Abstracting Membership and Profile Components

DotNetNuke Roadmap

Charles Nurse

Version 1.0.0
Last Updated: June 21, 2006
Category: DotNetNuke 3.3/4.1

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

This document describes a proposed abstraction of the MemberRole components of DotNetNuke..

Contents

Introduction

Introduction

DotNetNuke 3.0 introduced the use of a prototype of the new Membership, Role Manager and Provider components (MemberRole) that were introduced in ASP.NET 2. v3.2 still uses this component while v4.0 uses the version that is part of the .NET Framework

Unfortunately, the way it was introduced into the 3.0 code-base, restricts us from enhancing our Membership, Role and Profile offerings.

Some examples of the deficiencies of the MemberRole capabilities are as follows:

- Not a trivial process to provide Single-Sign-On
- Limited search capabilities for users – other than username or email
- Very limited Role capabilities!!
- Profile properties cannot be defined on a per portal basis
- Profile properties do not have a sort order capability (in East Asian cultures names should be displayed in a Last Name, First Name order

To allow DotNetNuke to provide enhanced Membership, Role and Profile capabilities this paper proposes that we introduce a new layer of abstraction, by introducing MembershipProvider, RoleProvider, and ProfileProvider abstract classes in the core, which will be used to replace the use of the MemberRole components.

In addition, this paper proposes that to reduce upgrade issues the default implementations of these abstract classes will be based on the MemberRole components.

Membership Provider

Introduction

In this section we will review the functions that are used from the MemberRole components today, and describe a new abstract set of providers, that should fulfill our needs

Membership Functionality

Let's start by reviewing the functionality of the Membership class in MemberRole that we use today.

Method/Property	Membership	Used in DotNetuke
ApplicationName		We don't actually use the property directly, but we set the property of the Provider to {objectQualifer}PortalId
CreateUser	4 overloads	We only use the overload with all the possible parameters (username, password, passwordQuestion, passwordAnswer, isApproved, status)
DeleteUser	2 overloads	We only use the overload with two parameters (username,

Method/Property	Membership	Used in DotNetuke
		deleteAllRelatedData)
EnablePaswordReset		True – by default
EnablePasswordRetrieval		True – by default
FindUsersByEmail	2 Overloads	We only use the paged overload (email, pageIndex, pageSize, TotalRecords)
FindUsersByName	2 Overloads	We only use the paged overload (email, pageIndex, pageSize, TotalRecords)
GetAllUsers	2 Overloads	We use both the paged overload (pageIndex, pageSize, TotalRecords) and the no params overload
GetNumberOfUsersOnline		
GetUser	6 Overloads	We use three of the overloads: GetUser(), GetUser(userOnline), GetUser(username)
GetUserNameByEmail		
MaxInvalidPasswordAttempts		

Method/Property	Membership	Used in DotNetuke
MinRequiredNonAlpha-NumericCharacters		0 – by default
MinRequiredPasswordLength		4 – by default
PasswordAttemptWindow		
PasswordStrengthRegular-Expression		
Provider		
Providers		
RequiresQuestionAndAnswer		
UpdateUser	1 Overload	Used
UserIsOnlineTimeWindow		
ValidateUser	1 Overload	Used

MembershipUser Functionality

The next most important class is the MembershipUser class. Most (if not all) of the properties that are designated as used, have a corresponding property in the core

UserMembership class, and the extent of their use is restricted to data-transport to-and-from the data store.

Method/Property	MembershipUser	Used in DotNetuke
ChangePassword	1 overload	Used
ChangePasswordQuestionAndAnswer	1 overload	
Comment		
CreationDate		Used
Email		Used
GetPassword	2 overloads	We only use the simple no parameters overload
IsApproved		Used
IsLockedOut		Used
IsOnline		Not used in Core (Users Online?)
LastActivityDate		Used
LastLockOutDate		Used

Method/Property	MembershipUser	Used in DotNetuke
LastLoginDate		Used
LastPasswordChanged-Date		
PasswordQuestion		
ProviderName		
ProviderUserKey		
ResetPassword		
UnlockUser		Used
UserName		Used

Other Membership related Methods and Properties

Method/Property		Used in DotNetuke
MembershipProvider.Requires-UniqueEmail		False – by default

Method/Property		Used in DotNetuke
MembershipCreateStatus enum		

Deficiencies of Membership

As mentioned in the introduction, there are a number of deficiencies in the Membership components of MemberRole. Some of these include

- It is not a trivial process to provide Single-Sign-On
- Limited search capabilities for users – other than username or email
- Enhancements are restricted to core API provided by Microsoft

In addition there are some deficiencies in how we implement the MemberRole components, which could and should be addressed.

The biggest problem is the number of database hits required. For instance, logging in as a SuperUser (measured from the Click of the Login button in the SignIn Form until the main page is displayed) requires the following 23 database calls:

- GetUserByUsername
- aspnet_Profile_GetProperties
- aspnet_Membership_GetUserByName
- aspnet_Membership_GetPassword
- aspnet_Membership_GetPasswordWithFormat
- aspnet_Membership_UpdateLastLoginAndActivityDates
- GetUserByUsername
- aspnet_Profile_GetProperties

-- Redirect after successful login

- aspnet_Membership_GetUserByName
- aspnet_Membership_GetUserByName

- GetUserByUsername
- aspnet_Profile_GetProperties
- GetRolesByUser
- GetUser
- aspnet_Profile_GetProperties
- aspnet_Membership_GetUserByName
- aspnet_Membership_GetPassword
- aspnet_Membership_GetUserByName
- aspnet_Membership_GetPassword
- GetUser
- aspnet_Profile_GetProperties
- aspnet_Membership_GetUserByName
- aspnet_Membership_GetPassword

Admittedly some of these call are from the Redirect after a successful login, but there are still 6 calls to aspnet_Membership_GetUserByName, where maybe you should expect 2 – one on the Login Postback and one on the redirect.

Lets look at the first batch of calls, those required to Log the user in to the portal, in a little more detail:

- GetUserByUsername
- aspnet_Profile_GetProperties
- aspnet_Membership_GetUserByName
- aspnet_Membership_GetPassword
- aspnet_Membership_GetPasswordWithFormat
- aspnet_Membership_UpdateLastLoginAndActivityDates
- GetUserByUsername
- aspnet_Profile_GetProperties

The first call **GetUserByName** is a call to the DNN tables to get the User. It is called at the very beginning of the UserLogin method in PortalSecurity.vb, by the UserController method GetUserByUserName.

```
' get the user based on username from the local data store
Dim objUser As UserInfo = objUsers.GetUserByUsername(PortalID, Username)
```

The second call **aspnet_Profile_GetProperties** is called by the FillUserInfo function that is itself called by the GetUserByUserName method.

```
objUserInfo.FirstName = Convert.ToString(Null.SetNull(dr("FirstName"),
objUserInfo.FirstName))
```

The read on the FirstName property triggers a call to the **aspnet_Profile_GetProperties** method as an attempt is made to fetch the property from the Profile if the FirstName is an empty string.

```
If _FirstName = "" Then
    'Try Profile
     FirstName = Profile.FirstName
End If
```

Thus the attempt at lazy loading the Membership properties never works because the FillUserInfo method reads the value of the FirstName property before setting it. Every call to **GetUserByUserName** will cause a **aspnet_Profile_GetProperties** call, even if there is no need for the Profile properties.

Next the UserLogin method evaluates the LockedOut property of the Membership property of the User.

```
' if the lockout period has expired then unlock the account
If objUser.Membership.LockedOut Then
```

As the Membership property is also lazy loaded, this causes a call to the **aspnet_Membership_GetUserByName** stored procedure in order to fill the Membership property,

```
_Membership = objUserController.FillUserMembership(Me)
```

followed by the following in FillUserMembership()

```
Dim objMembershipUser As Microsoft.ScalableHosting.Security.MembershipUser
objMembershipUser =
Microsoft.ScalableHosting.Security.Membership.GetUser(objUserInfo.Username)
```

Further down the FilUserlMembership method is a call to the GetPassword() method of

the AspNet MembershipUser class which in turn calls the **aspnet_Membership_GetPassword** stored procedure.

The next two stored procdedure calls, **aspnet_Membership_GetPasswordWithFormat** and **aspnet_Membership_UpdateLastLoginAndActivityDates** are called when the AspNet Membership.ValidateUser method is called to validate the user credentials.

Interestingly, because of the earlier GetPassword call, there actually is no need to make the call to ValidateUser, as we could compare the password provided by the user with the password stores in the Membership property, which was fetched on the previous call. The only benefit of this call is that it would automatically handle the password encryption in the case of SHA1 hashing, as there would be an exception to the GetPassword call if the passwords are hashed.

A New Abstract MembershipProvider

So having reviewed the capabilities of the Membership components as well as some of the deficiencies, we now turn to the requirements of a new MembershipProvider.

Methods to support Existing Functionality

First let's look at the methods the provider would need to provide to support the functionality we have today.

ChangePassword

ChangePassword allows the user's password to be modified. We therefore need to pass the old password and the new password. A Boolean is returned indicating success or failure.

```
Public MustOverride Function ChangePassword(ByVal portalId As Integer, ByVal user As
UserInfo, ByVal oldPassword As String, ByVal newPassword As String) As Boolean
```

There is some debate about whether the oldPassword should be provided as well as the newPassword.

CreateUser

In MemberRole the CreateUser method returns a MembershipUser object, and a MembershipCreateStatus enum value by reference. In a new Provider model we could work in a number of ways.

However, the simplest and most obvious way would be to create a new UserInfo object and pass this object to the CreateUser function, returning a UserCreateStatus. The UserInfo object would be passed by ref, so that the new User Id would be accessible.

```
Public MustOverride Function CreateUser(ByRef user As UserInfo) As UserCreateStatus
```

The user has an IsSuperUser property which will determine if a SuperUser would be added, otherwise the user would be added to the Portal.

DeleteUser

In MemberRole there are two DeleteUser overloads. It is probably not necessary to provide both overloads, as there is no Use Case that would not delete the related data. A Boolean is returned indicating success or failure.

```
Public MustOverride Function DeleteUser(ByVal user As UserInfo) As Boolean
```

FindUser / GetUser

FindUser and GetUser are essentially the same thing. They are methods to retrieve a User. MemberRole provides 13 variations of Find/Get User. Many of these are not necessary if the correct parameter selections are made, and in fact the internal abstract provider only provides the all parameters versions of these methods, so it is being passed default parameters.

The following methods are necessary to support the existing functionality, which return either a UserInfo object or an ArrayList of Info Objects (List Of UserInfo – in .NET 2):

```
Public MustOverride Function GetUser(ByVal portalId As Integer, ByVal userId As Integer,
ByVal isHydrated As Boolean, ByVal userIsOnline As Boolean) As UserInfo

Public MustOverride Function GetUserByUserName(ByVal portalId As Integer, ByVal username
As String, ByVal isHydrated As Boolean, ByVal userIsOnline As Boolean) As UserInfo
```

```
Public MustOverride Function GetUsers(ByVal portalId As Integer, ByVal isHydrated As
Boolean, ByVal pageIndex As Integer, ByVal pageSize As Integer, ByRef totalRecords As
Integer) As ArrayList

Public MustOverride Function GetUsersByEmail(ByVal portalId As Integer, ByVal isHydrated
As Boolean, ByVal emailToMatch As String, ByVal pageIndex As Integer, ByVal pageSize As
Integer, ByRef totalRecords As Integer) As ArrayList

Public MustOverride Function GetUsersByUserName(ByVal portalId As Integer, ByVal
isHydrated As Boolean, ByVal userNameToMatch As String, ByVal pageIndex As Integer, ByVal
pageSize As Integer, ByRef totalRecords As Integer) As ArrayList
```

GetPassword

GetPassword allows the user's password to be retrieved. If successful the password is retrieved. If unsuccessful – an empty string is returned.

```
Public MustOverride Function GetPassword(ByVal portalId As Integer, ByVal username As
String, ByVal passwordAnswer As String) As String
```

GetUserMembership

The GetUserMembership method is used to build the Membership property of the UserInfo object.

```
Public MustOverride Sub GetUserMembership(ByRef user As UserInfo)
```

UnLockUser

The UnLockUser method is used to unlock a User who has too many unsuccessful logins within a time window. A Boolean is returned indicating success or failure.

```
Public MustOverride Function UnLockUser(ByVal user As UserInfo) As Boolean
```

UpdateUser

Like CreateUser, UpdateUser should be passed a UserInfo object, although unlike CreateUser there is no return value.

```
Public MustOverride Sub UpdateUser(ByVal user As UserInfo)
```

UserLogin

The UserLogin Method attempts to log the user in. If successful it returns a UserInfo object. A UserLoginStatus enumeration is returned by ref, that indicates the result of the attempt to login.

The MemberRole component does not have this method. Logging a user in using MemberRole is a multi-step process in MemberRole (at least it is for DotNetNuke which not only needs to validate the user, but also needs to return a UserInfo object).

```
Public MustOverride Function UserLogin(ByVal portalId As Integer, ByVal username As
String, ByVal password As String, ByVal verificationCode As String, ByVal userIsOnline As
Boolean, ByRef loginStatus As UserLoginStatus) As UserInfo
```

Enumerations

UserCreateStatus

UserCreateStatus is a new Enumeration that will provide a means of returning the status of an attempt to create a new user. Its enumerated values will closely map to those included in the MembershipCreateStatus of MemberRole.

```
        AddUser = 0
        UsernameAlreadyExists = 1
        UserAlreadyRegistered = 2
        DuplicateEmail = 3
        DuplicateProviderUserKey = 4
        DuplicateUserName = 5
        InvalidAnswer = 6
        InvalidEmail = 7
        InvalidPassword = 8
        InvalidProviderUserKey = 9
        InvalidQuestion = 10
        InvalidUserName = 11
        ProviderError = 12
        Success = 13
        UnexpectedError = 14
        UserRejected = 15
```

UserLoginStatus

UserLoginStatus is a new Enum that will provide a means of returning the status of an attempt to login a user. Its enumerated values are as follows:

```
LOGIN_FAILURE = 0
LOGIN_SUCCESS = 1
LOGIN_SUPERUSER = 2
LOGIN_USERLOCKEDOUT = 3
LOGIN_USERNOTAPPROVED = 4
```

New Methods for UserController

In order to completely abstract the User Business Contol Layer from the UI related components, we need to include new methods, that correspond to the new Provider methods.

Any method that makes a direct reference in its method signature to the AspNet Membership components will need to be removed. This is a breaking change and will need to be communicated to the module developer community prior to the method being removed

Currently, the UI layer is making calls directly to the MemberRole components. In abstracting the providers, the UI layer could now of course make a call directly to the new abstract provider, but this goes against our normal policy. The UI layer should properly call the BusinessLayer, which in turn calls the abstracted Provider layer.

Until, we actually get to doing the abstraction is difficult to predict what will need to be obsoleted and what will need to be added.

AspNetMembershipProvider

A new AspNetMembershipProvider concrete Provider will need to be created to provide an AspNet (MemberRole) implementation of the new abstract MembershipProvider. In order to minimize any upgrade issues, this implementation will need to be configured as the default provider.

One interesting phenomenon of the current Membership implementation is that DotNetNuke uses a hybrid of MemberRole and its own Data Store. A simple abstraction would therefore indicate that the ApsNetMembershipProvider implementation should also be a “hybrid” implementation, as it would need to access its own table – as well as some DotNetNuke tables.

In creating this provider a number of things will need to be done:

- All current methods in the UserController class will need to be modified to call abstract Provider methods.
- The code that was in these UserController methods will need to be moved to the relevant Provider method, so that there is no effective change in behaviour.
- Code in UI controls that directly references the AspNet classes will need to be refactored to call new or existing UserController methods, which will in turn call the appropriate Abstract Provider method. Controls that access methods of MemberRole include:
  - Register.ascx.vb
  - SignIn.ascx.vb
  - ManageUser.ascx.vb
  - Users.ascx.vb
- Core methods that manipulate the ApplicationName through the HttpContext will need to be "deprecated. Our usual policy is to deprecate a method and mark it as obsolete. In this case the methods should be removed. Any modules etc that are directly accessing MemberRole will need to be updated, in order to support an abstracted Membership system, as this is an "unsupported" Use Case.
- Core methods that use "SynchonizeUsers" to determine whether user information is syncronised between dnn local tables and aspnet global tables will need to be deprecated as above.
- While not required, a new private helper method (SetApplicationName) should be included so the various public methods can manipulate the ApplicationName property of the AspNet MembershipProvider, consistently.

DNNMembershipProvider

A new DNNMembershipProvider concrete Provider will need to be created to provide a purely DotNetNuke implementation of the new abstract MembershipProvider. In order to minimize any upgrade issues, this implementation will not be configured as the default provider.

This is beyond the scope of this document.

Role Provider

Introduction

We now turn our attention to the Roles component.

In this section we will review the functions that are used from the MemberRole components today, and describe a new abstract set of provider, that should fulfill our needs

Roles Functionality

Let's start by reviewing the functionality of the Roles class in MemberRole that we use today.

Method/Property	Membership	Used in DotNetuke
ApplicationName		We don't actually use the property directly, but we set the property of the Provider to {objectQualifer}PortalId
AddUsersToRole		
AddUsersToRoles		
AddUserToRole		Used

Method/Property	Membership	Used in DotNetuke
AddUserToRoles		Used
CreateRole		Used
DeleteRole	2 Overloads	Use DeleteRole(string)
FindUsersInRole		
GetAllRoles		Used
GetRolesForUser		Used
GetUsersInRole		Used
IsUserInRole	2 Overloads	Use IsUserInRole(string) overload
RemoveUserFromRole		Used
RemoveUserFromRoles		
RemoveUsersFromRole		Used
RemoveUsersFromRoles		

Method/Property	Membership	Used in DotNetuke
RoleExists		

This looks like we are using the Roles capability quite a lot. However, in reality we are only using the Roles capability of MemberRole to save the role names.

Deficiencies of Roles

As mentioned in the introduction, there is a major deficiency in the Roles components of MemberRole, which has required us to implement our tables

- Very limited description of Role properties

In addition, we are really only using the Roles capability of MemberRole to store the Roles, and even then we store the same information (as well as much more in our Roles tables).

One of the most important features of the MemberRole Roles capability is the fact that it automatically generates a "RolePrincipal" that it attaches to the User property of the Current Context. This is actually done by the RoleManagerModule.

We don't however use this capability, instead, we have implemented our own DNNMembership Module which deals with PortalAlias issues and Roles. The roles are persisted in a cookie (portalroles). In the DNNMembership Module this cookie is loaded and parsed, and the resulting list of roles is saved in the HttpContext. If this is the first request for the User, the roles are fetched from the Database and subsequently saved to the cookie.

All checks to determine whether a User has a role are processed through two methods in PortalSecurity.vb

- IsInRole(string)
- IsInRoles(string)

IsInRoles splits the string parameter into individual roles and calls IsInRole for each Role, so lets look at IsInRole which is the hub of all role checking.

```
Public Shared Function IsInRole(ByVal role As String) As Boolean
    Dim objUserInfo As UserInfo = UserController.GetCurrentUserInfo
    Dim context As HttpContext = HttpContext.Current

    If objUserInfo.IsSuperUser Or
            (role <> "" AndAlso Not role Is Nothing AndAlso
            ((context.Request.IsAuthenticated = False And role = glbRoleUnauthUserName)
            Or role = glbRoleAllUsersName)) Then
        Return True
    Else
        If AspNetSecurity.Roles.IsUserInRole(role) Then
            Dim strRoles As String
            strRoles = Convert.ToString(HttpContext.Current.Items("UserRoles"))
            If strRoles.IndexOf(";" + role + ";") >= 0 Then
                Return True
            End If
        End If
    End If
    Return False
End Function
```

IsInRole first checks that the role has automatic access, and returns true. If this is not the case it calls the IsUserInRole(role) method of MemberRole's Roles class, to check whether the user has the role required. This call does not use the DotNetNuke Roles, which are persisted in the cookie (as discussed above). The Roles Manager in MemberRole has its own cookie persistence scheme.

If the user is determined by MemberRole to be in the role requested, the DotNetNuke roles are then checked.

It would appear here, that we are not using the Roles capability of MemberRole for anything that we don't already do using our own mechanism. In fact the only reason for storing the roles in the AspNet tables is so a 3rd party application can access the roles information independent of DotNetNuke.

A New Abstract RolesProvider

So having reviewed the capabilities of the Roles components as well as some of the deficiencies, we now turn to the requirements of a new RolesProvider.

Methods to support Existing Functionality

First let's look at the methods the provider would need to provide to support the functionality we have today.

AddUserToRole

AddUserToRole adds an existing user to an existing role.

```
Public MustOverride Function AddUserToRole(ByVal portalId As Integer, ByVal user As
UserInfo, ByVal userRole As UserRoleInfo) As Boolean
```

CreateRole

CreateRole creates a new role. Similarly to CreateUser we pass a Role by reference and return a Boolean indicating success or failure.

```
Public MustOverride Function CreateRole(ByVal portalId As Integer, ByRef role As
RoleInfo) As Boolean
```

DeleteRole

DeleteRole deletes a role.

```
Public MustOverride Sub DeleteRole(ByVal portalId As Integer, ByRef role As RoleInfo)
```

GetRole(s)

These methods either retrieve a RoleInfo object from the Database or an ArrayList of Roles.

```
Public MustOverride Function GetRole(ByVal portalId As Integer, ByVal roleId As Integer)
As RoleInfo

Public MustOverride Function GetRole(ByVal portalId As Integer, ByVal roleName As String)
As RoleInfo

Public MustOverride Function GetRoles(ByVal portalId As Integer) As ArrayList
```

GetRoleNames

GetRoleNames fetches a string array of role names, which is used by the application to do most of the role checking.

```
Public MustOverride Function GetRoleNames(ByVal portalId As Integer, ByVal userId As
Integer) As String()
```

GetUserRole(s)

These methods either retrieve a UserRoleInfo object from the Database or an ArrayList of UserRoles.

```
Public MustOverride Function GetUserRole(ByVal PortalId As Integer, ByVal UserId As
Integer, ByVal RoleId As Integer) As UserRoleInfo

Public MustOverride Function GetUserRoles(ByVal PortalId As Integer, ByVal UserId As
Integer) As ArrayList

Public MustOverride Function GetUserRoles(ByVal PortalId As Integer, ByVal Username As
String, ByVal Rolename As String) As ArrayList
```

GetUsersInRole

GetUsersInRole gets an ArrayList of Users in a Role.

```
Public MustOverride Function GetUsersInRole(ByVal PortalId As Integer, ByVal RoleName As
String) As ArrayList
```

RemoveUserFromRole

RemoveUserFromRole removes a User from a Role.

```
Public MustOverride Sub RemoveUserFromRole(ByVal portalId As Integer, ByVal user As
UserInfo, ByVal userRole As UserRoleInfo)
```

UpdateRole

UpdateRole updates an existing role.

```
Public MustOverride Sub UpdateRole(ByVal role As RoleInfo)
```

UpdateUserRole

UpdateUserRole updates an existing UserRole.

```
Public MustOverride Sub UpdateUserRole(ByVal userRole As UserRoleInfo)
```

AspNetRolesProvider

A new AspNetRolesProvider concrete Provider will need to be created to provide an AspNet (MemberRole) implementation of the new abstract RolesProvider. In order to minimize any upgrade issues, this implementation will need to be configured as the default provider.

As with the Membership Provider a number of things will need to be done:

- Current methods in the RoleController class will need to be modified to call abstract Provider methods.
- The code that was in these RoleController methods will need to be moved to the relevant Provider method, so that there is no effective change in behaviour.
- Code in UI controls that directly references the AspNet classes will need to be refactored to call new or existing RoleController methods, which will in turn call the appropriate Abstract Provider method. Controls that access methods of MemberRole include:
  - Roles.ascx.vb
  - SecurityRoles.ascx.vb
  - Register.ascx.vb
  - BulkEmail.ascx.vb

DNNRolesProvider

A new DNNRolesProvider concrete Provider will need to be created to provide a purely DotNetNuke implementation of the new abstract RolesProvider. In order to minimize any upgrade issues, this implementation will not be configured as the default provider.

This is beyond the scope of this document.

Profile Provider

Introduction

Finally we turn our attention to the last of the components of MemberRole, the Profile capabilities in ProfileBase.

Profile Functionality

Let's start by reviewing the functionality of the ProfileBase class in MemberRole that we use today.

Method/Property	Membership	Used in DotNetuke
ApplicationName		We don't actually use the property directly, but we set the property of the Provider to {objectQualifer}PortalId
Context		
Create	2 Overloads	Use the Create(String, Boolean) overload
GetProfileGroup		

Method/Property	Membership	Used in DotNetuke
GetPropertyValue		
IsAnonymous		
IsDirty		
IsSynchronized		
Item		
LastActivityDate		
LastUpdatedDate		
Properties		Used
PropertyValues		
Providers		
Save		Used
SetPropertyValue		

Method/Property	Membership	Used in DotNetuke
UserName		

As is clear from the table we use only the essentials of the class – the ability to create a Profile, access its properties and Save the profile.

Even though it appears we don't use much of the Profile capabilities, this is an area where the default Profile implementation does limit our ability to extend our capabilities.

A New Abstract ProfileProvider

So having reviewed the capabilities of the Profile components as well as some of the deficiencies, we now turn to the requirements of a new RolesProvider.

Methods to support Existing Functionality

First let's look at the methods the provider would need to provide to support the functionality we have today.

GetUserProfile

The GetUserProfile method is used to build the Profile property of the UserInfo object.

```
Public MustOverride Sub GetUserProfile(ByRef user As UserInfo)
```

UpdateUserProfile

The UpdateUserProfile method is used to save the Profile property of the UserInfo object.

```
Public MustOverride Sub UpdateUserProfile(ByVal user As UserInfo)
```

AspNetProfileProvider

A new AspNetProfileProvider concrete Provider will need to be created to provide an AspNet (MemberRole) implementation of the new abstract ProfileProvider. In order to minimize any upgrade issues, this implementation will need to be configured as the default provider.

All of the code that manages profile today is either already included in UserController methods or is sprinkled about the core UI components with corresponding membership code.

DNNProfileProvider

A new DNNProfileProvider concrete Provider will need to be created to provide a purely DotNetNuke implementation of the new abstract ProfileProvider. In order to minimize any upgrade issues, this implementation will not be configured as the default provider.

This is beyond the scope of this document.

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	Dec 16, 2005	Charles Nurse	• First Draft
1.01	Jan 23, 2006	Charles Nurse	• Second Draft, moved Enhancements to separate Document