DotNetNuke Module Developers Guide

Patrick Santry

Version 1.0.12
Last Updated: June 20, 2006
Category: Modules

DotNetNuke Module Developers Guide

Abstract

In order to clarify the intellectual property license granted with contributions of software from any person or entity (the "Contributor"), Perpetual Motion Interactive Systems Inc. must have a Contributor License Agreement on file that has been signed by the Contributor.

Contents

DotNetNuke Module Developers Guide

Introduction

The purpose of this guide to assist the developer in extending DotNetNuke (DNN) by developing modules that plug into the core framework of DNN.

In this guide we will cover every stage of developing modules, from creating your Visual Studio.NET project, to creating a distribution package for installing your new module in a portal site. The goal of this guide is to get you started developing modules in a short amount of time. Initially, we want to go through some basic functionality of a module that your user would see in order to introduce you to what a user would experience in your DNN portal, and then get into some architectural and coding concepts for you to get started on developing your own modules.

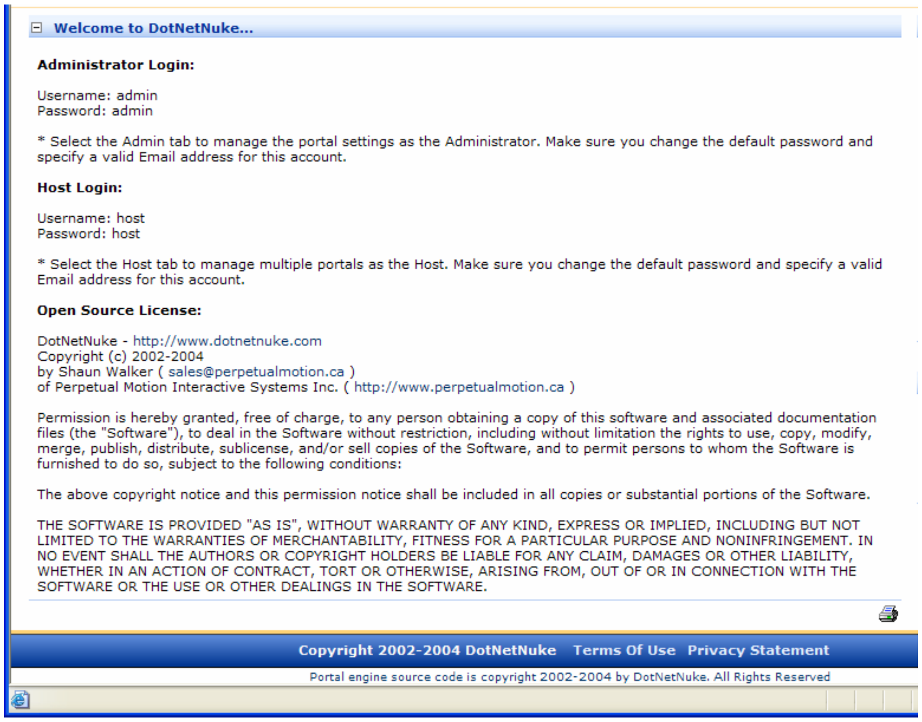
Throughout this guide we will refer to actual examples that are included as parts of the main DotNetNuke distribution file that can be downloaded from http://www.dotnetnuke.com. The examples we will focus on for the majority of the guide will be the Survey module. We also pull code from other items included within DotNetNuke, this is so all code listed within this guide is easily accessible to you.

What Are Modules?

Modules provide developers with the ability to extend the functionality of DotNetNuke. DotNetNuke provides a pluggable framework that can be expanded by the development of modules. A module container is provided by DotNetNuke to host a custom module. Modules can be developed in any .NET language, even though DNN is developed in VB.NET, a C# developer can still create a module that plugs into the core framework provided by DNN. This pluggable framework is accomplished by creating compiled private assemblies that expose and utilize interfaces specific to DNN. Once you compile the assembly, and then just create a user interface (UI) in the form of ascx files that allow your user to interact with your module.

Modules provide you with maximum code reusability; you can key off the module ID value (provided by the DNN framework) which allows you to display unique data for each implementation of your module.

From the user's perspective a module is an area of functionality within their DNN installation. For example, in Figure 1, you see an example of the Text/HTML module that comes standard with DNN. This module provides the user with the ability to edit content inline via their Web browser. It provides several functions such as basic text editing and formatting, and publishing of content.



Welcome to DotNetNuke...

Administrator Login:

Username: admin
Password: admin

* Select the Admin tab to manage the portal settings as the Administrator. Make sure you change the default password and specify a valid Email address for this account.

Host Login:

Username: host
Password: host

* Select the Host tab to manage multiple portals as the Host. Make sure you change the default password and specify a valid Email address for this account.

Open Source License:

DotNetNuke - http://www.dotnetnuke.com
Copyright (c) 2002-2004
by Shaun Walker (sales@perpetualmotion.ca)
of Perpetual Motion Interactive Systems Inc. (http://www.perpetualmotion.ca)

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

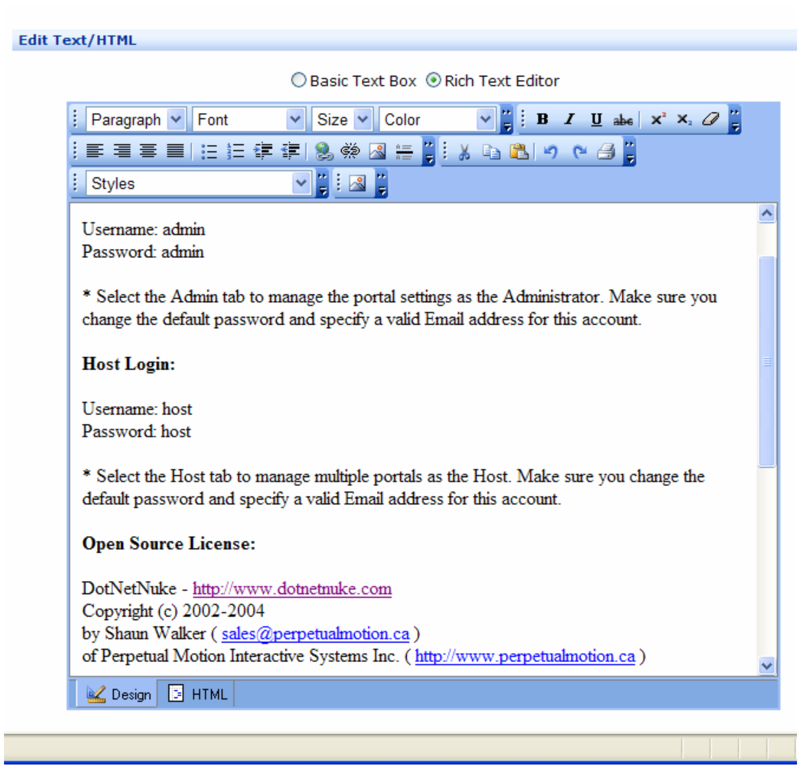
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Copyright 2002-2004 DotNetNuke Terms Of Use Privacy Statement

Portal engine source code is copyright 2002-2004 by DotNetNuke. All Rights Reserved

Figure 1 – User view of the Text/HTML module.

In this specific module example, we have two distinct views or behaviors of this module. In figure 1 we have a user view, which is the HTML that is the final presentation, and an edit view (see Figure 2), which is where the portal administrator, or content author can edit content directly on the portal. For example, a portal administrator can configure a role that has permissions to edit the content within the Text/HTML module, and once a user logs onto the portal which is part of that configured role, they will be presented with a menu that provides various options like editing (see Figure 3). The user can then select from the options of the menu to edit the content.



Username: admin
Password: admin

* Select the Admin tab to manage the portal settings as the Administrator. Make sure you change the default password and specify a valid Email address for this account.

Host Login:

Username: host
Password: host

* Select the Host tab to manage multiple portals as the Host. Make sure you change the default password and specify a valid Email address for this account.

Open Source License:

DotNetNuke - http://www.dotnetnuke.com
Copyright (c) 2002-2004
by Shaun Walker (sales@perpetualmotion.ca)
of Perpetual Motion Interactive Systems Inc. (http://www.perpetualmotion.ca)

Figure 2 – The edit view of the Text/HTML module.

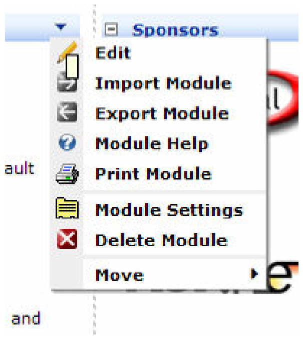


Figure 3 – Menu Options for the Text/HTML Module

Another menu item that you should be aware of is the "Module Settings". Module Settings is a global module menu item that provides you with the ability to modify some behaviors of the module container. The module container provides you with options for configuring security, the look, and caching for the module. You'll also notice within the menu items that you can position, import, and export modules.

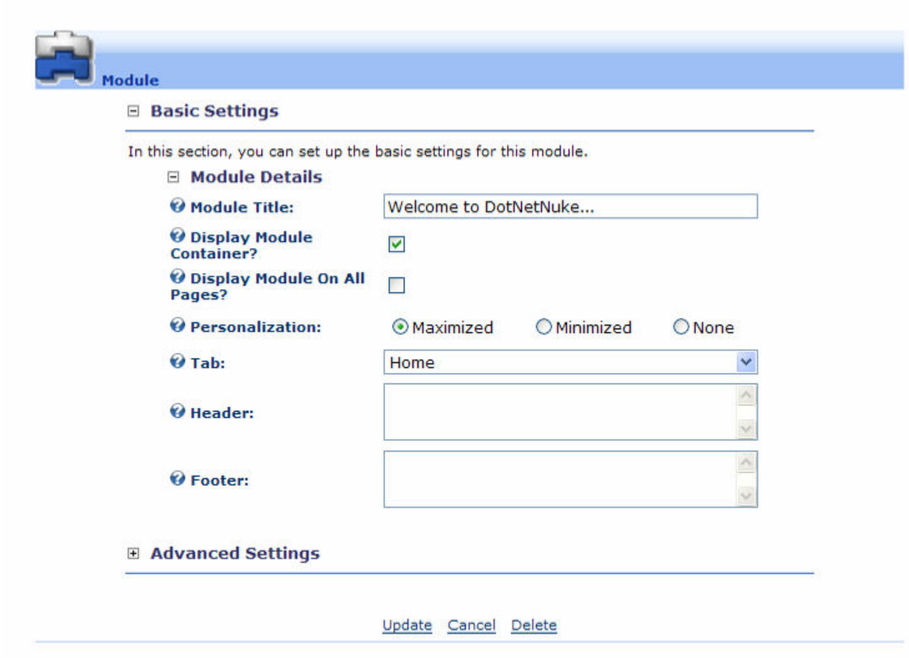
Module Containers

Module containers are the host container or interface for your application within DNN which gives you the control to position, change the look, and the other functionality that you see listed in the menu options. Let's cover the features provided by the container. First select "Module Settings" from the module menu.

In Module Settings you will see two areas:

Basic Settings (see Figure 4)
Provides your module with the following options:

- **Module Title** – To be displayed at the top of the module container.
- **Display Module Container** – Select this option if you wish to display the graphical element of the module container which surrounds the module.
- **Display Module on All Pages** – Use this option if you want the module to be a global module and appear on all pages within your DNN portal.
- **Personalization** – Select this option to allow your users to customize the appearance of the module display on the page, for example minimize and maximize.
- **Tab** – Select the current page that the module will be hosted on.
- **Header** – In addition to the Title, you can display some header text to be shown at the top of the module.
- **Footer** – Enter in text for display as a footer for this module implementation.



Module

Basic Settings

In this section, you can set up the basic settings for this module.

Module Details

Module Title:	Welcome to DotNetNuke...
Display Module Container?	✓
Display Module On All Pages?	☐
Personalization:	[✓ Maximized] Minimized None
Tab:	Home
Header:	
Footer:	

Advanced Settings

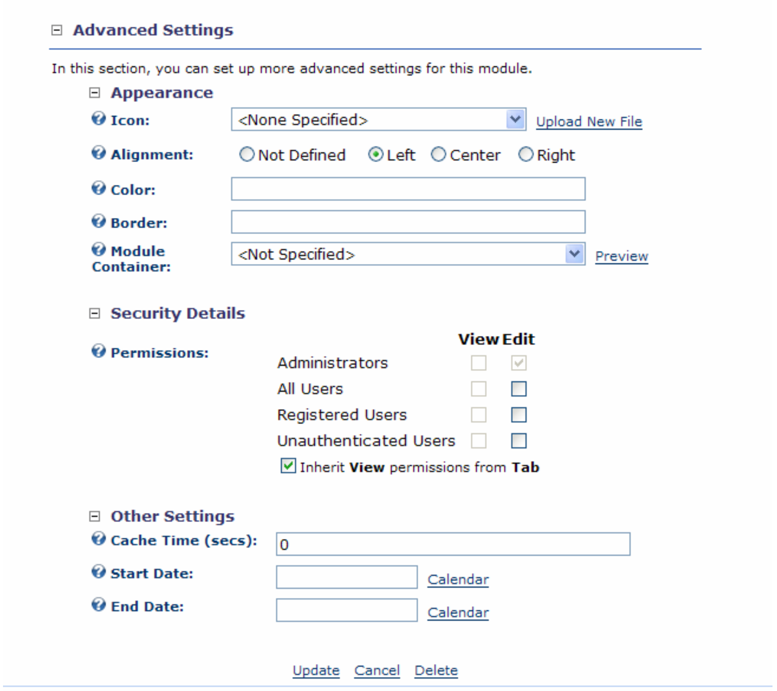
Figure 4 – Basic Settings for a Module

Advanced Settings
In addition to these basic settings DNN offers some advanced settings which govern the look and feel of the module implementation. Click on the plus sign (⊞) on the left side of the heading to extend the Advanced Settings area (see Figure 5). You will then be able to modify the following settings:

- Icon – Select an icon to be displayed next to the Module Title (you must first upload a file via the File Manager in order for it to be displayed in the drop down list)
- Alignment – Select how you wish to align the module, this affects the HTML tags by changing the align attribute for the HTML table cell tag which surrounds your module interface.
- Color – Changes the background color attribute for the table cell which contains the module control.
- Border – Changes the border color attribute for the table cell containing the module control.
- Module Container – This drop down list contains module containers installed within DNN. This module container is the graphical portion of the module and can be designed using a standard editor. Module containers can provide a separation of the look from the functionality. In this guide we are covering developing modules, for

more information on creating the container files refer to the skinning guide in the DNN distribution.

- Security Details – DNN provides your modules with a security wrapper, here you can define which roles have edit permissions for your module and which roles have view permissions for your module. By default permissions are inherited from the parent tab or page.
- Cache Time – If you module does not need to connect to a database on every request then you can specify a cache timeout. Use this for applications that provide somewhat static data, it can increase performance of your portal by reducing calls over the network to the database.
- Start Date – Allows you to set a publication date for the module. Specify a date when the module will first be displayed.
- End Date – An expiration date for display of the module. Enter in a date for this module to stop being displayed on the page.



Advanced Settings

In this section, you can set up more advanced settings for this module.

Appearance

Icon:	<None Specified> Upload New File
Alignment:	Not Defined [✓ Left] Center Right
Color:	
Border:	
Module Container:	<Not Specified> Preview

Security Details

Permissions:

	View	Edit
Administrators	☐	✓
All Users	☐	☐
Registered Users	☐	☐
Unauthenticated Users	☐	☐

✓ Inherit **View** permissions from **Tab**

Other Settings

Cache Time (secs):	0
Start Date:	Calendar
End Date:	Calendar

Figure 5 - Advanced Settings for a Module

Positioning Modules

The module container provides the editor with the ability to position the module with various panes on a DNN page. A pane is an area specified by a skin designer when designing skins for a DNN portal, typically a pane will be defined within an HTML table cell "<td>" tag. When panes are defined with the skin they will be listed within the menu options for the module as in Figure 6. A skin designer can create as many panes they wish within a page for module placement.

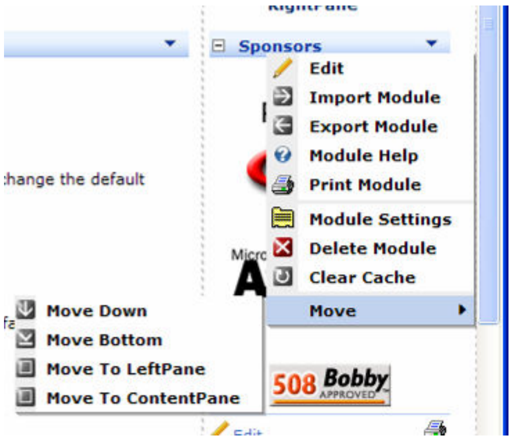


Figure 6 – Position options with a module's menu.

Interfacing a Module with DotNetNuke

Now that we have covered the basic features of a module, we need to understand how a module interfaces with DNN. In this section we will cover how to manually interface a module within DNN. In a subsequent section on deployment we will cover how to create a definition file which automates the install process of interfacing with DNN.

Importing and Exporting Modules

A new feature in DNN 3 is the ability to import and export modules. This allows you to import content of another module of the same type. As mentioned previously a module provides reusability by keying off a Module ID value provided by the DNN framework, the import/export features of DNN duplicates the data for the module by copying and replacing the old Module ID value with the new Module ID value of the target container.

The process of exporting a module is pretty simple; simply select the Export module option with the module's menu list (see Figure 6). You will be presented with a link button with the value of "Export" just click on the link button to save this export definition within DNN.

To import the previously exported module data, just create a new instance of the module within a page, from here select import. You will then be presented with a screen containing a drop down list of all previously exported modules. Select the module data you wish to import into this module instance, and click on the "Import" link button. Later in this guide we'll discuss how you can implement this in your own modules.

DotNetNuke Architecture

In this section we want to provide some basic concepts on the DNN architecture and how it affects you as a module developer. Before you begin development we want to make sure you are familiar with some of the architectural concepts that comprise the DNN framework.

The following diagram (see Figure 7) is taken from the DotNetNuke documentation. This provides a look into how DNN is architected. DNN uses a three-tier architectural approach, first with the ascx controls which provide the user interface. The interface then communicates with the Business Logic Layer (BLL), as in the diagram this BLL provides all data for our user interface.

Below the BLL we have the Abstract Data Provider; this class provides an abstraction layer for our application. This provider is not database specific; rather our data provider class will provide methods that override the abstraction class and interaction with our specific database.

Below the abstraction layer we have our Data Access Layer, this class is specific to a vendor database and is unique based on what database we want our module to interact with. This class is the SQLDataProvider project as in the image above. Finally there is the Microsoft.ApplicationBlocks.Data which provides functions for our specific database interaction, and frees a developer from having to write specific SQL Server code.

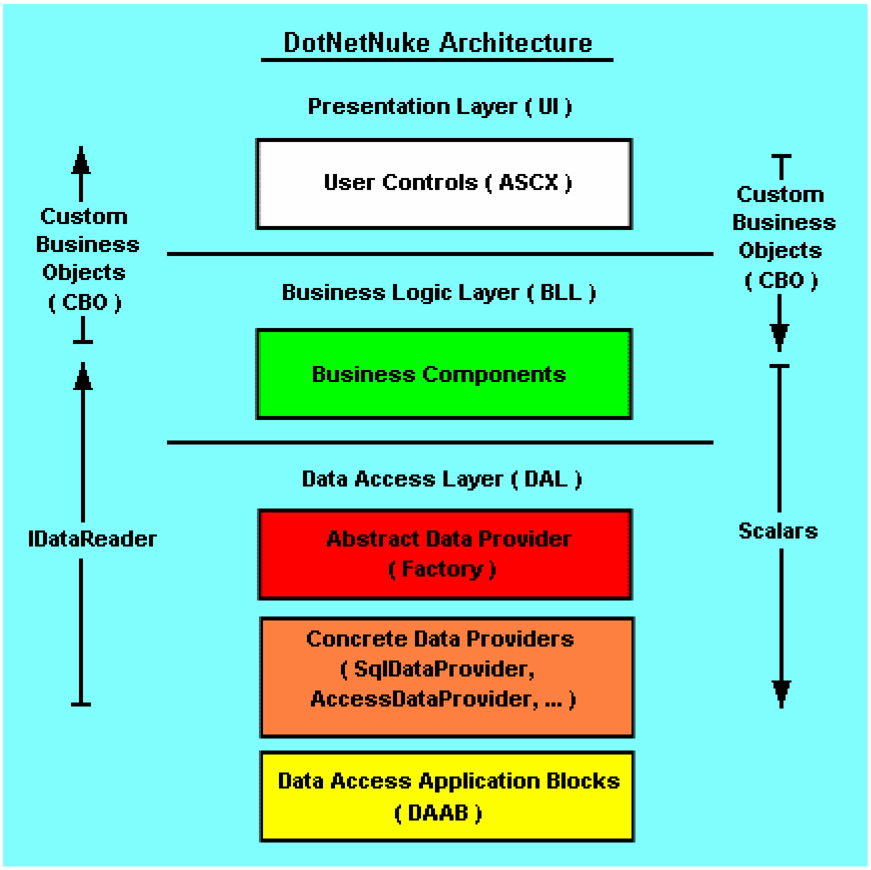

DotNetNuke Architecture
Presentation Layer (UI)
User Controls (ASCX)
Custom Business Objects (CBO)
Custom Business Objects (CBO)
Business Logic Layer (BLL)
Business Components
Data Access Layer (DAL)
IDataReader
Abstract Data Provider (Factory)
Scalars
Concrete Data Providers (SqlDataProvider, AccessDataProvider, ...)
Data Access Application Blocks (DAAB)

Figure 7 - DotNetNuke Application Architecture

Provider Model

DNN uses the provider model extensively. A provider pattern enables you to abstract out certain functionality. For example in the architectural diagram (see Figure x) you can see how there is a Abstract Data Provider between the Business Components, and the Concrete Data Provider, this enables you to plug in different providers without having to modify or recompile the core code within DNN.

DNN uses a provider pattern extensively in it's architecture. We talked about the database provider, but several other features use this pattern as well.

- Security and Membership Provider
- Text/HTML Provider
- Logging Provider

- Scheduler
- Friendly URLs

As mentioned, provider patterns enable you to abstract out certain functions, but the actual physical providers are then defined in the web.config. For example, in DNN if you open the web.config you will see a section for the providers:

```
<dotnetnuke>
        <htmlEditor defaultProvider="Ftb3HtmlEditorProvider">
                <providers>
                        <clear />
                        <add name="FtbHtmlEditorProvider"
                        type="DotNetNuke.HtmlEditor.FtbHtmlEditorProvider,
DotNetNuke.FtbHtmlEditorProvider"
providerPath="~\Providers\HtmlEditorProviders\FtbHtmlEditorProvider\" />
                        <add name="Ftb3HtmlEditorProvider"
 type="DotNetNuke.HtmlEditor.Ftb3HtmlEditorProvider,
DotNetNuke.Ftb3HtmlEditorProvider"
ProviderPath="~\Providers\HtmlEditorProviders\Ftb3HtmlEditorProvider\" />
                </providers>
        </htmlEditor>
        <searchIndex defaultProvider="ModuleIndexProvider">
                <providers>
                        <clear />
                        <add name="ModuleIndexProvider"
type="DotNetNuke.Services.Search.ModuleIndexer, DotNetNuke.Search.Index"
providerPath="~\Providers\SearchProviders\ModuleIndexer\" />
                </providers>
        </searchIndex>
        <searchDataStore defaultProvider="SearchDataStoreProvider">
                <providers>
                        <clear />
                        <add name="SearchDataStoreProvider"
type="DotNetNuke.Services.Search.SearchDataStore,
DotNetNuke.Search.DataStore"
providerPath="~\Providers\SearchProviders\SearchDataStore\" />
                </providers>
        </searchDataStore>
        <data defaultProvider="SqlDataProvider">
                <providers>
                        <clear />
                        <add name="SqlDataProvider"
type="DotNetNuke.Data.SqlDataProvider, DotNetNuke.SqlDataProvider"
```

```
connectionStringName="SiteSqlServer"
upgradeConnectionString=""
providerPath="~\Providers\DataProviders\SqlDataProvider\"
objectQualifier=""
databaseOwner="dbo" />
                </providers>
        </data>
        <logging defaultProvider="XMLLoggingProvider">
                <providers>
                        <clear />
                        <add name="XMLLoggingProvider"
type="DotNetNuke.Services.Log.EventLog.XMLLoggingProvider,
DotNetNuke.XMLLoggingProvider"
configfilename="LogConfig.xml.resources"
providerPath="~\Providers\LoggingProviders\XMLLoggingProvider\" />
                </providers>
        </logging>
        <scheduling defaultProvider="DNNScheduler">
                <providers>
                        <clear />
                        <add name="DNNScheduler"
type="DotNetNuke.Services.Scheduling.DNNScheduling.DNNScheduler,
DotNetNuke.DNNScheduler"
providerPath="~\Providers\SchedulingProviders\DNNScheduler\"
debug="false"
maxThreads="-1"
enabled="true" />
                </providers>
        </scheduling>
        <friendlyUrl defaultProvider="DNNFriendlyUrl">
                <providers>
                        <clear />
                        <add name="DNNFriendlyUrl"
type="DotNetNuke.Services.Url.FriendlyUrl.DNNFriendlyUrlProvider,
DotNetNuke.HttpModules.UrlRewrite" />
                </providers>
        </friendlyUrl>
</dotnetnuke>
```

By reviewing the web.config definition you can see that each provider is defined by a key, and then specific configuration properties are specified in order to define which physical provider to use. For example in the data section you see the following:

```
<data defaultProvider="SqlDataProvider">
```

The defaultProvider is defined as SQLDataProvider, then within the data key you can see the configuration information for the SQLDataProvider:

```
			<providers>
				<clear />
				<add name="SqlDataProvider"
type="DotNetNuke.Data.SqlDataProvider, DotNetNuke.SqlDataProvider"
connectionStringName="SiteSqlServer"
upgradeConnectionString=""
providerPath="~\Providers\DataProviders\SqlDataProvider\"
objectQualifier=""
databaseOwner="dbo" />
			</providers>
```

You can see within the provider section of the web.config is the actual physical database information. You could have as many providers as you want defined here, and then switch the defaultProvider value to point to a new physical database provider. Of course you would need to have a provider developed and loaded within your bin folder in order for it to be used. In addition to having a provider for DNN, you would also need to have a provider developed for your module that uses the same physical database as DNN.

Quick Tip: Learn More About Provider Patterns

Rob Howard details out the Provider Pattern in a two part series at Microsoft's MSDN site:
Provider Model Design Pattern and Specification, Part 1:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnaspnet/html/asp02182004.asp
Provider Design Pattern, Part 2:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnaspnet/html/asp04212004.asp?frame=true

Folder Structure

DotNetNuke strives to logically structure the folders within the application root folder. The following list provides the folders under the root folder that are important to your development efforts, and what they contain.

- <app root>/Admin – Contains interfaces to the various administrative functions within DNN, for example, the scheduler, tab management, search manager, and any interface that is to be secured and accessible by a portal host or admin.
- <app root>/App_GlobalResources – Contains application specific data like time zone information, and application settings.
- <app root>/bin – Contains the compiled assemblies of your DNN solution, and supporting assemblies.
- <app root>/Components – Contains core functionality for DNN.
- <app root>/controls – Contains the various controls for enhancing the user interface of DNN, for example the Text/HTML provider is located within this folder.
- <app root>/DesktopModules – Contains all module projects for DNN. This is where you would place your directory for your module project.
- <app root>/HttpModules – All HTTPModule projects are contained in here, for example, the exception handler, URLRewriter, Scheduler, and others.
- <app root>/images – Contains any images used by DNN such as icons.
- <app root>/js – Contains any client side scripts in use by DNN, for example the calendar popup control.
- <app root>/portals – Contains the default portal folder, and will contain any portal folders for newly created portals. This provides any new child portal with it's own personal folder for storing files, and portal skins. Portal folders are identified by a unique integer value which matches up with the Portal ID field value in the Portals table in the DNN database.
- <app root>/Providers – Contains the providers for your DNN portal, for example the database providers are located in a sub directory off of this folder.
- <app root>/Resources
- <app root>/Web references – This folder contains references for Web services, currently the exceptions Web service which allows a portal admin to optionally notify the DNN core team about errors being generated by the portal.

Beginning Module Development

Now that we have covered what modules are, and how they interface with DNN, let's get started developing modules. In the next sections we will cover how to set up a Visual Studio.Net project, and begin development.

Configuring Your Visual Studio.NET Project

Some basic assumptions of this guide are that you have a development machine configured with a default DNN installation within a virtual directory. You should also have some basic knowledge of developing ASP.NET applications in Visual Studio.NET. We are going to build off our default installation of DNN and add our module projects to the solution. By adding to the main DNN solution you ensure you are compiling to the DNN bin directory, and enable debugging for your module project.

Creating the Module Project

In order to create our module project you should have your DNN solution open, and we'll need to add a new project to the solution in order to begin.

There are various methods for creating modules for DNN, but we prefer to configure a separate project for module creation and then compile the assemblies within each module's own bin folder. The following procedure provides a method for configuring a portal module project in Visual Studio.NET.

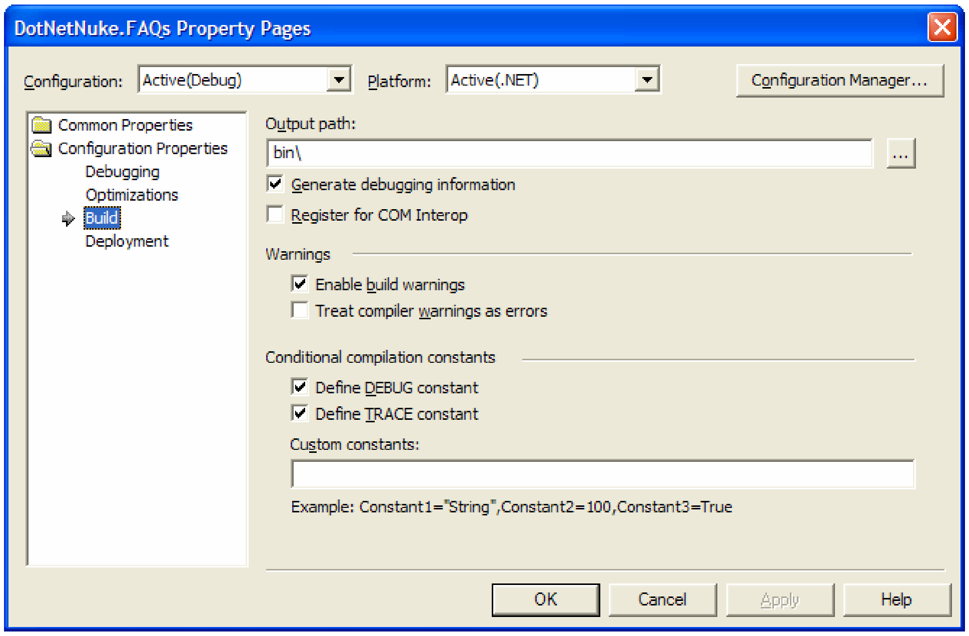
1. Create a new class project, be sure to use a logical naming structure. For example: CompanyName.ModuleName. By using this format you ensure your module will not conflict with modules developed by other companies or developers.

2. When creating the new project folder, create a folder off of the DotNetNuke Root\DesktopModules\ directory.

3. Clear any namespace from the project properties. Right click on properties of the project, under general clear the root namespace box.

4. Add a reference to the DotNetNuke project in your new module project.

5. In the BuildSupport project in the <app root>\Solutions\DotNetNuke.DesktopModules\BuildSupport\ directory add a reference to your project. By adding the reference to the BuildSupport project will take the references from your solution to be built within the main DotNetNuke bin directory.



Configuration: Active(Debug)
Platform: Active(.NET)
Output path: bin\
☑ Generate debugging information
☐ Register for COM Interop
Warnings
☑ Enable build warnings
☐ Treat compiler warnings as errors
Conditional compilation constants
☑ Define DEBUG constant
☑ Define TRACE constant
Custom constants:
Example: Constant1="String",Constant2=100,Constant3=True

Figure 8 – Class Project Properties Dialog

Now that your project has been created we will create some user controls for your module interface. A simple module can be as basic as a view control to display an interface for your user, and then a settings control for specifying unique values that are specific for a module instance.

This is one area of difficulty when developing module projects for DNN. Since we want to be able to see the results of our development within the portal, the method to accomplish this is by using a class project which allows you in Visual Studio.NET to change the compilation path. Since we are working within a class project the ability to create user controls in Visual Studio.NET is limited and only available when you're in a Web project. As of this writing, this author has included the Quick Tip below to point you to templates that will make the creation of the controls easier. For the purposes of this guide, we will manually create an item and add our code.

In our example we work with the Survey module that is included in the DNN distribution. This module is located within the /DesktopModules/Survey directory included in the distribution.

You see some controls, as well as some class files within this folder. Initially when starting a project, you will need to create a few user controls for your user to interact with your module, and for your administrators to configure some module instance settings.

Quick Tip: Ease DNN File Creation

In this guide we cover creating a module project from scratch; you can streamline this development process by using resources available on the Web. These we're initially targeted at DNN 2.x, but should also work for DNN 3.x. Try using DNN Project Templates available http://dnnjungle.vmasanas.net/Default.aspx?tabid=28 for creating your module projects, and controls.

Creating the Data Provider Project

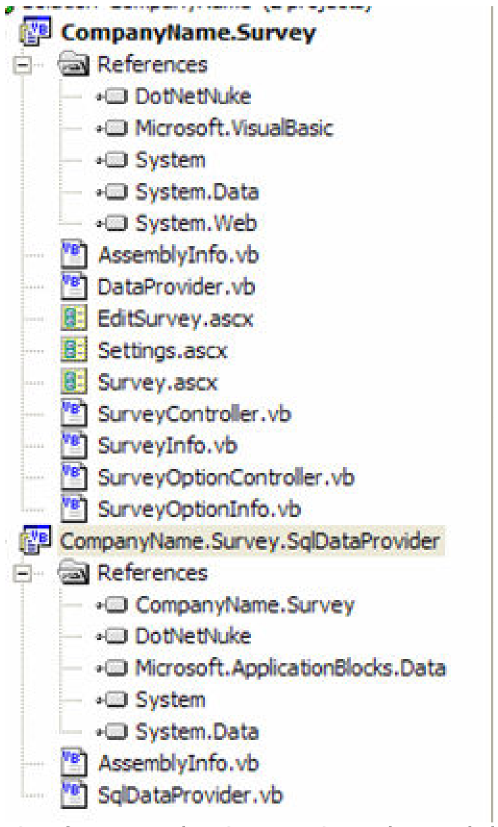
The Data Provider project is for developing the methods for interfacing with the physical database. In this guide we discuss SQL Server 2000 as our physical database provider, but your actual provider could be particularly any database such as Access, Oracle, or even MySQL. One of the big advantages of the DNN architecture is a provider model pattern, which enables to you to plug in any provider you wish for the physical database provider. We will discuss the provider pattern later in this guide.

In module development you create a separate project for each physical dataprovider. You do this similar to the way you configured your module project.

1. Create a new class project, be sure to use a logical naming structure. For example: CompanyName.ModuleName.SQLDataProvider. Again, this will avoid naming conflicts with other modules, and keeps you within the naming structure of the module that this provider is going to support.
2. When creating the new project folder, create a folder off of the DotNetNuke Root\DesktopModules\ModuleName\ directory, call this directory "Providers", and create a subdirectory for each provider type you will develop, for example:
<app root>\DesktopModules\ModuleName\Providers\SQLDataProvider\
3. Clear any namespace from the project properties. Right click on properties of the project, under general clear the root namespace box.

4. Now add project references to the main DotNetNuke and CompanyName.ModuleName projects.
5. Add a reference to the BuildSupport project as we did in the module setup.

Once both projects are configured they should look similar to Figure 9.



```
CompanyName.Survey
  References
    DotNetNuke
    Microsoft.VisualBasic
    System
    System.Data
    System.Web
  AssemblyInfo.vb
  DataProvider.vb
  EditSurvey.ascx
  Settings.ascx
  Survey.ascx
  SurveyController.vb
  SurveyInfo.vb
  SurveyOptionController.vb
  SurveyOptionInfo.vb
CompanyName.Survey.SqlDataProvider
  References
    CompanyName.Survey
    DotNetNuke
    Microsoft.ApplicationBlocks.Data
    System
    System.Data
  AssemblyInfo.vb
  SqlDataProvider.vb
```

Figure 9 – Configuring the Survey Sample Module in DNN.

Configuring Your Database

Finally we will need to create tables and since we're using SQL Server as our database backend; stored procedures to support our module. There are several methods of creating database objects, which is out of the scope of this guide. We'll provide the structure of the module here in the guide, and you can build from there. One easy way to build the structure is to open the database provider script for the module and remove all occurrences of the {databaseOwner}{objectQualifier} within notepad or another text editor, and then run the resulting script in Query Analyzer.

This specific module has the following database structure:

- Table: SurveyOptions
  - Field: SurveyOptionID
  - Field: SurveyID
  - Field: ViewOrder
  - Field: OptionName
  - Field: Votes
- Table: Surveys
  - Field: SurveyID
  - Field: ModuleID
  - Field: Question
  - Field: ViewOrder
  - Field: OptionType
  - Field: CreatedByUser
  - Field: CreatedDate
- Stored Procedure: GetSurveys
- Stored Procedure: GetSurvey
- Stored Procedure: AddSurvey
- Stored Procedure: UpdateSurvey
- Stored Procedure: DeleteSurvey
- Stored Procedure: GetSurveyOptions
- Stored Procedure: AddSurveyOption
- Stored Procedure: SurveyOptions
- Stored Procedure: UpdateSurveyOption
- Stored Procedure: DeleteSurveyOption
- Stored Procedure: AddSurveyResult

When we get to the section on the data provider, you will see that each stored procedure will have a method within the provider class.

When creating custom modules, your database structure, and the amount of stored procedures will differ based on what it is you want to do.

Quick Tip: Ease Database Stored Procedure and Provider Creation

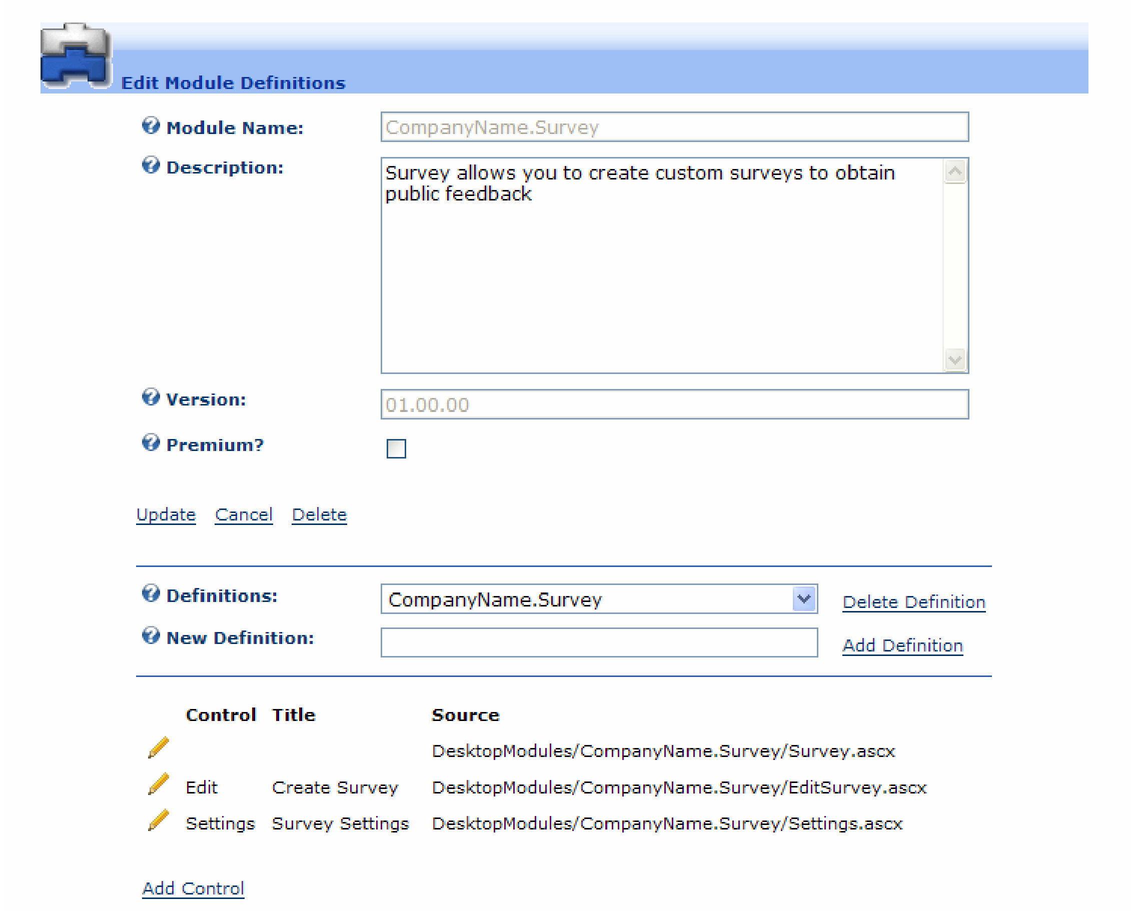
You can download Code Smith by Eric J. Smith, which will enable generation of stored procedures, and code for DNN with the appropriate templates. Download Code Smith at: http://www.ericjsmith.net/codesmith/. Scott McCulloch makes DNN templates available at his Weblog here: http://lucaslabs.net/blogs/mccullochs/archive/2004/02/15/487.aspx

Configuring DNN to Interface with Your Controls

Now that we have our project defined, controls created, and database structure done, we need to enable these controls to interface with our DNN portal. In our example for this guide we created three controls, a view, an edit, and an settings control. All of these controls need to be defined within the module definitions in our portal. In the following sections we will configure the user controls to interface with DNN; this will allow us to instantly see the results of a new compilation of our module within our test portal.

You may be familiar with Private Assembly installation files, here you just upload a zip file to DNN and everything is installed automatically. When you're developing modules you usually have to do this process manually, and let DNN know where your ascx controls are. Your compiled assemblies are already being compiled and placed in the DNN bin folder so DNN can access them when needed, but you still need to let DNN know about your new module. You can manually define your modules using the following procedure:

1. Under Host Settings menu, go to Module Definitions
2. In Module Definitions Select Add New Definition from the module menu in the upper left hand corner.
3. You will then be presented with the Edit Module Definitions screen.
4. From here add a new definition.
5. Once your definition is created add your user controls to your module. For the survey module you will need to add the following:
   Survey.ascx - Type of View
   EditSurvey.ascx - Type Edit, Key is Edit.
   Settings.ascx - Type is Edit, Key is Settings.



Edit Module Definitions

Module Name:	CompanyName.Survey
Description:	Survey allows you to create custom surveys to obtain public feedback
Version:	01.00.00
Premium?	☐

Update Cancel Delete

Definitions:	CompanyName.Survey	Delete Definition
New Definition:		Add Definition

Control	Title	Source
		DesktopModules/CompanyName.Survey/Survey.ascx
Edit	Create Survey	DesktopModules/CompanyName.Survey/EditSurvey.ascx
Settings	Survey Settings	DesktopModules/CompanyName.Survey/Settings.ascx

Add Control

Figure 10 – Module Definitions, Survey Module Defined

Now that you finished adding the references to your module, you should see a screen similar to Figure 10, which displays all the controls that comprise your module, and the keys defined.

Module Architecture

So let's look at how modules relate to DNN. In figure 11, you can see at the top most level is a tab. For a user this is the page that is presented when clicking through the navigation. Each tab can contain several desktop modules. A desktop module can then be defined in various ways as in we discussed in previous examples where we defined a module via the host interface using module definitions. Within this module definition you can define various modules containing various different types of module controls. Module controls are basically user controls within ASP.NET, but we use a key to define the type of the control, these controls can be of Admin, Edit, or View within your desktop module.

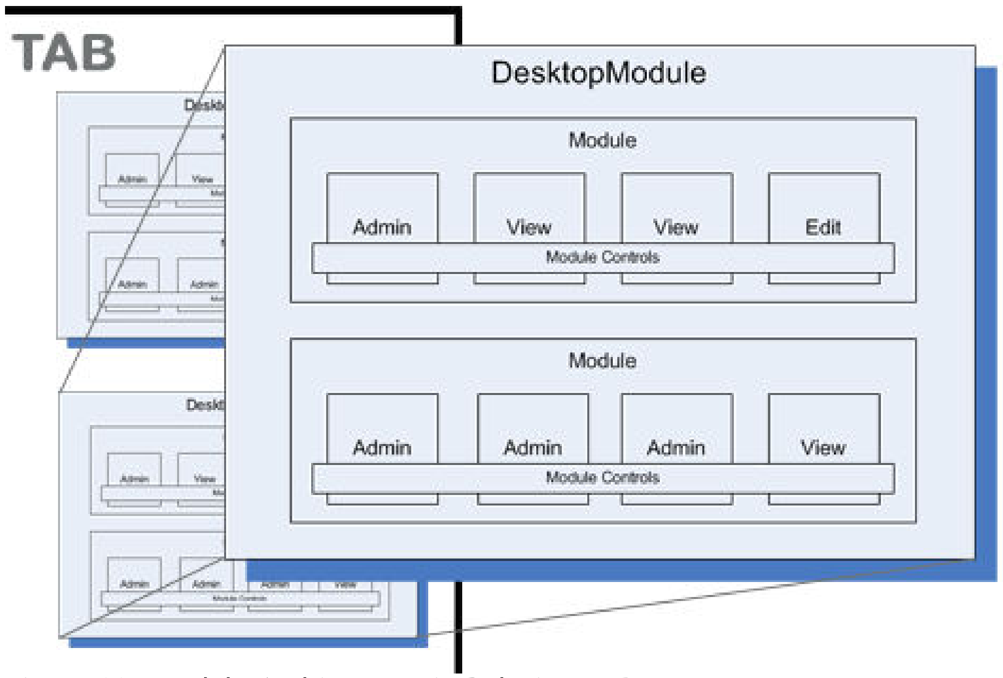


Figure 11 - Module Architecture in Relation to DNN

Now that we covered the high level abstraction of how the modules are architected within the DNN framework, let's begin writing some actual code, and then constructing a definition file for our module. These next sections should provide you with the background to begin developing your own module projects for DNN.

Three & n-Tier Architecture

Before getting into the different layers of developing DNN modules, we should first go over some basic architectural concepts of application development.

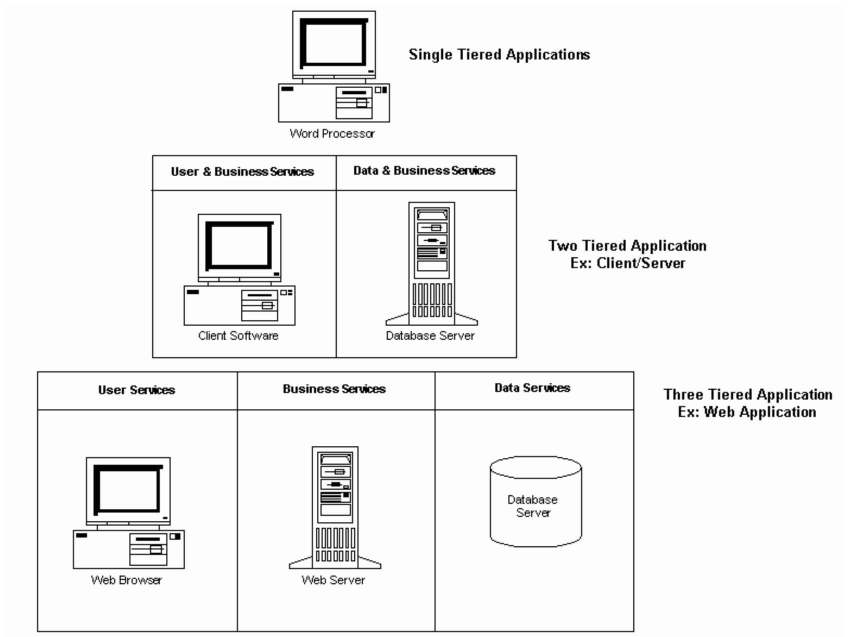


Figure 12 – Review of Application Three-Tiered Architecture

The top level of illustration (see Figure 12) is a single tiered word processing application that contains all logical service levels: the data services—the files it interfaces with; the business services—the logic used to define how the application interacts with the document files; and the user services—the GUI that the user interacts with in order to edit the documents. All these services are on one tier. No distribution of the application takes place.

At the middle level is a typical client/server application. This application has the user services at one tier being the GUI that the user interacts with. This level could also contain some of the business services, where you have some logic that enforces the business rules. This business level could also reside on the second tier, being the database on a back-end server that contains functions, or stored procedures. We still have three service layers, but the application is now divided into two distinct physical tiers.

At the bottom of the illustration, is the Web application model, where we clearly define a tier for each service layer. We distribute the application by placing the data services on a SQL Server or other data provider, the business services in a COM object, and the user services in the HTML or ASP code that is eventually distributed to the end-user via the Web browser.

We can further expand this application model to an n-tier level, meaning the business services could be split into multiple COM objects, or we could have several different data providers that interact with the application. Each additional level provided another tier to the application.

One Page Many Controls

The DNN user interface is driven from one primary document which is the default.aspx file contained in the application's root directory. All user requests are made to this document and it changes it's behavior based on certain parameters that are passed to it. For example, you may have multiple controls for your module which are then loaded into the main default.aspx file. Remember that the module load process is defined by the specific Tab Id for the page. Each control you create must then be defined within DNN (see "Configuring DNN to Interface with your Controls" later in this guide) with a key value which designates which control to load.

Pages are loaded based on the Tab ID passed within a request parameter to the DNN application. The Tab ID is a unique value within the Tabs table in the database. There are several other parameters of importance that you should be aware of when developing against the DNN framework. The following is a list of parameters and what they do:

- **TabId** – Pages within DNN are designated by a TabID. For example, if you create a page in your portal, it is assigned a primary key with the name of TabID within the database. This key now becomes associated with modules you insert into a page and the behavior.
- **Mid** – This parameter is for specifying the Module ID of a specific module. This is normally used when you want to change the settings within a module, or you need to load a specific user control in your process.
- **Ctl** – This is the control to load for your specific module. For example, you module may consist of multiple user controls, as long as these are defined in the module definitions of the portal, you can then pass a control's key value to the Ctl parameter to tell your module to load that specific control. This parameter must be used in conjunction with the Module ID (mid) parameter in order to designate which module instance the specified control is related to.

For example, say I have a module containing multiple user controls, each control would then be defined within the module definition with its own unique key; then when I want to load a specific control within a module, I would make a request back to the tab containing the module, but include a control to load parameter:

```
Response.Redirect(NavigateURL & "mid=" & ModuleID & "&ctl=ControlKey")
```

The preceding code block would have the effect of loading the current module, but with the specific control specified by passing its key name value via the CTL parameter. See the section on DNN user controls for more information on how controls work with your modules.

Data Layer

In this section we will cover our abstract class for interacting with the database. Since our module is using a provider model, our module only contains abstract classes for doing database interaction. One of the main benefits of this model, is the ability to swap different providers in our application without having to modify the code base of the primary module. All physical interaction is done within another assembly, and our abstraction provider in our module is only going to contain overridable methods for doing database operations.

Data Abstraction Provider

The DataProvider.vb class located within the module project is the class that contains the overridable methods for our module. When you open the DataProvider.vb class within the Survey project, you see it contains methods that must be overridden by our physical database provider assembly. Each one of the methods in this example corresponds to a stored procedure within our database.

```
Imports System
Imports DotNetNuke

Namespace CompanyName.Survey

    Public MustInherit Class DataProvider

        ' singleton reference to the instantiated object
        Private Shared objProvider As DataProvider = Nothing

        ' constructor
        Shared Sub New()
```

```
          CreateProvider()
       End Sub

       ' dynamically create provider
       Private Shared Sub CreateProvider()
                         objProvider =
CType(Framework.Reflection.CreateObject("data", "CompanyName.Survey",
"CompanyName.Survey"), DataProvider)
       End Sub

       ' return the provider
       Public Shared Shadows Function Instance() As DataProvider
          Return objProvider
       End Function

       ' all core methods defined below

       Public MustOverride Function GetSurveys(ByVal ModuleId As Integer) As
IDataReader
       Public MustOverride Function GetSurvey(ByVal SurveyID As Integer, ByVal
ModuleId As Integer) As IDataReader
       Public MustOverride Function AddSurvey(ByVal ModuleId As Integer, ByVal
Question As String, ByVal ViewOrder As Integer, ByVal OptionType As String, ByVal
UserName As String) As Integer
       Public MustOverride Sub UpdateSurvey(ByVal SurveyId As Integer, ByVal
Question As String, ByVal ViewOrder As Integer, ByVal OptionType As String, ByVal
UserName As String)
       Public MustOverride Sub DeleteSurvey(ByVal SurveyID As Integer)
       Public MustOverride Function GetSurveyOptions(ByVal SurveyId As Integer) As
IDataReader
       Public MustOverride Function AddSurveyOption(ByVal SurveyId As Integer,
ByVal OptionName As String, ByVal ViewOrder As Integer) As Integer
       Public MustOverride Sub UpdateSurveyOption(ByVal SurveyOptionId As
Integer, ByVal OptionName As String, ByVal ViewOrder As Integer)
       Public MustOverride Sub DeleteSurveyOption(ByVal SurveyOptionID As
Integer)
       Public MustOverride Sub AddSurveyResult(ByVal SurveyOptionId As Integer)

    End Class

End Namespace
```

These methods are then used by the business layer of your module to either perform some action against the database, like add, delete, or update operations, or to populate a business object.

Business Logic Layer (BLL)

We mentioned about populating business objects; now let’s cover an example on how to do this.

An object is a class of a specific type that exposes certain properties, and actions. In our example, we have to classes in our BLL, one is the info class, and the other is the controller class. In this example SurveyInfo, and SurveyController. Here's a description of these two classes:

- **ModuleInfo:** Sets and gets public properties for our objects being generated by our controller class. The properties exposed by this class, corresponds to field names and their types within the database table that supports our module. For example, in the Survey module there is a SurveyID, within the SurveyInfo class, there is a property with the same name.
- **ModuleController:** Used for populating values for objects, basically uses the custom business objects helper class to populate data from our data provider into a collection of objects.

Let’s look at the source code of the SurveyOptionsInfo.vb file located within the module project’s directory:

```
Imports System
Imports System.Data
Imports DotNetNuke

Namespace CompanyName.Survey

        Public Class SurveyOptionInfo

                ' local property declarations
                Private _SurveyOptionId As Integer
                Private _SurveyId As Integer
                Private _ViewOrder As Integer
                Private _OptionName As String
                Private _Votes As Integer

                ' initialization
                Public Sub New()
```

```
End Sub

' public properties
Public Property SurveyOptionId() As Integer
        Get
                Return _SurveyOptionId
        End Get
        Set(ByVal Value As Integer)
                _SurveyOptionId = Value
        End Set
End Property

Public Property SurveyId() As Integer
        Get
                Return _SurveyId
        End Get
        Set(ByVal Value As Integer)
                _SurveyId = Value
        End Set
End Property

Public Property ViewOrder() As Integer
        Get
                Return _ViewOrder
        End Get
        Set(ByVal Value As Integer)
                _ViewOrder = Value
        End Set
End Property

Public Property OptionName() As String
        Get
                Return _OptionName
        End Get
        Set(ByVal Value As String)
                _OptionName = Value
        End Set
End Property

Public Property Votes() As Integer
        Get
                Return _Votes
        End Get
        Set(ByVal Value As Integer)
                _Votes = Value
                End Set
            End Property

        End Class

End Namespace
```

You can see that the class exposes properties that correspond with the table structure in the database.

Now let’s look at the controller class for the Survey Options within the SurveyOptionController.vb file.

```
Imports System
Imports System.Data
Imports DotNetNuke

Namespace CompanyName.Survey

        Public Class SurveyOptionController

            Public Function GetSurveyOptions(ByVal SurveyId As Integer) As
ArrayList

                Return
CBO.FillCollection(DataProvider.Instance().GetSurveyOptions(SurveyId),
GetType(SurveyOptionInfo))

            End Function

            Public Sub DeleteSurveyOption(ByVal SurveyOptionID As Integer)

                DataProvider.Instance().DeleteSurveyOption(SurveyOptionID)

            End Sub

            Public Function AddSurveyOption(ByVal objSurveyOption As
SurveyOptionInfo) As Integer

                Return
CType(DataProvider.Instance().AddSurveyOption(objSurveyOption.SurveyId,
objSurveyOption.OptionName, objSurveyOption.ViewOrder), Integer)

            End Function
```

```
                Public Sub UpdateSurveyOption(ByVal objSurveyOption As
SurveyOptionInfo)

        DataProvider.Instance().UpdateSurveyOption(objSurveyOption.SurveyOptionId,
objSurveyOption.OptionName, objSurveyOption.ViewOrder)
                End Sub
                Public Sub AddSurveyResult(ByVal SurveyOptionID As Integer)
                        DataProvider.Instance().AddSurveyResult(SurveyOptionID)
                End Sub
        End Class
End Namespace
```

Each method within the controller class corresponds to a stored procedure within the database. Reviewing the code within the methods you will see that they create a call to DataProvider.Instance which is a call to our abstraction class talked about previously. These abstracted methods are passed the SurveyOptionInfo object for holding the values used for passing to the stored procedure defined in our physical database provider class. In the User Layer section we will show you how to implement a call to these classes and populate the values for the objects.

If you are returning data to the SurveyOptionInfo object for later binding in a control within your user interface, the Custom Business Object (CBO) helper class provides the hydration of the object. The CBO is part of the DNN core framework and provides the hydration of the objects when there is a return from a database call, as in this case a call to a stored procedure.

User Layer

In three or n-tier architecture, the user level is where the user interacts with your application, in DNN this layer is comprised of user controls (ascx) files. In the next sections we'll cover the various interfaces with DNN. This includes the view control, which your users will interact with, the administrative, and editing interfaces of your module.

Viewing Data from the User Perspective

The view control is the initial control that the user will see when accessing a page that contains the module. In our example, you can see the Survey.ascx. This contains all the controls we need for our initial survey view.

The view control for this module, this is the main Survey.ascx control, open the code behind file. In this class you'll notice the following:

```
Imports DotNetNuke
Namespace CompanyName.Survey
    Public MustInherit Class Survey
       Inherits DotNetNuke.Entities.Modules.PortalModuleBase
       Implements Entities.Modules.IActionable

        ::
        ::
End Class
```

PortalModuleBase

You'll notice instead of inheriting from the WebControls class as a standard user control would do, the module inherits from the DotNetNuke.Entities.Modules.PortalModuleBase provided by DNN. The PortalModuleBase class, located in the <app root>\Components\Modules\PortalModuleBase.vb file, provides some very important properties and methods exposed by DNN, for example:

- **IsEditable**: Boolean value determines whether or not the current user has edit permissions for the module.
- **LocalResourceFile** – If you're providing a localization resource for multiple languages.
- **SharedResourceFile**
- **ModuleSettings** – The ModuleSettings along with the Settings hash table provide the module developer with the ability to store module instance values.
- **Settings** – Hash table for storing module specific keys and values. This allows you to store and properties you define which are keyed off of the ModuleID for the current module instance, allowing you do implement the same module code base for various implementations.
- **HelpFile** – For storing a help file value for your module.

- **HelpURL** – This stores a value for online version of help stored on a Website.
- **Actions** – For storing menu items and their associated actions for your module.
- **CachedOutput** – For storing the cached output from the module. This cache setting is specified at the module level, and host levels.

In this specific example, we have a some basic interaction with a database via the BLL. Remember the entire code for this module is available in the DotNetNuke download as example of custom module development. Use this as a template for your own projects.

IActionable

Another area to talk about is the implementation of the IActionable interface. Here we are going to expose certain module actions in our menu control. In the Survey example you can see the following subroutine which exposes specific actions for our module.

```
Public ReadOnly Property ModuleActions() As
DotNetNuke.Entities.Modules.Actions.ModuleActionCollection Implements
DotNetNuke.Entities.Modules.IActionable.ModuleActions
        Get
                Dim Actions As New Entities.Modules.Actions.ModuleActionCollection
Actions.Add(GetNextActionID,
Localization.GetString(Entities.Modules.Actions.ModuleActionType.AddContent,
LocalResourceFile), Entities.Modules.Actions.ModuleActionType.AddContent, "", "",
EditUrl(), False, Security.SecurityAccessLevel.Edit, True, False)
                Return Actions
        End Get
End Property
```

In this code block, we are exposing menu actions as property of the module, of the type ModuleActions. Here we are going to add some actions which get raised to the moulde container in order to generate our menu structure. There are specific actions for our module such as adding an edit option to the menu items. If you look at the Add method for actions, you will see we are providing localized content if available, the URL of the edit control, and we're doing security checking to ensure that the menu item is only displayed to users of the portal that have edit permissions.

Displaying Data

One of the primary tasks of the view control is to display data from our database. In the survey example there are two controls in the Survey.ascx file which are bound to the

SurveyInfo object for displaying the questions on the survey and the results of the survey.

```
<%@ Control Language="vb" AutoEventWireup="false"
Inherits="CompanyName.Survey.Survey" CodeBehind="Survey.ascx.vb" %>
<br>
<asp:panel id="pnlSurvey" runat="server" visible="False">
        <asp:datalist id="lstSurvey" datakeyfield="SurveyId" cellpadding="4"
runat="server">
                <itemtemplate>
                        <asp:HyperLink id="cmdEdit1" ImageUrl="~/images/edit.gif"
NavigateUrl='<%# EditURL("SurveyId",DataBinder.Eval(Container.DataItem,"SurveyId"))
%>' Visible="<%# IsEditable %>" runat="server" />
                        <asp:Label ID="lblQuestion1" Runat="server"
CssClass="NormalBold" Text='<%#
FormatQuestion(DataBinder.Eval(Container.DataItem,"Question"),lstSurvey.Items.Count
+ 1) %>'>
                        </asp:label><br>
                        <asp:radiobuttonlist id="optOptions" runat="server"
cssclass="Normal" visible="False" repeatdirection="Vertical"
datavaluefield="SurveyOptionId" datatextfield="OptionName"></asp:radiobuttonlist>
                        <asp:checkboxlist id="chkOptions" runat="server"
cssclass="Normal" visible="False" repeatdirection="Vertical"
datavaluefield="SurveyOptionId" datatextfield="OptionName"></asp:checkboxlist>
                        <br>
                </itemtemplate>
        </asp:datalist>
        <asp:linkbutton id="cmdSubmit" resourcekey="cmdSubmit" runat="server"
cssclass="CommandButton">Submit Survey</asp:linkbutton>&nbsp;
        <asp:linkbutton id="cmdResults" resourcekey="cmdResults" runat="server"
cssclass="CommandButton">View Results</asp:linkbutton>
</asp:panel>
<asp:panel id="pnlResults" runat="server" visible="False">
        <asp:datalist id="lstResults" datakeyfield="SurveyId" cellpadding="4"
runat="server">
                <itemtemplate>
                        <asp:HyperLink id="cmdEdit2" ImageUrl="~/images/edit.gif"
NavigateUrl='<%# EditURL("SurveyId",DataBinder.Eval(Container.DataItem,"SurveyId"))
%>' Visible="<%# IsEditable %>" runat="server" />
                        <asp:Label ID="lblQuestion2" Runat="server"
CssClass="NormalBold" Text='<%#
FormatQuestion(DataBinder.Eval(Container.DataItem,"Question"),lstResults.Items.Count
+ 1) %>'>
                        </asp:label><br>
                        <asp:label id="lblResults" runat="server"
cssclass="Normal"></asp:label>
                        <br>
                </itemtemplate>
        </asp:datalist>
        <asp:linkbutton id="cmdSurvey" resourcekey="cmdSurvey" runat="server"
cssclass="CommandButton">View Survey</asp:linkbutton>
</asp:panel>
```

Let's look at the lstSurvey DataList control, and see how we bind our data to this control when the control is loaded on the first request.

```
Private Sub DisplaySurvey()

        Dim objSurveys As New SurveyController

        lstSurvey.DataSource = objSurveys.GetSurveys(ModuleId)
        lstSurvey.DataBind()

        pnlSurvey.Visible = True
        pnlResults.Visible = False

        If lstSurvey.Items.Count = 0 Then
                cmdSubmit.Visible = False
                cmdResults.Visible = False
        End If

End Sub
```

You can see in the above code, we create an instance of the SurveyController class, we then pass the value of ModuleID. Remember ModuleID is our key for the module instance, the value within ModuleID allows us to reuse this component in as many instances as we want within our DNN portal implementation. It is very important that when you design your database structure you include a field for holding the value of ModuleID. The ModuleID is passed to the stored procedure via our abstraction class, and then the physical provider class, and in turn returns our data, which populates our SurveryController object. This is of a type ArrayList which is bound to the DataList control, and finally populates our list to present to the user.

Adding Data

Now let's do an example of updating data for our module. This is done, by creating an instance of our SurveyOptionController object, calling the AddSurveyResults method, and passing our values, which in turn is executed via our abstraction class as covered earlier. Let's look at our Survey.ascx.vb class in the cmdSubmit_Click method.

```
::
::
Dim objSurveyOptions As New SurveyOptionController
::
::
If Request.Form(lstSurvey.UniqueID & ":_ctl" & intQuestion.ToString & ":optOptions") <>
"" Then

        objSurveyOptions.AddSurveyResult(Int32.Parse(Request.Form(lstSurvey.UniqueID
& ":_ctl" & intQuestion.ToString & ":optOptions")))
End If
::
::
```

Updating Data

Now let's update data for our survey. In order to update our data we need to create an instance of the Info class for our object that we want to update. We will pass the ID of the target object, and then pass the updated values to the object's properties. This object will then be passed to the controller class, which will pass the method to the abstract class, and in turn the physical provider will execute the corresponding stored procedure. Now let's look at an example of updating data, open the EditSurvey.ascx.vb file, and look for the Update_Click method. Within the Update_Click method you see the following code:

```
Dim objSurveyOptions As New SurveyOptionController
::
Dim objSurveyOption As New SurveyOptionInfo

objSurveyOption.SurveyOptionId = Integer.Parse(lstOptions.Items(intOption).Value)
objSurveyOption.SurveyId = SurveyId
objSurveyOption.OptionName = lstOptions.Items(intOption).Text
objSurveyOption.ViewOrder = intOption
objSurveyOption.Votes = 0

If Null.IsNull(objSurveyOption.SurveyOptionId) Then
      objSurveyOptions.AddSurveyOption(objSurveyOption)
Else
```

```
        objSurveyOptions.UpdateSurveyOption(objSurveyOption)
End If
```

In this code block we create an instance of our SurveyOptionInfo object, and then pass some values to the object for updating. You'll also notice here that we check to see if the value of the SurveyID which is our primary key for the Survey table is null, if it is null then that means we have a new item to add and call the AddSurveyOption method, if not, we then call the UpdateSurveyOption and pass our SurveyOptionInfo object with the populated values. The controller will then call the abstraction class, which in turns calls the physical database provider class, and finally executes the stored procedure with the updated values.

Deleting Data

Now that we covered adding, viewing, and updating, we still need to cover how to delete data from the tables. The way to perform a delete is similar to adding, and updating, but the only value that needs to be passed to the method is the primary key of the item you wish to delete. In this example, we make a call to our SurveyOptionController class and call the DeleteSurveyOption method. To view this code in the example find the cmdDeleteOption_Click method in the EditSurvey.ascx.vb file.

```
Dim objSurveyOptions As New SurveyOptionController
If lstOptions.SelectedIndex <> -1 Then

objSurveyOptions.DeleteSurveyOption(Integer.Parse(lstOptions.SelectedItem.Value))
        lstOptions.Items.RemoveAt(lstOptions.SelectedIndex)
End If
```

Editing Settings for Your Module

We covered adding, updating, and deleting data for your module, but we also need to talk about the item that drives the ability to create a distinct implementation of the same module multiple times throughout a portal or multiple portals. Remember previously in this guide we said that each module is assigned a unique Module ID, and this ID is used to key off for our modules. For example, say you have an invoicing module, but you need to point to different invoicing systems for two module implementations. You could create a module specific key and value pair within DNN to store the values for connection to each invoice system.

DNN contains a table within the database called ModuleSettings which hold key names and their values for each module instance. Let's go over the structure of the database:

- **ModuleID** – Contains the ID of the module instance that we are storing information for.
- **SettingName** – Contains a string value for storing the key name that we are going to be saving data for.
- **SettingValue** – This stores the actual value for the specific key for this specific module instance.

These values are set by making a call to the DotNetNuke.Entities.Modules.ModuleSettingsBase, let's review the Settings.ascx.vb file in the Survey module project.

If you look at the code in the Settings.ascx.vb file you'll see the two primary methods that we override in our base class, they are the LoadSettings, and the UpdateSettings methods. These are the methods called to populate the values from our settings hash table, and then subsequently save any settings that are entered via a form. In this case we are saving values that affect the look of our site.

Let's review the line of code for saving a setting to a key:

```
Public Overrides Sub UpdateSettings()
        Try
                Dim objModules As New
DotNetNuke.Entities.Modules.ModuleController

                Dim datClosingDate As Date =
Convert.ToDateTime(Null.SetNull(datClosingDate))
                If txtClosingDate.Text <> "" Then
                        datClosingDate = Convert.ToDateTime(txtClosingDate.Text)
                End If

                'Update Module Settings
                objModules.UpdateModuleSetting(ModuleId, "surveyclosingdate",
DotNetNuke.Common.Globals.DateToString(datClosingDate))
                objModules.UpdateModuleSetting(ModuleId, "surveytracking",
(rblstPersonal.SelectedIndex).ToString)
                objModules.UpdateModuleSetting(ModuleId, "surveyresultstype",
(rblstSurveyResults.SelectedIndex).ToString)
```

```
                'Update Tab Module Settings
                objModules.UpdateTabModuleSetting(TabModuleId,
"surveygraphwidth", txtGraphWidth.Text)

        Catch exc As Exception                      'Module failed to load
                ProcessModuleLoadException(Me, exc)
        End Try
End Sub
```

If you look at the preceeding code you can see we create an instance of the ModuleController class, and then call the method UpdateModuleSetting. The UpdateModuleSetting accepts the ID of the current module (ModuleID), a string value for the name of the key we're going to save, and then the value for the key.

Now let's look at how we obtain values from previously saved keys.

```
Public Overrides Sub LoadSettings()
        Try
                If (Page.IsPostBack = False) Then
                        cmdCalendar.NavigateUrl =
DotNetNuke.Common.Utilities.Calendar.InvokePopupCal(txtClosingDate)

                If Not CType(Settings("surveyclosingdate"), String) = "" Then
                                                txtClosingDate.Text =
CType(Settings("surveyclosingdate"), Date).ToShortDateString
                End If
                txtGraphWidth.Text = CType(Settings("surveygraphwidth"), String)
                rblstPersonal.SelectedIndex = CType(Settings("surveytracking"),
Integer)
                rblstSurveyResults.SelectedIndex =
CType(Settings("surveyresultstype"), Integer)
                End If
        Catch exc As Exception                      'Module failed to load
                ProcessModuleLoadException(Me, exc)
        End Try
End Sub
```

In the code you can see the only item we need to do is obtain the value within the Settings hash and pass it to our control.

Physical Database Providers

We reviewed the structure of DNN, and how we specify a provider via the web.config, in this section we will review the code required to create a SQL database provider. Remember each database type you connect to is going to need it's own specific provider.

For this section, switch to the CompanyName.ModuleName.SQLDataProvider project for the Survey module.

Compared to the parent Module project this is relatively simple in structure. This project contains the classes required for interfacing with your physical database. In our Survey example, we will use the SQLDataProvider project for connecting the SQL Server. In most cases this provider class will contain one to one methods for each stored procedure in your database for your specific module project. These methods are doing the actual interaction with the database and overriding methods within your abstract provider in order to interact with your module. This is one of the primary benefits of the provider model and one you should understand at this point, you can easily plug in different providers without having to recompile or modify your main project in any way.

To set up our class we need to import a few libraries:

```
Imports System
Imports System.Data
Imports System.Data.SqlClient
Imports Microsoft.ApplicationBlocks.Data
Imports DotNetNuke
```

The Microsoft.ApplicationBlocks.Data is going to provide us with some methods to ease our connection to the backend SQL database. More information on this class can be found at: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/daab-rm.asp

In our data provider class we will inherit the DataProvider class. The class is primarily constructed in way that mirrors the data provider class for the main DNN solution. We first need to declare some private members for our class.

```
Namespace CompanyName.Survey
    Public Class SqlDataProvider
        Inherits DataProvider
```

```
#Region "Private Members"
                Private Const ProviderType As String = "data"
                Private _providerConfiguration As
Framework.Providers.ProviderConfiguration =
Framework.Providers.ProviderConfiguration.GetProviderConfiguration(ProviderType)
                Private _connectionString As String
                Private _providerPath As String
                Private _objectQualifier As String
                Private _databaseOwner As String
```

You can see the first private member is a constant with the value of “data”, no go back to the web.config we listed earlier in this guide, the data value points to the value within the web.config. This is where our provider is going to pull it’s date for the connection string, provider path, object qualifier, and the database owner. All these values are defined in our provider class, which then get populated when the class is created. Let’s look at the “New” method to see what happens when the provider is instantiated.

```
Public Sub New()

        ' Read the configuration specific information for this provider
        Dim objProvider As Framework.Providers.Provider =
CType(_providerConfiguration.Providers(_providerConfiguration.DefaultProvider),
Framework.Providers.Provider)

        ' Read the attributes for this provider
        If objProvider.Attributes("connectionStringName") <> "" AndAlso _
        System.Configuration.ConfigurationSettings.AppSettings(objProvider.Attributes("connectionSt
<> "" Then
                _connectionString =
System.Configuration.ConfigurationSettings.AppSettings(objProvider.Attributes("connectionStringNar
        Else
                _connectionString = objProvider.Attributes("connectionString")
        End If

        _providerPath = objProvider.Attributes("providerPath")

        _objectQualifier = objProvider.Attributes("objectQualifier")
        If _objectQualifier <> "" And _objectQualifier.EndsWith("_") = False Then
                _objectQualifier += "_"
        End If

        _databaseOwner = objProvider.Attributes("databaseOwner")
        If _databaseOwner <> "" And _databaseOwner.EndsWith(".") = False Then
```

```
                _databaseOwner += "."
        End If

End Sub
```

You can see when this class is instantiated; it reads the information about the provider from the web.config. Let's review the data section from the web.config to see exactly what we're talking about.

```
<data defaultProvider="SqlDataProvider">
        <providers>
                <clear />
                <add name="SqlDataProvider"
                type="DotNetNuke.Data.SqlDataProvider,
DotNetNuke.SqlDataProvider"
                connectionStringName="SiteSqlServer"
                upgradeConnectionString=""
                providerPath="~\Providers\DataProviders\SqlDataProvider\"
                objectQualifier=""
                databaseOwner="dbo" />
        </providers>
</data>
```

You can see in the web.config where the values defined in the SQLDataProvider.vb file correspond with the values within the web.config.

We need to now expose our information via some public properties:

```
Public ReadOnly Property ConnectionString() As String
        Get
                Return _connectionString
        End Get
End Property

Public ReadOnly Property ProviderPath() As String
        Get
                Return _providerPath
        End Get
End Property

Public ReadOnly Property ObjectQualifier() As String
        Get
                Return _objectQualifier
        End Get
End Property

Public ReadOnly Property DatabaseOwner() As String
        Get
                Return _databaseOwner
        End Get
End Property
```

Let's move on now to the next section within the SQLDataProvider class. We now need to run the routines that do the actual database connection, and manipulation. You remember from the data abstraction layer that it contained overridable methods that we override from within our physical database provider. Let's review the methods used to connect to our physical SQL database.

```
Private Function GetNull(ByVal Field As Object) As Object
        Return DotNetNuke.Common.Utilities.Null.GetNull(Field, DBNull.Value)
End Function

Public Overrides Function GetSurveys(ByVal ModuleId As Integer) As IDataReader
        Return CType(SqlHelper.ExecuteReader(ConnectionString, DatabaseOwner
& ObjectQualifier & "GetSurveys", ModuleId), IDataReader)
End Function
Public Overrides Function GetSurvey(ByVal SurveyID As Integer, ByVal ModuleId As
Integer) As IDataReader
        Return CType(SqlHelper.ExecuteReader(ConnectionString, DatabaseOwner
& ObjectQualifier & "GetSurvey", SurveyID, ModuleId), IDataReader)
End Function
Public Overrides Function AddSurvey(ByVal ModuleId As Integer, ByVal Question As
String, ByVal ViewOrder As Integer, ByVal OptionType As String, ByVal UserName
As String) As Integer
        Return CType(SqlHelper.ExecuteScalar(ConnectionString, DatabaseOwner &
ObjectQualifier & "AddSurvey", ModuleId, Question, GetNull(ViewOrder),
OptionType, UserName), Integer)
End Function
Public Overrides Sub UpdateSurvey(ByVal SurveyId As Integer, ByVal Question As
String, ByVal ViewOrder As Integer, ByVal OptionType As String, ByVal UserName
As String)
        SqlHelper.ExecuteNonQuery(ConnectionString, DatabaseOwner &
ObjectQualifier & "UpdateSurvey", SurveyId, Question, GetNull(ViewOrder),
OptionType, UserName)
End Sub
Public Overrides Sub DeleteSurvey(ByVal SurveyID As Integer)
        SqlHelper.ExecuteNonQuery(ConnectionString, DatabaseOwner &
ObjectQualifier & "DeleteSurvey", SurveyID)
End Sub
::
::
```

On review of the code you can see there is a method that corresponds with each stored procedure in our database, in turn each method contained within our data abstraction layer is overridden and our methods with the physical database provider will return the data. This allows for any provider to be used within your DNN portal without a recompile to your base assembly.

In addition, also pay attention to the GetNull function within the class; use this to account for null values within your fields of your database.

Interfacing with DNN

In this section we are going to cover the most common tasks developers run into when writing DNN modules. DNN provides the developer with several providers for performing various tasks; these consist of editing content, logging events to the portal log, scheduling disconnected tasks, and more.

Many of these tools are using the provider model of DNN. We cover providers native to DNN, creating your own providers to replace the core providers are beyond the scope of this guide.

Personalization

DNN provides module developers with the ability of storing unique values based on the specific user accessing the portal. This enables you to create a custom experience for each user of your module. In the following code, we will check to see if personalization is enabled for the module by first checking the value within the "surveytracking" key in the Settings hash for this module instance. If we have enabled personalization, then we'll do a check against the DotNetNuke.Services.Personalization.Personalization.GetProfile method. The GetProfile method accepts the current Module ID value and returns the value of a key specified earlier in the application logic. Let's review the code within the Page_Load event in the Survey.ascx.vb file.

```
' Check the settings to see if this module is using Personalization for vote tracking
If Not Settings("surveytracking") Is Nothing Then
        blnPersonal = CType(Settings("surveytracking"), Boolean)
Else
        blnPersonal = False
End If
::
::
If blnPersonal = True Then
' This means the module vote tracking is using personalization, so check the database to
see if this user has voted
' We first must make sure the user is a registered user who is logged in
        If UserId <> -1 Then
                ' Check if the user has voted before
                If Not
DotNetNuke.Services.Personalization.Personalization.GetProfile(ModuleId.ToString,
"Voted") Is Nothing Then
                        blnVoted =
CType(DotNetNuke.Services.Personalization.Personalization.GetProfile(ModuleId.ToString,
"Voted"), Boolean)
                Else
                        blnVoted = False
                End If
                'If not voted before show the Survey items, otherwise just show the results
                If blnVoted = False Then
                        DisplaySurvey()
                Else
                        DisplayResults()
                End If
        Else
        ' The User is not logged in and the module is using personalization, so present the
user a message that he needs to login first
                DotNetNuke.UI.Skins.Skin.AddModuleMessage(Me,
Localization.GetString("Survey_MustBeSignedIn", Me.LocalResourceFile),
DotNetNuke.UI.Skins.Controls.ModuleMessage.ModuleMessageType.YellowWarning)
                                                        cmdSubmit.Visible = False
                                                        DisplaySurvey()
        End If
Else
' Survey is not personalized, so just show Survey items
        DisplaySurvey()
End If
```

Now that we covered how to read personalization values, while we're still in the Survey.ascx.vb class file, let's look at setting a value for personalization key within the cmdSubmit_Click method:

```
::
::
If blnPersonal = True Then
        ' This means the module vote tracking is using personalization, so set the profile to show they have voted

        DotNetNuke.Services.Personalization.Personalization.SetProfile(ModuleId.ToString, "Voted", True)
::
::
```

In this code block instead of doing a GetProfile as in the Page_Load event, we use the SetProfile method to save our value in the "Voted" key.

Scheduler

In DNN 2, the scheduling provider was implemented into the core framework. This enables developers to execute a process on a scheduled basis in its own process independent of a Web request.

The way the scheduler works is you enter an event definition in the Schedule table within the DNN database. Let's look at the structure and a sample entry for purging the site log, here are the field names along with associated values for this specific entry:

- **ScheduleID**: 2
- **TypeFullName**: DotNetNuke.Services.Log.SiteLog.PurgeSiteLog, DOTNETNUKE – This is the method that will be executed on a scheduled basis.
- **TimeLapse**: 1 – How long between intervals for this to be ran.
- **TimeLapseMeasurement**: d – The unit of measure, in this case, days.
- **RetryTimeLapse**: 2 – This is how long it will be between retrys.
- **RetryTimeLapseMeasurement**: h – The retry measure, in this case hours.
- **RetainHistoryNum**: 10 – How many incidents will be logged in the logging provider.

- **AttachToEvent**: NULL – Specifies if an event should be fired in the APPLICATION_ONSTART event.
- **CatchUpEnabled**: 0 – A flag to specify if this service will recover from missed events.
- **Enabled**: 1 – A flag to specify whether or not this item will be enabled.
- **ObjectDependencies**: SiteLog – Any dependencies for this class, in this case DotNetNuke.Services.Log.SiteLog.

So the idea here is you can enter an item into the schedule table upon module install or do this programmatically. You'll notice in the TypeFullName field we are passing the full name of the library or class that needs to be executed. Let's take a look at the structure of the DotNetNuke.Services.Log.SiteLog.PurgeSiteLog class and see how it is structured in order to log an event.

```
Imports System.Threading

Namespace DotNetNuke.Services.Log.SiteLog

        Public Class PurgeSiteLog
                Inherits DotNetNuke.Services.Scheduling.SchedulerClient
                Public Sub New(ByVal objScheduleHistoryItem As
DotNetNuke.Services.Scheduling.ScheduleHistoryItem)
                        MyBase.new()
                        Me.ScheduleHistoryItem = objScheduleHistoryItem
                End Sub

                Public Overrides Sub DoWork()
                        Try
                                'notification that the event is progressing
                                Me.Progressing()                        'OPTIONAL
                                PurgeSiteLog()
                                Me.ScheduleHistoryItem.Succeeded = True
                                Me.ScheduleHistoryItem.AddLogNote("Site Log
purged.")
                        Catch exc As Exception                  'REQUIRED
                                Me.ScheduleHistoryItem.Succeeded = False
                                Me.ScheduleHistoryItem.AddLogNote("Site Log purge
failed. " + exc.ToString)                       'OPTIONAL
                                'notification that we have errored
                                Me.Errored(exc)                         'REQUIRED
                                'log the exception
```

```
                                        LogException(exc)                     'OPTIONAL
                                End Try
                        End Sub

                        Private Sub PurgeSiteLog()
                      'run scheduled routine.
                      ::
                      ::
                        End Sub
                End Class

End Namespace
```

You can see from the class that we're inheriting the DotNetNuke.Services.Scheduling.SchedulerClient, this provides us with the methods needed to log events into the scheduler. Another item of interest is the DoWork method, you can see it overrides the DoWork method that is located in the SchedulerClient; this is the method that will be executed by the scheduler service.

Logging Provider

Another service provided by DNN is the ability to write items to a log. DNN includes an XML Logging Provider to capture events raised from the DNN core, and modules. This can be useful for writing error events, or any other information that a portal admin may want to keep track of. You can see in the section covering the Scheduling provider, that it makes a call to the Logging provider to add entries into the portal's log.

There are two primary event log entries that you can implement; a standard informational entry, or an entry for trapping an error within your code. First let's look at a regular informational entry:

```
'Write out the log entry for this event
''''''''''''''''''''''''''''''''''''''''''''''''''
Dim objEventLog As New Log.EventLog.EventLogController
Dim objEventLogInfo As New Log.EventLog.EventLogInfo
objEventLogInfo.AddProperty("THREAD ID",
Thread.CurrentThread.GetHashCode().ToString)
objEventLogInfo.AddProperty("TYPE", objSchedulerClient.GetType().FullName)
objEventLogInfo.AddProperty("SOURCE",
objSchedulerClient.ScheduleHistoryItem.ScheduleSource.ToString)
objEventLogInfo.AddProperty("ACTIVE THREADS", ActiveThreadCount.ToString)
objEventLogInfo.AddProperty("FREE THREADS", FreeThreads.ToString)
objEventLogInfo.AddProperty("READER TIMEOUTS", ReaderTimeouts.ToString)
objEventLogInfo.AddProperty("WRITER TIMEOUTS", WriterTimeouts.ToString)
objEventLogInfo.AddProperty("IN PROGRESS",
GetScheduleInProgressCount().ToString)
objEventLogInfo.AddProperty("IN QUEUE", GetScheduleQueueCount().ToString)
objEventLogInfo.LogTypeKey = "SCHEDULER_EVENT_STARTED"
objEventLog.AddLog(objEventLogInfo)
```

The previous example is pulled from the scheduler service. We first call the EventLogController, created an EventLogInfo object. You can see we create log event properties for our EventLogInfo object, and these then contain values that are generated from our logic. Finally the EventLogInfo object is passed to our controller class, which populates the event in the logging provider.

Now let's look at sending exception information to the logging provider:

```
Try
   'do some code.
Catch exc As Exception
    Dim objExceptionLog As New Log.EventLog.ExceptionLogController
    objExceptionLog.AddLog(exc,
Log.EventLog.ExceptionLogController.ExceptionLogType.MODULE_LOAD_EXCEPTION)
End Try
```

In the previous example, we will place our logging event in the catch statement for a try statement that surrounds some logic. If an error occurs, we then create an instance of the ExceptionLogController class, and pass the exception to the AddLog method. In addition to the exception, we will also define a type of Module_Load_Exception, since we're in a custom module.

Localization

A new feature with DNN 3 is localization by using resource files. Now a user can select a language of choice for a DNN portal by modifying their profile within DNN. You can also provide resource files with your module when you distribute it. Localizing your module is goes beyond what we can cover within this guide. More information can be found in the localization document in the documents folder included with the DotNetNuke distribution.

Importing and Exporting

Another feature in DNN is the ability to import and export modules within the portal. We covered this earlier in the guide, but now let's get into the code that enables you to implement this feature in your own modules. This functionality is provided by the DotNetNuke.Entities.Modules.IPortable interface. This interface provides you with the ability to import and export modules.

First thing we need to do is implement the interface within our class, we'll take this from the Text/HTML module:

```
Public MustInherit Class RDMView
        Inherits RDMUserControl
        Implements Entities.Modules.IActionable
        Implements Entities.Modules.IPortable
```

Now that the interface is implemented in the class we need to handle two events. These are the import and exporting of modules. First we'll look in the HTMLModule.ascx file, at the bottom of the class are two functions:

```
Public Function ExportModule(ByVal ModuleID As Integer) As String Implements
Entities.Modules.IPortable.ExportModule
           ' included as a stub only so that the core knows this module Implements
Entities.Modules.IPortable
End Function

Public Sub ImportModule(ByVal ModuleID As Integer, ByVal Content As String, ByVal
Version As String, ByVal UserId As Integer) Implements
Entities.Modules.IPortable.ImportModule
            ' included as a stub only so that the core knows this module Implements
Entities.Modules.IPortable
End Sub
```

You can see that the methods are included in the class to let the core know that we are going to provide import/export capabilities in the function. Now we need to handle the actual data contained in the module so it can be exported. Open the HTMLTextController.vb file to see how this accomplished. As in the HTMLModule.ascx file, we need to implement the IPortable interface in our class, then we need to again create the ExportModule and ImportModule methods, but this time we will be handling the export and import:

```
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>" &
XMLEncode(objHtmlText.DeskTopHTML) & "</desktophtml>"
                        strXML += "<desktopsummary>" &
XMLEncode(objHtmlText.DesktopSummary) & "</desktopsummary>"
                        strXML += "</htmltext>"
            End If
            Return strXML
End Function

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")
     UpdateHtmlText(ModuleID,
xmlHtmlText.SelectSingleNode("desktophtml").InnerText,
xmlHtmlText.SelectSingleNode("desktopsummary").InnerText, UserId)
End Sub
```

In the ExportModule function, we are going to call our GetHtmlText method and pass the current ModuleID, this will return the data for the module instance, which we build an XML export string from. This XML will be saved within the database and can later be imported into another module instance using the ImportModule function.

In the ImportModule function, we call the GetContent method to obtain the data from the exported module, and then populate the current module instance by calling the UpdateHtmlText method of our controller class.

Searching

New to DNN 3.0 is the ability to index module content for searching within a unified interface. In order to enable your module for searching you will need implement the DotNetNuke.Services.Search.ISearchable Interface in your class file for your control. For example, the following code is from the Announcements module which utilizes the search interface of DNN. In the code behind you would implement the ISearchable interface in your class:

```
::
::
Public Class AnnouncementsController
    Implements DotNetNuke.Services.Search.ISearchable
::
```

Now that you have implemented the class, you now need to populate the catalog for searching:

```
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
```

You can see from the preceding code block we made a call to our Info class for our module, just as we would when we bind to a control within our ascx file, but in this case the results are going to populate the SearchItemInfo, which will populate the DNN index with data from our module. Now that the index is populated with data, users of your portal will be able to search your module's information from a unified interface within DNN.

Text Editor

Another provider in DNN is the Text Editor, currently the default provider is the FreeTextBox control by John Dyer (http://www.freetextbox.com/). You can swap this provider for another by creating your own provider and changing the web.config to point to the new provider. In this guide, we will provide a brief overview on how you can implement the FTB into your own modules.

In the following sample code you can find in the Text/HTML module project in the DesktopModules directory.

In order to implement the Text Editor we first need to create a reference to the control in our ascx file. At the top of the ascx control page right under the class call for the appropriate code behind page:

```
<%@ Control language="vb" Inherits="Santry.ResDirMgr.ucEditRecord"
CodeBehind="ucEditRecord.ascx.vb" AutoEventWireup="false" Explicit="True" %>
<%@ Register TagPrefix="dnn" TagName="TextEditor"
Src="~/controls/TextEditor.ascx"%>
```

You can see we registered the tag prefix for the TextEditor control so we can insert it into our control. Later in the code you can see where the control is inserted into the control for display:

```
<dnn:TextEditor id="TextEditor1" Width="600" Height="400" runat="server"
HtmlEncode="False" />
```

Now you can have the functionality of the TextEditor in your own modules. We'll go over the code behind page to see how we declare the module and then modify it's properties. The first thing we do is reference a new interface called the DotNetNuke.UI.UserControls.TextEditor:

```
Protected WithEvents TextEditor1 As DotNetNuke.UI.UserControls.TextEditor
```

Now that we created our reference, we modify properties and handle events raised by the control. Like getting the text from the control land assigning it to a variable:

```
strHTML = TextEditor1.Text
```

You don't have to worry about writing code for changing from text to html view, and all the other features that it provides. In addition, you can create your own provider or purchase third party provider and plug it into the portal.

DotNetNuke User Controls

A new interface to DNN 3 is the availability of user controls. With DNN you have more than the standard Web controls provided by ASP.NET, as covered in the Text Editor section, you have the DotNetNuke.UI.UserControls class, that provides you with controls that take advantage of the core framework. Among the most common controls contained in this interface are:

- **Address** – This control provides the address entry control that you see in DNN when registering for an account within the portal.
- **DesktopModuleTitle** – Provides the title area above a module.
- **DesktopPortalBanner** – The banner ad area within the portal, includes the login, banner ad, logo, and other items.
- **DesktopPortalFooter** – The footer control for a portal.
- **DualListControl** – A control containing two list controls for moving data between the two.
- **Help** – Help control for a module. Accepts localization information, and help file references.
- **LabelControl** – Contains properties for storing localization information, help information, and other information provided the DNN core.
- **ListItemComparer** – Used to compare items within a list.
- **ModuleAuditControl** – For auditing a module, such as who created the module and when the module was created.
- **SectionHeadControl** – For creating collapsible areas within your module.
- **TextEditor** – Discussed previously
- **UrlControl** -
- **URLTrackingControl**

Use the object browser within Visual Studio to view more information about the controls and the DNN interfaces. We recommend to use DNN user control wherever possible in order to take advantage of localization, and help integration within the DNN core framework.

Error Handling

DotNetNuke provides developers with a mechanism to present a friendly error to the user, and determine whether to show detailed information on who is logged in. In addition to displaying of errors, DNN will also provide a mechanism for logging the errors into the logging provider for later review.

```
Try
    'Do some code.
Catch exc As Exception                  'Module failed to load
      ProcessModuleLoadException(Me, exc)
End Try
```

In the previous block of code you can see in the try catch statement we make a call to the ProcessModuleLoadException and pass our error to the method. This will raise the error to the portal and perform the error handling features of DNN.

Inter-Module Communication

The Inter Module Communication (IMC) is a very versatile and pointed way to transfer data within modules in your portal. Although IMC can do many things for you, the name is misleading and this section will attempt to organize its merits as well as improper uses.

In 2.0.1 beta DNN, the IMC was enhanced to allow four new properties in the interface. In the original version carried over from IBuySpy to DotNetNuke 1.x series, there was one property: [TEXT], and you could pass a string from a sender module to a receiver module. However there was no way to determine the Sender or Target, or the Type of message, and there was no way to pass an Object. This was remedied and properties were added to facilitate this required functionality. However, the enhancements spent about 6 months in limbo while 2.0 was being compiled, and as it works out, this is a fortunate thing.

A last minute change resolved the dependency of IMC on caching and isolates it from the skinning functionality resolving the interface to be a base function, and IMC in 2.0.1 beta DNN works as desired. Now, the question is, what do I use IMC for, and what do I NOT use it for?

This section will showcase several modules that use IMC currently, and how they implement it. As well it will show circumstances that IMC will not be functional in, even though it "seems" like an appropriate use.

How It Works

Inter Module Communications provides a simple, direct one way communication to other modules within your portal. The nature of this communication is through a raised event I will call iTalk, which is raised to the next postback of that page (tab) and is raised somewhere during PageInit. I say somewhere, because the actual event is raised within the control and event tree, so these will vary depending on the arrangement of your tab.

The values in this event are made accessible to all modules existing within that tab and they can listen for chatter that is aimed at the module type, target, and sender values, I call this iListen.

The ModuleCommunicator class is inherited into your module.

Then ModuleCommunication event is raised within the iTalk module that carries your chatter to the next pageInit, where the OnModuleCommunication Sub in your iListen module can intercept and utilize the values.

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


_

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

How Not to Use It

The most common assumption of IMC is that it will enable you to pass objects between modules. Although this is true, the value will only persist for a single postback. When the page is refreshed the IMC will become nothing. This is easily maintained by the use of Session or viewstate variables that are declared within your modules that look for IMC and store it for that module, user, etc. If the data needs to be intercepted and stored in the database, or used in another concurrent postback, you would implement a custom Listener to attach to that data and process it accordingly.

In no case will the IMC string persist beyond the initial post without programmatic help.

The IMC is often mistaken for a medium to transfer data without the use of a database. In some cases this is correct, but to use IMC in this way and access the data beyond the next postback or request requires a more robust layer, including session or database.

How to Use It

In all cases the IMC is meant to be used within modules on a single tab, or in modules that redirect to another tab and that tab has a module on it that can receive the IMC objects.

The beauty is that you can efficiently pass objects between modules, or cast them to an interim listening device.

1. **Step One**: Determine the [SENDER]: Here you give your modules the ability to attach to only IMC object sets from a certain Sender. This is usually the ModuleName of the sending module, as set in the module definitions. If a Sender is included in the IMC object set, and a Listener module is configured to Receive from the Sender, then it will only attach to an IMC object set for that Sender.
2. **Step Two**: Determine the [TARGET]: Here you give your modules the ability to attach to only IMC object sets aimed at a certain Target. This is usually the ModuleName of the receiving module, as set in the module definitions. If a Module is configured to listen to only the Target of MyModule then it will only attach to an IMC object set for that target.
3. **Step Three**: Determine the [TYPE]:: a configurable property where you set the type in the Talker and then the Listener looks for that type. If a Module is configured to Listen to only the Type of MyModule then it will only attach to a IMC object set of that type.
4. **Step Four**: Determine the [VALUE]: This is the only Object property in IMC, and you can pass any complex object between modules. This includes a dataset, a control within state, or any other thing you can imagine. The requirement is for a listener to know what to do with the Object and its state.
5. **Step Five**: Determine the [TEXT]: this is a relatively straightforward answer... a simple text string. In the original IMC, this was the only property.

Distributing Your Module

We covered every layer of the DotNetNuke module architecture; now let's get the module ready for distribution. In the following steps we're going to create a private assembly package out of our survey module so it can distributed easily to other DNN portals.

Creating Your Database for Distribution

We need to create a SQL script for generating the database structure of our database. Remember the database owner and object qualifier strings specified in the data provider class? Well we will need to specify these variables in our SQL scripts as well for deployment. DotNetNuke will then look check the values located. Creating this script takes a little bit of extra effort than simply going to SQL Query Analyzer and exporting creation scripts from your database. Yes, that's how we start off, by first generating creation scripts via Query Analyzer or Enterprise Manager, but we need to add some additional information to these scripts in order for DNN to properly create your database structure.

After you generate SQL creation scripts, you will need to add the following prefixes in front of your table, and stored procedure references within the newly created scripts.

{databaseOwner}{objectQualifier}

These prefixes will then be replaced by the values defined within the web.config at database generation time during module installation. So in the following example of a SQL script to generate a table:

```
if not exists (select * from dbo.sysobjects where
 id = object_id(N'{databaseOwner}{objectQualifier}[SurveyOptions]')
 and OBJECTPROPERTY(id, N'IsTable') = 1)
CREATE TABLE {databaseOwner}{objectQualifier}SurveyOptions (
	[SurveyOptionID] [int] IDENTITY (1, 1) NOT NULL ,
	[SurveyID] [int] NOT NULL ,
	[ViewOrder] [int] NOT NULL ,
	[OptionName] [nvarchar] (500) NOT NULL ,
	[Votes] [int] NOT NULL
) ON [PRIMARY]

GO
```

You can see where we highlighted the prefixes, then in your stored procedure generation script you will see these are added in any references to the tables:

```
create procedure {databaseOwner}{objectQualifier}UpdateSurveyOption

@SurveyOptionID int,
@OptionName     nvarchar(500),
@ViewOrder      int

as

update {objectQualifier}SurveyOptions
set    OptionName = @OptionName,
       ViewOrder = @ViewOrder
where  SurveyOptionID = @SurveyOptionID

GO
```

Once your SQL script is complete, save it using the following naming convention:

versionnum.dataprovidertype, for example: 01.00.00.SQLDataProvider

This will then get included in your distribution package. Depending on which data provider is configured in DNN, it will look for the appropriate database generation script based on the extension value.

One other file you should create as part of your distribution is an uninstall script. This script basically contains drop routines for deleting your tables and stored procedure in the event a portal admin wants to remove your module from their DNN install. The naming convention of this uninstall script is as follows:

uninstall.dataprovidertype, for example: uninstall.SqlDataProvider

NOTE: Object Qualifier, and DatabaseOwner

It is a requirement to have these prefixes added to your module database scripts. These prefixes allow multiple DNN implementations exist within a single database. In addition, if your module tables need to perform joins with other tables in the DNN core structure, these prefixes need to be defined in order for those SQL statements to work.

Creating Your DNN Distribution Definition

The DNN assembly install definition file is an XML file that describes the structure of your module. It contains several pieces of information in order for DNN to install your module within the application. This definition file will mirror the architecture for your specific module which is going to vary based upon how you structure your module. Let's look at a sample that comes with the survey module.

```
<?xml version="1.0" encoding="utf-8" ?>
   <dotnetnuke version="3.0" type="Module">
   <folders>
     <folder>
       <name>CompanyName - Survey</name>
       <description>Survey allows you to create custom surveys
          to obtain public feedback</description>
       <version>01.00.00</version>
       <resourcefile>SurveyResources.zip</resourcefile>
       <businesscontrollerclass>CompanyName.Survey.SurveyController,
CompanyName.Survey</businesscontrollerclass>
       <modules>
         <module>
           <friendlyname>CompanyName - Survey</friendlyname>
           <controls>
             <control>
               <src>Survey.ascx</src>
               <type>View</type>
               <helpurl>http://www.dotnetnuke.com</helpurl>
             </control>
             <control>
               <key>Edit</key>
               <title>Create Survey</title>
               <src>EditSurvey.ascx</src>
               <iconfile>icon_survey_32px.gif</iconfile>
               <type>Edit</type>
             </control>
             ::
           </controls>
        </module>
      </modules>
```

Let's look at the XML above. You can see we define our module at the top most level. Then we need to let DNN know what our module consists of, this will be the primary user interface or ascx files. In the survey module we have three defined, we provide two here in this example, a view control defined first, and the edit control. You can see within the edit control we define the name of the file used, a key name, title value for the module container, icon, and the type of the control., in this case it is an edit control. Since it is an edit control, we can use the Boolean value provided by DNN to check permissions to see if the user has access to edit functions on the module (remember "IsEditable" from the PortalModuleBase class?).

Also be aware that in the modules you develop you can create a very different definition file than the one used here in this guide. A module can contain several modules, admin, and edit controls. This will all be determined on the complexity of the module being developed and the business needs. Let's look at a Weblogs module as an example of a module containing multiple modules, within one definition file:

```
::
<modules>
	<module>
		<friendlyname>Weblogs</friendlyname>
		<controls>
			<control>
				<src>Weblogs.ascx</src>
				<type>0</type>
			</control>
			<control>
				<key>Edit</key>
				<title>Add Weblog entry</title>
				<src>EditWebLogs.ascx</src>
				<iconfile></iconfile>
				<type>1</type>
			</control>
			<control>
				<key>Options</key>
				<title>Weblog Options</title>
				<src>EditWebLogsOptions.ascx</src>
				<iconfile></iconfile>
				<type>1</type>
				</control>
			</controls>
	</module>
	<module>
		<friendlyname>Weblogs Calendar</friendlyname>
```

```
                <controls>
                        <control>
                                <src>NavCalendar.ascx</src>
                                <type>0</type>
                        </control>
                </controls>
        </module>
        <module>
                <friendlyname>Weblogs Archive</friendlyname>
                <controls>
                        <control>
                                <src>NavArchive.ascx</src>
                                <type>0</type>
                        </control>
                </controls>
        </module>
::
```

You can see in the preceding definition file, we had multiple module sections defined within this one module installation. This is due to the module being a bit complex and requiring multiple controls in order for it to work properly. In this case we have a calendar, and archive modules which are associated with the Weblog. You can add as many module sections as your install needs, and then associate the controls with the modules for your users to interface with.

So now that we defined the basic structure of our module, we need to define the actual files that make up our module. These files include the DLL, the ascx controls, data providers, images, and any other files we use to support our specific application.

```
     ::
    <files>
      <file>
        <name>Survey.ascx</name>
      </file>
      <file>
        <name>EditSurvey.ascx</name>
      </file>
      <file>
        <name>Settings.ascx</name>
      </file>
      <file>
        <name>YourCompanyName.Survey.dll</name>
      </file>
```

```
        <file>
          <name>YourCompanyName.Survey.SqlDataProvider.dll</name>
        </file>
        <file>
          <name>01.00.00.SqlDataProvider</name>
        </file>
        <file>
          <name>Uninstall.SqlDataProvider</name>
        </file>
        <file>
          <name>YourCompanyName.Survey.AccessDataProvider.dll</name>
        </file>
        <file>
          <name>01.00.00.AccessDataProvider</name>
        </file>
        <file>
          <name>Uninstall.AccessDataProvider</name>
        </file>
        <file>
        <file>
              <path>App_LocalResources</path>
              <name>Survey.ascx.resx</name>
            </file>
            <file>
              <path>App_LocalResources</path>
              <name>EditSurvey.ascx.resx</name>
            </file>
            <file>
              <path>App_LocalResources</path>
              <name>Settings.ascx.resx</name>
            </file>
       ::
      </files>
    </folder>
  </folders>
</dotnetnuke>
```

If you look at the code snip, you can see that we define our files to be included in the module, ensure you have all the files included in your zip file. In addition, we have the <path> node to define if a subdirectory to place the files in, in this example, we have resource files for localization. The resource files will be placed in the App_LocalResources directory off of the module's home directory.

Now that this file is created take all your files for your module and zip them up into a compressed zip file. This is the private assembly that you can distribute. Remember, you don't need to include any source files in this distribution, just the compiled assembly, ascx files, and other supporting files---no filename.vb class files are needed.

Now let's cover some points when creating your own definition files.

- Make sure you add the appropriate DNN version attribute value in the DotNetNuke node.
- In addition to the nodes covered in the example you can add a helpurl node to define a Web page for a specific edit, admin, or view control.
- A new node called the businesscontrollerclass has been added to the definition file. This is for defining the associated class for your module, for example:

```
<businesscontrollerclass>CompanyName.Survey.SurveyController   CompanyName.Survey</businessc
```

The Module Definition Validator

DNN provides you with a tool to check the definition file to ensure it is valid before distributing it. To check the validity of your .dnn file, navigate to the Module Definitions tab, which is located under the Host tab within your portal. Once at the Module Definitions tab, scroll to the bottom of the page to view the validator. Click on the browse button to find your .dnn file located on your local machine. This will open an explorer dialog, find your .dnn file, and then click on "Open". You should now see the local path to your .dnn file in the textbox. Click on the "Validate" link button to begin the validation process. If your file is valid, you should see the following message:

Validation Results
CompanyName.Survey.dnn is a valid Module Definition File.

If the file is not valid, it will notify you of the problem.

Packing Up Your Module

Your module will be a compressed zip file that contains all files relevant for the operation of the module. Typically a module distribution is going to contain the following files:

- Company.ModuleName.SQLDataProvider.dll – This is the compiled code from your data provider project. In this example we are using SQL as our backend, but you would name this file according to the physical database provider you are using.
- Company.ModuleName.dll – This is the main module project that is compiled into an assembly.
- All ASCX Files – This includes all user interface files, in this example we would include our Survey.ascx, EditSurvey.ascx, and Settings.ascx file.
- Images – If your module requires any image files, such as icons, or other images, package them in the zip file.
- CSS – If you need to include a specific look for your module. Refer to the next section on style sheet structure.
- Data Provider Definition – Include the physical database provider creation script. If you are going to provide multiple provider in order to support multiple providers for DNN, include them as well.
- Data Provider Uninstall – Include the uninstall script. Again, if you have multiple database providers include an uninstall file for each provider.
- Resource Files for Localization – If you plan on providing multiple languages for your module, you will need to distribute the supported resource files in your distribution.

Quick Tip: Compile In Release Mode

Make sure you compile your assemblies in release mode and not debug mode before adding to your distribution package. This will reduce the amount of overhead required in order to execute your package.

Note: You do not need to include the source code in your distribution package.

Style Sheets

One thing we want to cover here is some standard naming conventions for your cascading style sheets. Primarily your goal as a module developer in order to avoid CSS conflicts with the core modules or modules developed by other ISVs is to create a unique naming convention based on the CompanyName/ModuleName convention. For example, in the Survey module's CSS file:

```
/* Survey Custom Style */
.YourCompanyNameSurveyResults
{
   font-family: Tahoma, Arial, Helvetica;
   font-size: 12px;
   font-weight: normal;
}
```

If you do not need proprietary styles for your module you should then use the DNN styles which can be defined within each portal to provide the portal admin greater flexibility in defining styles for their site.

Installing the Assembly

Finally that we have our module packaged up into a distribution, let's cover how to install it into a DNN portal. DNN provides a very simple method for installing modules into the portal via the File Manager. That's the reason for all the work you did creating an assembly, a SQL creation script, and a .dnn file, now it all comes together so an admin can easily install it into their DNN installation. Use the following procedure to install your module:

1. Logon as Host access.
2. Go to the File Manager under the Host menu.
3. Click on the Upload button in the top toolbar of the File Manager.
4. In the Upload File screen, select Custom Module from the radio button selection.
5. Click on the browse button to find your assembly package.
6. Click Add to display the package in the list.
7. Click Upload to upload the assembly to the server.

Once the file is uploaded DNN will look for the .dnn file to find the information it needs in order to configure your module into the portal. The File Manager will display the status of the module installation on the screen. If any errors are encountered during the install process, DNN will let you know by displaying the specific problem on the screen in bold red text.

If everything installs correctly, you should now be able to add your module to a page by selecting it from the drop down menu.

Additional Documentation
For more information on DNN development refer to the appropriate documentation contained in the <approot>/documentation folder. Localization, the client API, Scheduler, and Membership provider are explained in more detail in each respective document.

Review

In this guide we covered the topics that should get you started developing DotNetNuke modules. We covered the implementation of modules within the DNN framework and how an administrator of a portal can modify the properties, like position, look, and security for a module instance.

We then went into the technical and created a Visual Studio.NET project for both the module and the physical database provider.

From there we covered three-tier architecture and how DNN module development falls into this architectural concept. We started with the Data Abstraction Layer, the Business Logic Layer, the User Layer, and finally the physical database provider.

We then covered the ways you can plug into the core framework, whether you want to search, schedule, or provide personalization for your module.

Finally we packed up our module for distribution and installed it into a portal.

We hope this guide will get you productive in developing highly interactive and functional modules. Be sure to visit the resources and be active in the community to stay on top of the most recent activity for DNN.

Best Regards,

The DotNetNuke Team

Additional Information

The DotNetNuke Portal Application Framework is constantly being revised and improved. To ensure that you have the most recent version of the software and this document, please visit the DotNetNuke website at:

http://www.dotnetnuke.com

The following additional websites provide helpful information about technologies and concepts related to DotNetNuke:

DotNetNuke Community Forums
http://www.dotnetnuke.com/tabid/795/Default.aspx

Microsoft® ASP.Net
http://www.asp.net

Open Source
http://www.opensource.org/

W3C Cascading Style Sheets, level 1
http://www.w3.org/TR/CSS1

Errors and Omissions

If you discover any errors or omissions in this document, please email marketing@dotnetnuke.com. Please provide the title of the document, the page number of the error and the corrected content along with any additional information that will help us in correcting the error.

Appendix A: Document History

Version	Last Update	Author(s)	Changes
1.0.0	2004	DNN Team	• Created document
1.0.1	2005	DNN Team	• Revised
1.0.2	Aug 16, 2005	Shaun Walker	• Applied new template