8

Core DotNetNuke APIs

DotNetNuke provides significant capability straight out of the box. Just install and go. Sometimes, however, you may need to extend the base framework. DotNetNuke provides a variety of integration points: from HTTP modules to providers to custom modules. To fully take advantage of the framework, it is important to understand some of the base services and APIs provided by DotNetNuke.

This chapter examines some of the core services provided by DotNetNuke. You can use these services from within your own code. Because most of the core services are built using the Provider design pattern, it's also possible to swap out the base functionality. If you need your events logged to a custom database or the Windows Event Logs, then just create your own provider.

The second part of this chapter covers several HTTP modules that are installed with DotNetNuke. They provide features like Friendly URLs, Exception Management, and Users Online. Many of the providers installed with DotNetNuke use HTTP modules to hook into the request-processing pipeline. By examining the code used in the core HTTP modules, you can build your own custom extensions that can be used in DotNetNuke as well as other ASP.NET applications.

The final section examines some of the core interfaces that you can implement in your own modules. These interfaces simplify the process of adding common features to your module, whether it is the module menu, searches, importing and exporting, or even custom upgrade logic. By using these interfaces in your modules, you can provide some of the same features you see in the core DotNetNuke modules with very little coding effort.

Event Logging

The Logging Provider in DotNetNuke provides a very configurable and extensible set of logging services. It is designed to handle a wide array of logging needs including exception logging, event auditing, and security logging. As you may have gathered from its name, the Logging Provider uses

the Provider Model design pattern. This allows the default DB Logging Provider to be replaced with another logging mechanism without having to make changes to the core code. This section covers the ways you can use the Logging Provider to log events in custom modules.

Before you dive into the details of how to use the Logging Provider API, it is important to understand some concepts and terminology that will be used in this section:

- **Log classification:** There are two different types of log classifications in the Logging Provider. The first is the *event log* classification. This encapsulates all log entries related to some type of event within DotNetNuke. For example, you can configure the Logging Provider to write a log entry when a login attempt fails. This would be considered an event log entry. The second log classification is the *exception* log classification. You can configure the Logging Provider to log exceptions and stack traces when exceptions are thrown within DotNetNuke. These two classifications are distinct only because they have different needs in terms of what type of information they log.
- **Log type:** A *log type* defines the type of event that creates the log entry. For example, an event log type is `LOGIN_FAILURE`. The Logging Provider can react differently for each log type. You can configure the Logging Provider to enable or disable logging for each of the log types. Depending on the Logging Provider, you can configure it to log each log type to a different file or to send e-mail notifications upon creating new log entries for that log type.
- **Log type configuration:** The Logging Provider is configured via a module that is accessible from the Log Viewer screen (the Edit Log Configuration Module Action). This enables you to configure each log type to be handled differently by the Logging Provider.

The API

The Logging Provider functionality lives in the DotNetNuke.Services.Log.EventLog namespace. In this namespace, you will find the classes that comprise the Logging Provider API. These are described in Table 8-1.

Table 8-1: Logging Provider Classes

Class	Description
EventLogController	Provides the methods necessary to write log entries with the event log classification. It inherits from LogController.
ExceptionLogController	Provides the methods necessary to write log entries with the exception log classification. It inherits from LogController.
LogController	Provides the methods that interact with the Logging Provider—the basic methods for adding, deleting, and getting log entries.
LogDetailInfo	Holds a single key/value pair of information from a log entry.
LoggingProvider	Provides the bridge to the implementation of the Logging Provider.
LogInfo	Container for the information that goes into a log entry.

Class	Description
LogInfoArray	Holds an array of LogInfo objects.
LogProperties	Holds an array of LogDetailInfo objects.
LogTypeConfigInfo	Container for the configuration data relating to how logs of a specific log type are to be handled.
LogTypeInfo	Container for the log type information.
PurgeLogBuffer	Scheduler task that can be executed regularly if Log Buffering is enabled.
SendLogNotifications	Scheduler task that can be executed regularly if any log type is configured to send e-mail notifications.

The Controller Classes

The controller classes, EventLogController and ExceptionLogController, are the two that bring the most functionality to custom modules. Many of the other classes are used in concert with the controllers.

EventLogController

The EventLogController provides the methods necessary to log significant system events. This class is located in the DotNetNuke.Library in the Components/Providers/Logging/EventLogging/EventLogController.vb file. The class also defines the EventLogType enumeration that lists each log type that is handled by the EventLogController. The enumerations are shown in Listing 8-1.

Listing 8-1: EventLogController.EventLogType Enumeration

```
Public Enum EventLogType
            USER_CREATED
            USER_DELETED
            LOGIN_SUPERUSER
            LOGIN_SUCCESS
            LOGIN_FAILURE
            CACHE_REFRESHED
            PASSWORD_SENT_SUCCESS
            PASSWORD_SENT_FAILURE
            LOG_NOTIFICATION_FAILURE
            PORTAL_CREATED
            PORTAL_DELETED
            TAB_CREATED
            TAB_UPDATED
            TAB_DELETED
            TAB_SENT_TO_RECYCLE_BIN
            TAB_RESTORED
            USER_ROLE_CREATED
            USER_ROLE_DELETED
            ROLE_CREATED
            ROLE_UPDATED
```

(continued)

Listing 8-1: *(continued)*

```
            ROLE_DELETED
            MODULE_CREATED
            MODULE_UPDATED
            MODULE_DELETED
            MODULE_SENT_TO_RECYCLE_BIN
            MODULE_RESTORED
            SCHEDULER_EVENT_STARTED
            SCHEDULER_EVENT_PROGRESSING
            SCHEDULER_EVENT_COMPLETED
            APPLICATION_START
            APPLICATION_END
            APPLICATION_SHUTTING_DOWN
            SCHEDULER_STARTED
            SCHEDULER_SHUTTING_DOWN
            SCHEDULER_STOPPED
            ADMIN_ALERT
            HOST_ALERT
        End Enum
```

The `EventLogController.AddLog()` method has several method overloads that enable a developer to log just about any values derived from an object or its properties. Following are descriptions of these overloaded methods, along with brief explanations of their parameters:

- The primary `AddLog` method is ultimately used by all of the other `AddLog` overloads and accepts a single LogInfo object. This method provides easy access to the base logging method in inherited LogController class:

```
Public Overloads Sub AddLog(ByVal objEventLogInfo As LogInfo)
```

- To log the property names and values of a Custom Business Object, use the following method:

```
Public Overloads Sub AddLog(ByVal objCBO As Object, ByVal _PortalSettings As _
PortalSettings, ByVal UserID As Integer, ByVal UserName As String, ByVal _
objLogType As Services.Log.EventLog.EventLogController.EventLogType)
```

Parameter	Type	Description
`objCBO`	Object	Custom Business Object.
`_PortalSettings`	PortalSettings	Current PortalSettings object.
`UserID`	Integer	UserID of the authenticated user of the request.
`UserName`	String	UserName of the authenticated user of the request.
`objLogType`	EventLogType	Event log type.

- To add a log entry that has no custom properties, use the following method:

```
Public Overloads Sub AddLog(ByVal _PortalSettings As PortalSettings, ByVal UserID _
As Integer, ByVal objLogType As _
Services.Log.EventLog.EventLogController.EventLogType)
```

This is useful if you simply need to log that an event has occurred, but you have no requirement to log any further details about the event.

Parameter	Type	Description
`_PortalSettings`	PortalSettings	Current PortalSettings object.
`UserID`	Integer	UserID of the authenticated user of the request.
`objLogType`	EventLogType	Event log type.

- To add a log entry that has a single property name and value, use the following method:

```
Public Overloads Sub AddLog(ByVal PropertyName As String, ByVal PropertyValue As _
String, ByVal _PortalSettings As PortalSettings, ByVal UserID As Integer, ByVal _
objLogType As Services.Log.EventLog.EventLogController.EventLogType)
```

Parameter	Type	Description
`PropertyName`	String	Name of the property to log.
`PropertyValue`	String	Value of the property to log.
`_PortalSettings`	PortalSettings	Current PortalSettings object.
`UserID`	Integer	UserID of the authenticated user of the request.
`objLogType`	EventLogType	Event log type.

- To add a log entry that has a single property name and value and the LogType is not defined in a core enumeration, use the following method:

```
Public Overloads Sub AddLog(ByVal PropertyName As String, ByVal PropertyValue As _
String, ByVal _PortalSettings As PortalSettings, ByVal UserID As Integer, ByVal _
LogType As String)
```

This is useful for custom modules that define their own log types.

Parameter	Type	Description
`PropertyName`	String	Name of the property to log.
`PropertyValue`	String	Value of the property to log.
`_PortalSettings`	PortalSettings	Current PortalSettings object.
`UserID`	Integer	UserID of the authenticated user of the request.
`LogType`	String	Event log type string.

- To add a log entry that has multiple property names and values, use the following method. To use this method, you must send into it a LogProperties object that is comprised of a collection of LogDetailInfo objects:

```
Public Overloads Sub AddLog(ByVal objProperties As LogProperties, ByVal _
_PortalSettings As PortalSettings, ByVal UserID As Integer, ByVal LogTypeKey As _
String, ByVal BypassBuffering As Boolean)
```

Parameter	Type	Description
`objProperties`	LogProperties	A collection of LogDetailInfo objects.
`_PortalSettings`	PortalSettings	Current PortalSettings object.
`UserID`	Integer	UserID of the authenticated user of the request.
`LogTypeKey`	String	Event log type.
`BypassBuffering`	Boolean	Specifies whether to write directly to the log (true) or to use log buffering (false) if log buffering is enabled.

Listings 8-2 through 8-6 show how to use the two most common overloaded methods for `EventLogController.AddLog()`. To exemplify the flexibility of this method, Listing 8-2 illustrates how you can send in a Custom Business Object and automatically log its property values.

Listing 8-2: EventLogController.AddLog Example

```
Private Sub TestUserInfoLog()
   Dim objUserInfo As New UserInfo
   objUserInfo.FirstName = "John"
   objUserInfo.LastName = "Doe"
   objUserInfo.UserID = 6
   objUserInfo.Username = "jdoe"
   Dim objEventLog As New Services.Log.EventLog.EventLogController
   objEventLog.AddLog(objUserInfo, PortalSettings, UserID, UserInfo.Username, _
       Services.Log.EventLog.EventLogController.EventLogType.USER_CREATED)
End Sub
```

The resulting log entry written by the XML Logging Provider for this example includes each property name and value in the objUserInfo object as shown in the `<properties/>` XML element in Listing 8-3.

Listing 8-3: EventLogController.AddLog Log Entry for the XML Logging Provider

```
<logs>
      <log LogGUID="92ca39e4-a135-475a-8c0c-7e4949c359b7" LogFileID="b86359bb-
e984-4483-891b-26a2b95bf9bd"
            LogTypeKey="USER_CREATED" LogUserID="-1" LogUserName="" LogPortalID="0"
LogPortalName="DotNetNuke"
            LogCreateDate="2005-02-04T14:33:46.9318672-05:00"
LogCreateDateNum="20050204143346931"
            LogServerName="DNNTEST">
            <properties>
                  <property>
                        <name>UserID</name>
                        <value>6</value>
```

```
                    </property>
                    <property>
                          <name>FirstName</name>
                          <value>John</value>
                    </property>
                    <property>
                          <name>LastName</name>
                          <value>Doe</value>
                    </property>
                    <property>
                          <name>UserName</name>
                          <value>jdoe</value>
                    </property>
              </properties>
        </log>
  </logs>
```

If you are using the default DB LoggingProvider, this same information is stored in the EventLog table. The properties element is saved in the LogProperties column as an XML fragment as shown in Listing 8-4. This format is similar to the properties node for the corresponding XML log version.

Listing 8-4: EventLogController.AddLog LogProperties for the DB Logging Provider

```
<ArrayOfAnyType
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <anyType xsi:type="LogDetailInfo">
            <name>UserID</name>
            <value>4</value>
      </anyType>
      <anyType xsi:type="LogDetailInfo">
            <name>FirstName</name>
            <value>Joe</value>
      </anyType>
      <anyType xsi:type="LogDetailInfo">
            <name>LastName</name>
            <value>Brinkman</value>
      </anyType>
      <anyType xsi:type="LogDetailInfo">
            <name>UserName</name>
            <value>tuser2</value>
      </anyType>
      <anyType xsi:type="LogDetailInfo">
            <name>Email</name>
            <value>joe.brinkman@tag-software.net</value>
      </anyType>
</ArrayOfAnyType>
```

Because the XML log format is a little easier to read, the XML Logging Provider is used for all additional examples in this chapter and will note any substantial differences with the DB Logging Provider.

This example logs each of the properties of a Custom Business Object. There are other overloaded `EventLogController.AddLog()` methods available if you need to log less information or information that isn't stored in a Custom Business Object. The example in Listing 8-5 shows how you can use `EventLogController.AddLog()` to add a single key/value pair to the log.

Listing 8-5: EventLogController.AddLog Example

```
    Private Sub TestCreateRole()

       Dim objRoleController As New RoleController
       Dim objRoleInfo As New RoleInfo

       'create and add the new role
       objRoleInfo.RoleName = "Newsletter Subscribers"
       objRoleInfo.PortalID = 5
       objRoleController.AddRole(objRoleInfo)

       'log the event
       Dim objEventLog As New Services.Log.EventLog.EventLogController
       objEventLog.AddLog("Role", objRoleInfo.RoleName, PortalSettings, _
       UserId, objEventLog.EventLogType.USER_ROLE_CREATED)

    End Sub
```

In this case, the key `Role` and the value `Newsletter Subscribers` will be logged. The resulting log entry written by the default XML Logging Provider for this example is shown in the `<properties/>` XML element in Listing 8-6.

Listing 8-6: EventLogController.AddLog Log Entry

```
<logs>
      <log LogGUID="2145856a-1e4a-4974-86f6-da1f0ae5dcca" LogFileID="b86359bb-
e984-4483-891b-26a2b95bf9bd"
            LogTypeKey="ROLE_CREATED" LogUserID="1" LogUserName="host"
LogPortalID="0" LogPortalName="DotNetNuke"
            LogCreateDate="2005-02-04T22:00:22.0413424-05:00"
LogCreateDateNum="20050204220022041"
            LogServerName="DNNTEST">
            <properties>
                  <property>
                        <name>RoleName</name>
                        <value>Newsletter Subscribers</value>
                  </property>
            </properties>
      </log>
</logs>
```

ExceptionLogController

The ExceptionLogController exposes the methods necessary for adding information about exceptions to the log. This controller class also defines four exception types in the ExceptionLogType enumeration: `GENERAL_EXCEPTION`, `MODULE_LOAD_EXCEPTION`, `PAGE_LOAD_EXCEPTION`, and `SCHEDULER_EXCEPTION`. By defining different log types for exceptions, the configuration of the Logging Provider can treat each exception log type differently regarding how and if the exceptions get logged.

The next section covers exceptions in more detail. For now, the focus is on how to log the exceptions. The `ExceptionLogController.AddLog()` method has three overloaded methods that enable you to pass in various types of exceptions. The first method enables you to send in a System.Exception or any exception that inherits System.Exception, as shown in Listing 8-7.

Listing 8-7: ExceptionLogController.AddLog Example

```
Public Sub test()
    Try
        If 1 = 1 Then
            Throw New Exception("Oh no, an exception!")
        End If
    Catch exc As Exception
        Dim objExceptionLog As New Services.Log.EventLog.ExceptionLogController
        objExceptionLog.AddLog(exc)
        'a shortcut to this is simply "LogException(exc)"
    End Try
End Sub
```

In this case, the properties of the System.Exception will be logged along with a collection of properties that are specific to the request. For instance, it will log the filename, line, and column number the exception occurred in if it is available. The resulting log entry written by the default XML Logging Provider for this example is shown in Listing 8-8.

Listing 8-8: ExceptionLogController.AddLog Log Entry

```
<logs>
      <log LogGUID="39c72059-bcd1-42ca-8886-002363d1c9dc" LogFileID="6b780a60-
cf46-4588-8a76-75ae9c577277"
            LogTypeKey="GENERAL_EXCEPTION" LogUserID="-1" LogUserName=""
LogPortalID="-1" LogPortalName=""
            LogCreateDate="2005-02-04T23:25:44.6873456-05:00"
LogCreateDateNum="20050204232544687"
            LogServerName="DNNTEST">
            <properties>
                  <property>
                        <name>AssemblyVersion</name>
                        <value>03.00.10</value>
                  </property>
                  <property>
                        <name>Method</name>
                        <value>DotNetNuke.Framework.CDefault.test</value>
                  </property>
                  <property>
                        <name>FileName</name>
                        <value>c:\public\dotnetnuke\Default.aspx.vb</value>
                  </property>
                  <property>
                        <name>FileLineNumber</name>
                        <value>481</value>
                  </property>
                  <property>
                        <name>FileColumnNumber</name>
                        <value>21</value>
```

(continued)

Listing 8-8: *(continued)*

```
                    </property>
                    <property>
                          <name>PortalID</name>
                          <value>0</value>
                    </property>
                    <property>
                          <name>PortalName</name>
                          <value>DotNetNuke</value>
                    </property>
                    <property>
                          <name>UserID</name>
                          <value>-1</value>
                    </property>
                    <property>
                          <name>UserName</name>
                          <value />
                    </property>
                    <property>
                          <name>ActiveTabID</name>
                          <value>36</value>
                    </property>
                    <property>
                          <name>ActiveTabName</name>
                          <value>Home</value>
                    </property>
                    <property>
                          <name>AbsoluteURL</name>
                          <value>/DotNetNuke/Default.aspx</value>
                    </property>
                    <property>
                          <name>AbsoluteURLReferrer</name>
                          <value />
                    </property>
                    <property>
                          <name>ExceptionGUID</name>
                          <value>128455d6-064a-4222-993f-b54fd302e21e</value>
                    </property>
                    <property>
                          <name>DefaultDataProvider</name>
                          <value>DotNetNuke.Data.SqlDataProvider,
DotNetNuke.SqlDataProvider</value>
                    </property>
                    <property>
                          <name>InnerException</name>
                          <value>Oh no, an exception!</value>
                    </property>
                    <property>
                          <name>Message</name>
                          <value>Oh no, an exception!</value>
                    </property>
                    <property>
                          <name>StackTrace</name>
                          <value>   at DotNetNuke.Framework.CDefault.test() in
c:\public\dotnetnuke\Default.aspx.vb:line 481</value>
```

```
                    </property>
                    <property>
                          <name>Source</name>
                          <value>DotNetNuke</value>
                    </property>
              </properties>
        </log>
  </logs>
```

Notice that Listing 8-8 does not tell you the portal module that the exception was thrown from. This is because a general exception was thrown (System.Exception). If a ModuleLoadException is thrown, more details about the portal module that throws the exception will be logged.

Exception Handling

The exception handling API in DotNetNuke provides a framework for handling exceptions uniformly and elegantly. Exception handling primarily uses four methods, most of which have several overloaded methods. Through these methods, developers can gracefully handle exceptions, log the exception trace and context, and display a user-friendly message to the end user.

The Exception Handling API

The exception handling API lives under the DotNetNuke.Services.Exceptions namespace. Table 8-2 lists the classes that comprise the Exception Handling API.

Table 8-2: Exception Handling Classes

Class	Description
BasePortalException	Inherits from System.Exception and contains many other properties specific to the portal application.
ErrorContainer	Generates formatting for the error message that will be displayed in the web browser.
ExceptionInfo	Stores information from the stack trace.
Exceptions	Contains most of the methods that are used in custom modules. It contains the methods necessary to process each type of portal exception.
ModuleLoadException	An exception type for exceptions thrown within portal modules. It inherits from BasePortalException.
PageLoadException	An exception type for exceptions thrown within pages.
SchedulerException	An exception type for exceptions thrown within the Scheduling Provider. It also inherits from BasePortalException.

The Exceptions Class

Although there are many classes in the exception handling namespace, the primary class that module developers deal with regularly is the Exceptions class. This class contains all of the methods necessary to gracefully handle exceptions in DotNetNuke. The most widely used method for exception handling is `DotNetNuke.Services.Exceptions.ProcessModuleLoadException()`.

ProcessModuleLoadException Method

The `ProcessModuleLoadException` method serves two primary functions: to log the exceptions that are thrown from within a module to the Logging Provider, and to display a friendly error message in place of the module that threw the exception. The friendly error message is displayed only if the host option Use Custom Error Messages is enabled on the Host Settings page (see Chapter 5).

`ProcessModuleLoadException` has seven overloaded methods:

- To process an exception that occurs in a portal module, use the following method. If the Custom Error Messages option has been enabled in Host Settings, this method will also handle displaying a user-friendly error message to the client browser:

```
Public Sub ProcessModuleLoadException(ByVal ctrlModule As _
Entities.Modules.PortalModuleBase, ByVal exc As Exception)
```

Parameter	Type	Description
`ctrlModule`	PortalModuleBase	Portal module object.
`exc`	Exception	Exception that was thrown.

- This method is the same as the previous one, although it provides the capability to suppress the error message from being displayed on the client browser:

```
Public Sub ProcessModuleLoadException(ByVal ctrlModule As _
Entities.Modules.PortalModuleBase, ByVal exc As Exception, ByVal _
DisplayErrorMessage As Boolean)
```

Parameter	Type	Description
`ctrlModule`	PortalModuleBase	Portal module object.
`exc`	Exception	Exception that was thrown.
`DisplayErrorMessage`	Boolean	Indicates whether the portal should render an error message to the client browser.

- This is the same as the previous method; however, it adds the capability to provide a custom friendly message to the client browser:

```
Public Sub ProcessModuleLoadException(ByVal FriendlyMessage As String, ByVal _
ctrlModule As Entities.Modules.PortalModuleBase, ByVal exc As Exception, _
ByVal DisplayErrorMessage As Boolean)
```

Parameter	Type	Description
FriendlyMessage	String	Friendly message to display to the client browser.
ctrlModule	PortalModuleBase	Portal module object.
exc	Exception	Exception that was thrown.
DisplayErrorMessage	Boolean	Indicates whether the portal should render an error message to the client browser.

- Use the following overloaded method if you are handling exceptions in a control that isn't directly in a portal module. For instance, if your portal module uses a server control, you can use this method to handle exceptions within that server control. It displays a friendly error message if custom error messages are enabled:

```
Public Sub ProcessModuleLoadException(ByVal FriendlyMessage As String, _
ByVal UserCtrl As Control, ByVal exc As Exception)
```

Parameter	Type	Description
FriendlyMessage	String	Friendly message to display to the client browser.
UserCtrl	Control	The control. It can be anything that inherits from System.Web.UI.Control.
exc	Exception	Exception that was thrown.

- This is the same as the previous method; however, it adds the capability to specify whether to display an error message to the client browser (the Host Settings option to Use Custom Error Messages takes precedence over this value):

```
Public Sub ProcessModuleLoadException(ByVal FriendlyMessage As String, _
ByVal ctrlModule As Control, ByVal exc As Exception, _
ByVal DisplayErrorMessage As Boolean)
```

Parameter	Type	Description
FriendlyMessage	String	Friendly message to display to the client browser.
ctrlModule	Control	The control. It can be anything that inherits from System.Web.UI.Control.
exc	Exception	Exception that was thrown.
DisplayErrorMessage	Boolean	Indicates whether the portal should render an error message to the client browser.

- This is a simple method that has only two parameters. It displays a generic error message to the client browser if custom error messages are enabled:

```
Public Sub ProcessModuleLoadException(ByVal UserCtrl As Control, _
ByVal exc As Exception)
```

Parameter	Type	Description
`UserCtrl`	Control	The control. It can be anything that inherits from System.Web.UI.Control.
`exc`	Exception	Exception that was thrown.

- This is the same as the previous method except it provides the capability to suppress the error message that is displayed in the client browser (the Host Settings option to Use Custom Error Messages takes precedence over this value):

```
Public Sub ProcessModuleLoadException(ByVal UserCtrl As Control, _
ByVal exc As Exception, ByVal DisplayErrorMessage As Boolean)
```

Parameter	Type	Description
`UserCtrl`	Control	The control. It can be anything that inherits from System.Web.UI.Control.
`exc`	Exception	Exception that was thrown.
`DisplayErrorMessage`	Boolean	Indicates whether the portal should render an error message to the client browser.

ProcessPageLoadException Method

Similar to the `ProcessModuleLoadException` method, the `ProcessPageLoadException` method serves two primary functions: to log the exceptions thrown from outside of a module to the Logging Provider, and to display a friendly error message on the page. The friendly error message will only be displayed if the host option Use Custom Error Messages is enabled on the Host Settings page (see Chapter 5).

`ProcessPageLoadException` has two overloaded methods:

- To process an exception that occurs in an ASPX file or in logic outside of a portal module, use the following overloaded method. If the Use Custom Error Messages option has been enabled in Host Settings, this method also handles displaying a user-friendly error message to the client browser:

```
Public Sub ProcessPageLoadException(ByVal exc As Exception)
```

Parameter	Type	Description
`exc`	Exception	Exception that was thrown.

- This is the same as the previous method; however, you must send in the URL parameter to redirect the request after logging the exception:

```
Public Sub ProcessPageLoadException(ByVal exc As Exception, _
ByVal URL As String)
```

Parameter	Type	Description
exc	Exception	Exception that was thrown.
URL	String	URL to redirect the request to.

LogException Method

The `LogException` method is used for adding exceptions to the log. It does not handle displaying any type of friendly message to the user. Instead, it simply logs the error without notifying the client browser of a problem. `LogException` has four overloaded methods:

- To log an exception thrown from a module, use the following overloaded method:

```
Public Sub LogException(ByVal exc As ModuleLoadException)
```

Parameter	Type	Description
exc	ModuleLoadException	Exception that was thrown.

- To log an exception thrown from a page or other logic outside of a module, use the following overloaded method:

```
Public Sub LogException(ByVal exc As PageLoadException)
```

Parameter	Type	Description
exc	PageLoadException	Exception that was thrown.

- To log an exception thrown from within a Scheduling Provider Task, use the following overloaded method:

```
Public Sub LogException(ByVal exc As SchedulerException)
```

Parameter	Type	Description
Exc	SchedulerException	Exception that was thrown.

- If you need to log an exception of another type, use the following overloaded method:

```
Public Sub LogException(ByVal exc As Exception)
```

Parameter	Type	Description
exc	Exception	Exception that was thrown.

ProcessSchedulerException Method

The `ProcessSchedulerException` method is used to log exceptions thrown from within a scheduled task. It simply logs the error.

To log an exception thrown from a scheduled task, use the following overloaded method:

```
Public Sub LogException(ByVal exc As ModuleLoadException)
```

Parameter	Type	Description
exc	ModuleLoadException	Exception that was thrown.

The exception handling API abstracts developers from the complexity of logging exceptions and presenting error messages gracefully. It provides several powerful methods that handle all of the logic involved in working with the Logging Provider and the presentation layer. The next section covers the various interfaces that module developers can take advantage of to bring more core features to life in their modules.

Scheduler

The Scheduler in DotNetNuke is a mechanism that enables developers to schedule tasks to run at defined intervals. It is implemented using the Provider pattern; therefore, it can easily be replaced without modifying core code. Creating a scheduled task is a fairly simple process. First, though, it's important to understand which types of tasks are suitable for the Scheduler.

Because the Scheduler is run under the context of the web application, it is prone to the same types of application recycles as a web application. In a web-hosting environment, it is a common practice to conserve resources by recycling the worker process for a site periodically. When this happens, the Scheduler stops running. Therefore, the tasks run by the Scheduler do not run 24 hours a day, 7 days a week. They are executed according to a defined schedule, but they can only be triggered when the worker process is alive. For this reason, you cannot specify that a task should run every night at midnight. It is not possible in the web environment to meet this type of use case. Instead, you can specify how often a task is run by defining the execution frequency for each task. The execution frequency is defined as every *x* minutes/hours/days.

To create a scheduled task, you must create a class that inherits from DotNetNuke.Services.Scheduling.SchedulerClient. This class must provide a constructor and a `DoWork` method. An example of a scheduled task is shown in Listing 8-9. This sample scheduled task will move all event log files to a folder named with the current date. By configuring this scheduled task to run once per day, the log files will be automatically archived daily, which keeps the log file sizes manageable.

Listing 8-9: Scheduled Task Example

```
Public Class ArchiveEventLog
    Inherits DotNetNuke.Services.Scheduling.SchedulerClient
    Public Sub New(ByVal objScheduleHistoryItem As _
        DotNetNuke.Services.Scheduling.ScheduleHistoryItem)
        MyBase.new()
        Me.ScheduleHistoryItem = objScheduleHistoryItem    'REQUIRED
    End Sub
    Public Overrides Sub DoWork()
        Try
            'notification that the event is progressing
            'this is optional
            Me.Progressing()    'OPTIONAL
            'get the directory that logs are written to
            Dim LogDirectory As String
            LogDirectory = Common.Globals.HostMapPath + "Logs\"

            'create a folder with today's date
            Dim FolderName As String
            FolderName = LogDirectory + Now.Month.ToString + "-" + _
                Now.Day.ToString + "-" + Now.Year.ToString + "\"
            If Not IO.Directory.Exists(FolderName) Then
                IO.Directory.CreateDirectory(FolderName)
            End If

            'get the files in the log directory
            Dim s As String()
            s = IO.Directory.GetFiles(LogDirectory)
            'loop through the files
            Dim i As Integer
            For i = 0 To s.Length - 1
                Dim OldFileInfo As New IO.FileInfo(s(i))
                'move all files to the new folder except the file
                'used to store pending log notifications
                If OldFileInfo.Name <> _
                    "PendingLogNotifications.xml.resources" Then
                    Dim NewFileName As String
                    NewFileName = FolderName + OldFileInfo.Name
                    'check to see if the new file already exists
                    If IO.File.Exists(NewFileName) Then
                        Dim errMessage As String
                        errMessage = "An error occurred archiving " + _
                            "log file to " + _
                            NewFileName + ".  The file already exists."
                        LogException(New _
                            BasePortalException(errMessage))
                    Else
                        IO.File.Move(OldFileInfo.FullName, NewFileName)
                        Me.ScheduleHistoryItem.AddLogNote("Moved " + _
                            OldFileInfo.FullName + _
                                " to " + FolderName + _
                                OldFileInfo.Name + ".")      'OPTIONAL
                    End If
```

(continued)

Listing 8-9: *(continued)*

```
                End If
            Next

            Me.ScheduleHistoryItem.Succeeded = True    'REQUIRED

        Catch exc As Exception    'REQUIRED

           Me.ScheduleHistoryItem.Succeeded = False    'REQUIRED

           Me.ScheduleHistoryItem.AddLogNote(String.Format( _
               "Archiving log files failed.", _
               exc.ToString))    'OPTIONAL

            'notification that we have errored
            Me.Errored(exc)    'REQUIRED

            'log the exception
            LogException(exc)    'OPTIONAL
        End Try
    End Sub
End Class
```

After the class has been compiled into the bin directory, the task can be scheduled from the Scheduling module under the Host page (see Chapter 6 for details). It is important to include each of the lines of code in Listing 8-9 that is labeled `REQUIRED`. These collectively ensure both the exception handling and schedule management are handled uniformly throughout all scheduled tasks.

HTTP Modules

ASP.NET provides a number of options for extending the path that data takes between client and server (known as the HTTP Pipeline). A popular method to extend the pipeline is through the use of custom components known as *HTTP modules*. An HTTP module enables you to add pre- and post-processing to each HTTP request coming into your application.

DotNetNuke implements a number of HTTP modules to extend the pipeline. They include features such as URL Rewriting, Exception Management, Users Online, Profile, Anonymous Identification, Role Management, DotNetNuke Membership, and Personalization.

Originally, a lot of the HTTP modules were implemented inside the core application (`global.asax.vb`). There were a number of reasons why the functionally was moved to HTTP modules:

- Administrators can optionally enable or disable an HTTP module.
- Developers can replace or modify HTTP modules without altering the core application.
- Provides templates for extending the HTTP Pipeline.

HTTP Modules 101

This section further examines the concepts of HTTP modules so you'll know when and where to implement them. To comprehend how HTTP modules work, it's necessary to understand the HTTP Pipeline and how ASP.NET processes incoming requests. Figure 8-1 shows the HTTP Pipeline.

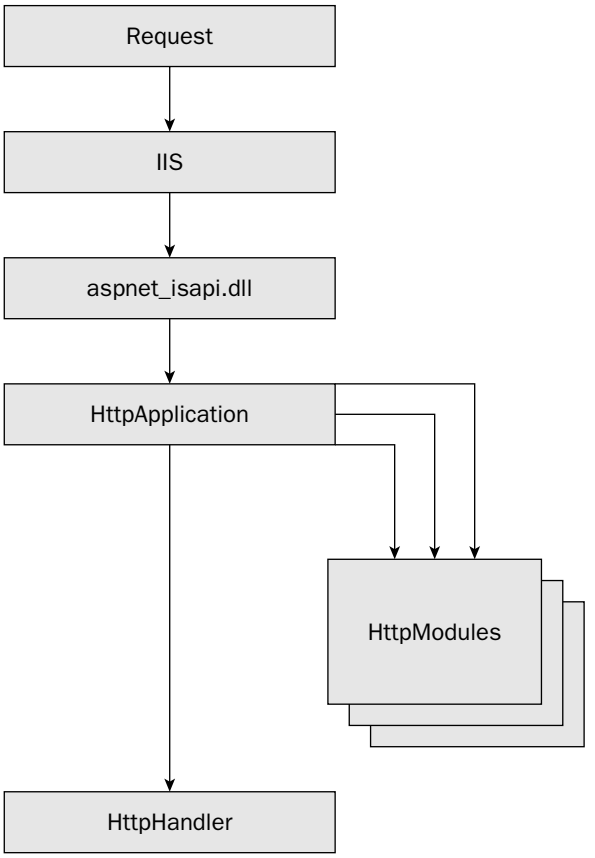

Figure 8-1

When a request is first made, it passes through a number of stages before it is actually handled by your application. The first participant in the pipeline is Microsoft Internet Information Server (IIS); its job is to route ASP.NET requests to the ASP.NET runtime. When an ASPX file is requested (or any other ASP.NET file), IIS forwards the request to the ASP.NET runtime (via an ISAPI extension).

Now that the request has been received by ASP.NET, it must pass through an instance of HttpApplication. The HttpApplication object handles application-wide methods, data, and events. It is also responsible for pushing the request through one or more HTTP module objects. The ASP.NET runtime determines which modules to load by examining the configuration files located at either machine level (`machine.config`) or application level (`web.config`). Listing 8-10 shows the HTTP modules configuration section of DotNetNuke.

Listing 8-10: HTTP Modules Configuration Section

```
<httpModules>
 <add name="UrlRewrite" type="DotNetNuke.HttpModules.UrlRewriteModule,
                              DotNetNuke.HttpModules.UrlRewrite" />

 <add name="Exception" type="DotNetNuke.HttpModules.ExceptionModule,
                             DotNetNuke.HttpModules.Exception" />

 <add name="UsersOnline" type="DotNetNuke.HttpModules.UsersOnlineModule,
                               DotNetNuke.HttpModules.UsersOnline" />

 <add name="Profile" type="Microsoft.ScalableHosting.Profile.ProfileModule,
                           MemberRole,
                           Version=1.0.0.0,
                           Culture=neutral,
                           PublicKeyToken=b7c773fb104e7562" />

 <add name="AnonymousIdentification"
      type="Microsoft.ScalableHosting.Security.AnonymousIdentificationModule,
            MemberRole,
            Version=1.0.0.0,
            Culture=neutral,
            PublicKeyToken=b7c773fb104e7562" />

 <add name="RoleManager"
      type="Microsoft.ScalableHosting.Security.RoleManagerModule,
            MemberRole,
            Version=1.0.0.0,
            Culture=neutral,
            PublicKeyToken=b7c773fb104e7562" />

 <add name="DNNMembership" type="DotNetNuke.HttpModules.DNNMembershipModule,
                                 DotNetNuke.HttpModules.DNNMembership" />

 <add name="Personalization" type="DotNetNuke.HttpModules.PersonalizationModule,
                                   DotNetNuke.HttpModules.Personalization" />

</httpModules>
```

To invoke each HTTP module, the `Init` method of each module is invoked. At the end of each request, the `Dispose` method is invoked to enable each HTTP module to clean up its resources. In fact, those two methods form the interface (IHttpModule) each module must implement. Listing 8-11 shows the IHttpModule interface.

Listing 8-11: The IHttpModule Interface Implemented by Each HTTP Module

```
Public Interface IHttpModule
    Sub Init(ByVal context As HttpApplication)
    Sub Dispose()
End Interface
```

During the Init event, each module may subscribe to a number of events raised by the HttpApplication object. Table 8-3 shows the events that are raised before the application executes. The events are listed in the order in which they occur.

Table 8-3: HTTP Module Events (Before the Application Executes)

Event	Description
`BeginRequest`	Signals a new request; guaranteed to be raised on each request.
`AuthenticateRequest`	Signals that the request is ready to be authenticated; used by the Security module.
`AuthorizeRequest`	Signals that the request is ready to be authorized; used by the Security module.
`ResolveRequestCache`	Used by the Output Cache module to short-circuit the processing of requests that have been cached.
`AcquireRequestState`	Signals that the per-request state should be obtained.
`PreRequestHandlerExecute`	Signals that the request handler is about to execute. This is the last event you can participate in before the HTTP handler for this request is called.

Table 8-4 shows the events that are raised after an application has returned. The events are listed in the order in which they occur.

Table 8-4: HTTP Module Events (After the Application Has Returned)

Event	Description
`PostRequestHandlerExecute`	Signals that the HTTP handler has completed processing the request.
`ReleaseRequestState`	Signals that the request state should be stored because the application is finished with the request.
`UpdateRequestCache`	Signals that code processing is complete and the file is ready to be added to the ASP.NET cache.
`EndRequest`	Signals that all processing has finished for the request. This is the last event called when the application ends.

In addition, there are three per-request events that can be raised in a nondeterministic order. They are described in Table 8-5.

Table 8-5: HTTP Module Events (Nondeterministic)

Event	Description
`PreSendRequestHeaders`	Signals that HTTP headers are about to be sent to the client. This provides an opportunity to add, remove, or modify the headers before they are sent.
`PreSendRequestContent`	Signals that content is about to be sent to the client. This provides an opportunity to modify the content before it is sent.
`Error`	Signals an unhandled exception.

After the request has been pushed through the HTTP modules configured for your application, the HTTP handler responsible for the requested file's extension (.ASPX) handles the processing of that file. If you are familiar with ASP.NET development, you'll be familiar with the handler for an ASPX page—System.Web.UI.Page. The HTTP handler then handles the life cycle of the page-level request raising events such as `Page_Init`, `Page_Load`, and so on.

DotNetNuke HTTP Modules

As stated earlier, DotNetNuke (like ASP.NET) comes with a number of HTTP modules. These modules enable developers to customize the HTTP Pipeline to provide additional functionality on each request. In this section, you explore several DotNetNuke HTTP modules, and examine their purpose and possibilities for extension.

URL Rewriter

The URL rewriter is an HTTP module that provides a mechanism for mapping virtual resource names to physical resource names at runtime—in other words, it provides a URL that is friendly. The term "friendly" has two aspects. One is to make the URL search-engine friendly, which is solved with the default implementation.

Most search engines ignore URL parameters, and because DotNetNuke relies on URL parameters to navigate to portal pages, the older application is not search-engine friendly. To effectively index your site, you need a parameterless mechanism for constructing URLs that search engines will process.

If you browse to a DotNetNuke site that is version 3.0 or greater, you may notice different URLs from earlier versions. Traditionally, a DotNetNuke URL looks something like the following:

```
http://www.dotnetnuke.com/default.aspx?tabid=622
```

With friendly URLs enabled, the preceding URL might look like this:

```
http://www.dotnetnuke.com/RoadMap/Friendly URLs/tabid/622/default.aspx
```

URL rewriter is invoked during the HTTP Pipeline's processing of a request and can optionally subscribe to application-wide events. The particular event of interest for this module is `BeginRequest`. This event enables you to modify the URL before the Page HTTP handler is invoked and make it believe the URL requested was that of the old non-friendly format.

The transformation process occurs through the use of regular expressions defined in SiteUrls.config in the root of your DotNetNuke installation. This file contains a number of expressions to `LookFor` and with corresponding URLs to `SendTo`. Listing 8-12 shows the default SiteUrls.config.

Listing 8-12: SiteUrls.config

```
<?xml version="1.0" encoding="utf-8" ?>
<RewriterConfig>
<Rules>
      <RewriterRule>
            <LookFor>.*/TabId/(\d+)(.*)/Logoff.aspx</LookFor>
            <SendTo>~/Admin/Security/Logoff.aspx?tabid=$1</SendTo>
      </RewriterRule>
      <RewriterRule>
            <LookFor>.*/TabId/(\d+)(.*)/rss.aspx</LookFor>
            <SendTo>~/rss.aspx?TabId=$1</SendTo>
      </RewriterRule>
      <RewriterRule>
            <LookFor>[^?]*/TabId/(\d+)(.*)</LookFor>
            <SendTo>~/Default.aspx?TabId=$1</SendTo>
      </RewriterRule>
</Rules>
</RewriterConfig>
```

The rules defined in this configuration file cover the default login and logoff page. You could potentially add any number of additional rules, and even hardcode some extra rules in there. For example, if you wanted to hardcode a link such as `http://www.dotnetnuke.com/FriendlyUrl.aspx` and have it map to another URL elsewhere, your entry might look like Listing 8-13.

Listing 8-13: SiteUrls.config with a Modified Rule

```
<?xml version="1.0" encoding="utf-8" ?>
<RewriterConfig>
<Rules>
      <RewriterRule>
            <LookFor>.*/FriendlyUrl.aspx</LookFor>
            <SendTo>~/default.aspx?tabid=622</SendTo>
      </RewriterRule>
      <RewriterRule>
            <LookFor>.*/TabId/(\d+)(.*)/Logoff.aspx</LookFor>
            <SendTo>~/Admin/Security/Logoff.aspx?tabid=$1</SendTo>
      </RewriterRule>
      <RewriterRule>
            <LookFor>.*/TabId/(\d+)(.*)/rss.aspx</LookFor>
            <SendTo>~/rss.aspx?TabId=$1</SendTo>
      </RewriterRule>
      <RewriterRule>
            <LookFor>[^?]*/TabId/(\d+)(.*)</LookFor>
            <SendTo>~/Default.aspx?TabId=$1</SendTo>
      </RewriterRule>
</Rules>
</RewriterConfig>
```

The preceding URL scheme is an excellent implementation for your own applications as well, particularly those with fixed pages. Unfortunately DotNetNuke has potentially any number of pages, so the team added some functionality that would transform any number of query string parameters.

Take a look at the default scheme for URL rewriting. You can see from the friendly URL shown earlier (`http://www.dotnetnuke.com/RoadMap/Friendly URLs/tabid/622/default.aspx`) that the requirement is met — that is, the URL would have no parameters. URLs generally adhere to the following pattern:

- `http://www.dotnetnuke.com/`: The site Host URL.
- `RoadMap/Friendly URLs/`: The breadcrumb path back to the home page.
- `tabid/622/`: The query string from the original URL transformed (`?tabid=622`).
- `default.aspx`: The standard web page for DotNetNuke.

The advantage of this scheme is that it requires no database lookups for the transformation, just raw regular expression processing that is typically quite fast.

For some situations, the breadcrumb path may not be desired. In those cases, simply modify the web.config `friendlyUrl` provider setting to turn off the feature. To turn it off, change the `includePageName` value shown in Listing 8-14 from `"true"` to `"false"`:

Listing 8-14: Modifying SiteUrls.config

```
<friendlyUrl defaultProvider="DNNFriendlyUrl">
    <providers>
        <clear />
        <add name="DNNFriendlyUrl"
        type="DotNetNuke.Services.Url.FriendlyUrl.DNNFriendlyUrlProvider,
        DotNetNuke.HttpModules.UrlRewrite" includePageName="true"
        regexMatch="[^a-zA-Z0-9 _-]" />
    </providers>
</friendlyUrl>
```

Earlier in this chapter, it was mentioned that there are two aspects of friendly URLs; so far, only one (search-engine friendly) has been discussed. The second aspect, known as human-friendly URLs, can sometimes impact performance.

A URL that is human friendly is easily remembered or able to be worked out by a human. For example, if you had a login to `dotnetnuke.com` and you wanted to visit your profile page without navigating to it, you might expect the URL to be `http://www.dotnetnuke.com/profile/smcculloch.aspx`.

That URL could easily be remembered, but would require additional processing on the request for two reasons:

- The URL contains no TabID. That would have to be looked up.
- The URL contains no UserID. A lookup on `smcculloch` is needed to find the corresponding UserID.

For these reasons, this approach was not chosen. Human-friendly URLs can be implemented by hardcoding the tabid and any other necessary parameters in the rewriter rules. You can see this in action on the Industrial Press web site at `www.industrialpress.com/en` — the left column contains links like `http://www.industrialpress.com/en/AutoCAD/default.aspx`. Listing 8-15 shows the rewriter rule for implementing this link.

Listing 8-15: Human Readable URL Example

```
<RewriterRule>
    <LookFor>[^?]*/AutoCad/(.*)</LookFor>
    <SendTo>~/Default.aspx?TabId=108</SendTo>
</RewriterRule>
```

So far, how incoming requests are interpreted has been explained, but how outgoing links are transformed into the friendly URL scheme have not. A number of options have been explored on how to transform the outgoing links, but the best option was to implement a provider-based component that would transform a given link into the chosen scheme. Figure 8-2 shows the URL rewriter architecture.

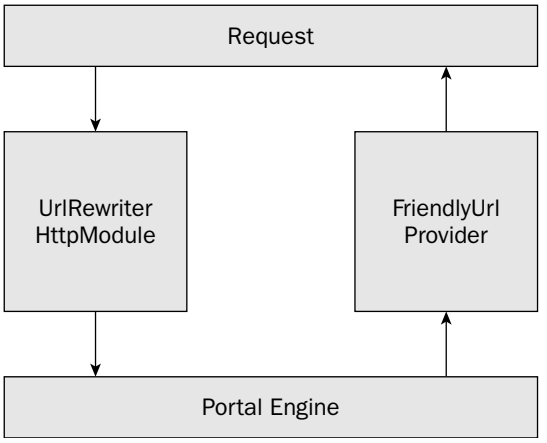

Figure 8-2

Luckily, DotNetNuke already had used two shortcut methods for building links within the application (`NavigateUrl` and `EditUrl`). It was relatively simple to place a call to the provider from each of these methods, effectively upgrading the site to the new URL format instantly.

This approach also tightly coupled the HTTP module with the provider, which is why you can find them in the same assembly (DotNetNuke.HttpModules.UrlRewrite.dll).

You can see from the architecture that it is quite plausible for you to write your own URL rewriting scheme. If it was more important for your site to have human-friendly URLs, you could write a scheme by creating a new provider to format outgoing URLs, and a new HTTP module to interpret the incoming requests.

Writing a new provider involves supplying new implementations of the methods in the FriendlyUrlProvider base class. Listing 8-16 shows these methods.

Listing 8-16: Friendly URL Provider Methods

```
Public MustOverride Function FriendlyUrl(ByVal objtab as TabInfo, _
    ByVal path As String) As String
Public MustOverride Function FriendlyUrl(ByVal objtab as TabInfo, _
    ByVal path As String, ByVal pageName As String) As String
Public MustOverride Function FriendlyUrl(ByVal objtab as TabInfo, _
    ByVal path As String, ByVal pageName As String, _
    ByVal settings As PortalSettings) As String
Public MustOverride Function FriendlyUrl(ByVal objtab as TabInfo, _
    ByVal path As String, ByVal pageName As String, ByVal portalAlias As String) _
    As String
```

As you can see, there are only four methods to implement so that you can write your URLs in your desired format. The most important part is to come up with a scheme and to find an efficient, reliable way of interpretation by your HTTP module. After you have written your provider, you can make an additional entry in the providers section of web.config as shown in Listing 8-17. Make sure to set the `defaultProvider` attribute.

Listing 8-17: Friendly URL Provider Configuration

```
<friendlyUrl defaultProvider="CustomFriendlyUrl">
  <providers>
    <clear />
    <add name="DNNFriendlyUrl"
      type="DotNetNuke.Services.Url.FriendlyUrl.DNNFriendlyUrlProvider,
      DotNetNuke.HttpModules.UrlRewrite" includePageName="true"
      regexMatch="[^a-zA-Z0-9 _-]"/>
    <add name="CustomFriendlyUrl"
        type="CompanyName.FriendlyUrlProvider, CompanyName.FriendlyUrlProvider" />
    </providers>
</friendlyUrl>
```

Exception Management

The exception management HTTP module subscribes to the error event raised by the HttpApplication object. Any time an error occurs within DotNetNuke, the error event is called. During the processing of this event, the last error to have occurred is captured and sent to the exception logging class, which calls the Logging Provider that handles the writing of that exception to a data store (the default is the DB Logging Provider).

Users Online

Users Online was implemented during version 2 of DotNetNuke. It allows other modules to interrogate the applications' data store for information regarding who is online, expressed as registered users and anonymous users. Previously it had been a custom add-on and was session-based. Before the addition of the functionality to the core (like many add-ons incorporated into the core), research was undertaken to investigate the best way to handle not only registered users, but also anonymous users.

The module subscribes to the `AuthorizeRequest` event. This event is the first chance an HTTP module has to examine details about the user performing the request. The HTTP module examines the request, determines whether the user is anonymous or authenticated, and stores the request in cache. Anonymous

users are also given a temporary cookie so they are not counted twice in the future. A scheduled job from the Scheduler executes every minute on a background thread, pulling the relevant details out of cache and updating them in the database. It also clears up any old records. The records are stored within two tables: AnonymousUsers and UsersOnline.

The HTTP module is a good module to disable (comment out of web.config) if you do not need this information within your portal. Alternatively, you can just disable it in Host Settings.

DNNMembership

The DNNMembership HTTP module performs tasks around the security of a user. It stores role information about a user in an HTTP cookie so the same information does not have to be requested again and performs security checks for users switching portals.

There is no real need to extend this module because it is critical to DotNetNuke's operation.

Personalization

The Personalization HTTP module is very similar to the Microsoft-provided Profile HTTP module, and in fact, was based on the same concept, just integrated much earlier. It loads a user's personalized information into a serialized XML object at the beginning of the request, and saves it at the end of the request.

If you are interested in storing personalized information about a user, see the personalization classes (described in Table 8-6) under /Components/Personalization/.

Table 8-6: Personalization Classes

Class	Purpose
`Personalization`	The primary API for using the personalization system. It encapsulates the few DotNetNuke business rules for using the personalization system.
PersonalizationController	Represents a low-level API that converts personalization database references into business objects.
PersonalizationInfo	The data transfer object that represents the data in a programming friendly object.

Module Interfaces

Modules represent a discrete set of functionality that can extend the portal framework. In past versions of DotNetNuke, module interactions with the portal were primarily limited to making method calls into the core portal APIs. Though this one-way interaction provides some capability to use portal services and methods within the module, it limits the capability of the portal to provide more advanced services.

To provide two-way interactions with modules, the portal needs to have a mechanism to make method calls into the module. There are several distinct mechanisms for allowing a program to call methods on an arbitrary set of code, where the module code is unknown at the time the portal is being developed. Three of these "calling" mechanisms are used within DotNetNuke:

- Inheritance
- Delegates
- Interfaces

As discussed previously, every module inherits from the PortalModuleBase class (located in the components/module directory). This base class provides a common set of methods and properties that can be used by the module as well as the portal to control the behavior of each module instance. Because the module must inherit from this class, the portal has a set of known methods that it can use to control the module. The portal could extend the base class to add methods to handle new services. One downside to this approach is that there is not an easy mechanism for determining whether a subclass implements custom logic for a specific method or property. Because of this restriction, inheritance is generally limited to providing services that are needed or required for every subclass.

A second method for interacting with the modules involves the use of delegates. A delegate is essentially a pointer to a method that has a specific set of parameters and return type. Delegates are useful when a service can be implemented with a single method call and are the underlying mechanism behind VB.NET's event handling. DotNetNuke uses delegates to implement callback methods for the Module Action menu action event. Although delegates are very useful in some situations, they are more difficult to implement and understand than alternative methods.

The third calling mechanism used by DotNetNuke is the use of interfaces. An interface defines a set of methods, events, and properties without providing any implementation details for these elements. Any class that implements an interface is responsible for providing the specific logic for each method, event, and property defined in the interface. Interfaces are especially useful for defining optional services that a module may implement. The portal can detect if a class implements a specific interface and can then call any of the methods, events, or properties defined in the interface.

Starting in version 3.0, DotNetNuke significantly extended its use of module interfaces. Six main interfaces are intended for use by modules:

- IActionable
- IPortable
- IUpgradeable
- IModuleCommunicator
- IModuleListener
- ISearchable

IActionable

Every module has a menu that contains several possible action items for activities like editing module settings, module movement, and viewing help. These menu items are called Module Actions. The module menu can be extended with your own custom actions. When your module inherits from the PortalModuleBase class, it receives a default set of actions, which are defined by the portal to handle common editing functions. Your module can extend these actions by implementing the IActionable interface.

Interface

As shown in Listing 8-18, the IActionable interface consists of a single method that returns a collection of Module Actions. The `ModuleActions` property is used when DotNetNuke renders the module.

Listing 8-18: IActionable Interface Definition

```
Namespace DotNetNuke.Entities.Modules
  Public Interface IActionable
    ReadOnly Property ModuleActions() As Actions.ModuleActionCollection
  End Interface
End Namespace
```

Listing 8-19 shows an example usage as implemented in the Announcements module. The first two lines tell the compiler that this method implements the `ModuleAction` method of the IActionable interface. It is a read-only method, so you only need to provide a `Get` function. The first step is to create a new collection to hold the custom actions. Then you use the collection's `Add` method to create a new action item in the collection. Finally, you return the new collection.

Listing 8-19: IActionable.ModuleActions Example

```
Public ReadOnly Property ModuleActions() As ModuleActionCollection _
  Implements IActionable.ModuleActions
  Get
    Dim Actions As New ModuleActionCollection

    Actions.Add(GetNextActionID, _
                Localization.GetString(ModuleActionType.AddContent, _
                                       LocalResourceFile), _
                ModuleActionType.AddContent, _
                "", _
                "", _
                EditUrl(), _
                False, _
                Security.SecurityAccessLevel.Edit, _
                True, _
                False)

    Return Actions
  End Get
End Property
```

This is a simple example that demonstrates the basic steps to follow for your own custom module menus. DotNetNuke provides extensive control over each Module Action.

ModuleAction API

To take full advantage of the power provided by Module Actions and the IActionable interface, you need to examine the classes, properties, and methods that make up the Module Action API.

Table 8-7 lists the classes that comprise the Module Action API.

Table 8-7: Module Action Classes

Class	Description
ModuleAction	Defines a specific function for a given module. Each module can define one or more actions that the portal will present to the user. Each module container can define the skin object used to render the Module Actions.
ModuleActionType	Defines a set of constants used for distinguishing common action types.
ModuleActionCollection	A collection of Module Actions.
ModuleActionEventListener	Holds callback information when a module registers for Action events.
ActionEventArgs	Passes data during the click event that is fired when a Module Action is selected by the user.
ActionEventHandler	A delegate that defines the method signature required for responding to the Action event.
ActionBase	Creates ModuleAction skin objects. The core framework includes three different implementations: SolPartActions.ascx, DropDownActions.ascx, and LinkActions.ascx.

The ModuleAction class is the heart of the API. Tables 8-8 and 8-9 show the properties and methods available in the ModuleAction class. Each item in the Module Action menu is represented by a single ModuleAction instance.

Table 8-8: ModuleAction Properties

Property	Type	Description
`Actions`	ModuleActionCollection	Contains the collection of Module Action items that can be used to form hierarchical menu structures. Every skin object that inherits from ActionBase may choose how to render the menu based on the capability to support hierarchical items. For example, the default SolpartActions skin object supports submenus, while the DropDownActions skin object only supports a flat menu structure.
`Id`	Integer	Every Module Action for a given module instance must contain a unique `Id`. The PortalModuleBase class defines the `GetNextActionId` method, which can be used to generate unique Module Action IDs.

Property	Type	Description
`CommandName`	String	Distinguishes which Module Action triggered an action event. DotNetNuke includes 19 standard ModuleActionTypes that provide access to standard functionality. Custom Module Actions can use their own string to identify commands recognized by the module.
`CommandArgument`	String	Provides additional information during action event processing. For example, the DotNetNuke core uses `CommandArgument` to pass the `ModuleID` for common commands like `DeleteModule.Action`.
`Title`	String	Sets the text that is displayed in the Module Action menu.
`Icon`	String	Name of the Icon file to use for the Module Action item.
`Url`	String	When set, this property allows a menu item to redirect the user to another web page.
`ClientScript`	String	JavaScript to run during the `menuClick` event in the browser. If the `ClientScript` property is present, it is called prior to the postback occurring. If the `ClientScript` returns `false`, the postback is canceled.
`UseActionEvent`	Boolean	Causes the portal to raise an Action Event on the server and notify any registered event listeners. If `UseActionEvent` is `false`, the portal handles the event, but does not raise the event back to any event listeners. The following CommandNames prevent the Action Event from firing: `ModuleHelp`, `OnlineHelp`, `ModuleSettings`, `DeleteModule`, `PrintModule`, `ClearCache`, `MovePane`, `MoveTop`, `MoveUp`, `MoveDown`, and `MoveBottom`.
`Secure`	SecurityAccessLevel	Determines the required security level of the user. If the current user does not have the necessary permissions, the Module Action is not displayed.
`Visible`	Boolean	If set to `false`, the Module Action will not be displayed. This property enables you to control the visibility of a Module Action based on custom business logic.
`NewWindow`	Boolean	Forces an action to open the associated URL in a new window. This property is not used if `UseActionEvent` is `true` or if the following CommandNames are used: `ModuleHelp`, `OnlineHelp`, `ModuleSettings`, or `PrintModule`.

Table 8-9: ModuleAction Method

Method	Return Type	Description
`HasChildren`	Boolean	Returns `true` if the `ModuleAction.Actions` property has any items (`Actions.Count > 0`).

DotNetNuke includes several standard Module Actions that are provided by the PortalModuleBase class or that are used by several of the core modules. These ModuleActionTypes are shown in Listing 8-20. ModuleActionTypes can also be used to access localized strings for the `ModuleAction.Title` property. This helps promote a consistent user interface for both core and third-party modules.

Listing 8-20: ModuleActionTypes

```
Public Class ModuleActionType
  Public Const AddContent As String = "AddContent.Action"
  Public Const EditContent As String = "EditContent.Action"
  Public Const ContentOptions As String = "ContentOptions.Action"
  Public Const SyndicateModule As String = "SyndicateModule.Action"
  Public Const ImportModule As String = "ImportModule.Action"
  Public Const ExportModule As String = "ExportModule.Action"
  Public Const OnlineHelp As String = "OnlineHelp.Action"
  Public Const ModuleHelp As String = "ModuleHelp.Action"
  Public Const PrintModule As String = "PrintModule.Action"
  Public Const ModuleSettings As String = "ModuleSettings.Action"
  Public Const DeleteModule As String = "DeleteModule.Action"
  Public Const ClearCache As String = "ClearCache.Action"
  Public Const MoveTop As String = "MoveTop.Action"
  Public Const MoveUp As String = "MoveUp.Action"
  Public Const MoveDown As String = "MoveDown.Action"
  Public Const MoveBottom As String = "MoveBottom.Action"
  Public Const MovePane As String = "MovePane.Action"
  Public Const MoveRoot As String = "MoveRoot.Action"
End Class
```

DotNetNuke provides standard behavior for the following ModuleActionTypes: `ModuleHelp`, `OnlineHelp`, `ModuleSettings`, `DeleteModule`, `PrintModule`, `ClearCache`, `MovePane`, `MoveTop`, `MoveUp`, `MoveDown`, and `MoveBottom`. All ModuleActionTypes in this subset will ignore the `UseActionEvent` and `NewWindow` properties. The ModuleActionTypes can be further subdivided into three groups:

- **Basic redirection**: The ModuleActionTypes that perform simple redirection and cause the user to navigate to the URL identified in the `URL` property: `ModuleHelp`, `OnlineHelp`, `ModuleSettings`, and `PrintModule`.
- **Module movement**: The ModuleActionTypes that change the order or location of modules on the current page: `MovePane`, `MoveTop`, `MoveUp`, `MoveDown`, and `MoveBottom`.

- **Custom logic**: The ModuleActionTypes with custom business logic that use core portal APIs to perform standard module-related actions: `DeleteModule` and `ClearCache`.

DotNetNuke uses a custom collection class for working with Module Actions. The ModuleActionCollection inherits from .Net System.Collections.CollectionBase class and provides a strongly typed collection class. That minimizes the possibility of typecasting errors that can occur when using generic collection classes such as ArrayList.

Most module developers only need to worry about creating the ModuleActionCollection to implement the IActionable interface. Listing 8-21 shows the two primary methods for adding ModuleActions to the collection. These methods wrap the ModuleAction constructor method calls.

Listing 8-21: Key ModuleActionCollection Methods

```
Public Function Add(ByVal ID As Integer, _
   ByVal Title As String, _
   ByVal CmdName As String, _
   Optional ByVal CmdArg As String = "", _
   Optional ByVal Icon As String = "", _
   Optional ByVal Url As String = "", _
   Optional ByVal UseActionEvent As Boolean = False, _
   Optional ByVal Secure As SecurityAccessLevel = SecurityAccessLevel.Anonymous, _
   Optional ByVal Visible As Boolean = True, _
   Optional ByVal NewWindow As Boolean = False) _
   As ModuleAction

Public Function Add(ByVal ID As Integer, _
    ByVal Title As String, _
    ByVal CmdName As String, _
    ByVal CmdArg As String, _
    ByVal Icon As String, _
    ByVal Url As String, _
    ByVal ClientScript As String, _
    ByVal UseActionEvent As Boolean, _
    ByVal Secure As SecurityAccessLevel, _
    ByVal Visible As Boolean, _
    ByVal NewWindow As Boolean) _
    As ModuleAction
```

The first method in Listing 8-21 uses optional parameters that are not supported by C#. This method is likely to be deprecated in future versions to simplify support for C# modules and its use is not recommended.

The ModuleAction framework makes it easy to handle simple URL redirection from a Module Action. Just like the `Delete` and `ClearCache` actions provided by the DotNetNuke framework, your module may require the use of custom logic to determine the appropriate action to take when the menu item is clicked. To implement custom logic, the module developer must create a response to a menu click event.

In the DotNetNuke architecture, the ModuleAction menu is a child of the module container. The module is also a child of the container. This architecture allows the framework to easily change out the menu implementation; however, it complicates communication between the menu and module. The menu never

has a direct reference to the module and the module does not have a direct reference to the menu. This is a classic example of the Mediator design pattern. This pattern is designed to allow two classes without direct references to communicate. Figure 8-3 shows the steps involved to implement this pattern.

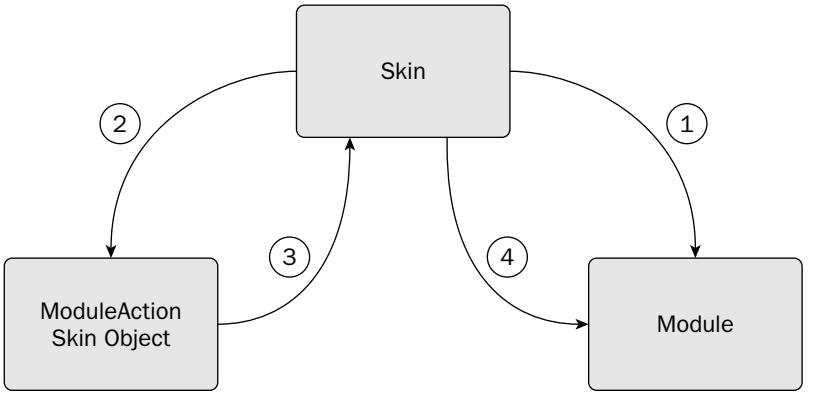

Figure 8-3

The following sections examine those steps and explore ways you can extend the framework.

Step 1: Register the Event Handler

The first step to implementing the Mediator pattern is to provide a mechanism for the module to register with the portal. The portal will use this information later when it needs to notify the module that a menu item was selected. Handling the click event is strictly optional. Your module may choose to use standard MenuActions, in which case you can skip this step. Because the module does not contain a direct reference to the page on which it is instantiated, you need to provide a registration mechanism.

The Skin class, which acts as the mediator, contains the `RegisterModuleActionEvent` method, which allows a module to notify the framework of the event handler for the action event (see Listing 8-22). Registration should occur in the module's `Page_Load` event to ensure that it happens before the event can be fired in the Skin class. The code in Listing 8-22 is from the HTML module and provides a working example of module-based event registration for the `ModuleAction` event. Although you could use another interface to define a known method to handle the event, the registration mechanism turns out to be a much more flexible design when implementing a single method.

Listing 8-22: Registering an Event Handler

```
'-------------------------------------------------------------------------------
'-                       Menu Action Handler Registration                      -
'-------------------------------------------------------------------------------
'This finds a reference to the containing skin
Dim ParentSkin As UI.Skins.Skin = UI.Skins.Skin.GetParentSkin(Me)

'We should always have a ParentSkin, but need to make sure
If Not ParentSkin Is Nothing Then

    'Register our EventHandler as a listener on the ParentSkin so that it may
```

```
        'tell us when a menu has been clicked.
        ParentSkin.RegisterModuleActionEvent(Me.ModuleId, AddressOf ModuleAction_Click)
    End If
    '-----------------------------------------------------------------------------
```

Listing 8-23 shows the `ModuleAction_Click` event handler code from the HTML module.

Listing 8-23: Handling the Event

```
Public Sub ModuleAction_Click(ByVal sender As Object, _
                              ByVal e As Entities.Modules.Actions.ActionEventArgs)

  'We could get much fancier here by declaring each ModuleAction with a
  'Command and then using a Select Case statement to handle the various
  'commands.
  If e.Action.Url.Length > 0 Then
    Response.Redirect(e.Action.Url, True)
  End If

End Sub
```

The DotNetNuke framework uses a delegate (see Listing 8-24) to define the method signature for the event handler. The `RegisterModuleActionEvent` requires the address of a method with the same signature as the `ActionEventHandler` delegate.

Listing 8-24: ActionEventHandler Delegate

```
Public Delegate Sub ActionEventHandler(ByVal sender As Object, _
                                       ByVal e As ActionEventArgs)
```

Step 2: Display the Menu

Now that the skin (the Mediator class) can communicate with the module, you need a mechanism to allow the menu to communicate with the skin as well. This portion of the communication chain is much easier to code. Handling the actual click event and passing it to the skinning class is the responsibility of the ModuleAction rendering code.

Like much of DotNetNuke, the ModuleAction framework supports the use of custom extensions. In this case, skin objects handle rendering the Module Actions. Each ModuleAction skin object inherits from the DotNetNuke.UI.Containers.ActionBase class. The Skin class retrieves the Module Action collection from the module by calling the `IActionable.ModuleActions` property and passes the collection to the ModuleAction skin object for rendering. The ActionBase class includes the code necessary to merge the standard Module Actions with the collection provided by the Skin class.

Each skin object includes code in the pre-render event to convert the collection of Module Actions into an appropriate format for display using an associated server control. In the case of SolPartActions.ascx, the server control is a menu control that is capable of fully supporting all of the features of ModuleActions including submenus and icons. Other skin objects like the DropDownActions.ascx may only support a subset of the Module Action features (see Table 8-10).

Table 8-10: ModuleAction Skin Objects

Action Skin Object	Menu Separator	Icons	Submenus	Client-Side JavaScript
Actions or SolPartActions	Yes	Yes	Yes	Yes
DropDownActions	Yes	No	No	Yes
LinkActions	No	No	No	No

Step 3: Notify the Portal of a Menu Item Selection

Each skin object handles the click event of the associated server control. This event, shown in Listing 8-25, calls the `ProcessAction` method, which is inherited from the ActionBase class. `ProcessAction` is then responsible for handling the event as indicated by the ModuleAction properties. If you create your own ModuleAction skin object, follow this pattern.

Listing 8-25: Click Event Handler

```
Private Sub ctlActions_MenuClick(ByVal ID As String) Handles ctlActions.MenuClick
  Try
    ProcessAction(ID)
  Catch exc As Exception    'Module failed to load
    ProcessModuleLoadException(Me, exc)
  End Try
End Sub
```

Step 4: Notify the Module That a Custom ModuleAction Was Clicked

If the `UseActionEvent` is set to `True`, the `ProcessAction` method (see Listing 8-26) calls the `OnAction` method to handle actually raising the event (see Listing 8-27). This might seem like an extra method call when `ProcessAction` could just raise the event on its own. The purpose of `OnAction`, though, is to provide an opportunity for subclasses to override the default event handling behavior. Although this is not strictly necessary, it is a standard pattern in .NET and is a good example to follow when developing your own event handling code.

Listing 8-26: ProcessAction Method

```
Public Sub ProcessAction(ByVal ActionID As String)
  If IsNumeric(ActionID) Then
    Dim action As ModuleAction = GetAction(Convert.ToInt32(ActionID))
    Select Case action.CommandName
      Case ModuleActionType.ModuleHelp
        DoAction(action)
      Case ModuleActionType.OnlineHelp
        DoAction(action)
      Case ModuleActionType.ModuleSettings
        DoAction(action)
      Case ModuleActionType.DeleteModule
        Delete(action)
      Case ModuleActionType.PrintModule
```

```
            DoAction(action)
          Case ModuleActionType.ClearCache
            ClearCache(action)
          Case ModuleActionType.MovePane
            MoveToPane(action)
          Case ModuleActionType.MoveTop, _
               ModuleActionType.MoveUp, _
               ModuleActionType.MoveDown, _
               ModuleActionType.MoveBottom
            MoveUpDown(action)
          Case Else
            ' custom action
            If action.Url.Length > 0 And action.UseActionEvent = False Then
              DoAction(action)
            Else
              ModuleConfiguration))
            End If
        End Select
      End If
    End Sub
```

Listing 8-27: OnAction Method

```
    Protected Overridable Sub OnAction(ByVal e As ActionEventArgs)
      RaiseEvent Action(Me, e)
    End Sub
```

Because the skin maintains a reference to the ModuleAction skin object, the Skin class can handle the `Action` event raised by the skin object. As shown in Listing 8-28, the Skin class iterates through `ActionEventListeners` to find the associated module event delegate. When a listener is found, the code invokes the event, which notifies the module that the event has occurred.

Listing 8-28: Skin Class Handles the ActionEvent

```
    Public Sub ModuleAction_Click(ByVal sender As Object, ByVal e As ActionEventArgs)
      'Search through the listeners
      Dim Listener As ModuleActionEventListener
      For Each Listener In ActionEventListeners

        'If the associated module has registered a listener
        If e.ModuleConfiguration.ModuleID = Listener.ModuleID Then

          'Invoke the listener to handle the ModuleAction_Click event
          Listener.ActionEvent.Invoke(sender, e)
        End If
      Next
    End Sub
```

You are now ready to take full advantage of the entire ModuleAction API to create custom menu items for your own modules, handle the associated Action event when the menu item is clicked, and create your own custom ModuleAction skin objects.

IPortable

DotNetNuke provides the capability to import and export modules within the portal. Like many features in DotNetNuke, it is implemented using a combination of core code and module-specific logic. The IPortable interface defines the methods required to implement this feature on a module-by-module basis (see Listing 8-29).

Listing 8-29: IPortable Interface Definition

```
Public Interface IPortable
  Function ExportModule(ByVal ModuleID As Integer) As String

  Sub ImportModule(ByVal ModuleID As Integer, _
                   ByVal Content As String, _
                   ByVal Version As String, _
                   ByVal UserID As Integer)
End Interface
```

This interface provides a much-needed feature to DotNetNuke and is a pretty straightforward interface to implement. To fully support importing and exporting content, implement the interface within your module's business controller class.

As modules are being loaded by the portal for rendering a specific page, they are checked to determine whether they implement the IPortable interface. To simplify checking whether a module implements the interface, a shortcut property has been added to the ModuleInfo class. The ModuleInfo class provides a consolidated view of properties related to a module. When a module is first installed in the portal, a quick check is made to determine if the module implements the IPortable interface, and if so, the `IsPortable` flag is set on the base ModuleDefinition record. This property allows the portal to perform the interface check without unnecessarily loading the business controller class. Adding the check at the point of installation removes a requirement by previous DotNetNuke versions for a module control to implement unused stub methods. If the control implements the IPortable interface, DotNetNuke automatically adds the Import Content and Export Content menu items to your Module Action menu (see Figure 8-4).

Figure 8-4

Each module should include a controller class that is identified in the `BusinessControllerClass` property of the portal's ModuleInfo class. This class is identified in the module manifest file discussed later in the book. The controller class is where you implement many of the interfaces available to modules.

Adding the IPortable interface to your module requires implementing logic for the `ExportModule` and `ImportModule` methods shown in Listing 8-30 and Listing 8-31, respectively.

Listing 8-30: ExportModule Stub

```
Public Function ExportModule(ByVal ModuleID As Integer) As String _
  Implements Entities.Modules.IPortable.ExportModule

  Dim strXML As String = ""

  Dim objHtmlText As HtmlTextInfo = GetHtmlText(ModuleID)
  If Not objHtmlText Is Nothing Then
    strXML += "<htmltext>"
    strXML += "<desktophtml>{0}</desktophtml>"
    strXML += "<desktopsummary>{1}</desktopsummary>"
    strXML += "</htmltext>"

    String.Format(strXML, _
                  XMLEncode(objHtmlText.DeskTopHTML), _
                  XMLEncode(objHtmlText.DesktopSummary))
  End If

  Return strXML

End Function
```

Listing 8-31: ImportModule Stub

```
Public Sub ImportModule(ByVal ModuleID As Integer, _
                        ByVal Content As String, _
                        ByVal Version As String, _
                        ByVal UserId As Integer) _
  Implements Entities.Modules.IPortable.ImportModule

  Dim xmlHtmlText As XmlNode = GetContent(Content, "htmltext")

  Dim objText As HtmlTextInfo = New HtmlTextInfo

  objText.ModuleId = ModuleID
  objText.DeskTopHTML = xmlHtmlText.SelectSingleNode("desktophtml").InnerText
  objText.DesktopSummary = xmlHtmlText.SelectSingleNode("desktopsummary").InnerText
  objText.CreatedByUser = UserId
  AddHtmlText(objText)

End Sub
```

The complexity of the data model for your module determines the difficulty of implementing these methods. Take a look at a simple case as implemented by the HTMLText module.

In Listing 8-30, the `ExportModule` method is used to serialize the content of the module to an XML string. DotNetNuke saves the serialized string along with the module's FriendlyName and Version. The XML file is saved into the portal directory.

The `ImportModule` method in Listing 8-31 reverses the process by deserializing the XML string created by the `ExportModule` method and replacing the content of the specified module. The portal passes the version information stored during the export process along with the serialized XML string.

The IPortable interface is straightforward to implement and provides much needed functionality to the DotNetNuke framework. It is at the heart of DotNetNuke's templating capability and therefore is definitely an interface that all modules should implement.

IUpgradeable

One of DotNetNuke's greatest features is the capability to easily upgrade from one version to the next. The heart of that is the creation of script files that can be run sequentially to modify the database schema and move any existing data to the new version's schema. In later versions, DotNetNuke added a mechanism for running custom logic during the upgrade process. Unfortunately, this mechanism was not provided for modules. Therefore, third-party modules were forced to create their own mechanism for handling custom upgrade logic.

This was fixed in DotNetNuke 3.0 and updated again in 4.1. The IUpgradeable interface (see Listing 8-32) provides a standard upgrade capability for modules, and uses the same logic as used in the core framework. The interface includes a single method, `UpgradeModule`, which enables the module to execute custom business logic depending on the current version of the module being installed.

Listing 8-32: IUpgradeable Interface

```
Public Interface IUpgradeable
  Function UpgradeModule(ByVal Version As String) As String
End Interface
```

`UpgradeModule` is called once for each script version included with the module. It is called only for script versions that are later than the version of the currently installed module.

> **Due to the behavior of ASP.NET when a new assembly is added to the \bin directory, the IUpgradeable interface could fail during installation. This behavior has been corrected in the 3.3 and 4.1 releases. If your module needs this interface for proper installation, have your users upgrade to the latest version of DotNetNuke.**

Inter-Module Communication

DotNetNuke includes the capability for modules to communicate with each other through the Inter-Module Communication (IMC) framework. The IMC framework enables modules to pass objects rather than simple strings. Additionally, other properties enable a module to identify the Sender, the Target, and the Type of message. Take a look at the two main interfaces that provide this functionality to your module: IModuleCommunicator and IModuleListener.

IModuleCommunicator

The IModuleCommunicator interface defines a single event, ModuleCommunication, for your module to implement (see Listing 8-33).

Listing 8-33: IModuleCommunicator Interface

```
Public Interface IModuleCommunicator
  Event ModuleCommunication As ModuleCommunicationEventHandler
End Interface
```

To communicate with another module, first implement the IModuleCommunicator interface in your module. You should have an event declaration in your module as shown in Listing 8-34.

Listing 8-34: ModuleCommunication Event Implementation

```
Public Event ModuleCommunication(ByVal sender As Object, _
                                 ByVal e As ModuleCommunicationEventArgs) _
                                 Implements IModuleCommunicator.ModuleCommunication
```

IModuleListener

Whereas the IModuleCommunicator is used for sending messages, the IModuleListener interface (see Listing 8-35) is used for receiving messages.

Listing 8-35: IModuleListener Interface

```
Public Interface IModuleListener
  Sub OnModuleCommunication(ByVal s As Object, _
                            ByVal e As ModuleCommunicationEventArgs)
End Interface
```

This interface defines a single method, `OnModuleCommunication`, which is called when an IModuleCommunicator on the same page raises the ModuleCommunication event. What you do in response to this event notification is totally up to you.

> *DotNetNuke does not filter event messages. Any module that implements the IModuleListener interface is notified when the event is raised. It is the responsibility of the module to determine whether it should take any action.*

ISearchable

DotNetNuke provides a robust search API for indexing and searching content in your portal. The API is divided into three distinct parts:

- Core search engine
- Search data store
- Search indexer

Like the ModuleAction framework, the search framework also implements a Mediator pattern. When combined with the Provider pattern, this framework provides lots of flexibility. In Figure 8-5, you can see the relationship between these patterns and the three parts of the search API.

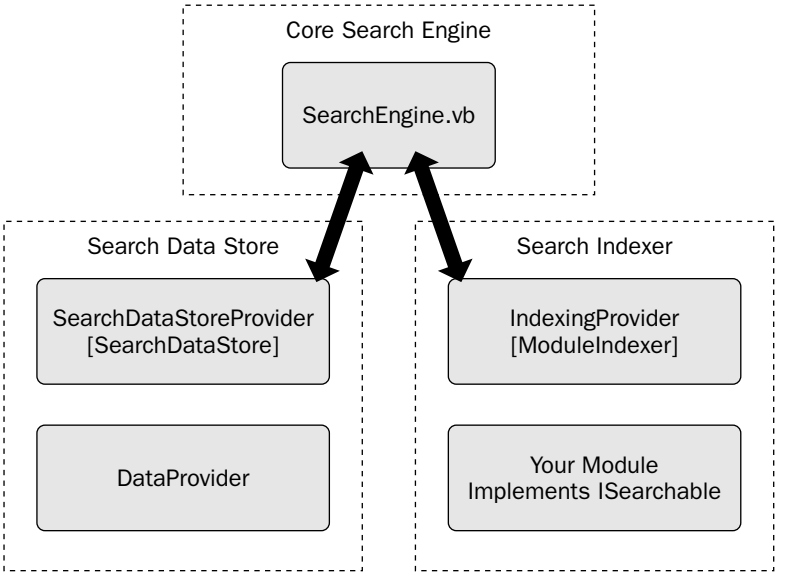

Figure 8-5

The core search engine provides a simple API for calling the IndexProvider and then storing the results using a SearchDataStoreProvider. This API is intended for use by the core framework. Future versions of the API will be extended to allow modules greater control over the indexing process.

DotNetNuke includes a default implementation of the SearchDataStoreProvider, which is meant to provide basic storage functionality, but could be replaced with a more robust search engine. As for other providers, third-party developers are implementing providers for many of the current search engines on the market. You can find links to these providers and more at `www.dotnetnuke.com` and in the DotNetNuke Marketplace at `http://marketplace.dotnetnuke.com`.

The IndexingProvider provides an interface between the core search engine and each module. DotNetNuke includes a default provider that indexes module content. This provider can be replaced to provide document indexing, web indexing, or even indexing legacy application content stored in another database. If you decide to replace it, keep in mind that DotNetNuke only allows for the use of a single provider of a given type. This means that if you want to index content from multiple sources, you must implement this as a single provider. Future versions of the framework may be enhanced to overcome this limitation.

When using the ModuleIndexer, you can incorporate a module's content into the search engine data store by implementing the ISearchable interface shown in Listing 8-36.

Listing 8-36: ISearchable Interface

```
Public Interface ISearchable
  Function GetSearchItems(ByVal ModInfo As ModuleInfo) As SearchItemInfoCollection
End Interface
```

This interface is designed to allow almost any content to be indexed. By passing in a reference to the module and returning a collection of SearchItems, the modules are free to map their content to each SearchItem as they see fit. Listing 8-37 shows a sample implementation from the Announcements module included with DotNetNuke.

Listing 8-37: Implementing the Interface

```
Public Function GetSearchItems(ByVal ModInfo As Entities.Modules.ModuleInfo) _
  As Services.Search.SearchItemInfoCollection _
  Implements Services.Search.ISearchable.GetSearchItems
    Dim SearchItemCollection As New SearchItemInfoCollection

    Dim Announcements As ArrayList = GetAnnouncements(ModInfo.ModuleID)

    Dim objAnnouncement As Object
    For Each objAnnouncement In Announcements
      Dim SearchItem As SearchItemInfo
      With CType(objAnnouncement, AnnouncementInfo)

        Dim UserId As Integer
        If IsNumeric(.CreatedByUser) Then
          UserId = Integer.Parse(.CreatedByUser)
        Else
          UserId = 2
        End If
        SearchItem = New SearchItemInfo(ModInfo.ModuleTitle & " - " & .Title, _
                     ApplicationURL(ModInfo.TabID), _
                     .Description, _
                     UserId, _
                     .CreatedDate, _
                     ModInfo.ModuleID, _
                     "Anncmnt" & ModInfo.ModuleID.ToString & "-" & .ItemId, _
                     .Description)
        SearchItemCollection.Add(SearchItem)
      End With
    Next

    Return SearchItemCollection
End Function
```

In this code, you make a call to your module's Info class, just as you would when you bind to a control within your ASCX file, but in this case the results are going to populate the SearchItemInfo, which will populate the DNN index with data from the module.

The key to implementing the interface is figuring out how to map your content to a collection of SearchItem Info objects. Table 8-11 lists the properties of the SearchItemInfo class.

Table 8-11: SearchItemInfo Properties

Property	Description
`SearchItemId`	An ID assigned by the search engine. It's used when deleting items from the data store.
`Title`	A string that is used when displaying search results.
`Description`	Summary of the content and is used when displaying search results.
`Author`	Content author.
`PubDate`	Date that allows the search engine to determine the age of the content.
`ModuleId`	ID of the module whose content is being indexed.
`SearchKey`	Unique key that can be used to identify each specific search item for this module.
`Content`	The specific content that will be searched. The default search data store does not search on any words that are not in the content property.
`GUID`	Another unique identifier that is used when syndicating content in the portal.
`ImageFileId`	Optional property used to identify image files that accompany a search item.
`HitCount`	Maintained by the search engine and used to identify the number of times that a search item is returned in a search.

Now that the index is populated with data, users of your portal can search your module's information from a unified interface within DNN.

Summary

This chapter examined many of the core APIs that provide the true power behind DotNetNuke. By leveraging common APIs, you can extend the portal in almost any direction. You can replace core functions or just add a custom module — the core APIs are what make it all possible. Now that you know how to use most of the core functions, the next several chapters examine how to create your own custom modules to really take advantage of this power.