Lecture 3

Profiles to personalize a web site

A profile feature allows you to store some useful information about a user such as the user's mailing address or the preferences that a user has for working with your web application. You can also use it as an easy way to store and retrieve session data like a shopping cart. The profile feature is designed:

1. To work with authenticated users (by default)
2. To work with anonymous users

An introduction to profiles

The ***profile feature*** is a part of the ***personalization feature*** that includes the forms-based authentication. The use of profiles has a several advantages, along with a few potential pitfalls.

The profile feature lets you use the *Profile Common* class to store and retrieve data that's associated with a user, which can be referred to as a profile. This works similarly to storing user data in session state, but the profile feature has several advantages over the *HttpSessionState* class.

Advantages of using the profile feature

Persistent data storage
Unlike session state, which only stores data for the current session, the profile future stores data in a persistent data store. As a result, you can retrieve the data in later sessions.

Strongly typed data access
Unlike session state, which required that you use keys and casting to retrieve data, the profile feature uses properties that have data types. This allows you to use the *IntelliSense* feature to display the properties that are stored in the profile, and it allows you to retrieve data without casting.

Efficient data retrieval
Unlike session state, which retrieves data each time a page is posted, the profile feature only retrieves data when necessary.

Disadvantages of using the profile feature

Difficulty accessing legacy data
By default, the profile feature creates and lets you work with the data in a SQL Server Express database. However, if you have data that's stored in an existing database and want to use the profile feature to work with that data, you need to write a custom profile provider and configure your web application to use that profile provider.

Data that's stale and redundant
Since the profile feature stores data persistently about each user, it is easy for your database to become full of stale and redundant data. This is especially true if you store data about anonymous users. If this begins to slow the performance of you site, you'll need to develop a strategy for managing this data.

Overview of how profiles work

To start, you define the properties of the *Profile Common* class in the *web.config* file for the application. Then, when you run the application, ASP.NET generates the *ProfileCommon* class based on the properties you defined. It also creates an instance of that class that you can refer to using the *Profile* property of the *HttpContext* object.

A *web.config* file that configures a profile

```
<system.web>
  <anonymousIdentification enabled="true" />

  <profile>
   <properties>
    <add name="FirstName" />

    <add name="LastActivityDate" type="System.DateTime" />

    <add name="Cart" type="Cart" serializedAs="Binary" allowAnonymous="true" />
   </properties>
  </profile>
</system.web>
```

C# code that retrieves profile data

```
protected void btnAdd_Click(object sender, EventArgs e)
{
        Profile.FirstName = txtFirstName.Text;
        Profile.LastActivitydate = DateTime.Now;
        Profile.Cart = Cart;
}
```

C# code that retrives profile data

```
ShoppingCart cart;

protected void Page_Last(object sender, EventArgs e)
{
        If (!IsPostBack)
        {
                txtFirstName.Text = Profile.FirstName;
                txtLastActivityDate.Text = Profile.LastActivityDate.ToShortDateString();

                cart = Profile.Cart;
        }
}
```

By default, the data that's stored by the profile feature can only be read by authenticated user. As a result, the user must log in or submit a cookie with a valid authentication ticket before retrieving or storing profile data. However, it's possible to use the profile feature with another new ASP.NET 2.0 feature known as ***anonymous identification*** that allows anonimous users to use the profile feature. Anonomous identification lets ASP.NET identify users by storing a unique identifier for each user, usually as a cookie within the user's browser.

In example, presented above, the *web.config* file allows anomimous identification by setting the *Enable* attribute of the *anonimousIdentification* attribute to *true*. Then, to allow the *Cart* property to use anonomous identification, the *AllowAnonomous* attribute of the *Cart* property is set to true. As a result, users can persistently store items in their cart even if they don't log in.

Once you've defined the profile properties in the *web.config* file, you can use the *Profile* property of the *HttpContext* object to get and set these properties. Since the properties are strongly typed, you don't need to use casting when you retrieve data from the profile. However, if you want to display non-string data in a text box, you need to use a *ToString* method to convert the data type to a string.

The default profile

The *machine.config* file sets the defaults for all web applications on the current machine. Below you can see the elements that set the default profile provider.

The elements of the *machine.config* file that are used by the profile feature

The default connection string for SQL Server Express

```
<connectionString>
    <add name="LocalSqlserver"
        connectionString="data source=.\SQLEXPRESS;
                          Integrated Security=SSPI;
                          AttachDBFilename=|DataDirectory|aspnetdb.mdf;
                          User Instance=true"
        providerName="System.Data.SqlClient" />
</connectionString>
```

The default profile provider

```
<system.web>

    <profile>
        <providers>
            <add name="AspNetSqlProfileProvider"
                connectionStringName="LocalSqlserver"
                applicationName="/"
                type="System.Web.Profile.SqlProfileProvider,
                    System.web, Version=2.0.0.0 Culture=neutral,
                    PublicKeyToken=b03f5f7f11d50a3a" />
        </providers>
    </profile>
```

</system.web>

The default connection string is named *LocalSqlServer* and it connects to a SQL Server Express database named *AspNetDb.mdf* that's stored in the *App_Data* directory of the application.This is the same connection string and database that's used for the membership and role providers.

Then, example above shows the code that configures the profile feature so it uses a profile provider named *AspNetSqlProfileProvider*. This provider uses the *LocalSqlserver* connection string to connect to the *AspNetDb.mdf* databaseusing the *SqlProfileProvider* class that's stored in the *System.web.Profile* namespace.

If you're developing a web application from the scratch, you may want to use this default provoder as is. It works well for prototyping and may be acceptable for some small to medium-sized web applications. Then, since the database is automatically created and configured, you can use the *ProfileCommon* class to store and retrieve data without having to write any data access code.

However, if you need to work with an existing data store, or if you need to gain control over low-level fata access details, the default profile provider won't work for your application. If, for example, you need to access data that's stored in an existing SQL Server database that uses a different database schema, the default provider won't work. Or, if you need to access data that's stored in an Oracle database, the default provider won't work.

In that case, one option is to avoid the profile feature altogether. After all, if you have an existing database, you may already have existing classes that read and write profile data to and from the database. If not, you can write this data access code yourself. Typically, that means storing temporary profile data in the session state object. Although this alternative may not be as slick as using the profile feature, it will give you more control over when and how profile data is stored.

Another option is to implement a custom profile provider. To do that, you need to write a class that implement all of the abstract methods of the abstract *ProfileProvider* class. Then, you need to modify the *web.config* file for your application so it uses this custom profile provider.

Profiles with authenticated users

By default, the profile feature only works, for authenticated users. We assume that the user has been authenticated before accessing the specified web pages.

How to define profile properties in the *web.config* file

```
<system.web>
        <profile>
                <properties>
                        <add name="FirstName"/>
                        <add name="LastName"/>
                </properties>
        </profile>
</system.web>
```

C# code that retrieves profile data

```
Protected void Page_Load(object sender, EventArgs e)
{
        if (!IsPostBack)
        {
                txtFirstName.Text = Profile.FirstName;
                txtLastName.Text = Profile.FirstName;
        }
}
```

C# code that stores profile data

```
protected void btnSave_Click(object sender, EventArgs e)
{
        Profile.FirstName = txtFirstName.Text;
        Profile.LastName = txtLastName.Text;
}
```

Since the *Add* elements don't specify a data type, the *String* data type is used by default.

Specifying data types

How to specify data type for profile properties in the *web.config* file

```
<!—System.string is the default type -->
<add name="FirstName" />
<add name="BirthDate" type="System.DateTime" />
<add name="AccessCount" type="System.Int32" />
```

C# code that retrieves typed profile data

```
txtFirstName.Text = Profile.FirstName;
txtBirthDate.Text = Prifile.BirthDate.ToShortString();
lblCount.Text = Profile.AccessCount.ToString();
```

C# code that stores typed profile data

```
Profile.FirstName = txtFirstName.Text;
Profile.BirthDate = Convert.ToDateTime(txtBirthDate.Text);
Profile.AccessCount++;          // updtes and stores the access count
```

In addition, if you use a custom data type and that data type is stored within the application's *App_Code* folder, you only need to include the namespace if the custom data type is stored in a namespace. For example, let's assume that you store a class named *Customer* within the *App_Code*

folder. Let's also assume that this class isn't coded within a namespace. In that case, you can specify the *Customer* type like this:

```
<add name="Customer" type="Customer">
```

If this class is coded within the *Business* namespace, you must specify the *Customer* type like this:

```
<add name="Customer" type="Business.Customer">
```

Grouping profile properties

To group profile properties you code a *Group* element within the *Profile* and *Properties* elements. Within this *Group* element, you must code a *Name* attribute that specifies the name of the group. See example:

```
<group name="Preferences">
        <add name="SendPromotions" type="System.Boolean" />
        <add name="SendNewProducts" type="System.Boolean" />
</group>
```

C# code that retrieves grouped profile properties

```
chkPromotions.Checked = Profile.Preferences.SendPromotions;
chkNewProducts.Checked = Profile.Preferences.SendNewProducts;
```

C# code that stores grouped profile properties

```
Profile.Preferences.SendPromotions = chkPromotions.Checked;
Profile.Preferences.SendNewProducts = chkNewProducts.Checked;
```

You can't nest a second *Group* element within the first *Group* element.

How to specify default value and read-only properties

This example shows how to use the *DefaultValue* attribute of the *Add* element to specify a default value for a profile property:

```
<add name="SendPromotions" type="System.Boolean" DefaultValue="true" />
```

By default, you can read and write all profile properties. However, if you want to make a profile property read-only, you can set the *ReadOnly* attribute of the *Add* element to true:

```
<add name="DisplayName" readOnly="true" />
```

How to use custom types

Example below shows how to use a custom data type named *Customer*:

```
[Serializable]
public class Customer
{
   public string Email;
   public string LastName;
   public string FirstName;
   public string Address;
   public string City;
   public string State;
   public string ZipCode;
   public string Phone;
}
```

The *Serializable* attribute at the start of the class lets the .NET Framework know that it's OK to convert this class to the specified format. In the example, shown above, each of the eight public fields of the *Customer* class store string data. However, you can use any of the data types for a class like this. If, for example, you want to store a *DateTime* data type, you can do that. Or, if you want to store a custom data type named *Address* to store street address information, you can that as long as you mark that data type as serializable too. To specify the format you want to use to store a custom data type, you use the *SerializeAs* attribute of the *Add* element for a profile property. The defaul value of this attribute is *String*, which stores all the public fields and properties of a data type as plain text. We recommend that you mark all your custom data types as serializable in case you decide to save the data in a different format later on.

If a custom data type contains a reference to another object, or if you need to save the private fields of the data type, you'll need to set the *SerializAs* attribute to binary. Although neither of these conditions apply to the *Customer* data type, we used binary serialization here for illustrative purposes. Because of that, a *Customer* object will be converted to binary data before it's saved to a column in the database. Conversaly, the binary data will be converted to a *Customer* object when the column is retrieved from tht database.

Specifying a custom data type for a profile property

```
<add name="Customer" type="Customer" serializeAs="Binary" />
```

Possible values for the *SerializeAs* attribute

Value	Description
String	Converts the object to plain text and stores it within the specified data store for the application. This is default setting.
Binary	Converts the object to binary data and stores it within the specified data store for the application. For this to work, the .NET Framework must be able to serialize the object.
XML	Converts the object to XML and stores it in the specified data store for the application.
ProviderSpecific	Allows a custom profile provider to decide how to store the object

C# code that retrieves typed data

```
Customer c = Profile.Customer;
```

C# code that stores profile data

```
Customer c = New Customer();
c.Email = txtEmail.Text;
c.LastName = txtLastName.Text;
c.FirstName = txtFirstName.Text;
Profile.Customer = c;
```

Using profiles with anonymous users

By default, the profile feature works with authenticated users. But sometimes, you may want to use profiles with anonymous users. For example, you probably don't want to force a user to log in before he/she can add items to a shopping cart. Fortunately, the new anonymous identification feature that came with ASP.NET lets you use profiles with anonymous.

Before you can identify anonymous users, you need to enable the anonymous identification feature as shown below:

```
<system.web>
    <anonymousIdentification enabled="true" />

    <profile>
        <properties>
            <add name="Customer" type="Customer" serializeAs="Binary" />
            <add name="Cart" type="ShoppingCart"
                serializeAs="Binary" allowAnonymous="True" />
        </properties>
```

```
    </profile>
</system.web>
```

Once you do that, a cookie will be stored for each user that visits your web site. This cookie will be named *.ASPXANONYMOUS*, and it will be stored on the user's machine for 100,000 minutes, which is almost 70 days. That way, each time an anonymous user visits your site, the user's browser passes a cookie to the site that contains an identifier that uniquely identifies the user.

Most of the time, that's all you need to do to enable anonymous identification. However, if you want to change the name of the cookie, you can use the *CookieName* attribute. Or, if you want to shorten or length the amount of time that the cookie is stored on the user's system, you can use the *CookieTimeout* attribute. Or, if you need to make sure that this works even if the user's browser doesn't support cookies (which is rare), you can use the *Cookieless* attribute. In that case, you typically set the *Cookieless* attribute to *AutoDetect* so cookies can be used if they're available.

Attributes of the *anonymousIdentification* element

Attribute	Description
Enabled	Enables or disables anonymous identification. The default value is *False*.
CookieName	Sets the name of the cookie that's stored for each anonymous user. The default value is *.ASPXANONYMOUS.*
CookieTimeout	Sets the length of time that the cookie will be stored on the user's machine in minutes. The default value setting is 100,000 minutes, which is almost 70 days.
Cookieless	Specifies how to identify anonymous users. You can use one of the four values shown below.

Possible values for the *Cookieless* attribute

Attribute	Description
UseCookies	Uses cookies to store the anonymous identifier. This is the default setting.
UseUrl	Stores the anonymous identifier in the URL
AutoDetect	Allows ASP.NET to detect whether the browser can support cookies. If so, it uses cookies to store the anonymous identifier. If no, it uses the URL.
UseDeviceProfile	Configures the anonymous identifier for the device or browser.

How to migrate data from anonymous to authenticate users

Once you begin storing personalized data for anonymous users, you need to develop a strategy for what to do if the anonymous user creates an account and authenticates. Let's assume an anonymous user has added items to his or her cart and that this cart has been saved to the data store. Then, the anonymous user decided to purchase the items in the cart. At this point, the user must get authenticated by creating an account or by logging in, and you must decide what to do with the data that's stored in the anonymous user's cart.

You can use the *Profile_MigrateAnonymous* event handler to handle this situation. To start, you can create a *global.asax* file for your application if one doesn't exist. Then, you can code the

Profile_MigrateAnonymous event handler in this file. Because *Profile_MigrateAnonymous* is an application-wide event, it will be called whenever a user authenticates. In fact, unless you remove the cookie that contains the anonymous identifier as shown in the second example below, this method may be called twice each time the user authenticates

Code that copies the data for an anonymous user to an authenticated user

```
void Profile_MigrateAnonymous(object sender, ProfileMigrateEventArgs e)
{
        ProfileCommon p = Profile.GetProfile(e.AnonymousID);
        Profile.Cart = p.Cart;
}
```

Code that merges data for an anonymous user with data for an authenticated user

```
void Profile_MigrateAnonymous(object sender, ProfileMigrateEventArgs e)
{
        // get the anonymous profile
        ProfileCommon p = Profile.GetProfile(e.AnonymousID);

        if  (p.Cart.Count > 0 && Profile.Cart.Count == 0)
        {
                // swap carts
                Profile.Cart = p.Cart;
        }
        else if (Profile.Cart.Count > 0 && p.Cart.Count > 0)
        {
                // put all anonymous items in the authenticated cart
                CartItem[] items = p.Cart.GetItems();

                foreach  (CartItem item in items)
                {
                        Profile.Cart.addItem(item);
                }
        }

        // delete the anonymous profile data from the data store
        ProfileManager.DeleteProfile(e.AnonymousID);

        // clear the cookie that identifies the anonymous user
        AnonymousIdentificationModule.ClearAnonymousIdentifier();
}
```

Although this situation works, it doesn't provide for an anonymous user that already has stored items in the cart for his authenticated account. In that case, you may not want to replace the authenticated *Cart* property with the anonymous *Cart* property. Instead, you may want to merge the items in the two carts

as shown in the second example. In that case item is in both carts, the *AddItem* method of the *ShoppingCart* class increases the quantity for the item appropriately,

After the items in the two carts have been merged, this code uses a method of the *ProfileManager* class to delete the anonymous profile from the data store. Then, it uses a method of the *AnonymousIdenticationModule* class to clear the cookie that stores the anonymous identifier from the client's browser. This reduce the size of the data store, it deletes all items from the anonymous cart, and it prevents the *Profile_MigrateAnonymous* event handler from being run twice each time a user authenticates.