DotNetNuke Data Access

Shaun Walker

Version 1.0.0
Last Updated: June 20, 2006
Category: Data Access

Information in this document, including URL and other Internet Web site references, is subject to change without notice. The entire risk of the use or the results of the use of this document remains with the user.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, places, or events is intended or should be inferred.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Perpetual Motion Interactive Systems, Inc. Perpetual Motion Interactive Systems may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Perpetual Motion, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

DotNetNuke® and the DotNetNuke logo are either registered trademarks or trademarks of Perpetual Motion Interactive Systems, Inc. in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

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 Data Access

Introduction

The ultimate goal of DotNetNuke is to create a portal framework which provides a solid foundation of building block services that developers can leverage to create a robust application. One of the key functions in any application is data access. The .NET Framework provides a variety of methods for performing data access and from an architecture perspective it feels a bit like navigating a complex maze when it comes to selecting the optimal solution to meet your needs. This whitepaper will attempt to reduce the complexity and provide justification for the data access strategy implemented in the DotNetNuke application.

Strategy

Although a variety of literature exists on the various data access methods in the .NET Framework, the majority of it is too high level to actually apply in real world situations. The various methods are usually discussed in terms of their specific strengths and weaknesses but at the end of the day, the developer is still left with lingering questions as to the best overall choice for their application. The ultimate dilemma is centered around the fact that each data access method is best suited to different application use cases. In theory this is great; however in practice, each developer is looking for a consistent data access strategy to apply across the entire enterprise.

A consistent data access strategy has some key benefits to the organization. The fact that the data access pattern is consistently defined results in higher productivity as the developer is not required to waste time selecting a data access method for each task. This results in improved code maintainability as the pattern is implemented consistently in all application areas. The risk of poor data access choices are minimized and the integrity of the code is increased through the centralization of data access components.

The concept of a consistent data access strategy certainly opposes the notion that each business requirement needs to be matched to an optimal data access method.

The philosophy of choosing a specific data access method for each task should theoretically result in the best overall performance for your application (assuming you made the appropriate selection in all cases). However, this perceived benefit is far overshadowed by the liability in terms of inconsistent development practices.

Falling back on traditional concepts known as the 80/20 rule, DotNetNuke has focused on providing a consistent data access strategy which achieves the optimal goals in 80 percent of application use cases. In the other 20 percent, it is up to the organization to decide whether the performance requirements necessitate the implementation of a different data access method for the task; while at the same time accepting the consequences outlined above.

Requirements

One of the key requirements of DotNetNuke is to provide an implementation of the application that supports multiple data stores.

Due to the fact we require the ultimate in flexibility and performance in terms of communicating with external data stores, we chose to discard the generic data access approach and build the application to take advantage of the database-native feature set (ie. .NET managed providers, proprietary SQL dialects, stored procedures, etc…). The tradeoff that we made when choosing to go with a database-specific access class was that we would need to write a separate data access layer for each database platform we wanted to support and hence the application would contain more code. While the data access layers share much common code, each is clearly targeted for use with a specific database.

In order to simplify the use of the database access classes we elected to use the Provider Pattern (also known as the Factory Design Pattern as outlined by the Gang of Four – GOF), with reflection being used to dynamically load the correct data access objects at runtime. The factory is implemented as follows: an abstract class is created with a method declared for each method that must be exposed by the database access classes. For each database that we want to support, we create a concrete class that implements the database specific code to perform each of the operations in the abstract class or "contract." To support the runtime determination of which concrete class to load, we also include an Instance() method which is the factory itself and relies on our generic Provider class to read a value from configuration file to determine which assembly to load using reflection. Due to the fact that reflection is very expensive in terms of application performance, we store the constructor of the data provider class in the cache.

Why abstract classes instead of interfaces? This is due to the fact that interfaces are immutable (static) and as a result do not lend themselves to versioning. Because

interfaces do not support implementation inheritance, the pattern that applies to classes does not apply to interfaces. Adding a method to an interface is equivalent to adding an abstract method to a base class; any class that implements the interface will break because the class does not implement the new method.

The following diagram shows how the business logic, factory, and databases access classes interact with each other. The key advantage of the solution built is that the database access class can be compiled after the business logic classes as long as the data access class implements the DataProvider abstract class methods. This means that should we want to create an implementation for another database, we do not need to change the business logic tier (or UI tier). The steps to create another implementation are:

- Create the database access classes for the new database which implement the DataProvider abstract class.
- Compile the class into an assembly.
- Test and deploy the new data assembly to a running server.
- Change the configuration file to point to the new database access class.
- No changes or recompiles need to be performed on the business logic components.

Configuration

The web.config file contains a number of critical sections to enable the DataProvider pattern. The first section registers the Providers and their corresponding ConfigurationSectionHandlers. Although in this instance we only have a single section specified for the DotNetNuke group, we could use the exact same method to configure other providers (ie. abstract authentication providers, etc…). The only catch is that the section name value must be implemented elsewhere in the web.config file.

```
    <configSections>
        <sectionGroup name="dotnetnuke">
            <section name="data" type="DotNetNuke.ProviderConfigurationHandler,
DotNetNuke" />
        </sectionGroup>
    </configSections>
```

The following section is retained for legacy modules which rely on the old method of data access:

```
   <appSettings>
      <add key="connectionString"
value="Server=localhost;Database=DotNetNuke;uid=sa;pwd=;" />
   </appSettings>
```

And finally the meat of the Provider model. The <data> section name within the <dotnetnuke> group (as described in the configSections above) should contain a defaultProvider attribute which relates to a specific instance in the <providers> collection below.

The defaultProvider is used as the single switch for changing from one provider to another. If a default provider is not specified, the first item in the collection is considered the default.

The <data> section also includes a <providers> collection specification where all of the implementations for <data> are identified. At a bare minimum, each provider must include a name, type, and providerPath attribute (name is generic but usually refers to the classname, type refers to the strong classname of the provider, and providerPath refers to the location where provider specific resources such as scripts can be found). Each provider can also have any number of custom attributes as well.

```
    <dotnetnuke>
        <data defaultProvider="SqlDataProvider" >
            <providers>
        <clear/>
                <add name = "SqlDataProvider"
                        type = "DotNetNuke.Data.SqlDataProvider,
DotNetNuke.SqlDataProvider"
                        connectionString =
"Server=localhost;Database=DotNetNuke;uid=sa;pwd=;"
                       providerPath = "~\Providers\DataProvider\SqlDataProvider\"
                        objectQualifier = "DotNetNuke"
                        databaseOwner = "dbo"
                />
                <add name = "AccessDataProvider"
                        type = "DotNetNuke.Data.AccessDataProvider,
DotNetNuke.AccessDataProvider"
                        connectionString = "PROVIDER=Microsoft.Jet.OLEDB.4.0;"
                       providerPath = "~\ Providers\DataProvider\AccessDataProvider\"
                        objectQualifier = "DotNetNuke"
                        databaseFilename = "DotNetNuke.resources"
                />
            </providers>
        </data>
    </dotnetnuke>
```

The following specification rules are in effect for defining nodes within the "providers" collection.

The <providers> configuration section contains one or more <add>, <remove>, or <clear> elements. The following rules apply when processing these elements:

It is not an error to declare an empty <providers /> element.
Providers inherit items from parent configuration <add> statements.
It is an error to redefine an item using <add> if the item already exists or is inherited.
It is an error to <remove> a non-existing item.
It is not an error to <add>, <remove>, and then <add> the same item again.
It is not an error to <add>, <clear>, and then <add> the same item again.
<clear> removes all inherited items and items previously defined, e.g. an <add> declared before a <clear> is removed while an <add> declared after a <clear> is not removed.

<add>	
Description	Adds a data provider.
Attributes	name – Friendly name of the provider. type – A class that implements the required provider interface. The value is a fully qualified reference to an assembly. providerPath - the location where provider specific resources such as scripts can be found Other name/value pairs – Additional name value pairs may be present. All name/value pairs are the responsibility of the provider to understand.

<remove>	
Description	Removes a named data provider.
Attributes	name – Friendly name of the provider to remove.

<clear>	
Description	Removes all inherited providers.

\Components\Provider.vb

The Provider.vb class provides all of the implementation details for loading the provider information from the web.config file and applying the <add>, <remove>, <clear> processing rules. It is a generic class which is not only applicable to data access.

\Components\DataProvider.vb

The DataProvider.vb is the abstract class containing all data access methods for DotNetNuke. It contains an Instance() method which is the factory itself and loads the appropriate assembly at runtime based on the web.config specification.

```
        ' provider constants - eliminates need for Reflection later
        Private Const [ProviderType] As String = "data" ' maps to <sectionGroup> in
web.config

        ' create a variable to store the reference to the instantiated object
        Private Shared objProvider As DataProvider

        Public Shared Function Instance() As DataProvider

            ' does the provider reference already exist?
            If objProvider Is Nothing Then

                Dim strCacheKey As String = [ProviderType] & "provider"

                ' use the cache because the reflection used later is expensive
                Dim objType As Type = CType(DataCache.GetCache(strCacheKey), Type)

                If objType Is Nothing Then

                    ' Get the provider configuration based on the type
                    Dim objProviderConfiguration As ProviderConfiguration =
ProviderConfiguration.GetProviderConfiguration([ProviderType])

                    ' The assembly should be in \bin or GAC, so we simply need to get an
instance of the type
                    Try

                        ' Get the typename of the Core DataProvider from web.config
                        Dim strTypeName As String =
CType(objProviderConfiguration.Providers(objProviderConfiguration.DefaultProvider),
Provider).Type

                        ' use reflection to get the type of the class that implements the
provider
                        objType = Type.GetType(strTypeName, True)

                        ' insert the type into the cache
                        DataCache.SetCache(strCacheKey, objType)

                    Catch e As Exception

                        ' Could not load the provider - this is likely due to binary
compatibility issues

                    End Try
```

```
                End If

                ' save the reference
                objProvider = CType(Activator.CreateInstance(objType), DataProvider)

            End If

            Return objProvider

        End Function
```

All data access methods are defined as MustOverride which means that any data provider derived from this class must provide implementations for these methods. This defines the abstract class contract between the Business Logic Layer and the Data Access Layer.

```
        ' links module
        Public MustOverride Function GetLinks(ByVal ModuleId As Integer) As IDataReader
        Public MustOverride Function GetLink(ByVal ItemID As Integer, ByVal ModuleId As
Integer) As IDataReader
        Public MustOverride Sub DeleteLink(ByVal ItemID As Integer)
        Public MustOverride Sub AddLink(ByVal ModuleId As Integer, ByVal UserName As
String, ByVal Title As String, ByVal Url As String, ByVal MobileUrl As String, ByVal
ViewOrder As String, ByVal Description As String, ByVal NewWindow As Boolean)
        Public MustOverride Sub UpdateLink(ByVal ItemId As Integer, ByVal UserName As
String, ByVal Title As String, ByVal Url As String, ByVal MobileUrl As String, ByVal
ViewOrder As String, ByVal Description As String, ByVal NewWindow As Boolean)
```

Data Access Layer (DAL)

The Data Access Layer must implement the methods contained in the DataProvider abstract class. However, each DAL provider may be very different in its actual implementation of these methods. This approach allows the provider the flexibility to choose its own database access protocol (ie. managed .NET, OleDB, ODBC, etc…). It also allows the provider to deal with proprietary differences between database platforms (ie. stored procedures, SQL language syntax, @@IDENTITY).

Each data provider must specify an implementation for its custom attributes defined in the web.config file.

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

Namespace DotNetNuke.Data
```

```
    Public Class SqlDataProvider

        Inherits DataProvider

        Private Const ProviderType As String = "data"

        Private _providerConfiguration As ProviderConfiguration =
ProviderConfiguration.GetProviderConfiguration(ProviderType)
        Private _connectionString As String
        Private _providerPath As String
        Private _objectQualifier As String
        Private  databaseOwner As String

        Public Sub New()

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

            ' Read the attributes for this provider
            _connectionString = objProvider.Attributes("connectionString")

            _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

        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
```

Data access methods must be designed as simple queries (ie. single SELECT, INSERT, UPDATE, DELETE) so that they can be implemented on all database platforms. Business logic such as conditional branches, calculations, or local variables should be implemented in the Business Logic Layer so that it is abstracted from the database and centralized within the application. This simplistic approach to data access may take some getting used to if you frequently work with rich SQL languages variants that allow you to implement programming logic at the database level (ie. Transact-SQL stored procedures which perform either an INSERT or UPDATE based on the specification of an identifier).

The SQL Server / MSDE DataProvider included with DotNetNuke uses Stored Procedures as a best practice data access technique.

```
        ' links module
        Public Overrides Function GetLinks(ByVal ModuleId As Integer) As IDataReader
            Return CType(SqlHelper.ExecuteReader(ConnectionString, DatabaseOwner &
ObjectQualifier & "GetLinks", ModuleId), IDataReader)
        End Function
        Public Overrides Function GetLink(ByVal ItemId As Integer, ByVal ModuleId As
Integer) As IDataReader
            Return CType(SqlHelper.ExecuteReader(ConnectionString, DatabaseOwner &
ObjectQualifier & "GetLink", ItemId, ModuleId), IDataReader)
        End Function
        Public Overrides Sub DeleteLink(ByVal ItemId As Integer)
            SqlHelper.ExecuteNonQuery(ConnectionString, DatabaseOwner & ObjectQualifier &
"DeleteLink", ItemId)
        End Sub
        Public Overrides Function AddLink(ByVal ModuleId As Integer, ByVal UserName As
String, ByVal Title As String, ByVal Url As String, ByVal MobileUrl As String, ByVal
ViewOrder As String, ByVal Description As String, ByVal NewWindow As Boolean) As Integer
            Return CType(SqlHelper.ExecuteScalar(ConnectionString, DatabaseOwner &
ObjectQualifier & "AddLink", ModuleId, UserName, Title, Url, MobileUrl,
GetNull(ViewOrder), Description, NewWindow), Integer)
        End Function
        Public Overrides Sub UpdateLink(ByVal ItemId As Integer, ByVal UserName As
String, ByVal Title As String, ByVal Url As String, ByVal MobileUrl As String, ByVal
ViewOrder As String, ByVal Description As String, ByVal NewWindow As Boolean)
            SqlHelper.ExecuteNonQuery(ConnectionString, DatabaseOwner & ObjectQualifier &
"UpdateLink", ItemId, UserName, Title, Url, MobileUrl, GetNull(ViewOrder), Description,
NewWindow)
        End Sub
```

The Microsoft Access data provider included with DotNetNuke also uses stored procedures (stored queries) but does not have the Auto Parameter Discovery feature (CommandBuilder.DeriveParameters) therefore the parameters must be explicitly declared. Also notice that Access does not support @@IDENTITY to return the unique auto number generated; therefore, it must be retrieved in a subsequent database call (GetLinkIdentity).

```
        ' links module
        Public Overrides Function GetLinks(ByVal ModuleId As Integer) As IDataReader
            Return CType(OleDBHelper.ExecuteReader(ConnectionString,
CommandType.StoredProcedure, ObjectQualifier & "GetLinks", _
                New OleDbParameter("@ModuleId", ModuleId)), IDataReader)
        End Function
        Public Overrides Function GetLink(ByVal ItemId As Integer, ByVal ModuleId As
Integer) As IDataReader
            Return CType(OleDBHelper.ExecuteReader(ConnectionString,
CommandType.StoredProcedure, ObjectQualifier & "GetLink", _
                New OleDbParameter("@ItemId", ItemId), _
                New OleDbParameter("@ModuleId", ModuleId)), IDataReader)
        End Function
        Public Overrides Sub DeleteLink(ByVal ItemId As Integer)
            OleDBHelper.ExecuteNonQuery(ConnectionString, CommandType.StoredProcedure,
ObjectQualifier & "DeleteLink", _
                New OleDbParameter("@ItemId", ItemId))
        End Sub
        Public Overrides Function AddLink(ByVal ModuleId As Integer, ByVal UserName As
String, ByVal Title As String, ByVal Url As String, ByVal MobileUrl As String, ByVal
ViewOrder As String, ByVal Description As String, ByVal NewWindow As Boolean) As Integer
            OleDBHelper.ExecuteNonQuery(ConnectionString, CommandType.StoredProcedure,
ObjectQualifier & "AddLink", _
                New OleDbParameter("@ModuleId", ModuleId),
                New OleDbParameter("@UserName", UserName), _
                New OleDbParameter("@Title", Title),
                New OleDbParameter("@Url", Url), _
                New OleDbParameter("@MobileUrl", MobileUrl), _
                New OleDbParameter("@ViewOrder", GetNull(ViewOrder)), _
                New OleDbParameter("@Description", Description),
                New OleDbParameter("@NewWindow", NewWindow))

            Return CType(OleDBHelper.ExecuteScalar(ConnectionString,
CommandType.StoredProcedure, ObjectQualifier & "GetLinkIdentity", _
                New OleDbParameter("@ModuleId", ModuleId),
                New OleDbParameter("@Title", Title)), Integer)
        End Function
        Public Overrides Sub UpdateLink(ByVal ItemId As Integer, ByVal UserName As
String, ByVal Title As String, ByVal Url As String, ByVal MobileUrl As String, ByVal
ViewOrder As String, ByVal Description As String, ByVal NewWindow As Boolean)
            OleDBHelper.ExecuteNonQuery(ConnectionString, CommandType.StoredProcedure,
ObjectQualifier & "UpdateLink", _
                New OleDbParameter("@ItemId", ItemId),
                New OleDbParameter("@UserName", UserName), _
                New OleDbParameter("@Title", Title), _
                New OleDbParameter("@Url", Url), _
                New OleDbParameter("@MobileUrl", MobileUrl),
                New OleDbParameter("@ViewOrder", GetNull(ViewOrder)), _
                New OleDbParameter("@Description", Description),
                New OleDbParameter("@NewWindow", NewWindow))
        End Sub
```

Database Scripts

DotNetNuke contains an Auto Upgrade feature which allows the application to upgrade the database automatically when a new application version is deployed. Scripts must be named according to version number and dataprovider (ie. 02.00.00.SqlDataProvider) and must be located in the directory specified in the providerPath attribute in the web.config file. Dynamic substitution can be implemented in scripts by overriding the ExecuteScript method in the provider implementation. This can be useful for object naming and security specifications.

```
create procedure {databaseOwner}{objectQualifier}GetLinks

@ModuleId int

as

select ItemId,
       CreatedByUser,
       CreatedDate,
       Title,
       Url,
       ViewOrder,
       Description,
       NewWindow
from {objectQualifier}Links
where  ModuleId = @ModuleId
order by ViewOrder, Title

GO
```

Since many databases do not have a rich scripting language, each provider also has the ability to implement the UpgradeDatabaseSchema method which can be used to programmatically alter the database structure.

```
        Public Overrides Sub UpgradeDatabaseSchema(ByVal Major As Integer, ByVal Minor As
Integer, ByVal Build As Integer)
            ' add your database schema upgrade logic related to a specific version. This
is used for data stores which do not have a rich scripting language.
            Dim strVersion As String = Major.ToString & "." & Minor.ToString & "." &
Build.ToString
            Select Case strVersion
                Case "02.00.00"
            End Select
        End Sub
```

SQL Language Syntax

SQL Server and MSDE have a rich scripting language called Transact-SQL which supports local variables, conditional branches (if …. then … else), and looping. Script can be nicely formatted using tabs and spaces for improved readability.

```
drop procedure {databaseOwner}{objectQualifier}GetPortalTabModules
go

create procedure {databaseOwner}{objectQualifier}GetPortalTabModules

@PortalId int,
@TabId int

as

select {objectQualifier}Modules.*,
       {objectQualifier}Tabs.AuthorizedRoles,
       {objectQualifier}ModuleControls.ControlSrc,
       {objectQualifier}ModuleControls.ControlType,
       {objectQualifier}ModuleControls.ControlTitle,
       {objectQualifier}DesktopModules.*
from {objectQualifier}Modules
inner join {objectQualifier}Tabs on {objectQualifier}Modules.TabId =
{objectQualifier}Tabs.TabId
inner join {objectQualifier}ModuleDefinitions on {objectQualifier}Modules.ModuleDefId =
{objectQualifier}ModuleDefinitions.ModuleDefId
inner join {objectQualifier}ModuleControls on
{objectQualifier}ModuleDefinitions.ModuleDefId =
{objectQualifier}ModuleControls.ModuleDefId
inner join {objectQualifier}DesktopModules on
{objectQualifier}ModuleDefinitions.DesktopModuleId =
{objectQualifier}DesktopModules.DesktopModuleId
where  ({objectQualifier}Modules.TabId = @TabId or ({objectQualifier}Modules.AllTabs = 1
and {objectQualifier}Tabs.PortalId = @PortalId))
and    ControlKey is null
order by ModuleOrder
GO
```

MS Access has its own proprietary SQL language called JET SQL. In contrast to Transact-SQL, JET SQL is an extremely limited scripting language as it has no support for local variables, conditional branches, or looping. This limits the queries to very simple CRUD operations. A few syntactical items to note are JET does not accept script formatting using tabs, stored queries need () around the parameter list and parameter names must be wrapped in [@], NULL must be handled using the isnull() function, the word "outer" is dropped from the join clause, the Now() function returns the current date, bit fields use boolean True and False specification, and the query must be completed with a semi-colon (";").

```
drop procedure {objectQualifier}GetPortalTabModules
go

create procedure {objectQualifier}GetPortalTabModules ( [@PortalId] int, [@TabId] int )
as
select {objectQualifier}Modules.*,
       {objectQualifier}Tabs.AuthorizedRoles,
       {objectQualifier}ModuleControls.ControlSrc,
       {objectQualifier}ModuleControls.ControlType,
       {objectQualifier}ModuleControls.ControlTitle,
       {objectQualifier}DesktopModules.*
from   {objectQualifier}Modules, {objectQualifier}Tabs, {objectQualifier}DesktopModules,
{objectQualifier}ModuleDefinitions, {objectQualifier}ModuleControls
where  {objectQualifier}Modules.TabId = {objectQualifier}Tabs.TabId
and    {objectQualifier}Modules.ModuleDefId =
{objectQualifier}ModuleDefinitions.ModuleDefId
and    {objectQualifier}ModuleDefinitions.ModuleDefId =
{objectQualifier}ModuleControls.ModuleDefId
and    {objectQualifier}DesktopModules.DesktopModuleId =
{objectQualifier}ModuleDefinitions.DesktopModuleId
and   ({objectQualifier}Modules.TabId = [@TabId] or ({objectQualifier}Modules.AllTabs =
True and {objectQualifier}Tabs.PortalId = [@PortalId]))
and    isnull(ControlKey) = True
order by ModuleOrder;
GO
```

Another item to note is that MS Access is a file-based database and as a result, requires write access at the file system level. In addition, the *.mdb file extension is not secured by default; therefore, you need to take precautions to ensure your data can not be compromised. As a result, the default web.config file specifies the database filename as DotNetNuke.resources. The *.resources file extension is protected by ASP.NET using the HttpForbiddenHandler in the machine.config and prevents unauthorized download. The Template.resources file distributed with the AccessDataProvider is a template database which contains a baseline schema and data for creating new Access databases.

Database Object Naming

The web.config file includes an attribute named objectQualifer which allows you to specify a prefix for your database obects (ie. DNN_). Web hosting plans usually offer only one SQL Server database that you have to share among all web applications in your account. If you do not specify an object prefix you may have a naming conflict with an application that already uses a specific object name (ie. Users table). Another benefit of prefixing object names is they will be displayed grouped together in tools such as Enterprise Manager for SQL Server when listed alphabetically which simplifies management.

If you are upgrading a pre-2.0 DotNetNuke database, you will want to set the objectQualifier to "". This is because you may have third party modules which do not

use the new DAL architecture and are dependent on specific object names. Setting the objectQualifier on an upgrade will rename all of your core database objects which may result in errors in your custom modules.

Application Blocks

The Microsoft Data Access Application Block (MSDAAB) is a .NET component that contains optimized data access code that helps you call stored procedures and issue SQL text commands against a SQL Server database. We use it as a building block in DotNetNuke to reduce the amount of custom code needed to create, test, and maintain data access functions in the application. We also created an OleDB.ApplicationBlocks.Data assembly for the Microsoft Access data provider based on the MSDAAB code.

In terms of implementation we chose to use the MSDAAB as a black box component rather than include the actual source code in our DAL implementation. This decision helps prevent modification of the MSDAAB code which enables us to upgrade the component seamlessly as new features/fixes become available.

Data Transport

DotNetNuke uses the DataReader for passing collections of data from the Data Access Layer (DAL) to the Business Logic layer (BLL) layer. The DataReader was chosen because it is the fastest performing data transport mechanism available in ADO.NET (a forward only, read only, stream of data). The IDataReader is the base interface for all .NET compatible DataReaders. The abstract IDataReader allows us to pass data between layers without worrying about the data access protocol being used in the actual data provider implementation (ie. SqlClient, OleDB, ODBC, etc…).

Business Logic Layer (BLL)

Good object oriented design recommends the abstraction of the data store from the rest of the application. Abstraction allows the application to build upon an independent set of logical interfaces; thereby reducing the dependency on the physical implementation of the underlying database.

The Business Logic Layer for DotNetNuke is effectively defined in the \Components subfolder. The Business Logic Layer contains the abstract classes which the

Presentation Layer calls for various application services. In terms of data access, the Business Logic Layer forwards API calls to the appropriate data provider using the DataProvider factory mechanism discussed earlier in this document.

Custom Business Objects are an object oriented technique of encapsulating data in user defined structures. Custom Business Objects require some custom coding but the payoff comes in terms of a type safe programming model, disconnected data storage, and serialization. Custom Business Objects offer the maximum flexibility as they allow the application to define the data structures in its own abstract terms; eliminating the dependency on proprietary data containers (ie. RecordSet, DataSet).

What is a type-safe programming model? Please consider the following data access code sample: variable = DataReader("fieldname"). You will notice that a database field value is being assigned to a variable. The problem with this code is that there is no way to ensure the data type of the field matches the data type of the variable; and any errors in the assignment will be reported at run-time. If we use Custom Business Objects the data access code will look like: variable = Object.Property. In this case the compiler will tell us immediately if the data types do not match. Type-safe programming also provides intellisense and improved code readability.

A group of Objects is called a Collection. In DotNetNuke we use a standard ArrayList to represent a group of Custom Business Objects. ArrayLists are intrinsic ASP.NET objects which contain all of the features you would expect in a base collection (add, remove, find, and iterate). The most important feature of the ArrayList for our purposes is that it implements the IEnumerable interface which allows it to be databound to any ASP.NET web control.

In DotNetNuke the Data Access Layer passes information to the Business Logic Layer in the form of a DataReader. A question which may arise in regards to this implementation is why DotNetNuke relies on the DataReader as a data transport container rather than serving up Custom Business Objects directly from the DAL. The answer is that although both options are viable, we believe the complete isolation of the DAL from the BLL has some advantages. For example, consider the task of adding an additional property to a custom business object. In this case the property is only used in the Presentation Layer and does not need to be stored in the database. Using DotNetNuke's approach, there are absolutely no changes required to the DAL implementations as they are not dependent in any way upon the BLL. However, if the DAL was architected to serve custom business objects directly, all of the DAL implementations would need to be recompiled to make them compatible with the BLL structure.

Custom Business Object Helper (CBO)

In an effort to minimize the mundane coding task of populating custom business objects with information from the data layer (passed as DataReader), a generic utility class has been created for this purpose. The class contains two public functions - one for hydrating a single object instance and one for hydrating a collection of objects (ArrayList). The routine assumes that each property in the class definition has a corresponding field in the DataReader. The atomic mapping of information must be identical in terms of Name and Data Type. The code uses Reflection to fill the custom business object with data and close the DataReader.

```
    Public Class CBO

        Private Shared Function GetPropertyInfo(ByVal objType As Type) As ArrayList

            ' Use the cache because the reflection used later is expensive
            Dim objProperties As ArrayList = CType(DataCache.GetCache(objType.Name),
ArrayList)

            If objProperties Is Nothing Then
                objProperties = New ArrayList
                Dim objProperty As PropertyInfo
                For Each objProperty In objType.GetProperties()
                    objProperties.Add(objProperty)
                Next
                DataCache.SetCache(objType.Name, objProperties)
            End If

            Return objProperties

        End Function

        Private Shared Function GetOrdinals(ByVal objProperties As ArrayList, ByVal dr As
IDataReader) As Integer()

            Dim arrOrdinals(objProperties.Count) As Integer
            Dim intProperty As Integer

            If Not dr Is Nothing Then
                For intProperty = 0 To objProperties.Count - 1
                    arrOrdinals(intProperty) = -1
                    Try
                        arrOrdinals(intProperty) =
dr.GetOrdinal(CType(objProperties(intProperty), PropertyInfo).Name)
                    Catch
                        ' property does not exist in datareader
                    End Try
                Next intProperty
            End If

            Return arrOrdinals

        End Function

        Private Shared Function CreateObject(ByVal objType As Type, ByVal dr As
IDataReader, ByVal objProperties As ArrayList, ByVal arrOrdinals As Integer()) As Object
```

```
            Dim objObject As Object = Activator.CreateInstance(objType)
            Dim intProperty As Integer

            ' fill object with values from datareader
            For intProperty = 0 To objProperties.Count - 1
                If CType(objProperties(intProperty), PropertyInfo).CanWrite Then
                    If arrOrdinals(intProperty) <> -1 Then
                        If IsDBNull(dr.GetValue(arrOrdinals(intProperty))) Then
                            ' translate Null value
                            CType(objProperties(intProperty),
PropertyInfo).SetValue(objObject, Null.SetNull(CType(objProperties(intProperty),
PropertyInfo)), Nothing)
                        Else
                            Try
                                ' try implicit conversion first
                                CType(objProperties(intProperty),
PropertyInfo).SetValue(objObject, dr.GetValue(arrOrdinals(intProperty)), Nothing)
                            Catch ' data types do not match
                                Try
                                    Dim pType As Type = CType(objProperties(intProperty),
PropertyInfo).PropertyType
                                    'need to handle enumeration conversions differently
than other base types
                                    If pType.BaseType.Equals(GetType(System.Enum)) Then
                                        CType(objProperties(intProperty),
PropertyInfo).SetValue(objObject, System.Enum.ToObject(pType,
dr.GetValue(arrOrdinals(intProperty))), Nothing)
                                    Else
                                        ' try explicit conversion
                                        CType(objProperties(intProperty),
PropertyInfo).SetValue(objObject,
Convert.ChangeType(dr.GetValue(arrOrdinals(intProperty)), pType), Nothing)
                                    End If
                                Catch
                                    ' error assigning a datareader value to a property
                                End Try
                            End Try
                        End If
                    Else ' property does not exist in datareader
                        CType(objProperties(intProperty),
PropertyInfo).SetValue(objObject, Null.SetNull(CType(objProperties(intProperty),
PropertyInfo)), Nothing)
                    End If
                End If
            Next intProperty

            Return objObject

        End Function

        Public Shared Function FillObject(ByVal dr As IDataReader, ByVal objType As Type)
As Object

            Dim objFillObject As Object
            Dim intProperty As Integer

            ' get properties for type
            Dim objProperties As ArrayList = GetPropertyInfo(objType)

            ' get ordinal positions in datareader
            Dim arrOrdinals As Integer() = GetOrdinals(objProperties, dr)

            ' read datareader
```

```
            If dr.Read Then
                ' fill business object
                objFillObject = CreateObject(objType, dr, objProperties, arrOrdinals)
            Else
                objFillObject = Nothing
            End If

            ' close datareader
            If Not dr Is Nothing Then
                dr.Close()
            End If

            Return objFillObject

        End Function

        Public Shared Function FillCollection(ByVal dr As IDataReader, ByVal objType As
Type) As ArrayList

            Dim objFillCollection As New ArrayList()
            Dim objFillObject As Object
            Dim intProperty As Integer

            ' get properties for type
            Dim objProperties As ArrayList = GetPropertyInfo(objType)

            ' get ordinal positions in datareader
            Dim arrOrdinals As Integer() = GetOrdinals(objProperties, dr)

            ' iterate datareader
            While dr.Read
                ' fill business object
                objFillObject = CreateObject(objType, dr, objProperties, arrOrdinals)
                ' add to collection
                objFillCollection.Add(objFillObject)
            End While

            ' close datareader
            If Not dr Is Nothing Then
                dr.Close()
            End If

            Return objFillCollection

        End Function

        Public Shared Function InitializeObject(ByVal objObject As Object, ByVal objType
As Type) As Object

            Dim intProperty As Integer

            ' get properties for type
            Dim objProperties As ArrayList = GetPropertyInfo(objType)

            ' initialize properties
            For intProperty = 0 To objProperties.Count - 1
                If CType(objProperties(intProperty), PropertyInfo).CanWrite Then
                    CType(objProperties(intProperty), PropertyInfo).SetValue(objObject,
Null.SetNull(CType(objProperties(intProperty), PropertyInfo)), Nothing)
                End If
            Next intProperty

            Return objObject
```

```
        End Function
        Public Shared Function Serialize(ByVal objObject As Object) As XmlDocument
            Dim objXmlSerializer As New XmlSerializer(objObject.GetType())
            Dim objStringBuilder As New StringBuilder
            Dim objTextWriter As TextWriter = New StringWriter(objStringBuilder)
            objXmlSerializer.Serialize(objTextWriter, objObject)
            Dim objStringReader As New StringReader(objTextWriter.ToString())
            Dim objDataSet As New DataSet
            objDataSet.ReadXml(objStringReader)
            Dim xmlSerializedObject As New XmlDocument
            xmlSerializedObject.LoadXml(objDataSet.GetXml())
            Return xmlSerializedObject
        End Function
    End Class
```

NULL Handling

Nearly every data store has a construct to specify when a field value has not been explicitly specified. In most relational database management systems this construct is known as a NULL value. From an application perspective, passing NULL values between the Presentation Layer and Data Access Layer is an architectural challenge. This is because the Presentation Layer must be abstracted from database specific details; yet, it must be able to specify when a property value has not been explicitly set. This is further complicated by the fact that the .NET Framework native data types are not capable of representing the NULL value returned from the database (an exception will be thrown if you attempt to perform this operation). In addition, each data store has its own proprietary implementation for NULL. The only reasonable solution is to create an abstract translation service which can be used to encode/decode NULL values between application layers.

At first glance you may think the "Nothing" keyword in VB.NET would be a perfect candidate for this translation service. Unfortunately, research reveals that the .NET Framework native data types do not exhibit expected behavior when dealing with "Nothing". Although a property assignment to "Nothing" does not throw an exception, the actual property value will vary depending on the data type (String = Nothing, Date

= Date.MinValue, Integer = 0, Boolean = False, etc…) and as a result the native IsNothing() function will not yield consistent results.

In DotNetNuke we have created a generic class for dealing with the NULL issue in a consistent manner across all application layers. Essentially, constant values for each supported data type are used to represent the NULL condition within the application. The constants are then converted to actual database NULL values in each specific data store implementation. A variety of methods are included so that the physical details of the NULL translation service are abstracted from the application.

*Please note that this class only needs to be used in situations where the database field actually allows NULL values (refer to the Implementation Details section for examples). Also note that this class requires the field data type be consistent in both the DAL and BLL layers (ie. the data type of the property in the BLL Info class must be the same as the data type of the parameter passed in the DAL DataProvider).

```
    Public Class Null

        ' define application encoded null values
        Public Shared ReadOnly Property NullInteger() As Integer
            Get
                Return -1
            End Get
        End Property
        Public Shared ReadOnly Property NullDate() As Date
            Get
                Return Date.MinValue
            End Get
        End Property
        Public Shared ReadOnly Property NullString() As String
            Get
                Return ""
            End Get
        End Property
        Public Shared ReadOnly Property NullBoolean() As Boolean
            Get
                Return False
            End Get
        End Property

        ' sets a field to an application encoded null value ( used in Presentation layer
)
        Public Shared Function SetNull(ByVal objField As Object) As Object
            If Not objField Is Nothing Then
                If TypeOf objField Is Integer Then
                    SetNull = NullInteger
                ElseIf TypeOf objField Is Date Then
                    SetNull = NullDate
                ElseIf TypeOf objField Is String Then
                    SetNull = NullString
                ElseIf TypeOf objField Is Boolean Then
                    SetNull = NullBoolean
                Else
                    Throw New NullReferenceException
                End If
            Else ' assume string
                SetNull = NullString
```

```
            End If
        End Function
        ' sets a field to an application encoded null value ( used in BLL layer )
        Public Shared Function SetNull(ByVal objPropertyInfo As PropertyInfo) As Object
            Select Case objPropertyInfo.PropertyType.ToString
                Case "System.Int16", "System.Int32", "System.Int64", "System.Single",
"System.Double", "System.Decimal"
                    SetNull = NullInteger
                Case "System.DateTime"
                    SetNull = NullDate
                Case "System.String", "System.Char"
                    SetNull = NullString
                Case "System.Boolean"
                    SetNull = NullBoolean
                Case Else
                    ' Enumerations default to the first entry
                    Dim pType As Type = objPropertyInfo.PropertyType
                    If pType.BaseType.Equals(GetType(System.Enum)) Then
                        Dim objEnumValues As System.Array = System.Enum.GetValues(pType)
                        Array.Sort(objEnumValues)
                        SetNull = System.Enum.ToObject(pType, objEnumValues.GetValue(0))
                    Else
                        Throw New NullReferenceException
                    End If
            End Select
        End Function

        ' convert an application encoded null value to a database null value ( used in
DAL )
        Public Shared Function GetNull(ByVal objField As Object, ByVal objDBNull As
Object) As Object
            GetNull = objField
            If objField Is Nothing Then
                GetNull = objDBNull
            ElseIf TypeOf objField Is Integer Then
                If Convert.ToInt32(objField) = NullInteger Then
                    GetNull = objDBNull
                End If
            ElseIf TypeOf objField Is Date Then
                If Convert.ToDateTime(objField) = NullDate Then
                    GetNull = objDBNull
                End If
            ElseIf TypeOf objField Is String Then
                If objField Is Nothing Then
                    GetNull = objDBNull
                Else
                    If objField.ToString = NullString Then
                        GetNull = objDBNull
                    End If
                End If
            ElseIf TypeOf objField Is Boolean Then
                If Convert.ToBoolean(objField) = NullBoolean Then
                    GetNull = objDBNull
                End If
            Else
                Throw New NullReferenceException
            End If
        End Function

        ' checks if a field contains an application encoded null value
        Public Shared Function IsNull(ByVal objField As Object) As Boolean
            If objField.Equals(SetNull(objField)) Then
                IsNull = True
```

```
            Else
                IsNull = False
            End If
        End Function

    End Class
```

Implementation Details

The following section provides code samples to demonstrate how the various application layers interface with one another to accomplish data access.

Presentation Layer (UI)

The Presentation Layer is dependent upon the Business Logic Layer for application services. Custom Business Object properties and methods establish the sole interface between these two layers (the Presentation Layer should never reference any data access methods directly).

Get:

```
       ' create a Controller object
    Dim objAnnouncements As New AnnouncementsController

    ' get the collection
    lstAnnouncements.DataSource = objAnnouncements.GetAnnouncements(ModuleId)
    lstAnnouncements.DataBind()
```

Add/Update:

```
    ...
    Private itemId As Integer

    If Not (Request.Params("ItemId") Is Nothing) Then
        itemId = Int32.Parse(Request.Params("ItemId"))
    Else
        itemId = Null.SetNull(itemId)
    End If
    ...

    ' create an Info object
    Dim objAnnouncement As New AnnouncementInfo

    ' set the properties
    objAnnouncement.ItemId = itemId
    objAnnouncement.ModuleId = ModuleId
    objAnnouncement.CreatedByUser = Context.User.Identity.Name
    objAnnouncement.Title = txtTitle.Text
    objAnnouncement.Description = txtDescription.Text
    objAnnouncement.Url = txtExternal.Text
```

```
    objAnnouncement.Syndicate = chkSyndicate.Checked
    If txtViewOrder.Text <> "" Then
        objAnnouncement.ViewOrder = txtViewOrder.Text
    Else
        objAnnouncement.ViewOrder = Null.SetNull(objAnnouncement.ViewOrder)
    End If
    If txtExpires.Text <> "" Then
        objAnnouncement.ExpireDate = txtExpires.Text
    Else
        objAnnouncement.ExpireDate = Null.SetNull(objAnnouncement.ExpireDate)
    End If

    ' create a Controller object
    Dim objAnnouncements As New AnnouncementsController

    If Null.IsNull(itemId) Then
        ' add
        objAnnouncements.AddAnnouncement(objAnnouncement)
    Else
        ' update
        objAnnouncements.UpdateAnnouncement(objAnnouncement)
    End If

    ** Notice the use of the Null.SetNull() and Null.IsNull() helper methods
```

Delete:

```
       ' create a Controller object
    Dim objAnnouncements As New AnnouncementsController

    ' delete the record
    objAnnouncements.DeleteAnnouncement(itemId)
```

Business Logic Layer (BLL)

Each application business function has its own physical file which may consist of multiple related business object definitions. Each business object definition has an Info class to define its properties and a Controller class to define its methods.

```
    Public Class AnnouncementInfo

        ' local property declarations
        Private _ItemId As Integer
        Private  ModuleId As Integer
        Private _UserName As String
        Private  Title As String
        Private _Url As String
        Private _Syndicate As Boolean
        Private _ExpireDate As Date
        Private  Description As String
        Private _ViewOrder As Integer
        Private  CreatedByUser As String
        Private _CreatedDate As Date
        Private _Clicks As Integer

        ' constructor
        Public Sub New()
```

```
    ' custom initialization logic
      End Sub

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

      Public Property ModuleId() As Integer
          Get
              Return _ModuleId
          End Get
          Set(ByVal Value As Integer)
              _ModuleId = Value
          End Set
      End Property

      Public Property Title() As String
          Get
              Return _Title
          End Get
          Set(ByVal Value As String)
              _Title = Value
          End Set
      End Property

      Public Property Url() As String
          Get
              Return _Url
          End Get
          Set(ByVal Value As String)
               Url = Value
          End Set
      End Property

      Public Property Syndicate() As Boolean
          Get
              Return  Syndicate
          End Get
          Set(ByVal Value As Boolean)
              _Syndicate = 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 Description() As String
          Get
              Return _Description
          End Get
          Set(ByVal Value As String)
```

```
                Description = Value
            End Set
        End Property

        Public Property ExpireDate() As Date
            Get
                Return _ExpireDate
            End Get
            Set(ByVal Value As Date)
                _ExpireDate = Value
            End Set
        End Property

        Public Property CreatedByUser() As String
            Get
                Return _CreatedByUser
            End Get
            Set(ByVal Value As String)
                _CreatedByUser = Value
            End Set
        End Property

        Public Property CreatedDate() As Date
            Get
                Return _CreatedDate
            End Get
            Set(ByVal Value As Date)
                _CreatedDate = Value
            End Set
        End Property

        Public Property Clicks() As Integer
            Get
                Return _Clicks
            End Get
            Set(ByVal Value As Integer)
                 Clicks = Value
            End Set
        End Property
    End Class
```

Each field in the data store should map to a corresponding property in the Info class. To allow the generic CBO helper class to automate the transfer of data from IDataReader to Custom Business Object, the directly related class properties and database fields MUST be identical in terms of Name and DataType.

```
    Public Class AnnouncementsController

        Public Function GetAnnouncements(ByVal ModuleId As Integer) As ArrayList

            Return CBO.FillCollection(DataProvider.Instance().GetAnnouncements(ModuleId),
GetType(AnnouncementInfo))

        End Function


        Public Function GetAnnouncement(ByVal ItemId As Integer, ByVal ModuleId As
Integer) As AnnouncementInfo
```

```
            Return CType(CBO.FillObject(DataProvider.Instance().GetAnnouncement(ItemId,
ModuleId), GetType(AnnouncementInfo)), AnnouncementInfo)

        End Function


        Public Sub DeleteAnnouncement(ByVal ItemID As Integer)

            DataProvider.Instance().DeleteAnnouncement(ItemID)

        End Sub

        Public Sub AddAnnouncement(ByVal objAnnouncement As AnnouncementInfo)


            DataProvider.Instance().AddAnnouncement(objAnnouncement.ModuleId,
objAnnouncement.CreatedByUser, objAnnouncement.Title, objAnnouncement.Url,
objAnnouncement.Syndicate, objAnnouncement.ExpireDate, objAnnouncement.Description,
objAnnouncement.ViewOrder)

        End Sub

        Public Sub UpdateAnnouncement(ByVal objAnnouncement As AnnouncementInfo)

            DataProvider.Instance().UpdateAnnouncement(objAnnouncement.ItemId,
objAnnouncement.CreatedByUser, objAnnouncement.Title, objAnnouncement.Url,
objAnnouncement.Syndicate, objAnnouncement.ExpireDate, objAnnouncement.Description,
objAnnouncement.ViewOrder)

        End Sub

    End Class
```

You will notice that the Controller methods which send information to the database (ie. Add and Update) pass a Custom Business Object instance as a parameter. The benefit of this approach is that the object definition is isolated to the BLL which reduces the modifications required to the application when the class definition changes. The individual object properties are then extracted and passed as Scalar values to the Data Access Layer (this is because the DAL is not aware of the BLL object structures).

Data Access Layer (DAL)

DotNetNuke supports multiple data stores using a Provider technique explained earlier in this document. Essentially this involves a base class which forwards data access requests to a concrete data access class implementation determined at runtime.

DataProvider (Base Class)

```
    ' announcements module
    Public MustOverride Function GetAnnouncements(ByVal ModuleId As Integer) As
IDataReader
    Public MustOverride Function GetAnnouncement(ByVal ItemId As Integer, ByVal ModuleId
As Integer) As IDataReader
    Public MustOverride Sub DeleteAnnouncement(ByVal ItemID As Integer)
```

```
    Public MustOverride Sub AddAnnouncement(ByVal ModuleId As Integer, ByVal UserName As
String, ByVal Title As String, ByVal URL As String, ByVal Syndicate As Boolean, ByVal
ExpireDate As Date, ByVal Description As String, ByVal ViewOrder As Integer)
    Public MustOverride Sub UpdateAnnouncement(ByVal ItemId As Integer, ByVal UserName As
String, ByVal Title As String, ByVal URL As String, ByVal Syndicate As Boolean, ByVal
ExpireDate As Date, ByVal Description As String, ByVal ViewOrder As Integer)
```

SqlDataProvider (Concrete Class)

The following helper function is included in the concrete class to isolate the database specific NULL implementation (in this case DBNull.Value for SQL Server) and provide a simplified interface.

```
    ' general
    Private Function GetNull(ByVal Field As Object) As Object
        Return Null.GetNull(Field, DBNull.Value)
    End Function
```

Each method marked as MustOverride in the base class must be included in the concrete class implementation. Notice the use of the GetNull() function described above in the Add/Update methods.

```
    ' announcements module
    Public Overrides Function GetAnnouncements(ByVal ModuleId As Integer) As IDataReader
        Return CType(SqlHelper.ExecuteReader(ConnectionString, DatabaseOwner &
ObjectQualifier & "GetAnnouncements", ModuleId), IDataReader)
    End Function
    Public Overrides Function GetAnnouncement(ByVal ItemId As Integer, ByVal ModuleId As
Integer) As IDataReader
        Return CType(SqlHelper.ExecuteReader(ConnectionString, DatabaseOwner &
ObjectQualifier & "GetAnnouncement", ItemId, ModuleId), IDataReader)
    End Function
    Public Overrides Sub DeleteAnnouncement(ByVal ItemId As Integer)
        SqlHelper.ExecuteNonQuery(ConnectionString, DatabaseOwner & ObjectQualifier &
"DeleteAnnouncement", ItemId)
    End Sub
    Public Overrides Sub AddAnnouncement(ByVal ModuleId As Integer, ByVal UserName As
String, ByVal Title As String, ByVal URL As String, ByVal Syndicate As Boolean, ByVal
ExpireDate As Date, ByVal Description As String, ByVal ViewOrder As Integer)
        SqlHelper.ExecuteNonQuery(ConnectionString, DatabaseOwner & ObjectQualifier &
"AddAnnouncement", ModuleId, UserName, Title, URL, Syndicate, GetNull(ExpireDate),
Description, GetNull(ViewOrder))
    End Sub
    Public Overrides Sub UpdateAnnouncement(ByVal ItemId As Integer, ByVal UserName As
String, ByVal Title As String, ByVal URL As String, ByVal Syndicate As Boolean, ByVal
ExpireDate As Date, ByVal Description As String, ByVal ViewOrder As Integer)
        SqlHelper.ExecuteNonQuery(ConnectionString, DatabaseOwner & ObjectQualifier &
"UpdateAnnouncement", ItemId, UserName, Title, URL, Syndicate, GetNull(ExpireDate),
Description, GetNull(ViewOrder))
    End Sub
```

Caching

High traffic data access methods use web caching to offset the performance demands of the application by reducing the number of calls required to the backend database. The System.Web.Caching.Cache namespace provides tools to programmatically add and retrieve items from the cache. It has a dictionary interface whereby objects are referenced by a string key. This object has a lifetime tied to the application. When the application is restarted, the cache is recreated as well. Note that only serializable objects can be inserted into the cache.

The ASP.NET Cache also supports features for cache management. In cases where a large number of items could potentially bloat the cache, DotNetNuke uses a sliding expiration scale to remove the items if they have not been accessed for more than a specified number of seconds (configurable by the host). This feature is mainly used in the areas which cache Tab settings.

In DotNetNuke, we have created a centralized DataCache class which exposes some simple methods for managing the application cache.

```
    Public Enum CoreCacheType
        Host = 1
        Portal = 2
        Tab = 3
    End Enum

    Public Class DataCache

        Public Shared Function GetCache(ByVal CacheKey As String) As Object

            Dim objCache As System.Web.Caching.Cache = HttpRuntime.Cache

            Return objCache(CacheKey)

        End Function

        Public Shared Sub SetCache(ByVal CacheKey As String, ByVal objObject As Object)

            Dim objCache As System.Web.Caching.Cache = HttpRuntime.Cache

            objCache.Insert(CacheKey, objObject)

        End Sub

        Public Shared Sub SetCache(ByVal CacheKey As String, ByVal objObject As Object,
ByVal SlidingExpiration As Integer)

            Dim objCache As System.Web.Caching.Cache = HttpRuntime.Cache

            objCache.Insert(CacheKey, objObject, Nothing, DateTime.MaxValue,
TimeSpan.FromSeconds(SlidingExpiration))

        End Sub
```

```
        Public Shared Sub SetCache(ByVal CacheKey As String, ByVal objObject As Object,
ByVal AbsoluteExpiration As Date)

            Dim objCache As System.Web.Caching.Cache = HttpRuntime.Cache

            objCache.Insert(CacheKey, objObject, Nothing, AbsoluteExpiration,
TimeSpan.Zero)

        End Sub

        Public Shared Sub RemoveCache(ByVal CacheKey As String)

            Dim objCache As System.Web.Caching.Cache = HttpRuntime.Cache

            If Not objCache(CacheKey) Is Nothing Then
                objCache.Remove(CacheKey)
            End If

        End Sub

        Public Shared Sub ClearCoreCache(ByVal Type As CoreCacheType, Optional ByVal ID
As Integer = -1, Optional ByVal Cascade As Boolean = False)

            Select Case Type
                Case CoreCacheType.Host
                    ClearHostCache(Cascade)
                Case CoreCacheType.Portal
                    ClearPortalCache(ID, Cascade)
                Case CoreCacheType.Tab
                    ClearTabCache(ID)
            End Select

        End Sub

        Private Shared Sub ClearHostCache(ByVal Cascade As Boolean)

            Dim objCache As System.Web.Caching.Cache = HttpRuntime.Cache

            If Not objCache("GetHostSettings") Is Nothing Then
                objCache.Remove("GetHostSettings")
            End If

            If Not objCache("GetPortalByAlias") Is Nothing Then
                objCache.Remove("GetPortalByAlias")
            End If

            If Not objCache("CSS") Is Nothing Then
                objCache.Remove("CSS")
            End If

            If Cascade Then
                Dim objPortals As New PortalController
                Dim objPortal As PortalInfo
                Dim arrPortals As ArrayList = objPortals.GetPortals

                Dim intIndex As Integer
                For intIndex = 0 To arrPortals.Count - 1
                    objPortal = CType(arrPortals(intIndex), PortalInfo)
                    ClearPortalCache(objPortal.PortalID, Cascade)
                Next
            End If

        End Sub
```

```
        Private Shared Sub ClearPortalCache(ByVal PortalId As Integer, ByVal Cascade As
Boolean)

            Dim objCache As System.Web.Caching.Cache = HttpRuntime.Cache

            If Not objCache("GetPortalSettings" & PortalId.ToString) Is Nothing Then
                objCache.Remove("GetPortalSettings" & PortalId.ToString)
            End If

            If Not objCache("GetTabs" & PortalId.ToString) Is Nothing Then
                objCache.Remove("GetTabs" & PortalId.ToString)
            End If

            If Cascade Then
                Dim objTabs As New TabController
                Dim objTab As TabInfo
                Dim arrTabs As ArrayList = objTabs.GetTabs(PortalId)

                Dim intIndex As Integer
                For intIndex = 0 To arrTabs.Count - 1
                    objTab = CType(arrTabs(intIndex), TabInfo)
                    ClearTabCache(objTab.TabID)
                Next
            End If

        End Sub

        Private Shared Sub ClearTabCache(ByVal TabId As Integer)

            Dim objCache As System.Web.Caching.Cache = HttpRuntime.Cache

            If Not objCache("GetTab" & TabId.ToString) Is Nothing Then
                objCache.Remove("GetTab" & TabId.ToString)
            End If

            If Not objCache("GetBreadCrumbs" & TabId.ToString) Is Nothing Then
                objCache.Remove("GetBreadCrumbs" & TabId.ToString)
            End If

            If Not objCache("GetPortalTabModules" & TabId.ToString) Is Nothing Then
                objCache.Remove("GetPortalTabModules" & TabId.ToString)
            End If

        End Sub

    End Class
```

In terms of interacting with the DataCache object it is best to use a specific syntax for retrieving items from the cache so that your application does not suffer stability problems from threading issues.

```
            Me.HostSettings = CType(DataCache.GetCache("GetHostSettings"), Hashtable)
            If Me.HostSettings Is Nothing Then
                Me.HostSettings = GetHostSettings()
                DataCache.SetCache("GetHostSettings", Me.HostSettings)
            End If
```

Performance

In order to measure the performance of the application we used the Microsoft Application Center Test tool which allows you to simulate a large group of users by opening multiple connections to the server and rapidly sending HTTP requests. For comparison, we analyzed DotNetNuke 2.0 (with the new abstract DAL) against DotNetNuke 1.0.10 (using SqlCommandGenerator). Here are the results (special thanks to Kenny Rice for his assistance in running these tests).

DotNetNuke 2.0 (DAL enhancement)

Total number of requests:	93,254	
Total number of connections:		93,253
Average requests per second:		310.85
Average time to first byte (msecs):	2.37	
Average time to last byte (msecs):	2.46	
Average time to last byte per iteration (msecs):	29.58	
Number of unique requests made in test:	12	
Number of unique response codes:	1	

DotNetNuke 1.0.10 (SqlCommandGenerator)

Total number of requests:	42,350	
Total number of connections:		42,350
Average requests per second:		141.17
Average time to first byte (msecs):	6.02	
Average time to last byte (msecs):	6.15	
Average time to last byte per iteration (msecs):	116.94	
Number of unique requests made in test:	17	
Number of unique response codes:	2	

Development

DotNetNuke provides a flexible portal software architecture. The application core provides plumbing services for common functions such as membership, role security, personalization, management, site logging, navigation, and data access. It also provides the ability to extend the application with specific business functionality. In most cases it is recommended that specific business functionality be abstracted from

the core and implemented as Custom Modules. This preserves the integrity of the core and provides the best options for future upgradeability. However, if you absolutely must modify the core entities, you are not restricted from making your required changes.

Custom Modules

DotNetNuke allows for Custom Modules to be packaged as Private Assemblies for deployment to portal installations. With only minor modifications, custom modules can leverage the same technique as the core in terms of data access. This provides the added benefit of providing custom module versions for each supported database platform.

Before you move forward with the data access technique discussed below, you need to decide whether or not your component will actually need to support multiple data stores. DotNetNuke does not force you to create your custom modules using the Provider pattern. In fact, if you know your component will only be used on a single database platform, then the additional development effort required is not justified. It is the responsibility of the developer to make these decisions on a case by case basis.

\DesktopModules\Survey

The Survey custom module is architected the same way as the DotNetNuke core. It contains a Business Logic Layer class called SurveyDB.vb which contains the BLL methods. It also contains its own DataProvider.vb class (the fact that it uses its own unique namespace prevents the class name from colliding with the DataProvider class in the DotNetNuke core). In this case the ProviderType constant is set consistently with the ProviderType used by the DotNetNuke core ("data"). This is very important as it allows the concrete data provider to rely on the same configuration settings as the core concrete provider (ie. YourCompanyName.DataProvider will use the same settings from the web.config as DotNetNuke.DataProvider in terms of connection string, etc…).

```
Imports System
Imports System.Web.Caching
Imports System.Reflection

Namespace YourCompanyName.Survey

    Public MustInherit Class DataProvider

        ' provider constants - eliminates need for Reflection later
        Private Const [ProviderType] As String = "data" ' maps to <sectionGroup> in
web.config
```

```
        Private Const [NameSpace] As String = "YourCompanyName.Survey" ' project
namespace
        Private Const [AssemblyName] As String = "YourCompanyName.Survey" ' project
assemblyname

        Public Shared Shadows Function Instance() As DataProvider

            Dim strCacheKey As String = [NameSpace] & "." & [ProviderType] & "provider"

            ' Use the cache because the reflection used later is expensive
            Dim objConstructor As ConstructorInfo =
CType(DotNetNuke.DataCache.GetCache(strCacheKey), ConstructorInfo)

            If objConstructor Is Nothing Then
                ' Get the provider configuration based on the type
                Dim objProviderConfiguration As DotNetNuke.ProviderConfiguration =
DotNetNuke.ProviderConfiguration.GetProviderConfiguration([ProviderType])

                ' The assembly should be in \bin or GAC, so we simply need to get an
instance of the type
                Try

                    ' Override the typename if a ProviderName is specified ( this allows
the application to load a different DataProvider assembly for custom modules )
                    Dim strTypeName As String = [NameSpace] & "." &
objProviderConfiguration.DefaultProvider & ", " & [AssemblyName] & "." &
objProviderConfiguration.DefaultProvider

                    ' Use reflection to store the constructor of the class that
implements DataProvider
                    Dim t As Type = Type.GetType(strTypeName, True)
                    objConstructor = t.GetConstructor(System.Type.EmptyTypes)

                    ' Insert the type into the cache
                    DotNetNuke.DataCache.SetCache(strCacheKey, objConstructor)

                Catch e As Exception

                    ' Could not load the provider - this is likely due to binary
compatibility issues

                End Try
            End If

            Return CType(objConstructor.Invoke(Nothing), DataProvider)

        End Function
```

And similar to the DataProvider class in DotNetNuke, it contains the data access methods stubs.

```
        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 Sub AddSurvey(ByVal ModuleId As Integer, ByVal Question As
String, ByVal ViewOrder As String, ByVal OptionType As String, ByVal UserName As String)
        Public MustOverride Sub UpdateSurvey(ByVal SurveyId As Integer, ByVal Question As
String, ByVal ViewOrder As String, 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 Sub AddSurveyOption(ByVal SurveyId As Integer, ByVal
OptionName As String, ByVal ViewOrder As String)
        Public MustOverride Sub UpdateSurveyOption(ByVal SurveyOptionId As Integer, ByVal
OptionName As String, ByVal ViewOrder As String)
        Public MustOverride Sub DeleteSurveyOption(ByVal SurveyOptionID As Integer)
        Public MustOverride Sub AddSurveyResult(ByVal SurveyOptionId As Integer)
```

\Providers

Contains the data provider implementations for the custom module. Again you must specify an implementation for its custom attributes defined in the web.config file.

```
Namespace YourCompanyName.Survey

    Public Class SqlDataProvider

        Inherits DataProvider

        Private Const ProviderType As String = "data"

        Private _providerConfiguration As ProviderConfiguration =
ProviderConfiguration.GetProviderConfiguration(ProviderType)
        Private _connectionString As String
        Private _providerPath As String
        Private  objectQualifier As String
        Private _databaseOwner As String

        Public Sub New()

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

            ' Read the attributes for this provider
            _connectionString = objProvider.Attributes("connectionString")

             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

        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
```

Survey.dnn (deployment)

DotNetNuke uses a manifest file for deploying private assembly custom modules. The structure of this file has changed slightly with the addition of multiple providers.

```
<?xml version="1.0" encoding="utf-8" ?>
<dotnetnuke version="2.0" 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>
         <modules>
            <module>
               <friendlyname>CompanyName - Survey</friendlyname>
               <controls>
                       <control>
                              <src>Survey.ascx</src>
                              <type>View</type>
                       </control>
                       <control>
                              <key>Edit</key>
                              <title>Create Survey</title>
                              <src>EditSurvey.ascx</src>
                              <iconfile>icon_survey_32px.gif</iconfile>
                              <type>Edit</type>
                       </control>
                       <control>
                              <key>Options</key>
                              <title>Survey Options</title>
                              <src>EditSurveyOptions.ascx</src>
                              <iconfile>icon_survey_32px.gif</iconfile>
                              <type>Edit</type>
                       </control>
               </controls>
            </module>
         </modules>
         <files>
            <file>
               <name>Survey.ascx</name>
            </file>
            <file>
               <name>EditSurvey.ascx</name>
            </file>
            <file>
```

```
            <name>EditSurveyOptions.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>
            <name>help.txt</name>
         </file>
         <file>
            <name>icon_survey_32px.gif</name>
         </file>
         <file>
            <name>red.gif</name>
         </file>
         <file>
            <name>Module.css</name>
         </file>
      </files>
   </folder>
</folders>
</dotnetnuke>
```

Core Enhancements

Custom modules are the preferred method for adding additional functionality to the portal architecture. However, it is sometimes necessary to modify the core functionality to meet your specific needs as well. In order to make data access modifications to the core you must have a basic understanding of the object oriented programming principles governing the Provider model.

In theory, the Provider model uses a Factory design pattern which allows a base class to defer instantiation to subclasses. In implementation, the DataProvider class acts as a base class which defines all of the core data access methods for the application. All methods are defined as Public MustOverride which means they simply act as stubs and have no implementation included in the base class.

The DataProvider class acts as a contract which any subclass must implement completely or else object instantiation will fail. What this means is if a base class MustOverride method is modified in terms of parameter list or return value, then all subclass implementations must also be modified to reflect this modification or else they will fail to load correctly. Failing to load correctly does not simply mean that a call to that specific method will fail but, in fact, actual instantiation of the subclass will fail entirely. This contract mechanism instills a degree of fragility into the application but also ensures that each subclass meets a minimum criteria level in terms of implementation.

As an example which demonstrates the steps involved in extending the core, we will assume we are adding a new Field to a core Table.

- If necessary, change the Presentation Layer to display/edit the new field
- Modify the associated Business Logic Layer (BLL) class to add the Field to the necessary methods (typically AddTable, UpdateTable)
- Update the DataProvider base class with the necessary changes from Step #2 and recompile the application.
- Update each DataProvider subclass implementation (ie. SqlDataProvider, AccessDataProvider) with the necessary changes. Recompiling the applicatioin will reveal any discrepancies between the base class and implementation. The number of implementations which need to be modified is dependent on the number of different databases your application supports.
- Update each DataProvider subclass implementation script with the specific database alteration commands (ie. ALTER TABLE). In the case of database providers which use stored procedures, the new versions of the stored procedures must be scripted as well (with associated DROP and CREATE commands).

SqlCommandGenerator

Earlier versions of DotNetNuke contained a class called SqlCommandGenerator which simplified the effort of calling your SQL Server / MSDE database. The problem with the SqlCommandGenerator is it used Reflection on every database call which imposed a serious performance penalty on the application. This class has been retained for legacy purposes but Custom Module developers are encouraged to migrate to the DataProvider model for obvious reasons.

Credits

Rob Howard from the Microsoft ASP.NET team supplied a great deal of mentoring / sample code for implementing the DataProvider model (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnaspnet/html/asp02182004.asp) . The .NET Pet Shop 3.0 also provided some excellent reference material. The Microsoft Data Access Application Block was instrumental for rapid development of the SQL Server Provider implementation.

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	Aug 17, 2005	Shaun Walker	• Applied new template