`.cab`) file and define important metadata about those Features. After a Solution is installed on a server in the farm, you can then use SharePoint's Solution management features to automate the deployment of that Solution to other sites within the farm. This kind of hands-off deployment of reusable pieces of SharePoint functionality has never been possible in SharePoint before and developers are sure to love how easy it is to deploy new functionality in this version of SharePoint.

Programming with Features

SharePoint includes a robust object model for working with Features that allows developers to enumerate installed and activated Features, to turn Features on and off, and to control the installation or removal of Features.

The object model for Features includes the following key classes:

- `SPFeatureCollection/SPFeature`—Refers to a Feature state at a given site hierarchy level. The presence of an `SPFeature` instance within a property of type `SPFeatureCollection` indicates that the Feature is *active* at that level.
- `SPFeaturePropertyCollection/SPFeatureProperty`—Represents a single property on a Feature or a collection of those properties.
- `SPFeatureScope`—Represents an enumeration of the possible scopes in which Features can be activated. Possible values are: `Farm`, `WebApplication`, `Site`, and `Web`.
- `SPFeatureDefinition`—Represents the basic definition of a Feature, including its name, scope, ID, and version. You can also store and retrieve properties of a Feature. Note that Feature properties apply globally to a single Feature definition, not to instances of Features activated throughout the farm.
- `SPFeatureDependency`—Represents a Feature upon which another Feature depends.
- `SPElementDefinition`—Represents a single element that will be provisioned when the Feature is activated.

Feature collections can be accessed from the following properties on their respective classes:

- `Features` on `Microsoft.SharePoint.Administration.SPWebApplication`
- `Features` on `Microsoft.SharePoint.Administration.SPWebService`
- `FeatureDefinitions` on `Microsoft.SharePoint.Administration.SPFarm`
- `Features` on `Microsoft.SharePoint.SPSite`
- `Features` on `Microsoft.SharePoint.SPWeb`
- `ActivationDependencies` on `Microsoft.SharePoint.Administration.SPFeatureDefinition`

The next few sections of this chapter provide many examples of programming with the Features portion of the SharePoint application programming interface (API).

Enumerating Features and Feature Definitions

It is important to recognize the difference between a Feature and a Feature definition. A Feature definition, as far as the object model is concerned, is an abstraction around the Feature manifest contained in a Feature directory in the Features directory. Feature definitions are installed at the farm (or server, if there is no farm) level.

A Feature is an instance of a Feature definition. Features can be activated or deactivated, and they exist at the various levels of scope such as the site or web level.

To enumerate the list of Feature definitions that are currently installed within a farm (which includes single-server farms and standalone servers, which are another form of single-server farms), you need to use an instance of the `SPFarm` class and access the `FeatureDefinitions` property. To enumerate the list of *active* Features on a given site, you need to enumerate the `Features` property on the appropriate `SPWeb` or `SPSite` class instance. You might be tempted to iterate through the collection contained in the `Features` property and look for something like an `Active` property. However, *the only `SPFeature` instances that appear in the `Features` property are those features that are* active *in the current scope*.

Creating an instance of the `SPFarm` class might seem a little tricky at first. Rather than obtaining it through a context provided by the Web Part manager or from a site uniform resource locator (URL), you need to pass a connection string that points to the farm's configuration database to the constructor. The connection string should look familiar to anyone with ADO.NET experience connecting to SQL server, because it is just a SQL server connection string.

The code in Listing 3.1 shows how to create an instance of the `SPFarm` class, create an instance of the `SPSite` class, and use the two of those to enumerate the list of installed Feature definitions and determine which of those definitions are active on the given site. This code is for a Windows Forms application that adds the name and enabled status of each Feature definition to a `ListView` control. If you plan to copy this code and test it, be sure to add a reference to the `Microsoft.SharePoint.dll` Assembly.

LISTING 3.1 Enumerating Feature Definitions and Features

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using Microsoft.SharePoint;
using Microsoft.SharePoint.Administration;
using System.Windows.Forms;
```

LISTING 3.1 Continued

```
namespace FeatureEnumerator
{
public partial class Form1 : Form
{
  private SPSite _rootCollection;

public Form1()
{
    InitializeComponent();
}

private string GetFeatureEnabled(SPFeatureDefinition featureDefinition)
{
    foreach (SPFeature feature in _rootCollection.Features)
    {
        if (feature.Definition.Id == featureDefinition.Id)
            return "Yes";
    }
    return "No";
}

private void button1_Click(object sender, EventArgs e)
{
    featureList.Enabled = true;
    featureList.Items.Clear();
    string dbConn = @"server=localhost\OfficeServers;initial
➥catalog=SharePoint_Config_66140120-a9bf-4191-86b6-
➥ec21810ca019;IntegratedSecurity=SSPI;";

    _rootCollection = new SPSite(siteUrl.Text);
    SPFarm farm = SPFarm.Open(dbConn);
    statusLabel.Text = "Site Feature Status (" +
      farm.FeatureDefinitions.Count.ToString() +
        " Feature Definitions Installed)";

    foreach (SPFeatureDefinition featureDefinition in farm.FeatureDefinitions)
    {
        ListViewItem lvi = new ListViewItem(featureDefinition.DisplayName);
        if (featureDefinition.Hidden)
            lvi.ForeColor = Color.Gray;
        lvi.Tag = featureDefinition.Id;
        lvi.SubItems.Add(GetFeatureEnabled(featureDefinition));
        featureList.Items.Add(lvi);
```

LISTING 3.1 Continued

```
    }

}


}
}
```

This Windows Forms application is shown in Figure 3.1.

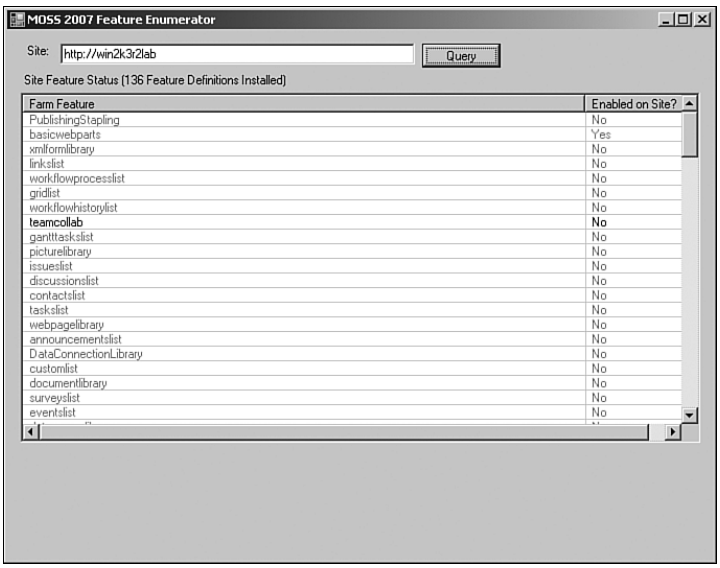

FIGURE 3.1 Feature Enumerator Windows Forms application.

The true power of Features should become immediately obvious as soon as you start using this tool to examine the list of active and inactive features on various sites. Most of the key functionality provided by SharePoint 2007 is implemented using Features. Previous versions of SharePoint did not provide anywhere near the amount of flexibility, customization, and enhancement capability that Features provide.

Activating and Deactivating Features

It follows that if the `SPFeatureCollection` instance represented by the `Features` property of a site or web object contains the list of active Features, then adding and removing to and from that collection should activate and deactivate Features. In fact, that is exactly how it works with one small exception: You cannot add or remove `SPFeature` or `SPFeatureDefinition` instances; you can only add or remove globally unique identifiers (GUIDs).

Before you start writing code that activates and deactivates Features, you should definitely create a test site that you don't mind destroying. Deactivating hidden features can have drastic consequences on the ability of your site to function properly. However, knowing what those consequences are can certainly provide deeper understanding of the Features system and time spent tinkering with Features is definitely time well spent.

To deactivate a Feature, simply remove it from the `Features` collection of the current site or web object:

```
currentSite.Features.Remove(new Guid("… guid of feature to remove …"));
```

Conversely, activating an installed Feature on a site or web object is accomplished simply by adding the Feature's GUID to the `Features` collection:

```
currentSite.Features.Add(new Guid(".. guid .."));
```

Keep in mind that an exception will be thrown if you attempt to activate a Feature that cannot be activated at the current scope.

Using Feature Properties

Every Feature installed in SharePoint maintains its own property bag.

NOTE

For those of you who don't know, a **property bag** is a special kind of name-value collection that first reared its head back in the days of Site Server and Commerce Server. For most developers, it remains a point of contention as to whether the head of a property bag is ugly or not.

Features and Feature definitions both have properties exposed through the `Properties` property, which is of type `SPFeaturePropertyCollection`—a collection of `SPFeatureProperty` objects. Regardless of whether you access the `Properties` property of a site Feature instance or of a Feature definition, the result will be the same. That is, you can think of Feature properties as static, global data that belongs to the definition itself but can also be accessed from an activated Feature `SPFeature` object.

The following code snippet illustrates how to enumerate through the properties associated with a given Feature (or Feature definition):

```
foreach (SPFeatureProperty property in myFeature.Properties)
{
    Console.WriteLine("{0} : {1}", property.Name, property.Value);
}
```

You don't have to worry about typecasting or conversion because the only value type acceptable to a property bag is `System.String`. Adding a new property to an existing Feature or Feature definition is quite simple:

```
SPFeatureProperty prop = new SPFeatureProperty("myProp", "myValue");
myFeature.Properties.Add(prop);
```

That's all there is to it—you don't need to explicitly call any methods to commit that change to SharePoint. It is just as easy to remove a property from a Feature:

```
SPFeatureProperty prop = myFeature.Properties[0];
myFeature.Properties.Remove(prop);
```

A few of the Features that are installed with SharePoint make use of properties such as the workflow features. You can use properties to do tremendously powerful things. You can think of the property bag assigned to each Feature as either a place for global configuration settings or for global state management for the Feature, or both.

Installing and Removing Feature Definitions

To install a Feature definition, you need to make use of one of the overloads of the `Add` method on the `SPFeatureDefinitionCollection` class:

- `Add(SPFeatureDefinition)`—Adds a new Feature definition based on the properties of the `SPFeatureDefinition` instance passed to the method.
- `Add(relative path to manifest, GUID of solution)`—Adds a new Feature definition that resides in the location indicated by the first parameter with the given solution ID.
- `Add(relative path to manifest, GUID of solution, force)`—Adds a new Feature definition that resides in the location indicated by the first parameter with the given solution ID and forces a reinstallation of the Feature.

The following code is a simple example of installing a new Feature in a farm (remember that to get an instance of the `SPFarm` class, you need the connection string of the farm's configuration database):

```
SPFeatureDefinitionCollection installedFeatures = theFarm.FeatureDefinitions;
installedFeatures.Add("newfeature", new Guid("- feature GUID -"));
```

Conversely, to uninstall a Feature, simply remove it from the collection. You can remove the Feature based on the relative path to the Feature manifest or the Feature's GUID:

```
installedFeatures.Remove(new Guid("- feature GUID -"));
```

Programming with Solutions

Solutions are the means by which collections of Features are installed on SharePoint farms and deployed to servers within those farms. After being installed on a farm, Solutions can be deployed to any server within that farm automatically using the web-based Solution management interface.

Installing and Removing Solutions

If you don't want to use the `stsadm.exe` command-line tool, or you simply can't for some reason, you can still programmatically manipulate the list of Solutions available for deployment within a farm.

As shown earlier in the chapter, any time you need to work with the `SPFarm` class, you need to pass a configuration database connection string to the constructor:

```
string dbConn = @"server=localhost\OfficeServers;initial
➥catalog=SharePoint_Config_66140120-a9bf-4191-86b6-
➥ec21810ca019;IntegratedSecurity=SSPI;";

_rootCollection = new SPSite(siteUrl.Text);
SPFarm farm = SPFarm.Open(dbConn);
```

After you have a reference to the farm, you can then access the list of installed Solutions in the farm with the `Solutions` property. To install a Solution, just add it to the `Solutions` collection. To uninstall a Solution, simply remove it from the collection.

The `SPSolution` class cannot be instantiated directly with a constructor. Instead, you have two options when adding to the `Solutions` collection. You can pass the Solution filename as a parameter, or you can pass the Solution filename and a locale identifier (a `UInt32`, such as `1033`):

```
farm.Solutions.Add("myapplication.cab");
farm.Solutions.Add("myapplication.cab", 1033);
```

When removing a Solution, you can either pass the name of the Solution as a parameter, or you can pass the GUID of the Solution:

```
farm.Solutions.Remove(new Guid("..."));
```

The `Add` and `Remove` methods of the `Solutions` collection correspond directly to the functionality provided by the `stsadm.exe` commands `"-o addsolution"` and `"-o deletesolution"`.

Enumerating Solutions

When working with Solutions, you might want to take a look at the list of Solutions currently installed in the farm. From this list, you can then control the deployment status (shown in the next section) of the Solution or inspect the various properties of the Solutions. The following code is a simple illustration of how to examine the list of installed Solutions in a farm:

```
Console.WriteLine("Solution:\tCAS Policy\tGAC Assembly\tWeb Resource\n");
foreach (SPSolution solution in farm.Solutions)
{
    Console.WriteLine("{0}:\t{1}\t{2}\t{3}",
```

```
        solution.DisplayName,
        solution.ContainsCasPolicy,
        solution.ContainsGlobalAssembly,
        solution.ContainsWebApplicationResource);
    foreach (SPServer deployedServer in solution.DeployedServers)
    {
        Console.WriteLine("\t\tDeployed to {0}", deployedServer.DisplayName);
    }
}
```

Table 3.1 describes many of the properties of the `SPSolution` class.

TABLE 3.1 `SPSolution` Properties

Property	Description
`Added`	Indicates whether a language-neutral Solution package has been added to the Solution
`ContainsCasPolicy`	Indicates whether the Solution contains a Code Access Security policy
`ContainsGlobalAssembly`	Indicates whether the Solution installs Assemblies into the GAC
`ContainsWebApplicationResource`	Indicates whether the Solution contains any application-specific resources
`Deployed`	Indicates whether the Solution has been deployed to one or more locations within the farm
`DeployedServers`	Indicates the list of servers to which the Solution has been deployed
`DeployedWebApplications`	Indicates the list of web applications to which the Solution has been deployed
`DeploymentState`	Indicates the current state of deployment for the Solution
`Id`	Indicates the GUID of the Solution
`IsWebPartPackage`	Indicates whether the Solution is a Web Part package
`LastOperationDetails`	Represents the details of the last operation performed while deploying the Solution
`LastOperationEndTime`	Indicates the time the last operation completed
`LastOperationResult`	Indicates the results of the last operation during deployment
`Name`	Indicates the name of the Solution
`Properties`	Indicates the property bag containing custom properties for the Solution
`SolutionFile`	Indicates the file associated with the Solution
`SolutionId`	Indicates the ID of the Solution as indicated by the Solution's manifest file

Controlling Solution Deployment

Controlling Solution deployment really boils down to two different methods on the `SPSolution` class: `Deploy` and `Retract`. The `Deploy` method deploys a Solution to the given location, whereas the `Retract` method removes the Solution from the given location while still remaining installed within the farm.

The deploy methods take the following arguments:

- `dt`—The date and time when the deployment should take place
- `globalInstallWPPackDlls`—A Boolean indicating whether to install the dynamic link libraries (DLLs) in the GAC (for Web Part packages only)
- `force`—A Boolean indicating whether the Solution can be redeployed

You can also optionally supply a collection of `SPWebApplication` instances to further refine the deployment. The `Retract()` method schedules a job for when the Solution should be retracted and can optionally take a list of web applications from which to retract the solution.

Summary

Features and Solutions are two of the most powerful new additions to SharePoint 2007. As a developer, you will probably be spending a considerable amount of time creating and manipulating Features and Solutions. This chapter included details on how to manipulate Features and Solutions programmatically using the SharePoint object model. At this point, you should not only be able to create your own Features and Solutions, but you should also be able to install, manipulate, and deploy them programmatically. For more information on the administration and maintenance of features unrelated to writing code, consult the SharePoint documentation or the Sams Publishing book, *Microsoft SharePoint 2007 Unleashed* (ISBN: 0672329476).