**Profile Provider Implementation Example**

This page is specific to

**Microsoft Visual Studio 2008/.NET Framework 3.5**

Other versions are also available for the following:

[Microsoft Visual Studio 2005/.NET Framework 2.0](http://msdn.microsoft.com/en-us/library/ta63b872(VS.80).aspx)

[.NET Framework 3.0](http://msdn.microsoft.com/en-us/library/ta63b872(VS.85).aspx)

[Microsoft Visual Studio 2010/.NET Framework 4](http://msdn.microsoft.com/en-us/library/ta63b872(VS.100).aspx)

The code example in this topic illustrates an ASP.NET profile provider implementation. For information about how to compile this code and use the provider, see [How to: Build and Run the Profile Provider Example](http://msdn.microsoft.com/en-us/library/8zs47k7y.aspx).

http://i.msdn.microsoft.com/Global/Images/clear.gif Example

**Code**

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl10_ctl00_ctl00_code');" \o "Copy Code)

Imports System.Web.Profile

Imports System.Configuration.Provider

Imports System.Collections.Specialized

Imports System

Imports System.Data

Imports System.Data.Odbc

Imports System.Configuration

Imports System.Diagnostics

Imports System.Web

Imports System.Collections

Imports Microsoft.VisualBasic

'

'

' This provider works with the following schema for the table of user data.

'

' CREATE TABLE Profiles

' (

' UniqueID AutoIncrement NOT NULL PRIMARY KEY,

' Username Text (255) NOT NULL,

' ApplicationName Text (255) NOT NULL,

' IsAnonymous YesNo,

' LastActivityDate DateTime,

' LastUpdatedDate DateTime,

' CONSTRAINT PKProfiles UNIQUE (Username, ApplicationName)

' )

'

' CREATE TABLE StockSymbols

' (

' UniqueID Integer,

' StockSymbol Text (10),

' CONSTRAINT FKProfiles1 FOREIGN KEY (UniqueID)

' REFERENCES Profiles

' )

'

' CREATE TABLE ProfileData

' (

' UniqueID Integer,

' ZipCode Text (10),

' CONSTRAINT FKProfiles2 FOREIGN KEY (UniqueID)

' REFERENCES Profiles

' )

'

'

Namespace Samples.AspNet.Profile

Public NotInheritable Class OdbcProfileProvider

Inherits ProfileProvider

'

' Global connection string, generic exception message, event log info.

'

Private eventSource As String = "OdbcProfileProvider"

Private eventLog As String = "Application"

Private exceptionMessage As String = "An exception occurred. Please check the event log."

Private connectionString As String

'

' If false, exceptions are thrown to the caller. If true,

' exceptions are written to the event log.

'

Private pWriteExceptionsToEventLog As Boolean

Public Property WriteExceptionsToEventLog As Boolean

Get

Return pWriteExceptionsToEventLog

End Get

Set

pWriteExceptionsToEventLog = value

End Set

End Property

'

' System.Configuration.Provider.ProviderBase.Initialize Method

'

Public Overrides Sub Initialize(name As String, config As NameValueCollection)

'

' Initialize values from web.config.

'

If config Is Nothing Then \_

Throw New ArgumentNullException("config")

If name Is Nothing OrElse name.Length = 0 Then \_

name = "OdbcProfileProvider"

If String.IsNullOrEmpty(config("description")) Then

config.Remove("description")

config.Add("description", "Sample ODBC Profile provider")

End If

' Initialize the abstract base class.

MyBase.Initialize(name, config)

If config("applicationName") Is Nothing OrElse config("applicationName").Trim() = "" Then

pApplicationName = System.Web.Hosting.HostingEnvironment.ApplicationVirtualPath

Else

pApplicationName = config("applicationName")

End If

'

' Initialize connection string.

'

Dim pConnectionStringSettings As ConnectionStringSettings = \_

ConfigurationManager.ConnectionStrings(config("connectionStringName"))

If pConnectionStringSettings Is Nothing OrElse \_

pConnectionStringSettings.ConnectionString.Trim() = "" \_

Then

Throw New ProviderException("Connection String cannot be blank.")

End If

connectionString = pConnectionStringSettings.ConnectionString

End Sub

'

' System.Configuration.SettingsProvider.ApplicationName

'

Private pApplicationName As String

Public Overrides Property ApplicationName As String

Get

Return pApplicationName

End Get

Set

pApplicationName = value

End Set

End Property

'

' System.Configuration.SettingsProvider methods.

'

'

' SettingsProvider.GetPropertyValues

'

Public Overrides Function GetPropertyValues(context As SettingsContext, \_

ppc As SettingsPropertyCollection) \_

As SettingsPropertyValueCollection

' The serializeAs attribute is ignored in this provider implementation.

Dim username As String = CType(context("UserName"), String)

Dim isAuthenticated As Boolean = CType(context("IsAuthenticated"), Boolean)

Dim svc As SettingsPropertyValueCollection = New SettingsPropertyValueCollection()

For Each prop As SettingsProperty In ppc

Dim pv As SettingsPropertyValue = New SettingsPropertyValue(prop)

Select Case prop.Name

Case "StockSymbols"

pv.PropertyValue = GetStockSymbols(username, isAuthenticated)

Case "ZipCode"

pv.PropertyValue = GetZipCode(username, isAuthenticated)

Case Else

Throw New ProviderException("Unsupported property.")

End Select

svc.Add(pv)

Next

UpdateActivityDates(username, isAuthenticated, True)

Return svc

End Function

'

' SettingsProvider.SetPropertyValues

'

Public Overrides Sub SetPropertyValues(context As SettingsContext, \_

ppvc As SettingsPropertyValueCollection)

' The serializeAs attribute is ignored in this provider implementation.

Dim username As String = CType(context("UserName"), String)

Dim isAuthenticated As Boolean = CType(context("IsAuthenticated"), Boolean)

Dim uniqueID As Integer = GetUniqueID(username, isAuthenticated, False)

If uniqueID = 0 Then uniqueID = CreateProfileForUser(username, isAuthenticated)

For Each pv As SettingsPropertyValue In ppvc

Select Case pv.Property.Name

Case "StockSymbols"

SetStockSymbols(uniqueID, CType(pv.PropertyValue, ArrayList))

Case "ZipCode"

SetZipCode(uniqueID, CType(pv.PropertyValue, String))

Case Else

Throw New ProviderException("Unsupported property.")

End Select

Next

UpdateActivityDates(username, isAuthenticated, False)

End Sub

'

' UpdateActivityDates

' Updates LastActivityDate and LastUpdatedDate when profile properties are accessed

' by GetPropertyValues and SetPropertyValues. Specifying activityOnly as true will update

' only the LastActivityDate.

'

Private Sub UpdateActivityDates(username As String, isAuthenticated As Boolean, activityOnly As Boolean)

Dim activityDate As DateTime = DateTime.Now

Dim conn As OdbcConnection = New OdbcConnection(connectionString)

Dim cmd As OdbcCommand = New OdbcCommand()

cmd.Connection = conn

If activityOnly Then

cmd.CommandText = "UPDATE Profiles Set LastActivityDate = ? " & \_

"WHERE Username = ? AND ApplicationName = ? AND IsAnonymous = ?"

cmd.Parameters.Add("@LastActivityDate", OdbcType.DateTime).Value = activityDate

cmd.Parameters.Add("@Username", OdbcType.VarChar, 255).Value = username

cmd.Parameters.Add("@ApplicationName", OdbcType.VarChar, 255).Value = ApplicationName

cmd.Parameters.Add("@IsAnonymous", OdbcType.Bit).Value = Not isAuthenticated

Else

cmd.CommandText = "UPDATE Profiles Set LastActivityDate = ?, LastUpdatedDate = ? " & \_

"WHERE Username = ? AND ApplicationName = ? AND IsAnonymous = ?"

cmd.Parameters.Add("@LastActivityDate", OdbcType.DateTime).Value = activityDate

cmd.Parameters.Add("@LastUpdatedDate", OdbcType.DateTime).Value = activityDate

cmd.Parameters.Add("@Username", OdbcType.VarChar, 255).Value = username

cmd.Parameters.Add("@ApplicationName", OdbcType.VarChar, 255).Value = ApplicationName

cmd.Parameters.Add("@IsAnonymous", OdbcType.Bit).Value = Not isAuthenticated

End If

Try

conn.Open()

cmd.ExecuteNonQuery()

Catch e As OdbcException

If WriteExceptionsToEventLog Then

WriteToEventLog(e, "UpdateActivityDates")

Throw New ProviderException(exceptionMessage)

Else

Throw e

End If

Finally

conn.Close()

End Try

End Sub

'

' GetStockSymbols

' Retrieves stock symbols from the database during the

' call to GetPropertyValues.

'

Private Function GetStockSymbols(username As String, isAuthenticated As Boolean) As ArrayList

Dim conn As OdbcConnection = New OdbcConnection(connectionString)

Dim cmd As OdbcCommand = New OdbcCommand("SELECT StockSymbol FROM Profiles " & \_

"INNER JOIN StockSymbols ON Profiles.UniqueID = StockSymbols.UniqueID " & \_

"WHERE Username = ? AND ApplicationName = ? And IsAnonymous = ?", conn)

cmd.Parameters.Add("@Username", OdbcType.VarChar, 255).Value = username

cmd.Parameters.Add("@ApplicationName", OdbcType.VarChar, 255).Value = ApplicationName

cmd.Parameters.Add("@IsAnonymous", OdbcType.Bit).Value = Not isAuthenticated

Dim outList As ArrayList = New ArrayList()

Dim reader As OdbcDataReader = Nothing

Try

conn.Open()

reader = cmd.ExecuteReader()

Do While reader.Read()

outList.Add(reader.GetString(0))

Loop

Catch e As OdbcException

If WriteExceptionsToEventLog Then

WriteToEventLog(e, "GetStockSymbols")

Throw New ProviderException(exceptionMessage)

Else

Throw e

End If

Finally

If Not reader Is Nothing Then reader.Close()

conn.Close()

End Try

Return outList

End Function

'

' SetStockSymbols

' Inserts stock symbol values into the database during

' the call to SetPropertyValues.

'

Private Sub SetStockSymbols(uniqueID As Integer, stocks As ArrayList)

Dim conn As OdbcConnection = New OdbcConnection(connectionString)

Dim cmd As OdbcCommand = New OdbcCommand("DELETE FROM StockSymbols WHERE UniqueID = ?", conn)

cmd.Parameters.Add("@UniqueID", OdbcType.Int).Value = uniqueID

Dim cmd2 As OdbcCommand = New OdbcCommand("INSERT INTO StockSymbols (UniqueID, StockSymbol) " & \_

"Values(?, ?)", conn)

cmd2.Parameters.Add("@UniqueID", OdbcType.Int).Value = uniqueID

cmd2.Parameters.Add("@StockSymbol", OdbcType.VarChar, 10)

Dim tran As OdbcTransaction = Nothing

Try

conn.Open()

tran = conn.BeginTransaction()

cmd.Transaction = tran

cmd2.Transaction = tran

' Delete any existing values.

cmd.ExecuteNonQuery()

For Each o As Object In Stocks

cmd2.Parameters("@StockSymbol").Value = o.ToString()

cmd2.ExecuteNonQuery()

Next

tran.Commit()

Catch e As OdbcException

Try

tran.Rollback()

Catch

End Try

If WriteExceptionsToEventLog Then

WriteToEventLog(e, "SetStockSymbols")

Throw New ProviderException(exceptionMessage)

Else

Throw e

End If

Finally

conn.Close()

End Try

End Sub

'

' GetZipCode

' Retrieves ZipCode value from the database during

' the call to GetPropertyValues.

'

Private Function GetZipCode(username As String, isAuthenticated As Boolean) As String

Dim zipCode As String = ""

Dim conn As OdbcConnection = New OdbcConnection(connectionString)

Dim cmd As OdbcCommand = New OdbcCommand("SELECT ZipCode FROM Profiles " & \_

"INNER JOIN ProfileData ON Profiles.UniqueID = ProfileData.UniqueID " & \_

"WHERE Username = ? AND ApplicationName = ? And IsAnonymous = ?", conn)

cmd.Parameters.Add("@Username", OdbcType.VarChar, 255).Value = username

cmd.Parameters.Add("@ApplicationName", OdbcType.VarChar, 255).Value = ApplicationName

cmd.Parameters.Add("@IsAnonymous", OdbcType.Bit).Value = Not isAuthenticated

Try

conn.Open()

zipCode = CType(cmd.ExecuteScalar(), String)

Catch e As OdbcException

If WriteExceptionsToEventLog Then

WriteToEventLog(e, "GetZipCode")

Throw New ProviderException(exceptionMessage)

Else

Throw e

End If

Finally

conn.Close()

End Try

Return zipCode

End Function

'

' SetZipCode

' Inserts the zip code value into the database during

' the call to SetPropertyValues.

'

Private Sub SetZipCode(uniqueID As Integer, zipCode As String)

If zipCode Is Nothing Then zipCode = String.Empty

Dim conn As OdbcConnection = New OdbcConnection(connectionString)

Dim cmd As OdbcCommand = New OdbcCommand("DELETE FROM ProfileData WHERE UniqueID = ?", conn)

cmd.Parameters.Add("@UniqueID", OdbcType.Int).Value = uniqueID

Dim cmd2 As OdbcCommand= New OdbcCommand("INSERT INTO ProfileData (UniqueID, ZipCode) " & \_

"Values(?, ?)", conn)

cmd2.Parameters.Add("@UniqueID", OdbcType.Int).Value = uniqueID

cmd2.Parameters.Add("@ZipCode", OdbcType.VarChar, 10).Value = zipCode

Dim tran As OdbcTransaction = Nothing

Try

conn.Open()

tran = conn.BeginTransaction()

cmd.Transaction = tran

cmd2.Transaction = tran

' Delete any existing values.

cmd.ExecuteNonQuery()

cmd2.ExecuteNonQuery()

tran.Commit()

Catch e As OdbcException

Try

tran.Rollback()

Catch

End Try

If WriteExceptionsToEventLog Then

WriteToEventLog(e, "SetZipCode")

Throw New ProviderException(exceptionMessage)

Else

Throw e

End If

Finally

conn.Close()

End Try

End Sub

'

' GetUniqueID

' Retrieves the uniqueID from the database for

' the current user and application.

'

Private Function GetUniqueID(username As String, isAuthenticated As Boolean, \_

ignoreAuthenticationType As Boolean) As Integer

Dim conn As OdbcConnection = New OdbcConnection(connectionString)

Dim cmd As OdbcCommand = New OdbcCommand("SELECT UniqueID FROM Profiles " & \_

"WHERE Username = ? AND ApplicationName = ?", conn)

cmd.Parameters.Add("@Username", OdbcType.VarChar, 255).Value = username

cmd.Parameters.Add("@ApplicationName", OdbcType.VarChar, 255).Value = ApplicationName

If Not ignoreAuthenticationType Then

cmd.CommandText &= " AND IsAnonymous = ?"

cmd.Parameters.Add("@IsAnonymous", OdbcType.Bit).Value = Not isAuthenticated

End If

Dim uniqueID As Integer = 0

Dim reader As OdbcDataReader = Nothing

Try

conn.Open()

reader = cmd.ExecuteReader(CommandBehavior.SingleRow)

If reader.HasRows Then \_

uniqueID = reader.GetInt32(0)

Catch e As OdbcException

If WriteExceptionsToEventLog Then

WriteToEventLog(e, "GetUniqueID")

Throw New ProviderException(exceptionMessage)

Else

Throw e

End If

Finally

If reader IsNot Nothing Then reader.Close()

conn.Close()

End Try

Return uniqueID

End Function

'

' CreateProfileForUser

' If no user currently exists in the database,

' a user record is created during the call to the

' GetUniqueID Private method.

'

Private Function CreateProfileForUser(username As String, isAuthenticated As Boolean) As Integer

' Check for valid user name.

If username Is Nothing Then \_

Throw New ArgumentNullException("username")

If username.Length > 255 Then \_

Throw New ArgumentException("User name exceeds 255 characters.")

If username.Contains(",") Then \_

Throw New ArgumentException("User name cannot contain a comma (,).")

Dim conn As OdbcConnection = New OdbcConnection(connectionString)

Dim cmd As OdbcCommand = New OdbcCommand("INSERT INTO Profiles (Username, " & \_

"ApplicationName, LastActivityDate, LastUpdatedDate, " & \_

"IsAnonymous) Values(?, ?, ?, ?, ?)", conn)

cmd.Parameters.Add("@Username", OdbcType.VarChar, 255).Value = username

cmd.Parameters.Add("@ApplicationName", OdbcType.VarChar, 255).Value = ApplicationName

cmd.Parameters.Add("@LastActivityDate", OdbcType.DateTime).Value = DateTime.Now

cmd.Parameters.Add("@LastUpdatedDate", OdbcType.VarChar).Value = DateTime.Now

cmd.Parameters.Add("@IsAnonymous", OdbcType.Bit).Value = Not isAuthenticated

Dim cmd2 As OdbcCommand = New OdbcCommand("SELECT @@IDENTITY", conn)

Dim uniqueID As Integer = 0

Try

conn.Open()

cmd.ExecuteNonQuery()

uniqueID = CType(cmd2.ExecuteScalar(), Integer)

Catch e As OdbcException

If WriteExceptionsToEventLog Then

WriteToEventLog(e, "CreateProfileForUser")

Throw New HttpException(exceptionMessage)

Else

Throw e

End If

Finally

conn.Close()

End Try

Return uniqueID

End Function

'

' ProfileProvider.DeleteProfiles(ProfileInfoCollection)

'

Public Overrides Function DeleteProfiles(profiles As ProfileInfoCollection) As Integer

Dim deleteCount As Integer = 0

Dim conn As OdbcConnection = New OdbcConnection(connectionString)

Dim tran As OdbcTransaction = Nothing

Try

conn.Open()

tran = conn.BeginTransaction()

For Each p As ProfileInfo In profiles

If DeleteProfile(p.UserName, conn, tran) Then deleteCount += 1

Next

tran.Commit()

Catch e As Exception

Try

tran.Rollback()

Catch

End Try

If WriteExceptionsToEventLog Then

WriteToEventLog(e, "DeleteProfiles(String())")

Throw New ProviderException(exceptionMessage)

Else

Throw e

End If

Finally

conn.Close()

End Try

Return deleteCount

End Function

'

' ProfileProvider.DeleteProfiles(string())

'

Public Overrides Function DeleteProfiles(usernames As String()) As Integer

Dim deleteCount As Integer = 0

Dim conn As OdbcConnection = New OdbcConnection(connectionString)

Dim tran As OdbcTransaction = Nothing

Try

conn.Open()

tran = conn.BeginTransaction()

For Each user As String In usernames

If (DeleteProfile(user, conn, tran)) Then deleteCount += 1

Next

tran.Commit()

Catch e As Exception

Try

tran.Rollback()

Catch

End Try

If WriteExceptionsToEventLog Then

WriteToEventLog(e, "DeleteProfiles(String())")

Throw New ProviderException(exceptionMessage)

Else

Throw e

End If

Finally

conn.Close()

End Try

Return deleteCount

End Function

'

' ProfileProvider.DeleteInactiveProfiles

'

Public Overrides Function DeleteInactiveProfiles( \_

authenticationOption As ProfileAuthenticationOption, \_

userInactiveSinceDate As DateTime) As Integer

Dim conn As OdbcConnection = New OdbcConnection(connectionString)

Dim cmd As OdbcCommand = New OdbcCommand("SELECT Username FROM Profiles " & \_

"WHERE ApplicationName = ? AND " & \_

" LastActivityDate <= ?", conn)

cmd.Parameters.Add("@ApplicationName", OdbcType.VarChar, 255).Value = ApplicationName

cmd.Parameters.Add("@LastActivityDate", OdbcType.DateTime).Value = userInactiveSinceDate

Select Case authenticationOption

Case ProfileAuthenticationOption.Anonymous

cmd.CommandText &= " AND IsAnonymous = ?"

cmd.Parameters.Add("@IsAnonymous", OdbcType.Bit).Value = True

Case ProfileAuthenticationOption.Authenticated

cmd.CommandText &= " AND IsAnonymous = ?"

cmd.Parameters.Add("@IsAnonymous", OdbcType.Bit).Value = False

End Select

Dim reader As OdbcDataReader = Nothing

Dim usernames As String = ""

Try

conn.Open()

reader = cmd.ExecuteReader()

Do While reader.Read()

usernames &= reader.GetString(0) + ","

Loop

Catch e As OdbcException

If WriteExceptionsToEventLog Then

WriteToEventLog(e, "DeleteInactiveProfiles")

Throw New ProviderException(exceptionMessage)

Else

Throw e

End If

Finally

If Not reader Is Nothing Then reader.Close()

conn.Close()

End Try

If usernames.Length > 0 Then

' Remove trailing comma.

usernames = usernames.Substring(0, usernames.Length - 1)

End If

' Delete profiles.

Return DeleteProfiles(usernames.Split(CChar(",")))

End Function

'

'

' DeleteProfile

' Deletes profile data from the database for the specified user name. Expects an OdbcConnection and

' an OdbcTransaction as it supports deleting multiple profiles in a transaction.

'

Private Function DeleteProfile(username As String, conn As OdbcConnection, tran As OdbcTransaction) As Boolean

' Check for valid user name.

If username Is Nothing Then \_

Throw New ArgumentNullException("username")

If username.Length > 255 Then \_

Throw New ArgumentException("User name exceeds 255 characters.")

If username.Contains(",") Then \_

Throw New ArgumentException("User name cannot contain a comma (,).")

Dim uniqueID As Integer = GetUniqueID(username, False, True)

Dim cmd1 As OdbcCommand = New OdbcCommand("DELETE \* FROM ProfileData WHERE UniqueID = ?", conn)

cmd1.Parameters.Add("@UniqueID", OdbcType.Int).Value = uniqueID

Dim cmd2 As OdbcCommand = New OdbcCommand("DELETE \* FROM StockSymbols WHERE UniqueID = ?", conn)

cmd2.Parameters.Add("@UniqueID", OdbcType.Int).Value = uniqueID

Dim cmd3 As OdbcCommand = New OdbcCommand("DELETE \* FROM Profiles WHERE UniqueID = ?", conn)

cmd3.Parameters.Add("@UniqueID", OdbcType.Int).Value = uniqueID

cmd1.Transaction = tran

cmd2.Transaction = tran

cmd3.Transaction = tran

Dim numDeleted As Integer = 0

' Exceptions will be caught by the calling method.

numDeleted += cmd1.ExecuteNonQuery()

numDeleted += cmd2.ExecuteNonQuery()

numDeleted += cmd3.ExecuteNonQuery()

If numDeleted = 0 Then

Return False

Else

Return True

End If

End Function

'

' ProfileProvider.FindProfilesByUserName

'

Public Overrides Function FindProfilesByUserName( \_

authenticationOption As ProfileAuthenticationOption, \_

usernameToMatch As String, \_

pageIndex As Integer, \_

pageSize As Integer, \_

ByRef totalRecords As Integer) As ProfileInfoCollection

CheckParameters(pageIndex, pageSize)

Return GetProfileInfo(authenticationOption, usernameToMatch, Nothing, \_

pageIndex, pageSize, totalRecords)

End Function

'

' ProfileProvider.FindInactiveProfilesByUserName

'

Public Overrides Function FindInactiveProfilesByUserName( \_

authenticationOption As ProfileAuthenticationOption, \_

usernameToMatch As String, \_

userInactiveSinceDate As DateTime, \_

pageIndex As Integer, \_

pageSize As Integer, \_

ByRef totalRecords As Integer) As ProfileInfoCollection

CheckParameters(pageIndex, pageSize)

Return GetProfileInfo(authenticationOption, usernameToMatch, userInactiveSinceDate, \_

pageIndex, pageSize, totalRecords)

End Function

'

' ProfileProvider.GetAllProfiles

'

Public Overrides Function GetAllProfiles( \_

authenticationOption As ProfileAuthenticationOption, \_

pageIndex As Integer, \_

pageSize As Integer, \_

ByRef totalRecords As Integer) As ProfileInfoCollection

CheckParameters(pageIndex, pageSize)

Return GetProfileInfo(authenticationOption, Nothing, Nothing, \_

pageIndex, pageSize, totalRecords)

End Function

'

' ProfileProvider.GetAllInactiveProfiles

'

Public Overrides Function GetAllInactiveProfiles( \_

authenticationOption As ProfileAuthenticationOption, \_

userInactiveSinceDate As DateTime, \_

pageIndex As Integer, \_

pageSize As Integer, \_

ByRef totalRecords As Integer) As ProfileInfoCollection

CheckParameters(pageIndex, pageSize)

Return GetProfileInfo(authenticationOption, Nothing, userInactiveSinceDate, \_

pageIndex, pageSize, totalRecords)

End Function

'

' ProfileProvider.GetNumberOfInactiveProfiles

'

Public Overrides Function GetNumberOfInactiveProfiles( \_

authenticationOption As ProfileAuthenticationOption, \_

userInactiveSinceDate As DateTime) As Integer

Dim inactiveProfiles As Integer = 0

Dim profiles As ProfileInfoCollection = \_

GetProfileInfo(authenticationOption, Nothing, userInactiveSinceDate, \_

0, 0, inactiveProfiles)

Return inactiveProfiles

End Function

'

' CheckParameters

' Verifies input parameters for page size and page index.

' Called by GetAllProfiles, GetAllInactiveProfiles,

' FindProfilesByUserName, and FindInactiveProfilesByUserName.

'

Private Sub CheckParameters(pageIndex As Integer, pageSize As Integer)

If pageIndex < 0 Then \_

Throw New ArgumentException("Page index must 0 or greater.")

If pageSize < 1 Then \_

Throw New ArgumentException("Page size must be greater than 0.")

End Sub

'

' GetProfileInfo

' Retrieves a count of profiles and creates a

' ProfileInfoCollection from the profile data in the database.

' Called by GetAllProfiles, GetAllInactiveProfiles,

' FindProfilesByUserName, FindInactiveProfilesByUserName,

' and GetNumberOfInactiveProfiles.

' Specifying a pageIndex of 0 retrieves a count of the results only.

'

Private Function GetProfileInfo( \_

authenticationOption As ProfileAuthenticationOption, \_

usernameToMatch As String, \_

userInactiveSinceDate As Object, \_

pageIndex As Integer, \_

pageSize As Integer, \_

ByRef totalRecords As Integer) As ProfileInfoCollection

Dim conn As OdbcConnection = New OdbcConnection(connectionString)

' Command to retrieve the total count.

Dim cmd As OdbcCommand = New OdbcCommand("SELECT COUNT(\*) FROM Profiles WHERE ApplicationName = ? ", conn)

cmd.Parameters.Add("@ApplicationName", OdbcType.VarChar, 255).Value = ApplicationName

' Command to retrieve the profile data.

Dim cmd2 As OdbcCommand = New OdbcCommand("SELECT Username, LastActivityDate, LastUpdatedDate, " & \_

"IsAnonymous FROM Profiles WHERE ApplicationName = ? ", conn)

cmd2.Parameters.Add("@ApplicationName", OdbcType.VarChar, 255).Value = ApplicationName

' If searching for a user name to match,

' add the command text and parameters.

If Not usernameToMatch Is Nothing Then

cmd.CommandText &= " AND Username LIKE ? "

cmd.Parameters.Add("@Username", OdbcType.VarChar, 255).Value = usernameToMatch

cmd2.CommandText &= " AND Username LIKE ? "

cmd2.Parameters.Add("@Username", OdbcType.VarChar, 255).Value = usernameToMatch

End If

' If searching for inactive profiles,

' add the command text and parameters.

If Not userInactiveSinceDate Is Nothing Then

cmd.CommandText &= " AND LastActivityDate <= ? "

cmd.Parameters.Add("@LastActivityDate", OdbcType.DateTime).Value = \_

CType(userInactiveSinceDate, DateTime)

cmd2.CommandText &= " AND LastActivityDate <= ? "

cmd2.Parameters.Add("@LastActivityDate", OdbcType.DateTime).Value = \_

CType(userInactiveSinceDate, DateTime)

End If

' If searching for a anonymous or authenticated profiles, add the command text

' and parameters.

Select Case authenticationOption

Case ProfileAuthenticationOption.Anonymous

cmd.CommandText &= " AND IsAnonymous = ?"

cmd.Parameters.Add("@IsAnonymous", OdbcType.Bit).Value = True

cmd2.CommandText &= " AND IsAnonymous = ?"

cmd2.Parameters.Add("@IsAnonymous", OdbcType.Bit).Value = True

Case ProfileAuthenticationOption.Authenticated

cmd.CommandText &= " AND IsAnonymous = ?"

cmd.Parameters.Add("@IsAnonymous", OdbcType.Bit).Value = False

cmd2.CommandText &= " AND IsAnonymous = ?"

cmd2.Parameters.Add("@IsAnonymous", OdbcType.Bit).Value = False

End Select

' Get the data.

Dim reader As OdbcDataReader = Nothing

Dim profiles As ProfileInfoCollection = New ProfileInfoCollection()

Try

conn.Open()

' Get the profile count.

totalRecords = CType(cmd.ExecuteScalar(), Integer)

' No profiles found.

If totalRecords <= 0 Then Return profiles

' Count profiles only.

If pageSize = 0 Then Return profiles

reader = cmd2.ExecuteReader()

Dim counter As Integer = 0

Dim startIndex As Integer = pageSize \* (pageIndex - 1)

Dim endIndex As Integer = startIndex + pageSize - 1

Do While reader.Read()

If counter >= startIndex Then

Dim p As ProfileInfo = GetProfileInfoFromReader(reader)

profiles.Add(p)

End If

If counter >= endIndex Then cmd.Cancel()

counter += 1

Loop

Catch e As OdbcException

If WriteExceptionsToEventLog Then

WriteToEventLog(e, "GetProfileInfo")

Throw New ProviderException(exceptionMessage)

Else

Throw e

End If

Finally

If Not reader Is Nothing Then reader.Close()

conn.Close()

End Try

Return profiles

End Function

'

' GetProfileInfoFromReader

' Takes the current row from the OdbcDataReader

' and populates a ProfileInfo object from the values.

'

Private Function GetProfileInfoFromReader(reader As OdbcDataReader) As ProfileInfo

Dim username As String = reader.GetString(0)

Dim lastActivityDate As DateTime = New DateTime()

If Not reader.GetValue(1) Is DBNull.Value Then \_

lastActivityDate = reader.GetDateTime(1)

Dim lastUpdatedDate As DateTime = New DateTime()

If Not reader.GetValue(2) Is DBNull.Value Then \_

lastUpdatedDate = reader.GetDateTime(2)

Dim isAnonymous As Boolean = reader.GetBoolean(3)

' ProfileInfo.Size not currently implemented.

Dim p As ProfileInfo = New ProfileInfo(username, \_

isAnonymous,lastActivityDate,lastUpdatedDate, 0)

Return p

End Function

'

' WriteToEventLog

' A helper function that writes exception detail to the

' event log. Exceptions are written to the event log as

' a security measure to prevent Private database details

' from being returned to the browser. If a method does not

' return a status or Boolean value indicating whether the action

' succeeded or failed, the caller also throws a generic exception.

'

Private Sub WriteToEventLog(e As Exception, action As String)

Dim log As EventLog = New EventLog()

log.Source = eventSource

log.Log = eventLog

Dim message As String = "An exception occurred communicating with the data source." & vbCrLf & vbCrLf

message &= "Action: " & action & vbCrLf & vbCrLf

message &= "Exception: " & e.ToString()

log.WriteEnTry(message)

End Sub

End Class

End Namespace

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl10_ctl00_ctl01_code');" \o "Copy Code)

using System.Web.Profile;

using System.Configuration.Provider;

using System.Collections.Specialized;

using System;

using System.Data;

using System.Data.Odbc;

using System.Configuration;

using System.Diagnostics;

using System.Web;

using System.Collections;

/\*

This provider works with the following schema for the table of user data.

CREATE TABLE Profiles

(

UniqueID AutoIncrement NOT NULL PRIMARY KEY,

Username Text (255) NOT NULL,

ApplicationName Text (255) NOT NULL,

IsAnonymous YesNo,

LastActivityDate DateTime,

LastUpdatedDate DateTime,

CONSTRAINT PKProfiles UNIQUE (Username, ApplicationName)

)

CREATE TABLE StockSymbols

(

UniqueID Integer,

StockSymbol Text (10),

CONSTRAINT FKProfiles1 FOREIGN KEY (UniqueID)

REFERENCES Profiles

)

CREATE TABLE ProfileData

(

UniqueID Integer,

ZipCode Text (10),

CONSTRAINT FKProfiles2 FOREIGN KEY (UniqueID)

REFERENCES Profiles

)

\*/

namespace Samples.AspNet.Profile

{

public sealed class OdbcProfileProvider: ProfileProvider

{

//

// Global connection string, generic exception message, event log info.

//

private string eventSource = "OdbcProfileProvider";

private string eventLog = "Application";

private string exceptionMessage = "An exception occurred. Please check the event log.";

private string connectionString;

//

// If false, exceptions are thrown to the caller. If true,

// exceptions are written to the event log.

//

private bool pWriteExceptionsToEventLog;

public bool WriteExceptionsToEventLog

{

get { return pWriteExceptionsToEventLog; }

set { pWriteExceptionsToEventLog = value; }

}

//

// System.Configuration.Provider.ProviderBase.Initialize Method

//

public override void Initialize(string name, NameValueCollection config)

{

//

// Initialize values from web.config.

//

if (config == null)

throw new ArgumentNullException("config");

if (name == null || name.Length == 0)

name = "OdbcProfileProvider";

if (String.IsNullOrEmpty(config["description"]))

{

config.Remove("description");

config.Add("description", "Sample ODBC Profile provider");

}

// Initialize the abstract base class.

base.Initialize(name, config);

if (config["applicationName"] == null || config["applicationName"].Trim() == "")

{

pApplicationName = System.Web.Hosting.HostingEnvironment.ApplicationVirtualPath;

}

else

{

pApplicationName = config["applicationName"];

}

//

// Initialize connection string.

//

ConnectionStringSettings pConnectionStringSettings = ConfigurationManager.

ConnectionStrings[config["connectionStringName"]];

if (pConnectionStringSettings == null ||

pConnectionStringSettings.ConnectionString.Trim() == "")

{

throw new ProviderException("Connection string cannot be blank.");

}

connectionString = pConnectionStringSettings.ConnectionString;

}

//

// System.Configuration.SettingsProvider.ApplicationName

//

private string pApplicationName;

public override string ApplicationName

{

get { return pApplicationName; }

set { pApplicationName = value; }

}

//

// System.Configuration.SettingsProvider methods.

//

//

// SettingsProvider.GetPropertyValues

//

public override SettingsPropertyValueCollection

GetPropertyValues(SettingsContext context,

SettingsPropertyCollection ppc)

{

string username = (string)context["UserName"];

bool isAuthenticated = (bool)context["IsAuthenticated"];

// The serializeAs attribute is ignored in this provider implementation.

SettingsPropertyValueCollection svc =

new SettingsPropertyValueCollection();

foreach (SettingsProperty prop in ppc)

{

SettingsPropertyValue pv = new SettingsPropertyValue(prop);

switch (prop.Name)

{

case "StockSymbols":

pv.PropertyValue = GetStockSymbols(username, isAuthenticated);

break;

case "ZipCode":

pv.PropertyValue = GetZipCode(username, isAuthenticated);

break;

default:

throw new ProviderException("Unsupported property.");

}

svc.Add(pv);

}

UpdateActivityDates(username, isAuthenticated, true);

return svc;

}

//

// SettingsProvider.SetPropertyValues

//

public override void SetPropertyValues(SettingsContext context,

SettingsPropertyValueCollection ppvc)

{

// The serializeAs attribute is ignored in this provider implementation.

string username = (string)context["UserName"];

bool isAuthenticated = (bool)context["IsAuthenticated"];

int uniqueID = GetUniqueID(username, isAuthenticated, false);

if (uniqueID == 0)

uniqueID = CreateProfileForUser(username, isAuthenticated);

foreach (SettingsPropertyValue pv in ppvc)

{

switch (pv.Property.Name)

{

case "StockSymbols":

SetStockSymbols(uniqueID, (ArrayList)pv.PropertyValue);

break;

case "ZipCode":

SetZipCode(uniqueID, (string)pv.PropertyValue);

break;

default:

throw new ProviderException("Unsupported property.");

}

}

UpdateActivityDates(username, isAuthenticated, false);

}

//

// UpdateActivityDates

// Updates the LastActivityDate and LastUpdatedDate values

// when profile properties are accessed by the

// GetPropertyValues and SetPropertyValues methods.

// Passing true as the activityOnly parameter will update

// only the LastActivityDate.

//

private void UpdateActivityDates(string username, bool isAuthenticated, bool activityOnly)

{

DateTime activityDate = DateTime.Now;

OdbcConnection conn = new OdbcConnection(connectionString);

OdbcCommand cmd = new OdbcCommand();

cmd.Connection = conn;

if (activityOnly)

{

cmd.CommandText = "UPDATE Profiles Set LastActivityDate = ? " +

"WHERE Username = ? AND ApplicationName = ? AND IsAnonymous = ?";

cmd.Parameters.Add("@LastActivityDate", OdbcType.DateTime).Value = activityDate;

cmd.Parameters.Add("@Username", OdbcType.VarChar, 255).Value = username;

cmd.Parameters.Add("@ApplicationName", OdbcType.VarChar, 255).Value = ApplicationName;

cmd.Parameters.Add("@IsAnonymous", OdbcType.Bit).Value = !isAuthenticated;

}

else

{

cmd.CommandText = "UPDATE Profiles Set LastActivityDate = ?, LastUpdatedDate = ? " +

"WHERE Username = ? AND ApplicationName = ? AND IsAnonymous = ?";

cmd.Parameters.Add("@LastActivityDate", OdbcType.DateTime).Value = activityDate;

cmd.Parameters.Add("@LastUpdatedDate", OdbcType.DateTime).Value = activityDate;

cmd.Parameters.Add("@Username", OdbcType.VarChar, 255).Value = username;

cmd.Parameters.Add("@ApplicationName", OdbcType.VarChar, 255).Value = ApplicationName;

cmd.Parameters.Add("@IsAnonymous", OdbcType.Bit).Value = !isAuthenticated;

}

try

{

conn.Open();

cmd.ExecuteNonQuery();

}

catch (OdbcException e)

{

if (WriteExceptionsToEventLog)

{

WriteToEventLog(e, "UpdateActivityDates");

throw new ProviderException(exceptionMessage);

}

else

{

throw e;

}

}

finally

{

conn.Close();

}

}

//

// GetStockSymbols

// Retrieves stock symbols from the database during the call to GetPropertyValues.

//

private ArrayList GetStockSymbols(string username, bool isAuthenticated)

{

OdbcConnection conn = new OdbcConnection(connectionString);

OdbcCommand cmd = new

OdbcCommand("SELECT StockSymbol FROM Profiles " +

"INNER JOIN StockSymbols ON Profiles.UniqueID = StockSymbols.UniqueID " +

"WHERE Username = ? AND ApplicationName = ? And IsAnonymous = ?", conn);

cmd.Parameters.Add("@Username", OdbcType.VarChar, 255).Value = username;

cmd.Parameters.Add("@ApplicationName", OdbcType.VarChar, 255).Value = ApplicationName;

cmd.Parameters.Add("@IsAnonymous", OdbcType.Bit).Value = !isAuthenticated;

ArrayList outList = new ArrayList();

OdbcDataReader reader = null;

try

{

conn.Open();

reader = cmd.ExecuteReader();

while (reader.Read())

{

outList.Add(reader.GetString(0));

}

}

catch (OdbcException e)

{

if (WriteExceptionsToEventLog)

{

WriteToEventLog(e, "GetStockSymbols");

throw new ProviderException(exceptionMessage);

}

else

{

throw e;

}

}

finally

{

if (reader != null) { reader.Close(); }

conn.Close();

}

return outList;

}

//

// SetStockSymbols

// Inserts stock symbol values into the database during

// the call to SetPropertyValues.

//

private void SetStockSymbols(int uniqueID, ArrayList stocks)

{

OdbcConnection conn = new OdbcConnection(connectionString);

OdbcCommand cmd = new OdbcCommand("DELETE FROM StockSymbols WHERE UniqueID = ?", conn);

cmd.Parameters.Add("@UniqueID", OdbcType.Int).Value = uniqueID;

OdbcCommand cmd2 = new OdbcCommand("INSERT INTO StockSymbols (UniqueID, StockSymbol) " +

"Values(?, ?)", conn);

cmd2.Parameters.Add("@UniqueID", OdbcType.Int).Value = uniqueID;

cmd2.Parameters.Add("@StockSymbol", OdbcType.VarChar, 10);

OdbcTransaction tran = null;

try

{

conn.Open();

tran = conn.BeginTransaction();

cmd.Transaction = tran;

cmd2.Transaction = tran;

// Delete any existing values;

cmd.ExecuteNonQuery();

foreach (object o in stocks)

{

cmd2.Parameters["@StockSymbol"].Value = o.ToString();

cmd2.ExecuteNonQuery();

}

tran.Commit();

}

catch (OdbcException e)

{

try

{

tran.Rollback();

}

catch

{

}

if (WriteExceptionsToEventLog)

{

WriteToEventLog(e, "SetStockSymbols");

throw new ProviderException(exceptionMessage);

}

else

{

throw e;

}

}

finally

{

conn.Close();

}

}

//

// GetZipCode

// Retrieves ZipCode value from the database during

// the call to GetPropertyValues.

//

private string GetZipCode(string username, bool isAuthenticated)

{

string zipCode = "";

OdbcConnection conn = new OdbcConnection(connectionString);

OdbcCommand cmd = new OdbcCommand("SELECT ZipCode FROM Profiles " +

"INNER JOIN ProfileData ON Profiles.UniqueID = ProfileData.UniqueID " +

"WHERE Username = ? AND ApplicationName = ? And IsAnonymous = ?", conn);

cmd.Parameters.Add("@Username", OdbcType.VarChar, 255).Value = username;

cmd.Parameters.Add("@ApplicationName", OdbcType.VarChar, 255).Value = ApplicationName;

cmd.Parameters.Add("@IsAnonymous", OdbcType.Bit).Value = !isAuthenticated;

try

{

conn.Open();

zipCode = (string)cmd.ExecuteScalar();

}

catch (OdbcException e)

{

if (WriteExceptionsToEventLog)

{

WriteToEventLog(e, "GetZipCode");

throw new ProviderException(exceptionMessage);

}

else

{

throw e;

}

}

finally

{

conn.Close();

}

return zipCode;

}

//

// SetZipCode

// Inserts the zip code value into the database during

// the call to SetPropertyValues.

//

private void SetZipCode(int uniqueID, string zipCode)

{

if (zipCode == null) { zipCode = String.Empty; }

OdbcConnection conn = new OdbcConnection(connectionString);

OdbcCommand cmd = new OdbcCommand("DELETE FROM ProfileData WHERE UniqueID = ?", conn);

cmd.Parameters.Add("@UniqueID", OdbcType.Int).Value = uniqueID;

OdbcCommand cmd2 = new OdbcCommand("INSERT INTO ProfileData (UniqueID, ZipCode) " +

"Values(?, ?)", conn);

cmd2.Parameters.Add("@UniqueID", OdbcType.Int).Value = uniqueID;

cmd2.Parameters.Add("@ZipCode", OdbcType.VarChar, 10).Value = zipCode;

OdbcTransaction tran = null;

try

{

conn.Open();

tran = conn.BeginTransaction();

cmd.Transaction = tran;

cmd2.Transaction = tran;

// Delete any existing values.

cmd.ExecuteNonQuery();

cmd2.ExecuteNonQuery();

tran.Commit();

}

catch (OdbcException e)

{

try

{

tran.Rollback();

}

catch

{

}

if (WriteExceptionsToEventLog)

{

WriteToEventLog(e, "SetZipCode");

throw new ProviderException(exceptionMessage);

}

else

{

throw e;

}

}

finally

{

conn.Close();

}

}

//

// GetUniqueID

// Retrieves the uniqueID from the database for the current user and application.

//

private int GetUniqueID(string username, bool isAuthenticated, bool ignoreAuthenticationType)

{

OdbcConnection conn = new OdbcConnection(connectionString);

OdbcCommand cmd = new OdbcCommand("SELECT UniqueID FROM Profiles " +

"WHERE Username = ? AND ApplicationName = ?", conn);

cmd.Parameters.Add("@Username", OdbcType.VarChar, 255).Value = username;

cmd.Parameters.Add("@ApplicationName", OdbcType.VarChar, 255).Value = ApplicationName;

if (!ignoreAuthenticationType)

{

cmd.CommandText += " AND IsAnonymous = ?";

cmd.Parameters.Add("@IsAnonymous", OdbcType.Bit).Value = !isAuthenticated;

}

int uniqueID = 0;

OdbcDataReader reader = null;

try

{

conn.Open();

reader = cmd.ExecuteReader(CommandBehavior.SingleRow);

if (reader.HasRows)

uniqueID = reader.GetInt32(0);

}

catch (OdbcException e)

{

if (WriteExceptionsToEventLog)

{

WriteToEventLog(e, "GetUniqueID");

throw new ProviderException(exceptionMessage);

}

else

{

throw e;

}

}

finally

{

if (reader != null) { reader.Close(); }

conn.Close();

}

return uniqueID;

}

//

// CreateProfileForUser

// If no user currently exists in the database,

// a user record is created during

// the call to the GetUniqueID private method.

//

private int CreateProfileForUser(string username, bool isAuthenticated)

{

// Check for valid user name.

if (username == null)

throw new ArgumentNullException("User name cannot be null.");

if (username.Length > 255)

throw new ArgumentException("User name exceeds 255 characters.");

if (username.Contains(","))

throw new ArgumentException("User name cannot contain a comma (,).");

OdbcConnection conn = new OdbcConnection(connectionString);

OdbcCommand cmd = new OdbcCommand("INSERT INTO Profiles (Username, " +

"ApplicationName, LastActivityDate, LastUpdatedDate, " +

"IsAnonymous) Values(?, ?, ?, ?, ?)", conn);

cmd.Parameters.Add("@Username", OdbcType.VarChar, 255).Value = username;

cmd.Parameters.Add("@ApplicationName", OdbcType.VarChar, 255).Value = ApplicationName;

cmd.Parameters.Add("@LastActivityDate", OdbcType.DateTime).Value = DateTime.Now;

cmd.Parameters.Add("@LastUpdatedDate", OdbcType.VarChar).Value = DateTime.Now;

cmd.Parameters.Add("@IsAnonymous", OdbcType.Bit).Value = !isAuthenticated;

OdbcCommand cmd2 = new OdbcCommand("SELECT @@IDENTITY", conn);

int uniqueID = 0;

try

{

conn.Open();

cmd.ExecuteNonQuery();

uniqueID = (int)cmd2.ExecuteScalar();

}

catch (OdbcException e)

{

if (WriteExceptionsToEventLog)

{

WriteToEventLog(e, "CreateProfileForUser");

throw new ProviderException(exceptionMessage);

}

else

{

throw e;

}

}

finally

{

conn.Close();

}

return uniqueID;

}

//

// ProfileProvider.DeleteProfiles(ProfileInfoCollection)

//

public override int DeleteProfiles(ProfileInfoCollection profiles)

{

int deleteCount = 0;

OdbcConnection conn = new OdbcConnection(connectionString);

OdbcTransaction tran = null;

try

{

conn.Open();

tran = conn.BeginTransaction();

foreach (ProfileInfo p in profiles)

{

if (DeleteProfile(p.UserName, conn, tran))

deleteCount++;

}

tran.Commit();

}

catch (Exception e)

{

try

{

tran.Rollback();

}

catch

{

}

if (WriteExceptionsToEventLog)

{

WriteToEventLog(e, "DeleteProfiles(ProfileInfoCollection)");

throw new ProviderException(exceptionMessage);

}

else

{

throw e;

}

}

finally

{

conn.Close();

}

return deleteCount;

}

//

// ProfileProvider.DeleteProfiles(string[])

//

public override int DeleteProfiles(string[] usernames)

{

int deleteCount = 0;

OdbcConnection conn = new OdbcConnection(connectionString);

OdbcTransaction tran = null;

try

{

conn.Open();

tran = conn.BeginTransaction();

foreach (string user in usernames)

{

if (DeleteProfile(user, conn, tran))

deleteCount++;

}

tran.Commit();

}

catch (Exception e)

{

try

{

tran.Rollback();

}

catch

{

}

if (WriteExceptionsToEventLog)

{

WriteToEventLog(e, "DeleteProfiles(String())");

throw new ProviderException(exceptionMessage);

}

else

{

throw e;

}

}

finally

{

conn.Close();

}

return deleteCount;

}

//

// ProfileProvider.DeleteInactiveProfiles

//

public override int DeleteInactiveProfiles(

ProfileAuthenticationOption authenticationOption,

DateTime userInactiveSinceDate)

{

OdbcConnection conn = new OdbcConnection(connectionString);

OdbcCommand cmd = new OdbcCommand("SELECT Username FROM Profiles " +

"WHERE ApplicationName = ? AND " +

" LastActivityDate <= ?", conn);

cmd.Parameters.Add("@ApplicationName", OdbcType.VarChar, 255).Value = ApplicationName;

cmd.Parameters.Add("@LastActivityDate", OdbcType.DateTime).Value = userInactiveSinceDate;

switch (authenticationOption)

{

case ProfileAuthenticationOption.Anonymous:

cmd.CommandText += " AND IsAnonymous = ?";

cmd.Parameters.Add("@IsAnonymous", OdbcType.Bit).Value = true;

break;

case ProfileAuthenticationOption.Authenticated:

cmd.CommandText += " AND IsAnonymous = ?";

cmd.Parameters.Add("@IsAnonymous", OdbcType.Bit).Value = false;

break;

default:

break;

}

OdbcDataReader reader = null;

string usernames = "";

try

{

conn.Open();

reader = cmd.ExecuteReader();

while (reader.Read())

{

usernames += reader.GetString(0) + ",";

}

}

catch (OdbcException e)

{

if (WriteExceptionsToEventLog)

{

WriteToEventLog(e, "DeleteInactiveProfiles");

throw new ProviderException(exceptionMessage);

}

else

{

throw e;

}

}

finally

{

if (reader != null) { reader.Close(); }

conn.Close();

}

if (usernames.Length > 0)

{

// Remove trailing comma.

usernames = usernames.Substring(0, usernames.Length - 1);

}

// Delete profiles.

return DeleteProfiles(usernames.Split(','));

}

//

// DeleteProfile

// Deletes profile data from the database for the

// specified user name.

//

private bool DeleteProfile(string username, OdbcConnection conn, OdbcTransaction tran)

{

// Check for valid user name.

if (username == null)

throw new ArgumentNullException("User name cannot be null.");

if (username.Length > 255)

throw new ArgumentException("User name exceeds 255 characters.");

if (username.Contains(","))

throw new ArgumentException("User name cannot contain a comma (,).");

int uniqueID = GetUniqueID(username, false, true);

OdbcCommand cmd1 = new OdbcCommand("DELETE \* FROM ProfileData WHERE UniqueID = ?", conn);

cmd1.Parameters.Add("@UniqueID", OdbcType.Int).Value = uniqueID;

OdbcCommand cmd2 = new OdbcCommand("DELETE \* FROM StockSymbols WHERE UniqueID = ?", conn);

cmd2.Parameters.Add("@UniqueID", OdbcType.Int).Value = uniqueID;

OdbcCommand cmd3 = new OdbcCommand("DELETE \* FROM Profiles WHERE UniqueID = ?", conn);

cmd3.Parameters.Add("@UniqueID", OdbcType.Int).Value = uniqueID;

cmd1.Transaction = tran;

cmd2.Transaction = tran;

cmd3.Transaction = tran;

int numDeleted = 0;

// Exceptions will be caught by the calling method.

numDeleted += cmd1.ExecuteNonQuery();

numDeleted += cmd2.ExecuteNonQuery();

numDeleted += cmd3.ExecuteNonQuery();

if (numDeleted == 0)

return false;

else

return true;

}

//

// ProfileProvider.FindProfilesByUserName

//

public override ProfileInfoCollection FindProfilesByUserName(

ProfileAuthenticationOption authenticationOption,

string usernameToMatch,

int pageIndex,

int pageSize,

out int totalRecords)

{

CheckParameters(pageIndex, pageSize);

return GetProfileInfo(authenticationOption, usernameToMatch,

null, pageIndex, pageSize, out totalRecords);

}

//

// ProfileProvider.FindInactiveProfilesByUserName

//

public override ProfileInfoCollection FindInactiveProfilesByUserName(

ProfileAuthenticationOption authenticationOption,

string usernameToMatch,

DateTime userInactiveSinceDate,

int pageIndex,

int pageSize,

out int totalRecords)

{

CheckParameters(pageIndex, pageSize);

return GetProfileInfo(authenticationOption, usernameToMatch, userInactiveSinceDate,

pageIndex, pageSize, out totalRecords);

}

//

// ProfileProvider.GetAllProfiles

//

public override ProfileInfoCollection GetAllProfiles(

ProfileAuthenticationOption authenticationOption,

int pageIndex,

int pageSize,

out int totalRecords)

{

CheckParameters(pageIndex, pageSize);

return GetProfileInfo(authenticationOption, null, null,

pageIndex, pageSize, out totalRecords);

}

//

// ProfileProvider.GetAllInactiveProfiles

//

public override ProfileInfoCollection GetAllInactiveProfiles(

ProfileAuthenticationOption authenticationOption,

DateTime userInactiveSinceDate,

int pageIndex,

int pageSize,

out int totalRecords)

{

CheckParameters(pageIndex, pageSize);

return GetProfileInfo(authenticationOption, null, userInactiveSinceDate,

pageIndex, pageSize, out totalRecords);

}

//

// ProfileProvider.GetNumberOfInactiveProfiles

//

public override int GetNumberOfInactiveProfiles(

ProfileAuthenticationOption authenticationOption,

DateTime userInactiveSinceDate)

{

int inactiveProfiles = 0;

ProfileInfoCollection profiles =

GetProfileInfo(authenticationOption, null, userInactiveSinceDate,

0, 0, out inactiveProfiles);

return inactiveProfiles;

}

//

// CheckParameters

// Verifies input parameters for page size and page index.

// Called by GetAllProfiles, GetAllInactiveProfiles,

// FindProfilesByUserName, and FindInactiveProfilesByUserName.

//

private void CheckParameters(int pageIndex, int pageSize)

{

if (pageIndex < 0)

throw new ArgumentException("Page index must 0 or greater.");

if (pageSize < 1)

throw new ArgumentException("Page size must be greater than 0.");

}

//

// GetProfileInfo

// Retrieves a count of profiles and creates a

// ProfileInfoCollection from the profile data in the

// database. Called by GetAllProfiles, GetAllInactiveProfiles,

// FindProfilesByUserName, FindInactiveProfilesByUserName,

// and GetNumberOfInactiveProfiles.

// Specifying a pageIndex of 0 retrieves a count of the results only.

//

private ProfileInfoCollection GetProfileInfo(

ProfileAuthenticationOption authenticationOption,

string usernameToMatch,

object userInactiveSinceDate,

int pageIndex,

int pageSize,

out int totalRecords)

{

OdbcConnection conn = new OdbcConnection(connectionString);

// Command to retrieve the total count.

OdbcCommand cmd = new OdbcCommand("SELECT COUNT(\*) FROM Profiles WHERE ApplicationName = ? ", conn);

cmd.Parameters.Add("@ApplicationName", OdbcType.VarChar, 255).Value = ApplicationName;

// Command to retrieve the profile data.

OdbcCommand cmd2 = new OdbcCommand("SELECT Username, LastActivityDate, LastUpdatedDate, " +

"IsAnonymous FROM Profiles WHERE ApplicationName = ? ", conn);

cmd2.Parameters.Add("@ApplicationName", OdbcType.VarChar, 255).Value = ApplicationName;

// If searching for a user name to match, add the command text and parameters.

if (usernameToMatch != null)

{

cmd.CommandText += " AND Username LIKE ? ";

cmd.Parameters.Add("@Username", OdbcType.VarChar, 255).Value = usernameToMatch;

cmd2.CommandText += " AND Username LIKE ? ";

cmd2.Parameters.Add("@Username", OdbcType.VarChar, 255).Value = usernameToMatch;

}

// If searching for inactive profiles,

// add the command text and parameters.

if (userInactiveSinceDate != null)

{

cmd.CommandText += " AND LastActivityDate <= ? ";

cmd.Parameters.Add("@LastActivityDate", OdbcType.DateTime).Value = (DateTime)userInactiveSinceDate;

cmd2.CommandText += " AND LastActivityDate <= ? ";

cmd2.Parameters.Add("@LastActivityDate", OdbcType.DateTime).Value = (DateTime)userInactiveSinceDate;

}

// If searching for a anonymous or authenticated profiles,

// add the command text and parameters.

switch (authenticationOption)

{

case ProfileAuthenticationOption.Anonymous:

cmd.CommandText += " AND IsAnonymous = ?";

cmd.Parameters.Add("@IsAnonymous", OdbcType.Bit).Value = true;

cmd2.CommandText += " AND IsAnonymous = ?";

cmd2.Parameters.Add("@IsAnonymous", OdbcType.Bit).Value = true;

break;

case ProfileAuthenticationOption.Authenticated:

cmd.CommandText += " AND IsAnonymous = ?";

cmd.Parameters.Add("@IsAnonymous", OdbcType.Bit).Value = false;

cmd2.CommandText += " AND IsAnonymous = ?";

cmd2.Parameters.Add("@IsAnonymous", OdbcType.Bit).Value = false;

break;

default:

break;

}

// Get the data.

OdbcDataReader reader = null;

ProfileInfoCollection profiles = new ProfileInfoCollection();

try

{

conn.Open();

// Get the profile count.

totalRecords = (int)cmd.ExecuteScalar();

// No profiles found.

if (totalRecords <= 0) { return profiles; }

// Count profiles only.

if (pageSize == 0) { return profiles; }

reader = cmd2.ExecuteReader();

int counter = 0;

int startIndex = pageSize \* (pageIndex - 1);

int endIndex = startIndex + pageSize - 1;

while (reader.Read())

{

if (counter >= startIndex)

{

ProfileInfo p = GetProfileInfoFromReader(reader);

profiles.Add(p);

}

if (counter >= endIndex)

{

cmd.Cancel();

break;

}

counter++;

}

}

catch (OdbcException e)

{

if (WriteExceptionsToEventLog)

{

WriteToEventLog(e, "GetProfileInfo");

throw new ProviderException(exceptionMessage);

}

else

{

throw e;

}

}

finally

{

if (reader != null) { reader.Close(); }

conn.Close();

}

return profiles;

}

//

// GetProfileInfoFromReader

// Takes the current row from the OdbcDataReader

// and populates a ProfileInfo object from the values.

//

private ProfileInfo GetProfileInfoFromReader(OdbcDataReader reader)

{

string username = reader.GetString(0);

DateTime lastActivityDate = new DateTime();

if (reader.GetValue(1) != DBNull.Value)

lastActivityDate = reader.GetDateTime(1);

DateTime lastUpdatedDate = new DateTime();

if (reader.GetValue(2) != DBNull.Value)

lastUpdatedDate = reader.GetDateTime(2);

bool isAnonymous = reader.GetBoolean(3);

// ProfileInfo.Size not currently implemented.

ProfileInfo p = new ProfileInfo(username,

isAnonymous, lastActivityDate, lastUpdatedDate,0);

return p;

}

//

// WriteToEventLog

// A helper function that writes exception detail to the event

// log. Exceptions are written to the event log as a security

// measure to prevent private database details from being

// returned to the browser. If a method does not return a

// status or Boolean value indicating whether the action succeeded

// or failed, the caller also throws a generic exception.

//

private void WriteToEventLog(Exception e, string action)

{

EventLog log = new EventLog();

log.Source = eventSource;

log.Log = eventLog;

string message = "An exception occurred while communicating with the data source.\n\n";

message += "Action: " + action + "\n\n";

message += "Exception: " + e.ToString();

log.WriteEntry(message);

}

}

}