Enhancing Profile

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.

Enhancing Profile

Abstract

This document describes proposed enhancements of the Profile capabilities of DotNetNuke.

Contents

Introduction

This document proposes a number of enhancements for Profiles.

- Extensive collection of Strongly Typed Properties.
- Module profile Properties – Modules should be able to add profile properties for module–specific information.
- Portal Properties – the Profile Properties should be defined at the Portal level (not the host level)
- Dynamic Definition – the Portal level properties should be managed by the Portal Administrator.
- Searchable – Profile Properties should be Searchable (ie we should be able to do Find Users By City or Find Users with Green Eyes)
- Property Order – the ability to define a custom order for the properties to be displayed.

This document focuses on the business requirements in Business Layer and the Provider requirements in the Data Layer. Adding or updating User Interface components, for instance the following enhancements, requires that the underlying API is in place.

- UI – Profile properties should be exposed through a custom user interface
- Support of bulk Export/Import of profile information

Strongly Typed Properties

Introduction

This enhancement would extend the base UserProfile object to provide a much richer Profile implementation. The idea here is to provide, by default, a rich strongly-typed object model for User Profiles (or non user profiles, such as contacts, anonymous users etc). These properties should be consistent with the W3C Consortium's Platform for Privacy. (http://www.w3.org/TR/P3P/)

Current UserProfile Class

Name Properties

UserProfile has three name properties (FullName is a ReadOnly calculated property - FirstName LastName)

- FirstName
- LastName
- FullName

Address Properties

- Unit
- Street
- City
- Region
- Country
- PostCode

Contact Properties

- Telephone
- Fax
- Cell
- IM
- Email (actually part of Membership)
- Website

Other Properties

- TimeZone
- Preferred Locale

New Deep Object Model for Profile

In order to provide an extended strongly typed collection of Profile properties it is proposed that a number of new classes be created.

New NameInfo Class

This class would extend the current naming capability to provide a richer name description. This class corresponds to the Platform for Privacy personname structure (http://www.w3.org/TR/P3P/#Names - Section 5.5.2).

- Prefix – eg Mr, Capt
- FirstName
- MiddleName – (or Middle Initial)
- LastName
- Suffix – eg BSc, MCSE etc
- FullName

It could be argued that a separate Name Class is not neccessary, and that the new Name properties (Prefix, MiddleName and Suffix) could be added to the UserProfile class.

New AddressInfo Class

This class corresponds to the Platform for Privacy postal structure (http://www.w3.org/TR/P3P/#Postal - Section 5.5.6.1). In addition to the fields described in the Platform for Privacy specification, the Type is added (an enumeration of

AddressTypes) as well as a Primary flag (Boolean representing whether the Address is the primary contact address for mailing purposes.)

- Type – Home, Business etc
- Primary – Flag that indicates whether the Address is the primary address (mailing address)
- Organization
- Unit
- Street
- City
- Region
- Country
- PostCode

Note: Using an enumerated type for the Address Type restricts the available types to those chosen by the Developer(s). It may be better to use the core Lists capabaility but this restricts the available types to those defined by a SuperUser.

New AddressCollection Class

This class would provide the ability for a Profile to have a collection of addresses. It should inherit from DictionaryBase, using the "AddressType" as a key. This would enable an implementer to quickly access an Address of a specific type.

```
User.Profile.Addresses(AddressType.Work)
```

The collection should also provide a "Primary" property, to enable the following

```
Dim primaryAddress as AddressInfo = User.Profile.Addresses.Primary
```

New PhoneInfo Class

This class corresponds to the Platform for Privacy telephonenum structure (http://www.w3.org/TR/P3P/#Telecommunication - Section 5.5.5). In addition to the fields described in the Platform for Privacy specification, the Type is added (an enumeration of PhoneTypes) as well as a Primary flag (Boolean representing whether the Phone Number is the primary contact no.)

- Type – Home, Business, Fax, Cell, Pager etc
- Primary – Flag that indicates whether the Phone Number is the primary number
- International Code - eg +1
- Local Area Code - eg (604)
- Number – eg 555-5555
- Extension - 28
- Note/Comment

Note: The same considerations with regard to Phone Type apply as was discussed above for Address Type.

New PhoneCollection Class

This class would provide the ability for a Profile to have a collection of phone numbers. It should inherit from DictionaryBase, using the "PhoneType" as a key. This would enable an implementer to quickly access a Phone Number of a specific type.

```
User.Profile.Phones(PhoneType.Work)
```

The collection should also provide a "Primary" property, to enable the following

```
Dim primaryPhone as PhoneInfo = User.Profile. Phones.Primary
```

New Uri Class

This class corresponds to the Platform for Privacy online structure (http://www.w3.org/TR/P3P/#Online - Section 5.5.6.3). In addition to the fields described in the Platform for Privacy specification, the Type is added (an enumeration of UriTypes) as well as a Primary flag (Boolean representing whether the Phone Number is the primary contact no.)

- Type – Email, BlogUrl, Website, IM
- Uri

Note: The same considerations with regard to Url Type apply as was discussed above for Address Type.

New UriCollection Class

This class would provide the ability for a Profile to have a collection of uri values. It should inherit from DictionaryBase, using the "UrlType" as a key. This would enable an implementer to quickly access a Uri of a specific type.

```
User.Profile.UriCollection(UriType.Email)
```

The collection should also provide a "Primary" property, to enable the following

```
Dim blogUrl as UriInfo = User.Profile.UriCollection.Primary
```

Changes to UserProfile

The UserProfile class would need 3 new properties to handle these new Property Collections.

Addresses – of type AddressCollection

Phones – of type PhoneCollection

UriCollection – of type UriCollection

In addition the existing properties should provide a wrapper to the contained collections/classes, for legacy purposes. (Note that these examples do not have the necessary error-checking for null references etc)

```
        Public Property FirstName() As String
            Get
                Return Name.FirstName
            End Get
            Set(ByVal Value As String)
                Name.FirstName = Value
            End Set
        End Property

        Public Property Country() As String
            Get
                Return Addresses.Primary.Country
            End Get
            Set(ByVal Value As String)
                Addresses.Primary.Country = Value
            End Set
        End Property

        Public Property Website() As String
```

```
            Get
                Return UriCollection(UriType.Website).Uri
            End Get
            Set(ByVal Value As String)
                UriCollection(UriType.Website).Uri = Value
            End Set
        End Property
```

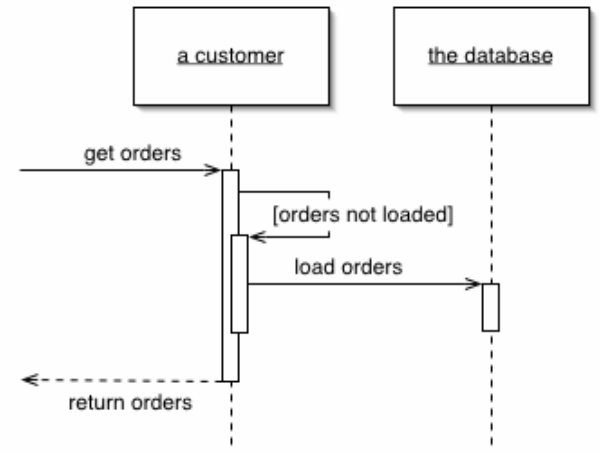
Proxy Pattern vs Lazy Initialization

If we now add these strongly-typed Profile classes to the object model we now have a 3-level object model and Hydration issues become even more important than before. It is possible that a Profile provider could be implemented in such a way that the profile is pulled back in one call to the database. This would depend on how the database is implemented. Having said that, some implementations may not be able to do this and so a lazy loading capability should be implemented.

Lazy Loading

Lazy loading is the concept of loading or hydrating an object when it is needed. Thus if we get all the users and only "need" to access whether the users are SuperUsers then we do not need to load the Membership and Profile properties, as this would represent an unnecessary performance hit. There are a number of design patterns that implement a lazy loading capability

Lazy Initialization



Lazy Initialization uses a marker in the class, that determines whether the object is loaded.[1] This is the design pattern we use, with the ObjectHydrated property of Profile and Membership. However, our use of this design pattern is not perfectly OO. In our implementation it is the parent (User) that determines whether the Object should be loaded, rather than the Object that determines how it should behave.

```
        Public Property Profile() As Entities.Users.UserProfile
            Get
                'implemented progressive hydration
                'this object will be hydrated on demand
                If Not _Profile.ObjectHydrated AndAlso Not Me.Username Is Nothing AndAlso
Me.Username.Length > 0 Then
                    Dim objUserController As New UserController
                    UserController.GetUserProfile(Me)
                    _Profile.ObjectHydrated = True
                End If
                Return _Profile
            End Get
            Set(ByVal Value As Entities.Users.UserProfile)
                 Profile = Value
                _Profile.ObjectHydrated = True
            End Set
        End Property
```

Thus the correct use of this design pattern (which would make Profile work the same independent of whether it was defined as a child of User), would be to implement the check in the Get Method of each property. There is a problem, however with this approach (in our situation at least), as the profile doesn't know to which user it belongs.

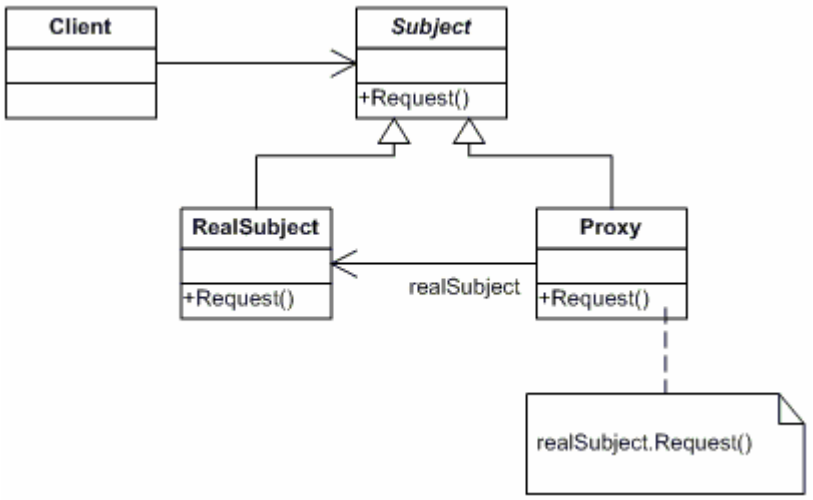
Proxy Pattern

Another pattern that can be used is a Proxy pattern. In this pattern, two different classes ProxyClass and RealClass both implement the insterface IClass, which defines the methods and properties of the object being lazy loaded. The parent class has a property of type IClass, which when instantiated is set to an instance of ProxyClass.[2]

ProxyClass does not "load" itself but instead contains an internal (private) reference to RealClass. When the consumer code makes a call to a property of the ProxyClass it check whether its reference to the real class is null, and if it is loads it. If it is not null it returns the value of the referenced RealClass's property.

In this design pattern we still run into the issue of how does the ProxyClass know how to load itself.

[1] See http://www.martinfowler.com/eaaCatalog/lazyLoad.html
[2] See http://www.dofactory.com/Patterns/PatternProxy.aspx



Controller Classes

Obviously new Controller classes would be necessary to manage these new Info classes and collections.

Dynamic Custom Properties

Introduction

This Enhancement provides the capability to define new custom properties at the portal level. The enhancement will allow new Properties to be created, existing properties to be enabled/disabled and a custom property order to be defined. It could also be used by Modules to add their own custom profile properties

Additions to UserProfile

The base UserProfile object already includes a ProfileProperties Property. This is a Hashtable, so can manage any kind of property (as long as it has a unique key"). For instance if I define a custom property in my Portal "MyCustomProperty", then I can access that property using:

```
User.Profile.ProfileProperties("MyCustomProperty")
```

Thus the main thrust of this enhancement is to provide a mechanism to define and manage custom properties, so that they are available to the UI developer.

Managing Custom Profile Properties

In order to manage custom properties by portal, we need a mechanism to manage the properties. It is therefore proposed that the following class (as well as relevant Controller class, DataProvider methods and DataBase table) be created to manage the properties.

New ProfilePropertyDefinitionInfo class

- PropertyId – Unique identifier

- PortalId – the Id of the Portal that will be using the property.
- ModuleDefId – the Id of the Module that defines a "custom" module specific property (-1 or DBNull for all others)
- PropertyCategory – the category of this property (allows the properties to be grouped, in the UI by section or tabpage)
- PropertyName – the name of the Property (would be used as the localization key (PropertyName.Text, PropertyName.Help)
- DefaultValue – the default Value to use
- CanDelete – Is this a default property, if so it cannot be deleted by the admin.
- Visible - Is this property visible (can be used to enable/disable default properties)
- Required – Is the property required
- DisplayControl – the type of control to use to display the property in the UI (textbox, optionbuttons, checkbox, drop-down etc)
- DataType – the data type of the property. While the actual storage of the property in the data store will depend on the provider implementation, the data would need to be validated (so that if a date is expected, a valid date must be entered)
- ValidationExpression – the concept here is to provide the capability for using a regular expression for data validation (for an email for instance)
- ViewOrder – the order this property will display

Note: In the Database a Unique constraint should be applied to the PortalId/PropertyName combination.

Using this structure will allow us to manage a number of custom behaviours:

- Add/Edit/Delete new custom properties by portal
- Configure the default properties (enable/disable)
- Indicate whether the property is required or not (for validation purposes)
- Set the ViewOrder of the properties.

In addition this structure would allow custom Modules to "Add" there own properties. If Module developers wanted to use theire own UI to manage these custom properties then the would define their properties with Visible=False (although an Administrator could redefine this to Visible=True. The module developer would be responsible for adding the property definitions to the table and the "Uninstall module" functionality would need to remove all the properties.

These properties could be identified as related to a specific Module, by appending the module name to the property name". For instance "Forum_Avatar" could be used by the forum module to define an Avatar for the user.

```
Dim avatar as string = User.Profile.ProfileProperties("Forum_Avatar")
```

There are two alternative approaches for the "default" properties. They can be defined by using a PortalId of "-1", so they only have to be defined "once", or they can be defined for each Portal.

The advantages of defining them with a PortalId of "-1" is that they are only created once. The disadvantage is that the ViewOrder/Visible and Required properties cannot be defined by "Portal".

The advantages of defining them by Portal are that the ViewOrder/Visible and Required properties can now be defined by Portal. The disadvantage is that they will need to be created (and deleted) when portals are added/deleted.

New ProfilePropertyDefinitionCollection class

It is also proposed that a new ProfilePropertyDefinitionCollection class be implemented. This would inherit from DictionaryBase so that a collection can be accessed by using the PropertyName as key.

```
Dim myProperty as ProfilePropertyDefinition = properties("MyProperty")
```

It is also recommended that the following properties be added to the collection:

- Required – returns all the properties that are required.
- Optional – returns all the properties that are not required.
- Visible – returns all the properties are Visible

The above properties would all return a ProfilePropertyDefinitionCollection so we could use the following:

```
Dim requiredProperties as ProfilePropertyDefinitionCollection = properties.Required()
```

Personalization Integration

Introduction

This enhancement would integrate the current Personalization Components into an all-inclusive profile management.

DotNetNuke Personalization Components

DotNetNuke provides a Personalization framework independent of the UserProfile. There are 4 components to the Personalization Framework.

- PersonalizationModule – an HttpModule that automatically persists the Users Personalization values to the Database, at the end of a Request
- PersonalizationInfo – an Entity class that describes a users Personalization
  - UserId – Id of the User
  - PortalId – Id of the Portal
  - IsModified – flag that indicates whether a profile has been modified (used by PerosnalizationModule to determine whether the profile needs to be updated in the Data Store
  - Profile – a Hashtable of "profile values"
- Personalization – a Helper class that provides Shared (static) methods
  - GetProfile – gets a "profile value" from the PersonalizationInfo.Profile Hashtable
  - SaveProfile – saves a "profile value" to the PersonalizationInfo.Profile Hashtable
  - RemoveProfile removes a "profile value" from the PersonalizationInfo.Profile Hashtable
- PersonalizationController – a Controller class that works with the Dataprovider to provide LoadProfile and SaveProfile methods. The actual implementation is that

the profile is serialized using XMLSerialization and saved to the DotNetNuke UserProfile table.

Integration of Personalization with Profile

There is no need to have two different Personalization/Profile implementations so this enhancement proposes that Personalization be deprecated, in favour of an enhanced profile management system.

- PersonalizationController and PersonalizationInfo classes would be deprecated.
- Personalization's shared/static methods would be refactored to save the "profile value" to the Profile object.
- The IsModified property of the PersonalizationInfo class would need to be added to the UserProfile class.
- PersonalizationModule would still be required, as many of the personalizations and many Module profile properties are only updated in the object, and need to be persisted. The difference is that instead of this module calling the PersonalizationController to persist the profile it would call UserController.UpdateUserProfile.

Searchable Profile Data

Introduction

This enhancement would provide the ability to Search the Data Store for Users that have a particular profile property.

Addition to UserController

In order to support a Search by Profile Property feature we will need to provide an additional User Controller method.

GetUsersByProfileProperty

The GetUsersByProfileProperty method will work similarly to the other GetUsers methods (GetUsersByName and GetUsersByEmail) to return a page of Users that satisfy the property match.

```
Public Shared Function GetUsersByProfileProperty (ByVal portalId As Integer, ByVal
isHydrated As Boolean, ByVal propertyName As String, ByVal propertyValue As String, ByVal
pageIndex As Integer, ByVal pageSize As Integer, ByRef totalRecords As Integer) As
ArrayList
```

This is trivial to implement in the Business Layer and Data Layer. The difficulty here is implementing this in the Data Base itself (see later)

Customised Profile Visibility

Introduction

This enhancement would allow a user to control the visibility of their profile (to other users), by providing a "Visible" checkbox next to every profile property.

This would be quite a significant enhancement as it would require quite a change to the profile object model as well as significant impact on how the data is stored in the database.

Having said that, we should not make any "implementation" decisions that would cause us extreme difficulty implementing this feature at some time in the future.

New ProfileProperty Class

A new profile property class would need to be added.

- PropertyId
- PropertyValue
- IsPublic

This class would be used for every profile property, so for example FirstName would be changed to

```
        Public Property FirstName() As String
            Get
                Dim retValue As String = Null.NullString
                Dim prop as ProfileProperty
                If _profileProperties.ContainsKey(cFirstName) = False OrElse
                       profileProperties(cFirstName) Is Nothing Then
                    prop = CType(_profileProperties(cFirstName), ProfileProperty)
                    retValue = prop.PropertyValue
                End If
                Return retValue
            End Get
```

```
            Set(ByVal Value As String)
                SetProfileProperty(cFirstName, Value)
                If Not ObjectHydrated Then
                    ObjectHydrated = True
                End If
            End Set
        End Property
```

and SetProfileProperty would need to be modified similarly. In addition a new SetVisibility method would need to be added to set the Visibility of a property.

ProfileProvider Enhancements

Introduction

So having significantly enhanced the Business Layer, how can we implement these enhancements in the profile provider. Ideally, we would like to find a way to be able to use the AspNetProfileProvider, as well as our own DNNProfileProvider, but we should be prepared to drop completely the AspNetProfileProvider if this does not prove feasible.

Profile Text Blob

The default MemberRole implementation of profile uses a Text Blob. The Database Table aspnet_Profile has five columns:

- UserId – unique identifier (primary key)
- PropertyNames – an ntext field that stores the property names together with a code that indicates the type, a start index and a length. These are stored as a colon delimited list (eg Website:S:0:0:TimeZone:S:0:1:PreferredLocale:S:1:5:Region:S:6:0:FirstName:S:6:9:PostalCode:S:15:0:Street:S:15:0:LastName:S:15:7:Unit:S:22:0:Telephone:S:22:0:Country:S:22:0:IM:S:22:0:Fax:S:22:0:City:S:22:0:Cell:S:22:0:)
- PropertyValuesString – an ntext field that store the values of the properties. The string can be decoded by using the definition provided in the PropertyNames field.
- PropertyValuesBinary – an image filed that store the values of properties serialized by Binary serialization
- LastUpdatedTime – a datetime field that store the last time the profile was updated.

The benefit of this approach is that it is easily extensible. The downside is that the properties must be defined in the web.config, which limits us from defining custom properties.

Also, it is not trivial to search on the profile property, but it can be done. For instance, we can define the following functions

```
ALTER FUNCTION [dbo].[fn GetElement]
(
   @ord AS INT,
   @str AS VARCHAR(8000),
   @delim AS VARCHAR(1) )

   RETURNS INT
AS
BEGIN
   -- If input is invalid, return null.
   IF  @str IS NULL
       OR LEN(@str) = 0
       OR @ord IS NULL
       OR @ord < 1
       -- @ord > [is the] expression that calculates the number of elements.
       OR @ord > LEN(@str) - LEN(REPLACE(@str, @delim, '')) + 1
      RETURN NULL

   DECLARE @pos AS INT, @curord AS INT

   SELECT @pos = 1, @curord = 1

   -- Find next element's start position and increment index.
   WHILE @curord < @ord
      SELECT
          @pos    = CHARINDEX(@delim, @str, @pos) + 1,
          @curord = @curord + 1
      RETURN    CAST(SUBSTRING(@str, @pos, CHARINDEX(@delim, @str + @delim, @pos) - @pos)
AS INT)
END
```

And

```
ALTER FUNCTION [dbo].[fn_GetProfileElement]
(
   @fieldName AS NVARCHAR(100),
   @fields AS NVARCHAR(4000),
   @values AS NVARCHAR(4000))

   RETURNS NVARCHAR(4000)
AS
BEGIN
    -- If input is invalid, return null.
    IF  @fieldName IS NULL
       OR LEN(@fieldName) = 0
       OR @fields IS NULL
       OR LEN(@fields) = 0
       OR @values IS NULL
       OR LEN(@values) = 0
```

```
      RETURN NULL

   -- locate FieldName in Fields
   DECLARE @fieldNameToken AS NVARCHAR(20)
   DECLARE @fieldNameStart AS INTEGER, @valueStart AS INTEGER, @valueLength AS INTEGER

   -- Only handle string type fields (:S:)
   SET @fieldNameStart = CHARINDEX(@fieldName + ':S',@Fields,0)

   -- If field is not found, return null
   IF @fieldNameStart = 0 RETURN NULL
   SET @fieldNameStart = @fieldNameStart + LEN(@fieldName) + 3

   -- Get the field token which I've defined as the start of the field offset to the end
of the length
   SET @fieldNameToken = SUBSTRING(@Fields,@fieldNameStart,LEN(@Fields)-@fieldNameStart)

   -- Get the values for the offset and length
   SET @valueStart = dbo.fn getelement(1,@fieldNameToken,':')
   SET @valueLength = dbo.fn_getelement(2,@fieldNameToken,':')

   -- Check for sane values, 0 length means the profile item was stored, just no data
   IF @valueLength = 0 RETURN ''

   -- Return the string
   RETURN SUBSTRING(@values, @valueStart+1, @valueLength)
END
```

These functions can then be used in SELECT statements such as

```
DECLARE @PropertyName AS NVARCHAR(20)
DECLARE @PropertyValue AS NVARCHAR(20)

SELECT
      u.UserName,
      m.Email,
      m.PasswordQuestion,
      m.Comment,
      m.IsApproved,
      m.IsLockedOut,
      m.CreateDate,
      m.LastLoginDate,
      u.LastActivityDate,
      m.LastPasswordChangedDate,
      u.UserId,
      m.LastLockoutDate
FROM  dbo.aspnet_Membership m
   INNER JOIN dbo.aspnet_Users u ON u.UserId = m.UserId
   INNER JOIN dbo.aspnet_Profile p ON p.UserId = u.UserId
WHERE (dbo.fn_GetProfileElement(@PropertyName, p.PropertyNames, p.PropertyValuesString) =
@PropertyValue)
```

This select statement will return the same datareader as FindUsersByName and FindUsersByEmail. However the drawback is that the performance of these statements is likely to be relatively poor, due to the amount of string manipulation.

Profile XML Blob

An extension of the Text Blob, this approach is similar to the Text Blob approach in that the profile is saved as a Blob into a single field. The difference here is that there is a structure to the text, which depends on how it is Serialized. There is precedence for using this mechanism as the DNN Personalization Framework uses XML Serialization to persist its profile.

The benefits to this approach are that the profile is easily extensible. At the same time, only one property needs to be defined in web.config (ProfileProperties, type=String), so the profile can contain any number of properties of any time (as long as they are xml serializable). This allows for portal level property definitions.

So, what are the downside of this persistence mechanism? The main downside is that again it is not easy to make the Profile searchable. SQL Server 2000 does include some XML capabilities in particular the OPENXML clause and the sp_xml_preparedocument and sp_xml_removedocument system stored procedures, which should be able to build functions similar to the functions defined above to manipulate the Text Blob.

Relational Tables

We could replace the current Profile implementation by using a set of relational tables. For instance we would need the following tables

UserProfile

- ProfileId
- UserId
- PortalId
- Prefix
- FirstName
- MiddleName
- LastName
- Suffix
- TimeZone
- Preferred Locale
- LastUpdated

Address

- AddressId
- Type
- Primary
- Organization
- Unit
- Street
- City
- Region
- Country
- PostCode

Phone

- PhoneId
- Type
- Primary
- InternationalCode
- LocalAreaCode
- Number
- Extension
- Note/Comment

Uri

- UriId
- Type
- Uri

As these tables could potentially be used by other entities (eg Contacts), it is proposed that the relationship is established using Link tables: eg

UserProfileAddress

- ProfileId
- AddressId

UserProfilePhone

- ProfileId
- PhoneId

UserProfileUri

- ProfileId
- UriId

where the ProfileId forms a Foreign Key into UserProfile, and AddressId, PhoneId, & UriId form a Foreign Key into their respective tables. These tables deal effectively with the standard "profile properties". Custom profile properties could then be handled by a UserProfileProperty table

UserProfileProperty

- ProfileId
- PropertyName
- PropertyValue

The benefits of this approach is that it is the most efficient storage mechanism for the Profile in terms of Performance, and searchability. It is easy to write a customized Stored Procedure to return all Users whose Uri contains the text "aol" for instance. However, writing a generic search Stored procedure as envisaged in the previous section would require a complicated Stored Procedure, with statements such as

```
IF @PropertyName = 'City'
   SELECT … WHERE Address.City = @PropertyValue

IF @PropertyName = 'LastName'
   SELECT … WHERE Profile.LastName = @PropertyValue

etc
```

One Single Normalized Profile Table

We could replace the current Profile implementation by using a single normalized table for the Profile Properties. Thus all profile properties would be saved to an enhanced version of the UserProfile table listed above.

UserProfile

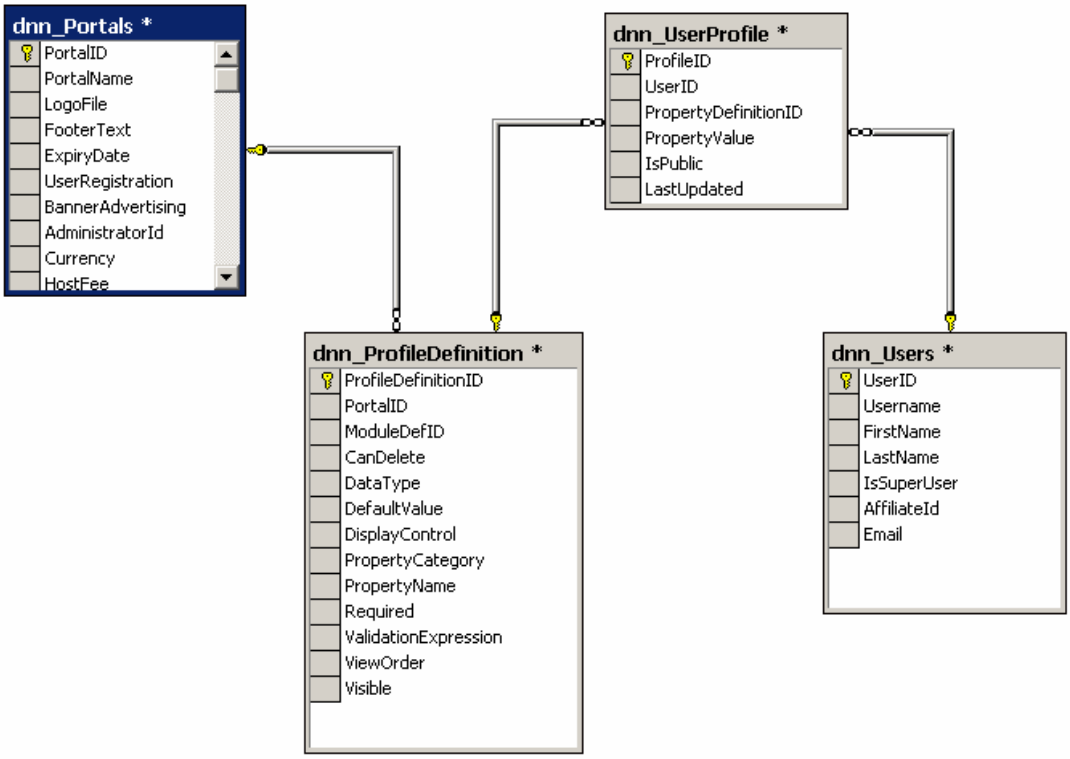
- ProfileId – A unique Id (primary key)
- UserId – The Id of the User
- PropertyDefinitionId – The Id of the property definition (this would include a reference to the portal)
- PropertyValue – the value of the property
- IsPublic – a flag that indicates whether the profile property is visible
- LastUpdated

The advantage of this approach is that is fairly quick and easy to find Users based on a Search requirement, with a simple join of Users and ProfileProperty. The disadvantage of this approach is that the table will become extremely large, quite quickly.

However, large tables with Terabytes of data, if well indexed is what databases are optimized to handle, so while it could become an issue (and an alternative provider might become required for some scenarios), it is recommended that this storage mechanism is optimum for the default DotNetNuke provider.

There remains one outstanding issue, if this approach was taken and that is – how would we update the profile in the database? One big update stored procedure with a huge number of parameters? Or a simple UpdateProfileProperty stored procedure that might get called 30-40 times for each profile change?

One solution to this would be a hybrid. An UpdateProfileProperty stored procedure would allow a single profile property to be updated, while an UpdateProfile stored procedure would take some form of delimited or structured text, parse it and call the UpdateProfile stored procedure for each individual property.



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	Jan 23, 2006	Charles Nurse	• First Draft