Chapter 7—Security And User Management

# Security And User Management

Most business applications include user management. Such applications must therefore handle security and user settings. In ASP.NET, you’ll find a comprehensive collection of features, services, classes and controls to manage users, as well as their roles, settings, and access conditions.

In this chapter, you’ll learn about:

\* The components forming security and user management

\* Extending the underlying providers, especially the Membership and Role providers

\* Customizing and extending the profile provider

All of these capabilities are based on the provider model introduced in chapter 4. (Of course, that is the resource for more about the basic design of providers and how to extend and configure them.)

## Built-In Capabilities

Conceptually, ASP.NET employs the principle of “gate keepers”. A gate keeper is a module that sits on top of the pipeline observing every incoming request. Like any other module, its implementation is based on the IHttpModule interface. There are usually several such modules in a row, each of them handling a specific kind of access security or authorization. As we’ve learned in previous chapters, the provider and module models are highly extensible. This gives you the opportunity to add new modules specific to the needs of your application and your desired security levels.

One of the basic concepts of web security is the statelessness of HTTP. Because the protocol essentially uses a “fire and forget” approach, the security aspects of every request must be checked. Since subsequent actions might require authorization or authentication, the security modules are positioned first in the line invoked by pipeline events.

Insert 19835f0701.tif

Figure 7-1. The chain of authentication and authorization providers

The pipeline fires events in a specific order. Firstly, the user is authenticated via the AuthenticateRequest event. After establishing the user’s identity, authorization occurs, and the AuthorizeRequest event allows the user access to certain resources. The order in which these events are fired is crucial as the session state becomes available only after both events are handled (see Figure 7-2). This means that no session-related data can be stored to support the authentication and authorization modules.

Insert 19835f0702.tif

Figure 7-2. Application events involved in the authentication and authorization cycle

After successful user identification, their credentials are stored as an object containing their information. These credentials consist of a username/password pair and, optionally, additional data such as roles or lifetime. Different authorization techniques, such as Windows authentication or Basic authentication, create different sets of credentials. Information which must be available during the whole processing cycle is bound to the current context, namely the HttpContext object. The user credentials are stored in the User property, and can be accessed in this manner:

System.Web.Security.IPrincipal user = Context.User;

Context is provided by the Page class. If you’re not using the Page class, HttpContext.Current.User retrieves the same object.

The IPrincipal interface is a simple definition for a credential store. It contains the user identity (via an object that implements IIdentity) and the IsInRole method, to provide basic authorization support. However, this isn’t sufficient for typical applications. Several different implementations of these interfaces are available that fill in the gaps. IIdentity provides additional support properties, while AuthenticationType indicates which method was used to authenticate a user—such as “Forms”, “NTLM”, or “Custom”. IsAuthenticated reveals that one module of the chain of authentication modules (illustrated in Figure 7-1) has identified the user. Each module can check whether the previous module has identified the user and, if so, create the appropriate identity object and skip the module’s own authentication process. Finally, the username is stored in the Name property of the IIdentity object as a unique key. There are several built-in implementations of the IIdentity interface:

\* System.Web.Security.FormsIdentity

\* System.Security.Principal.WindowsIdentity

\* System.Security.Principal.GenericIdentity

The namespaces reveal that the identity model is not limited to Web applications. Instead, the identity model is embedded within the framework’s security components, and as with most things in .NET, it is extensible.

### Authentication Modules

Several authentication modules are available in ASP.NET. Before building your own module, consider modifying an existing one. Identifying a module close to your requirements and customizing it is always a simpler solution than writing a new module from scratch. ASP.NET provides three core modules:

\* FormsAuthenticationModule

\* WindowsAuthenticationModule

\* PassportAuthenticationModule

The FormsAuthenticationModule provides a simple form-based technique which is highly compatible with most existing clients, but it is not the most secure. Unless the connection between the client and server uses SSL (secure socket layer), the username and password travel unencrypted at least once through the internet. For most applications, this isn’t acceptable. The WindowsAuthenticationModule, on the other hand, requires specific support from the client. Although most browsers handle it well, keep in mind that some devices (such as mobile gadgets) could fail. The PassportAuthenticationModule is listed simply for completeness—it isn’t a viable alternative. Microsoft’s Passport concept was discontinued and has now been replaced by Live ID, which has not yet attracted wide-spread client support.

#### Set up the Authentication Module

To set up an authentication module, you’ll need to make an entry in the *web.config* file. The element <authentication> is responsible for defining and configuring the appropriate module. As a module is responsible for handling all incoming requests, you can’t define the element in subfolders, and overriding the root settings would create a conflict within the pipeline. (However, this will be possible for the authorization method discussed later in the chapter.)

The mode attribute has three commonly used values:

<authentication mode="Forms" />

<authentication mode="Windows" />

<authentication mode="None" />

As there is no “Custom” value, you’ll have to utilize one of the existing techniques when you add your own membership system. (The “Custom” value isn’t provided because you’d need to support existing clients, and writing custom clients isn’t feasible.) Web applications that deal with private data are usually protected by a form of transport layer security which allows the usage of forms authentication for publicly accessible sites and Windows authentication for intranet applications. For the configuration of forms authentication, the child element <form> provides the appropriate settings.

### Authorization Modules

After authenticating the user, you need to determine what he or she is permitted to request. This is all about accessing resources. As there are two kinds of resources on a web server, files and URLs, there are two built-in authorization modules:

\* FileAuthorization

\* UrlAuthorization

If you have already set up the membership system and not defined the authorization explicitly, you might wonder how it works as expected. This is because ASP.NET uses the FileAuthorization module if Windows authentication is set. No additional configuration step is required. Thus, if Window authentication is used, the NTFS file access security is employed, based on Windows ACLs (access control lists).

#### Using Impersonation

In the context of Windows authentication, impersonation occurs when the identity of the logged-in user is changed and the new identity’s credentials are used to access resources. This is a common technique, but it’s dangerous. If the user base is large, and the roles and access policies grow, management becomes arduous. Furthermore, authentication mistakes create security holes, as parts of the application could run under an account other than that intended by the developer.

#### Set up Authorization

The URL authorization controls access to directories and files. It can be configured by setting the appropriate child elements of the current <authorization> element. Restrictions may be set based on both user and role. File authorization does not require any settings here.

### The User Management Interfaces

Included with ASP.NET are a number of tools to support user management, the details of which are beyond the scope of this book. However, I’ll give a general overview of the extensibility interfaces. There are three parts responsible for handling user management:

\* Membership Service

\* Role Service

\* Profile Service

“Membership” is about user identification and management. With the appropriate classes and controls, you can create, edit, and delete user accounts. This includes login and logout facilities based on controls, as well as several features which enhance user experience (such as password recovery or ability to view the current name and login status). ASP.NET includes several built-in management tools to assist administrators with handling users, based on a SQL Server database store.

“Role” performs authorization tasks. Roles provide access to resources and can be assigned to many users. If the user becomes a member of a role, he or she gains access to the resources the role is permitted to retrieve. The built-in management tools support the role model.

“Profile” complements the user management by adding a user-specific data store. This allows you to maintain per-user settings between visits. Profiles are not intended to support session-related processes. Instead, they allow a user to customize a site specifically for their own preferences—think of a “MySite” type of functionality. ASP.NET includes many features, services and object types to support profiles.

#### Extensibility Issues

The extensibility model is based on providers. The three services mentioned above use providers to save and retrieve their data. By default, data is stored in a SQL Server database. It’s often necessary to use alternative or existing storage, Active Directory, LDAP, or a custom solution. Several built-in providers are included with each service in order to support different stores.

Insert 19835f0700.tif

Figure 7-3. The class hierarchy for built-in Membership and Role providers

The default providers use a SQL Server database. These are the SqlMembershipProvider and the SqlRoleProvider. ASP.NET also supports Active Directory (AD) out-of-the-box, via the ActiveDirectoryMembershipProvider.

The following table shows all the basic members of the membership provider classes.

Table 7-1. Members of the MembershipProvider classes\*

|  |
| --- |
| Name Description |
| Methods |
| ChangePassword Changes the password. |
| ChangePasswordQuestionAndAnswer Changes the password question and answer. |
| CreateUser Creates a new user. |
| DecryptPassword Decrypts the password. |
| DeleteUser Deletes a user. |
| EncryptPassword Encrypts the password. |
| FindUsersByEmail Finds a user using his or her email address. |
| FindUsersByName Finds a user using his or her name. |
| GetAllUsers Retrieves a list of all users. |
| GetNumberOfUsersOnline Retrieves a list of all currently logged-in users. The behavior depends on the provider and does not necessarily report the exact value. |
| GetPassword Retrieves the password for a given name. |
| GetUser Retrieves a System.Web.Security.MembershipUser object based on the name. |
| GetUserNameByEmail Retrieves the name using an email address. |
| Initialize Initializes the provider. Inherited from ProviderBase. |
| ResetPassword Resets the current password with an automatically generated one. |
| UnlockUser Unlocks a locked user. |
| UpdateUser Updates the user’s information in the data store. |
| ValidateUser Checks whether the user with the provided password is a legitimate user. |
| Events |
| ValidatingPassword An event fired while the provider validates the password. |
| Properties |
| ApplicationName The name of the application. A membership provider can support several applications and supports different data storage for each application. Inherited from ProviderBase. |
| EnablePasswordReset Defines whether or not the user can reset their password. |
| EnablePasswordRetrieval Defines whether or not the user can retrieve their password. |
| MaxInvalidPasswordAttempts Determines the maximum number of false password attempts allowed before the account is locked. |
| MinRequiredNonAlphanumericCharacters Minimum number of non-alphanumeric characters a password must have. |
| MinRequiredPasswordLength Minimum password length. |
| Name Name of the provider. Inherited from ProviderBase. |
| PasswordAttemptWindow Prevents “brute force” attempts to guess a password. If the MaxInvalidPasswordAttempts occur within PasswordAttemptWindow (in minutes), that user’s account is locked. |
| PasswordFormat Format in which the password is saved in the storage. This is a value from the enumeration MembershipPasswordFormat, which provides the values Clear, Hashed, or Encrypted. Hashes are unidirectional encryption methods. The default hash format is SHA1. |
| PasswordStrengthRegularExpression A regular expression that is used to check whether or not the password meets the minimum password strength required by your application. |
| RequiresQuestionAndAnswer Determines whether the provider supports the password retrieval using the question and answer technique. |
| RequiresUniqueEmail Determines whether or not the email address must be unique in the database. |

\* Covers both SqlMembershipProvider and ActiveDirectoryMembershipProvider

For the role provider, things look a little different. The AuthorizationStoreRoleProvider class supports three kinds of role data storage: the Active Directory itself, an Active Directory Application Mode server (ADAM), or an XML file. For these three modes, a connection string defines the data source. The connection string begins with msxml:// for an XML file, or with msldap:// for Active Directory or ADAM . The WindowsTokenRoleProvider uses the local Windows user database to retrieve the role information. Don’t confuse this with Windows groups. Direct access to Windows group information is not supported by this provider. While you can obtain membership information about a group, you cannot write, delete, or change these memberships even if management tools support this. From the perspective of ASP.NET, the WindowsTokenRoleProvider provides read-only access.

None of these classes are sealed. This means that you can inherit from each class and override methods and properties to change behavior as desired.

The following table shows all basic members of the role provider classes:

Table 7-2. Members of the RoleProvider classes

|  |
| --- |
| Name Description Not supported in \* |
| Methods |
| AddUsersToRoles Add users to roles. Defined in RoleProvider. W |
| CreateRole Create a new role. Defined in RoleProvider. W |
| DeleteRole Delete a role. Defined in RoleProvider. W |
| FindUsersInRole Find all users within a role. Defined in RoleProvider. W, A |
| GetAllRoles Get a list of all roles. Defined in RoleProvider. W |
| GetRolesForUser Get all roles for a user. Defined in RoleProvider. |
| GetUsersInRole Get all users in a role. Defined in RoleProvider. W |
| IsUserInRole Indicates whether or not a user is in a role. Defined in RoleProvider. |
| RemoveUsersFromRoles Remove users from roles. Defined in RoleProvider. W |
| RoleExists Indicates whether the role exists. Defined in RoleProvider. |
| Properties |
| ApplicationName Name of the application. A role provider can support several applications and different data storage for each application. Defined in ProviderBase. |
| Description Description used in ProviderBase. |
| ScopeName Scope for the authorization store. W, S |
| CacheRefreshInterval Time (in minutes) the provider caches role information W, S |
| Name Name as used in ProviderBase. |

\* W = WindowsTokenRoleProvider, S = SqlRoleProvider, A = AuthorizationStoreRoleProvider

With this overview of the capabilities of different providers, we can now investigate how to extend them.

## Extending Membership and Role Providers

The functionality of the built-in providers suits most common types of applications. Before constructing your own provider, let’s take a look at when and why it’s more practical to create or extend a built-in one.

### Why Create a Membership Provider?

The purpose of a provider is to manage access to data storage. Extending the provider model is necessary if you want to use a different storage for your user database. The SqlMembershipProvider supports a SQL Server database, while the ActiveDirectoryMembershipProvider supports the Active Directory. Creating a service infrastructure following the principles of service oriented architectures (SOA) requires another method—Web services. One reason for creating a custom membership provider is the ability to use Web services. As shown in Table 7-1, the MembershipProvider base class supports an incredible number of features. Whether you require some or all of these depends on the needs of your application.

In the following example, I’ll demonstrate the basic principles of a custom membership provider in the particular case of authenticating users against a Web service. The service provides a transparent tier against a data store. This means that the service can run anywhere in your organization as a central point of service, and the users of the service—the provider—knows nothing about the data storage behind the service. Security is enhanced by hiding all data storage implementation details behind the publicly visible web service methods. For simplicity and demonstration purposes only, the example service stores user information in a simple XML file. This allows you to observe the file and learn exactly how the provider behaves.

In the following sections, I’ll show the sample code and explain the implementation details.

### Solution Details

Developing a custom provider firstly requires a data source. In this example, I’ll use an XML file as storage and a WCF (Windows Communication Foundation) service to access the file remotely. This demonstrates that it’s possible to transparently access a remote data store using standard framework techniques.

The solution consists of:

\* A WCF service project with two services, one for Membership and the other for Roles

\* A Web project configured to use the custom provider that has:

\* An implementation of a custom Membership Provider

\* An implementation of a custom Role Provider

\* A service reference to use the WCF service

## Developing Membership and Role Providers

Both Membership and Role providers support several features. Whether or not you implement all features, extend additional ones, or simply create a rudimentary service depends on your application and requirements. In this example, I’ll implement all the basic features via a simplified approach for the sake of brevity.

### Create Web Service Driven Membership Provider

The Web service driven membership provider consists of three parts:

1. The user data store—an XML file

2. The service tier—a Web service with several useful methods

3. The Membership Provider implementation

### Create Web Service Driven Role Provider

The Web service driven role provider also consists of three parts:

1. The same XML file as used in the user data store, extended with role properties

2. The service tier—a Web service with several useful methods

3. The Role Provider implementation

For both providers, we’ll need a simple test environment. Consider creating some pages using various Login controls, such as the CreateUserWizard. This functions well but requires additional work. To begin with, I recommend using the built-in ASP.NET configuration application. Launch it locally in Visual Studio from the *Project* and *ASP.NET configuration* menu items. Using the embedded Web server, Visual Studio starts a web application which allows you to manage users and roles with the current provider. Before doing so, you’ll need to configure the providers. However, we’ll begin by developing the entire application.

#### Developing the Service Tier

The service tier uses WCF to create two Web services, one for Membership management and one for Role management. To create the service:

\* Within Visual Studio create a new project via *File > New > Project > WCF > WCF Service Library*

\* Name the project “Membership”

\* Add two services to the project using *Add* from the context menu. Select *New Item* and in the following dialog choose *WCF Service*.

Insert 19835f070.tif

Figure 7-4. Adding a new WCF service to the service library

The wizard creates a service interface (the contract for the service), the class with the implementation and the required entries in *web.config* file to define the service endpoint. The endpoint is a definition used later in the web application where you can access the service. The contract is usually an interface decorated with the [ServiceContract] attribute. Each method that is exposed as a service call is decorated with the [OperationContract] attribute. Both attributes support several settings for modifying behavior and appearance. However, this is beyond the scope of this book, and default settings can be used for this example as well.

Tip: If you want to learn more about professional WCF development, refer to the book “Pro WCF: Practical Microsoft SOA Implementation” from Apress, ISBN 978-1-59059-702-6.

#### Creating the Membership Service

The Membership service consists of the following interfaces and classes:

\* IMembershipService interface

\* MembershipService service

\* MembershipService class

\* UserData class

\* User class

\* FileManager class

All the code to build the service is shown and explained below.

Note: For sake of brevity and space, the following code lacks error handling, “using” statements, and namespace definitions. Please refer to the sample code provided with the book for the full implementation.

Listing 7-1. The IMembershipService interface defines the contract

[ServiceContract]

public interface IMembershipService

{

[OperationContract]

bool ChangePassword(string username, string oldPassword, string newPassword);

[OperationContract]

bool ChangePasswordQuestionAndAnswer(string username, string password, ⮰

string newPasswordQuestion, ⮰

string newPasswordAnswer);

[OperationContract]

User CreateUser(string username, string password, string email, ⮰

string passwordQuestion, string passwordAnswer, ⮰

bool isApproved, object providerUserKey, ⮰

out MembershipCreateStatus status);

[OperationContract]

bool DeleteUser(string username, bool deleteAllRelatedData);

[OperationContract]

List<User> FindUsersByEmail(string emailToMatch, int pageIndex, ⮰

int pageSize, out int totalRecords);

[OperationContract]

List<User> FindUsersByName(string usernameToMatch, int pageIndex, ⮰

int pageSize, out int totalRecords);

[OperationContract]

List<User> GetAllUsers(int pageIndex, int pageSize, out int totalRecords);

[OperationContract]

int GetNumberOfUsersOnline();

[OperationContract]

string GetPassword(string username, string answer);

[OperationContract(Name = "GetUserbyName")]

User GetUser(string username, bool userIsOnline);

[OperationContract]

string GetUserNameByEmail(string email);

[OperationContract]

string ResetPassword(string username, string answer);

[OperationContract]

bool UnlockUser(string userName);

[OperationContract]

void UpdateUser(User user);

[OperationContract]

bool ValidateUser(string username, string password);

}

The interface is not unusual. Each method decorated with the OperationContract attribute is exposed by the service. As shown in the listing for the GetUser method, the method’s name can be modified. This is necessary if you wish to use overloaded methods (methods that have the same name, but a different parameter list signature). The service lacks this capability and requires unique names. The OperationContract attribute’s Name property separates internal logic from the external service façade.

Once the interface is completed, it must be implemented. Begin with a service file that links to a code-behind file.

Listing 7-2. The MembershipService.svc file is required in order to publish the service

<%@ ServiceHost Language="C#" Debug="true" ⮰

Service="Apress.Extensibility.Membership.MembershipService" ⮰

CodeBehind="MembershipService.svc.cs" %>

The logic of the service is coded in the code-behind file, shown in the next listing.

Listing 7-3. The MembershipService class is the implementation

[FileIOPermission(SecurityAction.LinkDemand)]

public class MembershipService : IMembershipService

{

public bool ChangePassword(string username, string oldPassword, ⮰

string newPassword)

{

throw new NotImplementedException();

}

public bool ChangePasswordQuestionAndAnswer(string username, string password, ⮰

string newPasswordQuestion, ⮰

string newPasswordAnswer)

{

throw new NotImplementedException();

}

public User CreateUser(string username, string password, string email, ⮰

string passwordQuestion, string passwordAnswer, ⮰

bool isApproved, object providerUserKey, ⮰

out System.Web.Security.MembershipCreateStatus status)

{

User user = null;

UserData ud = FileManager.Load();

// check users, consider adding more data here

var hasUser = from u in ud.Users where u.UserName.Equals(username) select u;

if (hasUser.Count() > 0)

{

status = MembershipCreateStatus.DuplicateUserName;

return null;

}

var hasEmail = from u in ud.Users where u.Email.Equals(email) select u;

if (hasEmail.Count() > 0)

{

status = MembershipCreateStatus.DuplicateEmail;

return null;

}

try

{

user = new User(

username,

email,

passwordQuestion,

"",

isApproved,

false,

DateTime.Now,

DateTime.MinValue,

DateTime.MinValue,

DateTime.Now,

DateTime.MinValue);

// Store Hash Only

user.Password = FileManager.CalculateSHA1(password);

ud.Users.Add(user);

FileManager.Save(ud);

status = MembershipCreateStatus.Success;

}

catch

{

status = MembershipCreateStatus.ProviderError;

}

return user;

}

public bool DeleteUser(string username, bool deleteAllRelatedData)

{

UserData ud = FileManager.Load();

var user = (from u in ud.Users

where u.UserName.Equals(username)

select u).FirstOrDefault<User>();

if (user != null)

{

ud.Users.Remove(user);

FileManager.Save(ud);

return true;

}

return false;

}

public List<User> FindUsersByEmail(string emailToMatch, int pageIndex, ⮰

int pageSize, out int totalRecords)

{

UserData ud = FileManager.Load();

var users = (from u in ud.Users

where u.Email.Equals(emailToMatch)

select u).ToList<User>();

totalRecords = users.Count();

return GetPaged(users, pageIndex, pageSize);

}

public List<User> FindUsersByName(string usernameToMatch, int pageIndex, ⮰

int pageSize, out int totalRecords)

{

UserData ud = FileManager.Load();

var users = (from u in ud.Users

where u.UserName.Equals(usernameToMatch)

select u).ToList<User>();

totalRecords = users.Count();

return GetPaged(users, pageIndex, pageSize);

}

public List<User> GetAllUsers(int pageIndex, int pageSize, out int totalRecords)

{

UserData ud = FileManager.Load();

totalRecords = ud.Users.Count;

return GetPaged(ud.Users, pageIndex, pageSize);

}

private List<User> GetPaged(List<User> ud, int pageIndex, int pageSize)

{

pageSize = Math.Min(ud.Count, pageSize);

return ud.GetRange(pageIndex \* pageSize, pageSize);

}

public int GetNumberOfUsersOnline()

{

// Users who logged in within the last 15 mins

UserData ud = FileManager.Load();

var users = (from u in ud.Users

where u.LastActivityDate.AddMinutes(15) > DateTime.Now

select u);

return users.Count();

}

public string GetPassword(string username, string answer)

{

UserData ud = FileManager.Load();

var user = (from u in ud.Users

where u.UserName.Equals(username) && ⮰

u.PasswordAnswer.Equals(answer)

select u).First<User>();

if (user != null)

{

return user.Password;

}

return null;

}

public User GetUser(string username, bool userIsOnline)

{

UserData ud = FileManager.Load();

var user = (from u in ud.Users

where u.UserName.Equals(username) &&

(userIsOnline) ?

u.LastActivityDate.AddMinutes(15) > DateTime.Now :

true // all users

select u).FirstOrDefault<User>();

return user;

}

public string GetUserNameByEmail(string email)

{

UserData ud = FileManager.Load();

var user = (from u in ud.Users

where u.Email.Equals(email)

select u).FirstOrDefault<User>();

if (user != null)

{

return user.UserName;

}

return null;

}

public string ResetPassword(string username, string answer)

{

UserData ud = FileManager.Load();

var user = (from u in ud.Users

where u.UserName.Equals(username) && ⮰

u.PasswordAnswer.Equals(answer)

select u).FirstOrDefault<User>();

if (user != null)

{

return user.Password;

}

return null;

}

public bool UnlockUser(string userName)

{

throw new NotImplementedException();

}

public void UpdateUser(User user)

{

UserData ud = FileManager.Load();

var userToUpdate = (from u in ud.Users

where u.UserName.Equals(user.UserName)

select u).FirstOrDefault<User>();

foreach (PropertyInfo pi in user.GetType().GetProperties(⮰

BindingFlags.Public | BindingFlags.Instance))

{

PropertyInfo piTarget = typeof(User).GetProperty(pi.Name);

if (piTarget != null)

{

piTarget.SetValue(userToUpdate, pi.GetValue(user, null), null);

}

}

FileManager.Save(ud);

}

public bool ValidateUser(string username, string password)

{

UserData ud = FileManager.Load();

string hash = FileManager.CalculateSHA1(password);

var user = (from u in ud.Users

where u.UserName.Equals(username) &&

u.Password.Equals(hash)

select u).FirstOrDefault<User>();

return (user != null);

}

}

Most of these methods follow the same code pattern. The current content of the membership data store —an XML file—is loaded by calling FileManager.Load (see Listing 7-6). This ensures that changes made by parallel calls are handled properly. Each method uses a LINQ query to obtain the required data. If the method needs to persist changes, the updated UserData object is saved back to the storage file.

Tip: If you want to learn more about LINQ, I recommend “LINQ for Visual C# 2008”, published by Apress (ISBN 978-1-4302-1580-6).

Passwords require special treatment. They aren’t usually stored in an unencrypted format, so you’ll need to choose to either encrypt the password or store a hash of the password.

Hashes are a one-way encryption technique. To validate a user logging in, you simply hash their supplied password and compared the result with the stored hash value. Storing a hash is good practice, because the hash algorithm was designed to make it almost impossible to derive the original text from the hash value. On the other hand, the hash doesn’t support password retrieval. Instead, the user receives a new password when he or she requests a lost one. This requires more effort, as the provider must create generic passwords, and the application must support password change forms, and so forth. In the example code, the password is stored as a hash, despite there being no password generator. The password is hashed using the SHA1 hash algorithm. This is safe and easy to use. Listing 7-6 shows the FileManager.CalculateSHA1 method. If you plan to implement the provider in an application, consider using an encrypted password that you can decrypt on request.

The code also has a hard coded setting for the calculation of the number of users online (the GetNumberOfUsersOnline method). Based on the time of last activity, all users that have logged in within the last 15 minutes are counted as being online. This is an extremely simplified algorithm, but it demonstrates one possible approach.

One significant method is CreateUser. It checks several conditions in order to avoid duplicate usernames, email addresses, or other unwanted data. The example checks just two of a wide range of possible conditions. First, it tests for duplicate user names. If this occurs, the method sets the property status to MembershipCreateStatus.DuplicateUserName . Second, it checks for duplicate email addresses and sets the status to MembershipCreateStatus.DuplicateEmail, if required. You can refer to the MembershipCreateStatus enumeration to find more return values and conditions to test. If everything checks out, the MembershipCreateStatus.Success value is set. The MembershipCreateStatus.ProviderError condition is used if unexpected errors are caught by the try-catch statement.

The class uses the UserData class heavily. This class is straightforward, containing a serializable collection of User and Role objects:

Listing 7-4. The UserData class is the container for serializing users and roles

public class UserData

{

public UserData()

{

}

[XmlElement]

public List<User> Users

{

get;

set;

}

[XmlElement]

public List<string> Roles

{

get;

set;

}

}

Because the serialization uses the XmlSerializer (shown in Listing 7-6), all elements requiring serialization are tagged with the XmlElement attribute. The User object contains all data pertaining to a specific account. This class is shown next.

Listing 7-5. The User class serializes a single user

[Serializable]

[DataContract]

public class User

{

public User()

{

}

public User(string name, string email, string passwordQuestion, ⮰

string comment, bool isApproved, bool isLockedOut, ⮰

DateTime creationDate, DateTime lastLoginDate, ⮰

DateTime lastActivityDate, DateTime lastPasswordChangedDate, ⮰

DateTime lastLockoutDate)

{

this.UserName = name;

this.Email = email;

this.Comment = comment;

this.IsApproved = isApproved;

this.IsLockedOut = isLockedOut;

this.CreationDate = creationDate;

this.LastLoginDate = lastLoginDate;

}

[DataMember]

[XmlElement]

public string Comment { get; set; }

[DataMember]

[XmlElement]

public DateTime CreationDate { get; set; }

[DataMember]

[XmlElement]

public string Email { get; set; }

[DataMember]

[XmlAttribute]

public bool IsApproved { get; set; }

[DataMember]

[XmlAttribute]

public bool IsLockedOut { get; set; }

[DataMember]

[XmlElement]

public DateTime LastActivityDate { get; set; }

[DataMember]

[XmlElement]

public DateTime LastLockoutDate { get; set; }

[DataMember]

[XmlElement]

public DateTime LastLoginDate { get; set; }

[DataMember]

[XmlElement]

public DateTime LastPasswordChangedDate { get; set; }

[DataMember]

[XmlElement]

public string PasswordQuestion { get; set; }

[DataMember]

[XmlElement]

public string PasswordAnswer { get; set; }

[DataMember]

[XmlElement]

public string UserName { get; set; }

[DataMember]

[XmlElement]

public string Password { get; set; }

[DataMember]

[XmlArray(ElementName="Roles"), XmlArrayItem(ElementName="Role")]

public List<string> Roles

{

get;

set;

}

public bool ChangePassword(string oldPassword, string newPassword)

{

if (Password.Equals(oldPassword))

{

Password = newPassword;

return true;

}

return false;

}

public bool ChangePasswordQuestionAndAnswer(string password, ⮰

string newPasswordQuestion, string newPasswordAnswer)

{

if (Password.Equals(password))

{

PasswordQuestion = newPasswordQuestion;

PasswordAnswer = newPasswordAnswer;

return true;

}

return false;

}

public string GetPassword(string passwordAnswer)

{

return Password;

}

public string ResetPassword()

{

return Password;

}

public string ResetPassword(string passwordAnswer)

{

return Password;

}

public bool UnlockUser()

{

return IsLockedOut = false;

}

}

This class supports two features. It must be transferred via the service and it is part of the data contract. A data contract in WCF defines the structure of complex data. Applying the DataContract attribute to a class makes that class into a data contract. Each serializable member in the data contract class is tagged with the DataMemberAttribute. In our example the same class is used internally to serialize the user objects to the XML file. The XmlElement attributes mark the elements to be serialized.

Each user can be the member of none, one, or many roles. The roles are simple strings stored in another collection. To reference the roles in the user object, a copy of the role’s name is stored. A List<string> object is used internally. However, in XML, the format should be more readable. The XmlArray and XmlArrayItem attributes ensure that the element names are correct. Refer to Listing 7-7 to view the created XML.

Finally, the helper class is shown here to complete the code needed to build the service.

Listing 7-6. A helper class that is used to access the data file

internal static class FileManager

{

private const string DATAPATH = "App\_Data\\UserData.xml";

private static readonly string dataPath;

private static object locker = new object();

static FileManager()

{

Uri codeUri = new Uri(typeof(UserData).Assembly.CodeBase);

dataPath = Path.Combine(Directory.GetParent( ⮰

Path.GetDirectoryName(codeUri.LocalPath)).FullName, DATAPATH);

// check file permissions

FileIOPermission permission = new FileIOPermission( ⮰

FileIOPermissionAccess.AllAccess, dataPath);

permission.Demand();

}

internal static UserData Load()

{

lock (locker)

{

XmlSerializer xs = new XmlSerializer(typeof(UserData));

UserData ud = null;

try

{

using (FileStream fs = new FileStream(dataPath, FileMode.Open))

{

ud = xs.Deserialize(fs) as UserData;

}

}

catch

{

// save rudimentary format

ud = new UserData();

Save(ud);

}

return ud;

}

}

internal static void Save(UserData ud)

{

lock (locker)

{

XmlSerializer xs = new XmlSerializer(typeof(UserData));

using (FileStream fs = new FileStream(dataPath, FileMode.Create))

{

xs.Serialize(fs, ud);

}

}

}

internal static string CalculateSHA1(string text)

{

byte[] buffer = Encoding.ASCII.GetBytes(text);

SHA1CryptoServiceProvider cryptoTransformSHA1 = ⮰

new SHA1CryptoServiceProvider();

string hash = BitConverter.ToString( ⮰

cryptoTransformSHA1.ComputeHash(buffer));

return hash;

}

}

The class is static and provides three methods. With Load, the caller can retrieve the current contents of the stored data. In Save, the data is written back to disk. Both methods block concurrent threads because this is a multi-threaded environment. The thread handling is in neither way optimized but gives you the idea what you should be aware of.

The CalculateSHA1 method creates the SHA1 hash of a given string. It’s used internally to hash the password.

If everything works as intended, the service creates a XML file similar to the one shown in the following code listing.

Listing 7-7. The data file filled with some users and roles

<?xml version="1.0"?>

<UserData xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<Users IsApproved="false" IsLockedOut="false">

<Comment>Test</Comment>

<CreationDate>2009-04-30T13:12:23.6008895+02:00</CreationDate>

<Email>krause@comzept.de</Email>

<LastActivityDate>0001-01-01T01:00:00+01:00</LastActivityDate>

<LastLockoutDate>0001-01-01T01:00:00+01:00</LastLockoutDate>

<LastLoginDate>0001-01-01T01:00:00+01:00</LastLoginDate>

<LastPasswordChangedDate>0001-01-01T01:00:00+01:00</LastPasswordChangedDate>

<UserName>JoergKrause</UserName>

<Roles />

</Users>

<Users IsApproved="true" IsLockedOut="false">

<Comment />

<CreationDate>2009-04-30T19:06:42.3928895+02:00</CreationDate>

<Email>nissan@comzept.de</Email>

<LastActivityDate>0001-01-01T00:00:00</LastActivityDate>

<LastLockoutDate>0001-01-01T00:00:00</LastLockoutDate>

<LastLoginDate>0001-01-01T00:00:00</LastLoginDate>

<LastPasswordChangedDate>0001-01-01T00:00:00</LastPasswordChangedDate>

<UserName>BerndAlbrecht</UserName>

<Password>56-5E-E9-0F-A9-60-2C-0C-16-49-1A-7A-0F-3F-6C-70-D9-17-A3-2B</Password>

<Roles>

<Role>User</Role>

<Role>Contributor</Role>

</Roles>

</Users>

<Users IsApproved="true" IsLockedOut="false">

<Comment />

<CreationDate>2009-04-30T19:39:44.1528895+02:00</CreationDate>

<Email>joerg@krause.net</Email>

<LastActivityDate>0001-01-01T00:00:00</LastActivityDate>

<LastLockoutDate>0001-01-01T00:00:00</LastLockoutDate>

<LastLoginDate>0001-01-01T00:00:00</LastLoginDate>

<LastPasswordChangedDate>0001-01-01T00:00:00</LastPasswordChangedDate>

<UserName>Joerg</UserName>

<Password>4E-B8-C5-DE-4C-76-60-80-C5-91-C6-94-D5-47-5D-B8-E3-53-B0-F3</Password>

<Roles>

<Role>Editor</Role>

</Roles>

</Users>

<Roles>Admin</Roles>

<Roles>User</Roles>

<Roles>Contributor</Roles>

<Roles>Editor</Roles>

</UserData>

The collection of Roles elements at the end contains the list of roles used in the provider. In the Users element, the Roles element contains the assigned roles for that user. The other elements contain the properties. IsApproved and IsLockedOut are stored as attributes because they only contain scalar values. The Password element displays the hash in plain text format.

In this section, we’ve looked at the basic operations of the Membership service and its storage classes. The roles are already in this schema and the User object accepts setting the assigned roles. All available roles are stored in separate Roles items at the end of the file—see the bold elements in Listing 7-7. From where do the roles originate? In the next section, we’ll look at a Role service. It requires less code, as it re-uses portions of the Membership service.

#### Creating the Role Service

The role service is much simpler than the Membership service. Instead of dealing with a “Role” object, we’ll simply store roles as strings. The functions of the role service are to assign users to existing roles, create or remove roles, and search for users that are members of specific roles. The data storage is the same as for the Membership service. Refer to the Membership classes to see how both parts operate together.

The role service requires three components:

\* IRoleService interface

\* RoleService service

\* RoleService class

As before, the interface defines the service contract, while the class implements the service. The service file is the service exposed by WCF as the endpoint, and it refers to the class’s implementation in a code-behind file.

Listing 7-8. The contract defined by an interface

[ServiceContract]

public interface IRoleService

{

[OperationContract]

void AddUsersToRoles(string[] usernames, string[] roleNames);

[OperationContract]

void CreateRole(string roleName);

[OperationContract]

bool DeleteRole(string roleName, bool throwOnPopulatedRole);

[OperationContract]

string[] FindUsersInRole(string roleName, string usernameToMatch);

[OperationContract]

string[] GetAllRoles();

[OperationContract]

string[] GetRolesForUser(string username);

[OperationContract]

string[] GetUsersInRole(string roleName);

[OperationContract]

bool IsUserInRole(string username, string roleName);

[OperationContract]

void RemoveUsersFromRoles(string[] usernames, string[] roleNames);

[OperationContract]

bool RoleExists(string roleName);

}

The service defines a direct mirror of the RoleProvider class. This makes it easier to call the service’s methods from the provider shown later. So far, this is very straightforward.

Listing 7-9. The service class

<%@ ServiceHost Language="C#" Debug="true" ⮰

Service="Apress.Extensibility.Membership.RoleService" ⮰

CodeBehind="RoleService.svc.cs" %>

The service class (.svc file) points to the code-behind file containing the implementation.

Listing 7-10. The implementation of the service

public class RoleService : IRoleService

{

public void AddUsersToRoles(string[] usernames, string[] roleNames)

{

UserData ud = FileManager.Load();

var users = (from u in ud.Users

where usernames.Contains(u.UserName)

select u);

foreach (User user in users)

{

user.Roles.RemoveAll(role => roleNames.Contains(role));

user.Roles.AddRange(roleNames);

}

FileManager.Save(ud);

}

public void CreateRole(string roleName)

{

UserData ud = FileManager.Load();

ud.Roles.Add(roleName);

FileManager.Save(ud);

}

public bool DeleteRole(string roleName, bool throwOnPopulatedRole)

{

UserData ud = FileManager.Load();

var userInRole = from u in ud.Users

where u.Roles.Contains(roleName)

select u;

if (userInRole.Count() > 0 && throwOnPopulatedRole)

{

return false;

}

ud.Roles.Remove(roleName);

FileManager.Save(ud);

return true;

}

public string[] FindUsersInRole(string roleName, string usernameToMatch)

{

UserData ud = FileManager.Load();

var userInRole = from u in ud.Users

where u.UserName.Contains(usernameToMatch) && ⮰

u.Roles.Contains(roleName)

select u.UserName;

return userInRole.ToArray<string>();

}

public string[] GetAllRoles()

{

UserData ud = FileManager.Load();

var roles = from r in ud.Roles select r;

return roles.ToArray<string>();

}

public string[] GetRolesForUser(string username)

{

UserData ud = FileManager.Load();

var roles = (from u in ud.Users

where u.UserName.Equals(username)

select u.Roles).First();

return roles.ToArray();

}

public string[] GetUsersInRole(string roleName)

{

UserData ud = FileManager.Load();

var roles = (from u in ud.Users

where u.Roles.Contains(roleName)

select u.UserName);

return roles.ToArray();

}

public bool IsUserInRole(string username, string roleName)

{

UserData ud = FileManager.Load();

var roles = (from u in ud.Users

where u.UserName.Equals(username) && u.Roles.Contains(roleName)

select u.UserName);

return roles.Count() > 0;

}

public void RemoveUsersFromRoles(string[] usernames, string[] roleNames)

{

UserData ud = FileManager.Load();

var roles = (from u in ud.Users

where usernames.Contains(u.UserName) && ⮰

roleNames.Intersect(u.Roles).Count() > 0

select u.UserName);

FileManager.Save(ud);

}

public bool RoleExists(string roleName)

{

UserData ud = FileManager.Load();

var roles = (from r in ud.Roles

where r.Equals(roleName)

select r);

return roles.Count() > 0;

}

}

Again, LINQ is used to access the data store. As the UserData class already covers the roles by providing the Roles element, the whole RoleService class doesn’t require its own data access. The FileManager.Save method is called for methods that persist roles data.

### Configuring the Services

Once the services have been defined, they must be configured. WCF has its own definition language which extends the *web.config* file. The definition has three parts, but some settings can be omitted if the default values are suitable. WCF is based on the ABC mnemonic, which stands for Address, Binding, and Contract. Contract is defined by adding specific attributes to a class or interface definition, as shown earlier. The binding and address are defined in the configuration. The address defines where a client can reach the service. The binding defines the protocol used between client and service. A complete description consisting of address and binding is also called an endpoint.

A sample XML configuration follows:

Listing 7-11. The configuration of the service

<system.serviceModel>

<services>

<service behaviorConfiguration="MembershipServiceBehavior" ⮰

name="Apress.Extensibility.Membership.MembershipService">

<endpoint address="" binding="wsHttpBinding" ⮰

contract="Apress.Extensibility.Membership.IMembershipService">

<identity>

<dns value="localhost" />

</identity>

</endpoint>

<endpoint address="mex" binding="mexHttpBinding" ⮰

contract="IMetadataExchange" />

<host>

<baseAddresses>

<add baseAddress="http://localhost/service" />

</baseAddresses>

</host>

</service>

<service behaviorConfiguration="RoleServiceBehavior" ⮰

name="Apress.Extensibility.Membership.RoleService">

<endpoint address="" binding="wsHttpBinding" ⮰

contract="Apress.Extensibility.Membership.IRoleService">

<identity>

<dns value="localhost" />

</identity>

</endpoint>

<endpoint address="mex" binding="mexHttpBinding" ⮰

contract="IMetadataExchange" />

</service>

</services>

<behaviors>

<serviceBehaviors>

<behavior name="MembershipServiceBehavior">

<serviceMetadata httpGetEnabled="true" />

<serviceDebug includeExceptionDetailInFaults="true" />

</behavior>

<behavior name="RoleServiceBehavior">

<serviceMetadata httpGetEnabled="true" />

<serviceDebug includeExceptionDetailInFaults="false" />

</behavior>

</serviceBehaviors>

</behaviors>

</system.serviceModel>

The system.serviceModel section contains all the settings required by our WCF services. Its definition consists of two services and two corresponding behaviors. Because we require a regular web service, the wsHttpBinding is used. An additional endpoint is defined for meta data exchange, using the mexHttpBinding. This allows the client to create a proxy which talks to the service. (The client is technically created when you reference the service in Visual Studio.) When this occurs, the meta data endpoint is called to retrieve the service’s description. This includes all methods, signatures, and data objects used by the service, as well as the endpoint configuration.

If you use and debug the whole solution you might encounter an exception. For security reasons, the details of an exception are not transferred to the client, as intruders could try to provoke an error by attacking the service. If the internally thrown exception exposes details, the hackers will receive information with which they can refine their attack. During the development cycle, however, you’ll need as much information as possible in order to troubleshoot errors. Set the includeExceptionDetailInFaults attribute to “true” while debugging and change it to “false” before publishing the service.

Now the service is up and running. You can check this by launching the .svc endpoints in the browser. Just right click the service and choose *View in browser*. You should see a service description similar to that shown in Figure 7-5.

Insert 19835f0704.tif

Figure 7-5. The service is up and running

Use the same procedure to check the RoleService.

### Implementing the Provider

The provider requires a few more steps:

\* Referencing the services

\* Implementing the Membership provider

\* Implementing the Role provider

\* Configuring the Web project so that it accepts the providers

The reference is easy. Simply choose *Add service reference* from the context menu of your project. Then click on *Discover* and *Services in Solution*.

Insert 19835f0706.tif

Figure 7-6. Adding the service references

Open the service in the tree view to the left and choose the contract. Give the service a suitable name. In the example, I’ve used MembershipService and RoleService respectively. Repeat the steps for the other service. You now have two service references in your project.

#### Provider Specific Configuration

The provider can be combined with its own configuration to support specific settings. Refer to chapter 4 for more information about the basic steps of provider implementation. For the sake of brevity, I have removed from the example any code not required for basic tasks.

#### Create the Membership Provider

The Membership provider implements the MembershipProvider base class. To avoid confusion I named the class WSMembershipProvider, where the WS prefix reminds us that it’s based on a Web service.

Listing 7-12. The WSMembershipProvider

public class WSMembershipProvider : MembershipProvider

{

private MembershipServiceClient client;

private bool \_enablePasswordReset;

public override void Initialize(string name, ⮰

System.Collections.Specialized.NameValueCollection config)

{

if (config == null)

throw new ArgumentNullException("config");

if (String.IsNullOrEmpty(name))

{

name = this.GetType().Name;

}

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

{

config.Remove("description");

config.Add("description", "WS Based Membership Provider");

}

base.Initialize(name, config);

client = new MembershipServiceClient();

// optional parameters

if (!String.IsNullOrEmpty(config["EnablePasswordReset"]))

{

\_enablePasswordReset = Boolean.Parse(config["EnablePasswordReset"]);

config.Remove("EnablePasswordReset");

}

// mandatory parameters

ApplicationName = config["ApplicationName"];

}

public override string ApplicationName

{

get;

set;

}

public override bool ChangePassword(string username, string oldPassword, ⮰

string newPassword)

{

return client.ChangePassword(username, oldPassword, newPassword);

}

public override bool ChangePasswordQuestionAndAnswer(string username, ⮰

string password, ⮰

string newPasswordQuestion,

string newPasswordAnswer)

{

return client.ChangePasswordQuestionAndAnswer(username, password, ⮰

newPasswordQuestion, ⮰

newPasswordAnswer);

}

public override MembershipUser CreateUser(string username, ⮰

string password, ⮰

string email, ⮰

string passwordQuestion, ⮰

string passwordAnswer, ⮰

bool isApproved, ⮰

object providerUserKey, ⮰

out MembershipCreateStatus status)

{

User user = client.CreateUser(out status, username, password, email, ⮰

passwordQuestion, passwordAnswer, ⮰

isApproved, providerUserKey);

if (user == null) return null;

MembershipUser mu = new MembershipUser(this.GetType().Name,

user.UserName,

providerUserKey,

user.Email,

user.PasswordQuestion,

"",

user.IsApproved,

user.IsLockedOut,

user.CreationDate,

user.LastLoginDate,

user.LastActivityDate,

user.LastPasswordChangedDate,

user.LastLockoutDate

);

return mu;

}

public override bool DeleteUser(string username, bool deleteAllRelatedData)

{

return client.DeleteUser(username, deleteAllRelatedData);

}

public override bool EnablePasswordReset

{

get { return \_enablePasswordReset; }

}

public override bool EnablePasswordRetrieval

{

get { throw new NotImplementedException(); }

}

public override MembershipUserCollection FindUsersByEmail(string emailToMatch, ⮰

int pageIndex, int pageSize, out int totalRecords)

{

return CopyToMembershipCollection(client.FindUsersByEmail(out totalRecords,⮰

emailToMatch, pageIndex, pageSize));

}

public override MembershipUserCollection FindUsersByName(string usernameToMatch,

int pageIndex, int pageSize, out int totalRecords)

{

return CopyToMembershipCollection(client.FindUsersByName(out totalRecords,

usernameToMatch, pageIndex, pageSize));

}

public override MembershipUserCollection GetAllUsers(int pageIndex, ⮰

int pageSize, out int totalRecords)

{

return CopyToMembershipCollection(client.GetAllUsers(out totalRecords, ⮰

pageIndex, pageSize));

}

private MembershipUserCollection CopyToMembershipCollection(User[] users)

{

MembershipUserCollection muc = new MembershipUserCollection();

foreach (User user in users)

{

muc.Add(CopyToMembershipUser(user));

}

return muc;

}

private MembershipUser CopyToMembershipUser(User user)

{

MembershipUser mu = new MembershipUser(this.GetType().Name,

user.UserName,

"",

user.Email,

user.PasswordQuestion,

user.Comment,

user.IsApproved,

user.IsLockedOut,

user.CreationDate,

user.LastLoginDate,

user.LastActivityDate,

user.LastPasswordChangedDate,

user.LastLockoutDate);

return mu;

}

public override int GetNumberOfUsersOnline()

{

return client.GetNumberOfUsersOnline();

}

public override string GetPassword(string username, string answer)

{

return client.GetPassword(username, answer);

}

public override MembershipUser GetUser(string username, bool userIsOnline)

{

return CopyToMembershipUser(client.GetUserbyName(username, userIsOnline));

}

public override MembershipUser GetUser(object providerUserKey, ⮰

bool userIsOnline)

{

throw new NotImplementedException();

}

public override string GetUserNameByEmail(string email)

{

return client.GetUserNameByEmail(email);

}

public override int MaxInvalidPasswordAttempts

{

get { return 5; }

}

public override int MinRequiredNonAlphanumericCharacters

{

get { return 1; }

}

public override int MinRequiredPasswordLength

{

get { return 6; }

}

public override int PasswordAttemptWindow

{

get { return 10; }

}

public override MembershipPasswordFormat PasswordFormat

{

get { return MembershipPasswordFormat.Clear; }

}

public override string PasswordStrengthRegularExpression

{

get { return ""; }

}

public override bool RequiresQuestionAndAnswer

{

get { return false; }

}

public override bool RequiresUniqueEmail

{

get { return false; }

}

public override string ResetPassword(string username, string answer)

{

return client.ResetPassword(username, answer);

}

public override bool UnlockUser(string userName)

{

throw new NotImplementedException();

}

public override void UpdateUser(MembershipUser user)

{

User u = new User();

foreach (PropertyInfo pi in u.GetType().GetProperties())

{

PropertyInfo piTarget = user.GetType().GetProperty(pi.Name);

if (piTarget != null)

{

pi.SetValue(u, piTarget.GetValue(user, null), null);

}

}

client.UpdateUser(u);

}

public override bool ValidateUser(string username, string password)

{

return client.ValidateUser(username, password);

}

}

The client is instantiated in the Initialize method. The provider’s methods subsequently use the client to call the appropriate functions remotely. The only issue requiring more work is the fact that the MembershipUser class cannot be used directly by the service. The internally employed User class has a similar structure but no dependencies on the underlying provider. In the example, I use reflection to iterate over the properties and, where a property is both public and available in the target class, copy the value from the MembershipUser class to the User class. This is in the UpdateUser method. The provider also has several hard coded settings. In a fully fledged application, these would be obtained from *web.config*.

#### Create the Role Provider

The Role Provider implements the RoleProvider base class. Again, to avoid confusion, our new class is named WSRoleProvider, where the WS reminds us that it’s based on a Web service.

Listing 7-13. The WSRoleProvider

**public class WSRoleProvider : RoleProvider**

{

private RoleServiceClient client;

public override void Initialize(string name, ⮰

System.Collections.Specialized.NameValueCollection config)

{

if (config == null)

throw new ArgumentNullException("config");

if (String.IsNullOrEmpty(name))

{

name = this.GetType().Name;

}

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

{

config.Remove("description");

config.Add("description", "WS Based Membership Provider");

}

base.Initialize(name, config);

client = new RoleServiceClient();

// mandatory parameters

ApplicationName = config["ApplicationName"];

}

public override void AddUsersToRoles(string[] usernames, string[] roleNames)

{

client.AddUsersToRoles(usernames, roleNames);

}

public override string ApplicationName

{

get;

set;

}

public override void CreateRole(string roleName)

{

client.CreateRole(roleName);

}

public override bool DeleteRole(string roleName, bool throwOnPopulatedRole)

{

return client.DeleteRole(roleName, throwOnPopulatedRole);

}

public override string[] FindUsersInRole(string roleName, ⮰

string usernameToMatch)

{

return client.FindUsersInRole(roleName, usernameToMatch);

}

public override string[] GetAllRoles()

{

return client.GetAllRoles();

}

public override string[] GetRolesForUser(string username)

{

return client.GetRolesForUser(username);

}

public override string[] GetUsersInRole(string roleName)

{

return client.GetUsersInRole(roleName);

}

public override bool IsUserInRole(string username, string roleName)

{

return client.IsUserInRole(username, roleName);

}

public override void RemoveUsersFromRoles(string[] usernames, ⮰

string[] roleNames)

{

client.RemoveUsersFromRoles(usernames, roleNames);

}

public override bool RoleExists(string roleName)

{

return client.RoleExists(roleName);

}

}

As you can see in the example code the WSRoleProvider class is simpler than the provider shown before. The methods predominantly call the corresponding service methods and do not add additional logic. Again, the Initialize method contains the instantiation of the RoleServiceClient class.

### Configuring the Provider

After the provider is implemented, it must be configured. As for any other provider, this is achieved within the *web.config* file:

Listing 7-14. Configuring the providers

<system.web>

<membership defaultProvider="WSMembershipProvider">

<providers>

<add name="WSMembershipProvider"

type="Apress.Extensibility.Membership.WSMembershipProvider,

Apress.Extensibility.Membership"

EnablePasswordReset="True"

/>

</providers>

</membership>

<roleManager defaultProvider="WSRoleProvider" enabled="true" >

<providers>

<clear/>

<add name="WSRoleProvider"

type="Apress.Extensibility.Membership.WSRoleProvider,

Apress.Extensibility.Membership" />

</providers>

</roleManager>

The provider’s add element requires the name, the type and optionally the parameters defined in the configuration. The type is the full class name including the assembly name. In this case, I’ve used a regular Visual Studio 2008 project which compiles into a DLL. The name and namespace of the DLL are specified here.

### Testing the Providers

You can now use the providers. However, you’ll need several additional pages in order to add users or roles, retrieve data, check users, assign roles, etc. Fortunately, Visual Studio contains an embedded tool which allows you to manage users and roles.

This tool is available from the *Project* menu and is called *ASP.NET configuration*. You can add and change users, add and assign roles, and test all common settings of the new providers without writing a single line of code. On the start page, choose the Security tab to view the current state of the providers. If you have already defined some users and roles, their quantities are displayed:

Insert 19835f0707.tif

Figure 7-7. The security tabs uses the custom provider if properly configured

You can use the tool to create or modify users and add or assign roles. If the tool does not work as expected, check whether or not the custom providers are properly registered. Open the Provider tab and click on *Select a different provider for each feature (advanced)*. Your custom providers (WSMembershipProvider and WSRoleProvider) should be selected, as shown in the next figure:

Insert 19835f0708.tif

Figure 7-8. The security tabs use the custom provider if properly configured

## Extending Profile Providers

The Profile service is an integrated module for storing and retrieving user settings. Settings are not limited to ASP.NET but are widely used in distributed multi-user environments; even for Win Forms application exist a similar concept. The Profile service employs a Profile provider to read data from, and save data to, a storage device. By default, the data store is a SQL Server or SQL Server Express instance, and the provider class is System.Web.Profile.SqlProfileProvider. This provider is also responsible for allowing users to update their settings. Common management functions, such as deleting a Profile after a period of inactivity, are located here as well.

Note: Profile providers are part of the provider model. It is described in detail in chapter 4.

### The Profile Service

The purpose of the Profile service is to store and retrieve user settings. With these settings, users can personalize their web pages. Typical information stored here includes:

\* User information, such as city and phone number

\* Preferences for accessibility, colors, and font size

\* Data related to the current session, such as the contents of a shopping basket

\* Individual selections of services, such as news feeds or newsletter topics

The personalization possibilities are limitless. Instead of building custom database modules from scratch to store the users’ settings, this is where the Profile service excels.

The services’ definition consists of two parts—Profile properties and related user. The Profile service retrieves properties for the current user from the underlying provider. In order to use Profiles, you must first activate this feature by define the Provider in the *web.config* file. By default, the SqlProfileProvider stores properties in the local SQL Server database. The properties are defined in the *web.config* file, too. Unlike application settings, where key/value pairs are stored in *web.config*, the Profile property definition consists of the property definition only. Actual values are stored in the Profile provider’s data store.

### Understanding the Profile Provider

Before you consider extending the Profile provider, let’s look at the features that the provider supports. The ProfileProvider class implements the ProviderBase base class and the System.Configuration.SettingsProvider. As the namespace implies, the concepts are not limited to ASP.NET. The Profile provider extends the settings concept with some features that allow for the management of user Profiles, such as deleting of Profiles and activity monitoring.

Table 7-1. Methods defined by the ProfileProvider class

|  |  |
| --- | --- |
| Method | Description |
| Initialize | Derived from ProviderBase. Contains the code for setting up the provider and reading the settings. |
| ApplicationName | Derived from SettingsProvider. Name of the current application. The data store can use this to handle multiple applications that use the same database. |
| GetPropertyValues | Derived from SettingsProvider. Retrieves a list of properties using a SettingsContext object containing information about the user. You could use this to retrieve Profile information for a user. The context provides information about authenticated or anonymous users. The method returns an object of type SettingsPropertyCollection., which is a collection of SettingsProperty objects, each containing the name and type of a property, as well as additional information such as default settings and read-only state. Using this method updates the LastActivityDate value that is used to track activity (and thus monitor logged on or inactive users). |
| SetPropertyValues | Derived from SettingsProvider. Uses a SettingsContext object and a SettingsPropertyValueCollection object to write property values back to the data store. Using this method also updates the LastActivityDate. |
| DeleteProfiles | Using an array of usernames, this method deletes the related Profiles. An overloaded version of the same method accepts ProfileInfo objects. |
| DeleteInactiveProfiles | Accepts a ProfileAuthenticationOption value and a DateTime object in order to delete inactive Profiles. The Profiles should be deleted if the supplied date and time value is equal to or less than the Profile’s LastActivityDate. |
| GetAllProfiles | Retrieves all available Profile objects using a ProfileAuthenticationOption value and an integer value for the page and the maximum number of Profiles. The page index allows a paged retrieval. The method returns a ProfileInfoCollection with ProfileInfo objects. |
| GetAllInactiveProfiles | Using the same conditions as GetAllProfiles, this method returns only Profiles that have not been used for a given time. |
| FindProfilesByUserName | Retrieves Profiles via their usernames. In derived classes, this method might accept wildcards, regular expressions, or any other kind of search pattern in order to retrieve multiple values matching the search string. This method behaves similarly to GetAllProfiles. |
| FindInactiveProfilesByUserName | Retrieves inactive Profiles via their usernames. In derived classes, this method might accept wildcards, regular expressions, or any other kind of search pattern in order to retrieve multiple values matching the search string. This method behaves similarly to GetAllProfiles. |
| GetNumberOfInactiveProfiles | Retrieves the number of inactive Profiles. |

When implementing a custom Profile provider, it must support as a minimum the above methods in order to be used transparently in applications.

The Profile data always has a specific scope, which is usually the current user’s username. However, the Profile provider also supports anonymous users. This means that a user not currently logged in gets a Profile as a non-authenticated user.

Consider a visitor who selects several items to purchase at a website, but they have not yet logged in. Their Profile contains several items in their shopping basket. It is therefore important to keep their Profile (containing their selections) separate from the Profiles of other non-authenticated users. To accomplish this, the Profile provider assigns each anonymous user an ID that is used as a key, in the absence of a username.

When the user leaves the session, the Profile becomes obsolete. This is why the last activity date is important. If you specify a minimum inactivity time period, the provider is able to delete expired Profiles. The ProfileAuthenticationOption parameter determines which of anonymous or authenticated Profiles are to be deleted. You can use a different time span for each type.

The data store itself is data agnostic—any properties can be stored there. The definition of acceptable values is set in the *web.config* file for all users.

#### Serializing and Deserializing

Values for custom settings can support several data types. Most storage systems require serialized values instead of .NET objects. Fortunately, the base class provides the code for serializing and deserializing objects. However, when retrieving values, this code is not called automatically. Consider the case where you have several settings, and deserializing them takes some time. If the calling party requires only one or two of these values, deserializing all of them will waste resources. To improve performance, you can call the base methods to deserialize the desired properties and ignore all the others. This method is known as “lazy deserialization”.

Listing 7-1. Typical strategy for retrieving and deserializing values

SettingsPropertyValueCollection settings = new SettingsPropertyValueCollection ();

foreach (SettingsProperty property in properties)

{

SettingsPropertyValue pp = new SettingsPropertyValue (property);

object val = GetPropertyValueFromDataSource (property.Name);

if (val == null)

{

pp.PropertyValue = null;

pp.Deserialized = true;

pp.IsDirty = false;

}

else

{

pp.PropertyValue = Deserialize(val);

pp.Deserialized = true;

pp.IsDirty = false;

}

settings.Add (pp);

}

return settings;

#### Defining Profile settings

A typical setting definition looks like:

Listing 7-2. Define available properties

<profile>

<properties>

<add name="Greeting" type="String" />

<add name="Count" type="Int32" defaultValue="0" />

</properties>

</profile>

The name of the property and the type are required. A default value can also be specified, which the provider will use if the data store does not retrieve one.

The serialization can be refined by adding the serializeAs attribute to the Profile property definition shown in the next section. This attribute allows the pre-selection of a predefined serializer:

\* String

\* Binary

\* Xml

\* ProviderSpecific

The provider can also be selected on a per-property basis. This allows for different data stores for different types of data.

Each property can be limited to authenticated users only by setting the attribute allowAnonymous to false.

For more complex scenarios, you can group settings with the <group> tag. If you plan on supporting hundreds of settings, groups can help you to manage your settings.

#### Using the Profile Data

There are two ways of reaching Profile data from within your code. In a web application project (as opposed to a web site project), the Profile class is not auto-generated from the current configuration. You will need to use the following two techniques to manage your Profile settings. To retrieve values, use the following pattern:

string myValue = (string) Context.Profile.GetPropertyValue("MyValue");

Setting values is similarly straightforward:

if (Context.Profile.UserName != null) {

Context.Profile.SetPropertyValue("MyValue", "something");

Context.Profile.Save();

}

In a web site project, Visual Studio generates a Profile class from the configuration settings, which exposes the Profile properties as typed properties:

string myValue = Profile.MyValue;

Several free tools are available for overcoming the limitation of Web site projects that doesn’t support Profile creation. The Web Profile Builder found at *http://code.msdn.microsoft.com/WebProfileBuilder* is a good starting point. However, digging deeper into the build techniques used in Visual Studio is beyond the scope of this book.

Note: To overcome the limitations of web application projects, in this chapter I use the web site project template—This is the opposite of what I recommend in other parts of the book. As long as you’re not using Profiles, you should stay with web application projects. If you do use Profiles, it depends on whether strongly typed access is important.

### Configuring Custom Profile Providers

The Profile settings use the embedded provider if no other is specified. The settings follow the schema for providers explained in chapter 4. The following code listing shows how to define settings for the example shown below, and how to configure the provider.

Listing 7-3. Define available properties

<profile defaultProvider="MyProfileProvider">

<properties>

<add name="FirstName" type="System.String" />

<add name="LastName" type="System.String" />

<add name="EmailAddress" type="System.String" />

<add name="MakeNamePublic" type="System.Boolean" />

<add name="NewsFeed" type="System.String" />

</properties>

<providers>

<add name="MyProfileProvider"

type="Apress.Extensibility.XmlProfileProvider" />

</providers>

</profile>

To implement the custom Profile provider, select the preferred data access method.

## Implementing a Custom Profile Provider

The custom Profile provider shown in the following example uses an XML file to store all users’ data. It does not implement usage functions such as checking inactivity or deleting Profiles. However, it does implement the save and load functions as well as the underlying serialization and deserialization methods.

A Profile provider requires several steps to run properly. Even if only one provider class is necessary for retrieving the code, you need several users and authentication capabilities in order to check the provider.

### Preparation Steps

To test a custom provider, which requires the ability to login and logout of different user accounts, I set up Forms Authentication in *web.config*. Furthermore, I defined the users’ credentials directly in *web.config* and force the Membership provider to use data from that file instead of from the default SQL Server. This is accomplished by handling the authentication event manually. The *web.config* section defining the users looks like this:

Listing 7-4. Define several users to test the application

<authorization>

<deny users="?"/>

</authorization>

<authentication mode="Forms">

<forms>

<credentials passwordFormat="Clear">

<user name="User1" password="User1"/>

<user name="User2" password="User2"/>

<user name="User3" password="User3"/>

</credentials>

</forms>

</authentication>

Caution: Storing user credentials in clear text format in web.config is not intended to run in a production environment. It is here only in order to rapidly set up a test environment .

Next, the application requires at least one page for changing and using Profile settings, plus a login page. The login page consists of a single Login control and an event handler.

Listing 7-5. The “core” component of the login page

<body>

<form id="form1" runat="server">

<div>

<asp:Login ID="Login1" runat="server" onauthenticate="Login1\_Authenticate">

</asp:Login>

<br />

Not yet registered? Create new account <asp:HyperLink ID="HyperLink1"

runat="server" NavigateUrl="~/CreateUser.aspx">here</asp:HyperLink>.

</div>

</form>

</body>

</html>

Listing 7-6. An event handler forces the login control to use the configured user names

public partial class Login : System.Web.UI.Page

{

protected void Page\_Load(object sender, EventArgs e)

{

}

protected void Login1\_Authenticate(object sender, AuthenticateEventArgs e)

{

e.Authenticated = FormsAuthentication.Authenticate(Login1.UserName, ⮰

Login1.Password);

}

}

The next page is also for testing purposes. It contains a simple form to save new Profile settings, display current settings, and change accounts.

Listing 7-7. ASPX page used to test the Profile provider

<body>

<form id="form1" runat="server">

<br />Set your Profile data,

<asp:LoginName ID="LoginName1" runat="server" />

:<br />

<br />E-Mail:

<asp:TextBox runat="server" ID="txtEmail"></asp:TextBox>

<br />Fore Color:

<asp:DropDownList ID="drpForeColor" runat="server">

<asp:ListItem>Red</asp:ListItem>

<asp:ListItem>Green</asp:ListItem>

<asp:ListItem>Blue</asp:ListItem>

<asp:ListItem></asp:ListItem>

</asp:DropDownList>

<br />Back Color:<asp:DropDownList ID="drpBackColor" runat="server">

<asp:ListItem>White</asp:ListItem>

<asp:ListItem>Beige</asp:ListItem>

<asp:ListItem>Yellow</asp:ListItem>

</asp:DropDownList>

<br />

<br />

<asp:Button ID="btnSend" runat="server" onclick="btnSend\_Click"

Text="Set Profile Data" />

<br />

<br />

Logout to use anonymous mode, login as different user to test different

settings:

<asp:LoginStatus ID="LoginStatus1" runat="server" />

&nbsp;(Predefined: <i>User1</i>, <i>User2</i>, <i>User3</i>, in each case use the

name as password, too)<br />

<br />Result of your current settings:<br />

<div>

<asp:Panel ID="PanelSettings" runat="server">

This panel&#39;s design is read from current user&#39;s Profile.

</asp:Panel>

</div>

</form>

</body>

The page allows you to set an email address and a color. The color type is used here to demonstrate serialization of non-scalar values.

Insert 19835f0709.tif

Figure 7-1. A simple form to set and retrieve Profile data

The page shows the currently logged-in user and the settings that format a Panel control. The code-behind section shows how the Profile provider is invoked.

Listing 7-8. Direct calls to the Profile provider and typed access

public partial class \_Default : System.Web.UI.Page

{

protected void Page\_Load(object sender, EventArgs e)

{

SetProfileData();

}

protected void btnSend\_Click(object sender, EventArgs e)

{

Profile.ForeColor = Color.FromName(drpForeColor.SelectedValue);

Profile.BackColor = Color.FromName(drpBackColor.SelectedValue);

Profile.User.Email = txtEmail.Text;

SetProfileData();

}

private void SetProfileData()

{

PanelSettings.BackColor = Profile.BackColor;

PanelSettings.ForeColor = Profile.ForeColor;

Label l = new Label();

l.Text = Profile.User.Email;

PanelSettings.Controls.Add(l);

}

}

The Profile class shown in the next code listing is auto-generated and derives from the ProfileBase. It provides typed access to properties. The Profile provider returns settings of type object and the access layer casts these types to the final values.

Listing 7-8. An auto-generated Profile class

//------------------------------------------------------------------------------

// <auto-generated>

// This code was generated by a tool.

// Runtime Version:2.0.50727.3074

//

// Changes to this file may cause incorrect behavior and will be lost if

// the code is regenerated.

// </auto-generated>

//------------------------------------------------------------------------------

using System;

using System.Web;

using System.Web.Profile;

public class ProfileGroupUser : System.Web.Profile.ProfileGroupBase {

public virtual string Name {

get {

return ((string)(this.GetPropertyValue("Name")));

}

set {

this.SetPropertyValue("Name", value);

}

}

public virtual string Email {

get {

return ((string)(this.GetPropertyValue("Email")));

}

set {

this.SetPropertyValue("Email", value);

}

}

}

public class ProfileCommon : System.Web.Profile.ProfileBase {

public virtual int Size {

get {

return ((int)(this.GetPropertyValue("Size")));

}

set {

this.SetPropertyValue("Size", value);

}

}

public virtual System.Drawing.Color ForeColor {

get {

return ((System.Drawing.Color)(this.GetPropertyValue("ForeColor")));

}

set {

this.SetPropertyValue("ForeColor", value);

}

}

public virtual System.Drawing.Color BackColor {

get {

return ((System.Drawing.Color)(this.GetPropertyValue("BackColor")));

}

set {

this.SetPropertyValue("BackColor", value);

}

}

public virtual ProfileGroupUser User {

get {

return ((ProfileGroupUser)(this.GetProfileGroup("User")));

}

}

public virtual ProfileCommon GetProfile(string username) {

return ((ProfileCommon)(ProfileBase.Create(username)));

}

}

To understand these settings, take a look at the *web.config* file where the properties are defined.

Listing 7-10. Definition of Profile properties

<profile automaticSaveEnabled="true" defaultProvider="XmlProfileProvider">

<properties>

<group name="User">

<add name="Name" type="System.String" />

<add name="Email" type="System.String" />

</group>

<add name="Size" type="System.Int32"/>

<add name="ForeColor" type="System.Drawing.Color"/>

<add name="BackColor" type="System.Drawing.Color"/>

</properties>

<providers>

<clear/>

<add name="XmlProfileProvider"

type="Apress.Extensibility.ProfileProvider.XmlProfileProvider"/>

</providers>

</profile>

The settings for username and email are grouped by the “User” group element. This is merely for organizational purposes, and has no direct influence on the behavior of the provider. The <providers> section contains the definition of the provider’s type. The name is used to set this provider as the default one.

### Implementing the Provider

We now have the requisite components. The last step is to construct the custom Profile provider itself, shown in the next listing. An explanation follows the code.

Listing 7-11. A custom Profile provider which can save and retrieve settings

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Configuration;

using System.Configuration.Provider;

using System.IO;

using System.Linq;

using System.Security.Permissions;

using System.Text;

using System.Web;

using System.Web.Profile;

using System.Xml.Linq;

namespace Apress.Extensibility.ProfileProvider

{

[SecurityPermission(SecurityAction.Assert,

Flags = SecurityPermissionFlag.SerializationFormatter)]

public class XmlProfileProvider : System.Web.Profile.ProfileProvider

{

private const string DATAPATH = "~/App\_Data/Profile\_Data";

public override string ApplicationName

{

get { throw new NotSupportedException(); }

set { throw new NotSupportedException(); }

}

public override void Initialize(string name, ⮰

System.Collections.Specialized.NameValueCollection config)

{

base.Initialize(name, config);

if (config.Count > 0)

throw new ProviderException("Unrecognized attribute: " + ⮰

config.GetKey(0));

}

public override System.Configuration.SettingsPropertyValueCollection ⮰

GetPropertyValues(System.Configuration.SettingsContext context, ⮰

System.Configuration.SettingsPropertyCollection collection)

{

SettingsPropertyValueCollection settings = ⮰

new SettingsPropertyValueCollection();

// Make sure we have an entry for this username in the XML data

string username = context["UserName"] as string;

if (!string.IsNullOrEmpty(username))

{

// Get the profile values for the user

Dictionary<string, object> usersProperties = GetUserProfile(username);

foreach (SettingsProperty property in collection)

{

if (property.PropertyType.IsPrimitive || ⮰

property.PropertyType == typeof(String))

property.SerializeAs = SettingsSerializeAs.String;

else

property.SerializeAs = SettingsSerializeAs.Xml;

SettingsPropertyValue setting = new SettingsPropertyValue(property);

if (usersProperties != null)

{

setting.IsDirty = false;

if (usersProperties.ContainsKey(property.Name))

{

setting.SerializedValue = usersProperties[property.Name];

setting.Deserialized = false;

}

}

settings.Add(setting); // Add the settings value to the collection

}

}

return settings; // Return the settings collection

}

protected virtual Dictionary<string, object> GetUserProfile(string username)

{

Dictionary<string, object> propertyValues = new Dictionary<string, object>();

XDocument xProfiles = XDocument.Load(ProfileFilePath);

var xProf = (from p in xProfiles.Root.Elements() ⮰

where p.Attribute("UserName").Value.Equals(username) ⮰

select p);

foreach (XElement xmlProperty in xProf.Elements())

{

SettingsSerializeAs ss = (SettingsSerializeAs)⮰

Enum.Parse(typeof(SettingsSerializeAs), ⮰

xmlProperty.Attribute("serializedAs").Value);

switch (ss)

{

case SettingsSerializeAs.Binary:

propertyValues.Add(

xmlProperty.Name.LocalName,

Encoding.ASCII.GetString(⮰

Convert.FromBase64String((((XCData) xmlProperty.FirstNode).Value))));

break;

case SettingsSerializeAs.String:

propertyValues.Add(

xmlProperty.Name.LocalName,

xmlProperty.Value);

break;

case SettingsSerializeAs.Xml:

if (xmlProperty.Attribute("typeConverter") != null)

{

TypeConverter converter = (TypeConverter) ⮰

Activator.CreateInstance(⮰

Type.GetType(xmlProperty.Attribute("typeConverter").Value));

propertyValues.Add( ⮰

xmlProperty.Name.LocalName, ⮰

converter.ConvertFromString(xmlProperty.Value));

}

break;

case SettingsSerializeAs.ProviderSpecific:

throw new NotSupportedException();

}

}

return propertyValues;

}

public override void SetPropertyValues(⮰

System.Configuration.SettingsContext context, ⮰

System.Configuration.SettingsPropertyValueCollection collection)

{

string username = context["UserName"] as string;

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

// If no username is specified, or if no properties are to be saved, exit

if (string.IsNullOrEmpty(username) || collection.Count == 0)

return;

if (!ExistsDirtyProperty(collection))

return;

XDocument xProfiles = XDocument.Load(ProfileFilePath);

// check elements

var xProf = (from p in xProfiles.Root.Elements() ⮰

where p.Attribute("UserName").Value.Equals(username) ⮰

select p).FirstOrDefault();

if (xProf == null)

{

// Add a default empty profile

xProf = new XElement("Profile", new XAttribute("UserName", username));

xProfiles.Root.Add(xProf);

xProfiles.Save(ProfileFilePath);

}

// assure empty element as write everything back

xProf.RemoveNodes();

foreach (SettingsPropertyValue setting in collection)

{

// If the user is not authenticated and the property does

// not allow anonymous access, skip serializing it

if (!userIsAuthenticated && ⮰

!(bool)setting.Property.Attributes["AllowAnonymous"])

continue;

// Skip the current property if it's not dirty and is currently

// assigned its default value

if (!setting.IsDirty && setting.UsingDefaultValue)

continue;

// Serialize data based on property's SerializeAs type

switch (setting.Property.SerializeAs)

{

case SettingsSerializeAs.String:

xProf.Add(new XElement(⮰

setting.Name, ⮰

Convert.ToString(setting.SerializedValue), ⮰

new XAttribute("serializedAs", setting.Property.SerializeAs)));

break;

case SettingsSerializeAs.Xml:

// instead of XML we ask the default converter

TypeConverter converter = TypeDescriptor.GetConverter(⮰

setting.Property.PropertyType);

string data = converter.ConvertToString(setting.PropertyValue);

xProf.Add(new XElement(⮰

setting.Name, ⮰

data, ⮰

new XAttribute("serializedAs", setting.Property.SerializeAs), ⮰

new XAttribute("typeConverter", converter.GetType().AssemblyQualifiedName)));

break;

case SettingsSerializeAs.Binary:

// encode the binary data using base64 encoding

string encodedBinaryData = Convert.ToBase64String(setting.SerializedValue as byte[]);

xProf.Add(new XElement(⮰

setting.Name, ⮰

new XCData(encodedBinaryData), ⮰

new XAttribute("serializedAs", setting.Property.SerializeAs)));

break;

default:

// unknown serialize type!

throw new ProviderException(string.Format("Invalid value for SerializeAs; expected String, Xml, or Binary, received {0}", System.Enum.GetName(setting.Property.SerializeAs.GetType(), setting.Property.SerializeAs)));

}

}

xProfiles.Save(ProfileFilePath);

}

protected virtual string ProfileFilePath

{

get

{

return Path.Combine(HttpContext.Current.Server.MapPath(DATAPATH), ⮰

"Profiles.xml");

}

}

protected virtual bool ExistsDirtyProperty(⮰

System.Configuration.SettingsPropertyValueCollection collection)

{

foreach (SettingsPropertyValue setting in collection)

if (setting.IsDirty)

return true;

// If we reach here, none are dirty

return false;

}

public override int DeleteInactiveProfiles(⮰

ProfileAuthenticationOption authenticationOption, ⮰

DateTime userInactiveSinceDate)

{

throw new Exception("The method or operation is not implemented.");

}

public override int DeleteProfiles(string[] usernames)

{

throw new Exception("The method or operation is not implemented.");

}

public override int DeleteProfiles(ProfileInfoCollection profiles)

{

throw new Exception("The method or operation is not implemented.");

}

public override ProfileInfoCollection FindInactiveProfilesByUserName(⮰

ProfileAuthenticationOption authenticationOption, ⮰

string usernameToMatch, ⮰

DateTime userInactiveSinceDate, ⮰

int pageIndex, int pageSize, out int totalRecords)

{

throw new Exception("The method or operation is not implemented.");

}

public override ProfileInfoCollection FindProfilesByUserName(⮰

ProfileAuthenticationOption authenticationOption, ⮰

string usernameToMatch, int pageIndex, int pageSize, out int totalRecords)

{

throw new Exception("The method or operation is not implemented.");

}

public override ProfileInfoCollection GetAllInactiveProfiles(⮰

ProfileAuthenticationOption authenticationOption, ⮰

DateTime userInactiveSinceDate, int pageIndex, ⮰

int pageSize, out int totalRecords)

{

throw new Exception("The method or operation is not implemented.");

}

public override ProfileInfoCollection GetAllProfiles(⮰

ProfileAuthenticationOption authenticationOption, ⮰

int pageIndex, int pageSize, out int totalRecords)

{

throw new Exception("The method or operation is not implemented.");

}

public override int GetNumberOfInactiveProfiles(⮰

ProfileAuthenticationOption authenticationOption, ⮰

DateTime userInactiveSinceDate)

{

throw new Exception("The method or operation is not implemented.");

}

}

}

The class has two principal methods, GetPropertyValues and SetPropertyValues, for retrieving and saving Profile settings. The GetPropertyValues method is called to obtain data from the Profile class. It then calls GetUserProfile to read the specific Profile from the XML file. Within the GetUserProfile method, a LINQ query retrieves the Profile. A loop reads each value and deserializes it depending on its settings. Binary data is saved in Base64 encoded format and protected within the <!CDATA> section. The content node is cast to Xcdata and string values are read directly. More complex types are treated as ProviderSpecific, which means that the provider is responsible for finding the right serialization. To support typical .NET types like System.Drawing.Color, without writing type specific code, the internal use type converter is used. This type converter usually supports the design time experience. For instance, to show colors within the PropertyGrid control, a type converter is used. In this example, the type converter’s type is saved into the document and retrieved from the typeConverter attribute. In the XML file, the fully qualified name is stored and used to create an instance of the converter. The converter provides a ConvertFromString method, which converts from string format to the specified object.

The SetPropertyValues method reverses the whole process. Firstly, the same LINQ statement is used to retrieve the specific user’s section. If it doesn’t exist, an empty section is added. The collection of settings defined in the *web.config* file is used to assemble the current user’s Profile. Each setting’s “type” (as defined in the *web.*config) is used to obtain the associated TypeConverter object, whose ConvertToString method serializes the setting’s value. If everything goes correctly, the XML fragment is saved in the XML file.

Note: This is a simplified scenario lacking error handling and multiuser support. It is only intended to show the construction of a custom provider and how to change the way the data is persisted.

The XML written using this code looks very simple:

Listing 7-12. XML that stores user Profile settings

<?xml version="1.0" encoding="utf-8"?>

<Profiles>

<Profile UserName="User1">

<BackColor serializedAs="ProviderSpecific"

typeConverter="System.Drawing.ColorConverter, System.Drawing,

Version=2.0.0.0, Culture=neutral,

PublicKeyToken=b03f5f7f11d50a3a">Red/BackColor>

<User.Email serializedAs="String">joerg@krause.net</User.Email>

<ForeColor serializedAs="ProviderSpecific"

typeConverter="System.Drawing.ColorConverter, System.Drawing,

Version=2.0.0.0, Culture=neutral,

PublicKeyToken=b03f5f7f11d50a3a">White</ForeColor>

</Profile>

<Profile UserName="User2">

<BackColor serializedAs="ProviderSpecific"

typeConverter="System.Drawing.ColorConverter, System.Drawing,

Version=2.0.0.0, Culture=neutral,

PublicKeyToken=b03f5f7f11d50a3a">Blue</BackColor>

<User.Email serializedAs="String">User3@user.de</User.Email>

<ForeColor serializedAs="ProviderSpecific"

typeConverter="System.Drawing.ColorConverter, System.Drawing,

Version=2.0.0.0, Culture=neutral,

PublicKeyToken=b03f5f7f11d50a3a">Red</ForeColor>

</Profile>

</Profiles>

Each Profile consists of a <Profile> element. This element has a UserName attribute which associates the settings with a particular user. The content depends on the property definition, as shown in code listing 7-10. Each element has the serializedAs attribute to indicate the serializer used. In the case of “ProviderSpecific” serializers, the typeConverter attribute defines how a value is converted into a string and back.

### A Client Side Driven Profile Provider

With AJAX (Asynchronous JavaScript and XML), you can also work with the custom Profile technology directly from your client-side script. This section is slightly different, as most of the code is JavaScript, instead of C#. The code extends the profile usage to the client and empowers script developers predominantly.

#### Exposing Profile Settings to AJAX using Web.config

There’s a minor addition required to the *web.config* file to enable AJAX to manage Profiles (Listing 7-12). You need to specify which properties in the Profile can be read from and which ones can be changed. The Profiles settings we saw in Listing 7-10 are unchanged.

Listing 7-12. Web.config settings to configure AJAX access to user Profile properties

<system.web.extensions>  
   <scripting>  
       <webServices>  
           <profileService enabled="true"  
               readAccessProperties="User.Name, User.Email,Size,ForeColor,BackColor"  
               writeAccessProperties="User.Email,ForeColor,BackColor" />  
       </webServices>  
   </scripting>  
</system.web.extensions>

All you need to do now is write the appropriate client code.

#### Creating the User Interface

Because we’ll use pure AJAX to communicate with the server, we don’t need server side controls. The form shown below uses basic HTML to build the UI. There are a few HTML input controls with an ID attribute assigned to each one:

Listing 7-12. Client side part of the solution—the HTML form

<form runat="server">

<asp:ScriptManager ID="ScriptManager" runat="server" />

<fieldset id="ContactFieldset">

<label>

E-Mail

<input type="text" id="eMail" /></label><br />

<label>

User Name

<input type="text" id="userName" disabled="disabled" /></label>

<label>

<br />

Fore Color:

<br />

Red

<input type="radio" name="ForeColor" id="fc1" value="Red" />

Blue

<input type="radio" name="ForeColor" id="fc2" value="Blue" />

Green

<input type="radio" name="ForeColor" id="fc3" value="Green" />

</label>

<br />

<label>

Back Color:

<br />

White

<input type="radio" name="BackColor" id="bc1" value="White" />

Beige

<input type="radio" name="BackColor" id="bc2" value="Beige" />

Yellow

<input type="radio" name="BackColor" id="bc3" value="Yellow" />

</label>

<br />

<button onclick="SaveProperties();">

Save</button>

</fieldset>

<hr />

<p id="Status">

</p>

</form>

The form above will appear like this:

Insert 19835f0710.tif

Figure 7-2. The form with simple HTML form controls

#### The JavaScript to Tie It Together

The final step is to add the JavaScript code which will communicate between the client and the server. Firstly, we’ll bring up the Profile for the current user and populate the form when the page loads. Then we’ll expose a “Save” button so that the user can update their Profile. The final step is to inform the user that their Profile has been saved. We'll accomplish this with a status label which will hide itself after 5 seconds.

Breaking this down into steps, we’ll firstly load the Profile for the authenticated user by calling the Load method of the Sys.Services.ProfileService object. If this call is successful, we’ll populate the form. If there was an error, we’ll alert the user. The callback methods are used to check success and error conditions. If successful, the values are read directly from the generic JavaScript class, using the same syntax for properties as the server.

Listing 7-15. JavaScript that stores user Profile settings

window.onload = function() {

Sys.Services.ProfileService.load(null, onLoadSuccess, onError);

}

function onLoadSuccess(obj) {

$get("eMail").value = Sys.Services.ProfileService.properties.User.Email;

$get("userName").value = Sys.Services.ProfileService.properties.User.Name;

var fc = Sys.Services.ProfileService.properties.ForeColor.Name;

$get("fc1").checked = (fc == 'Red');

$get("fc2").checked = (fc == 'Blue');

$get("fc3").checked = (fc == 'Green');

var bc = Sys.Services.ProfileService.properties.BackColor.Name;

$get("bc1").checked = (bc == 'White');

$get("bc2").checked = (bc == 'Beige');

$get("cc3").checked = (bc == 'Yellow');

}

function onError(error) {

$get("Status").innerHTML = error.get\_message();

}

The above code will attempt to load the Profile for the current user when the page loads. If this is successful, the onLoadSuccess function will fire and we can populate the form. If there was an error, the onError function will fire and an error message will be displayed to the user.

The final step is to build the “Save Profile” function. This is similar to the Load method that is called above. Assign the new values to the Profile object in JavaScript and then call Save on the Sys.Services.ProfileService object. Again, the callback methods are used to check for success (and errors).

Listing 7-12. Script code that stores user Profile settings

function SaveProperties() {

Sys.Services.ProfileService.properties.User.Email = $get("eMail").value;

Sys.Services.ProfileService.properties.User.Name = $get("username").value;

var fc = '';

fc += ($get("fc1").checked) ? 'Red' : '';

fc += ($get("fc2").checked) ? 'Blue' : '';

fc += ($get("fc3").checked) ? 'Green' : '';

Sys.Services.ProfileService.properties.ForeColor = fc;

var bc = '';

bc += ($get("fc1").checked) ? 'White' : '';

bc += ($get("fc2").checked) ? 'Beige' : '';

bc += ($get("fc3").checked) ? 'Yellow' : '';

Sys.Services.ProfileService.properties.BackColor = bc;

Sys.Services.ProfileService.save(null, onSaveSuccess, onError);

}

function onSaveSuccess() {

clearTimeout();

// Display the success message to the user.

$get("Status").innerHTML = "Your profile has been saved.";

// Reset the display after 5 seconds have passed.

setTimeout(function() { $get("Status").innerHTML = ""; }, 5000);

}

If all goes well, you should see a message when you click the “Save” button:

Insert 19835f0711.tif

Figure 7-2. The form response when the Profile access was successful

This is a very simple example which demonstrates how to access the Profile using the AJAX library. Its power comes from the library and its close relation to basic ASP.NET features. If you’re not confident using the ASP.NET AJAX libraries because of their size or performance, keep in mind that they support larger parts of ASP.NET out of the box. This is very useful for smaller projects, as well as for constructing complex features quickly. However, in large projects with heavy access you might consider different ways, like other third party libraries which support the specific parts required for Profiles only.

## Extending Web Parts Personalization Providers

Web Parts are specialized user controls that have extensive client support. Web Parts make up portions of a web page, and allow users to change the page design. For example, users can move Web Parts around, hide or close them, and change their behavior. Web Parts are typically used to create portal pages that users can personalize. ASP.NET comes with all the pieces needed to construct a completely personalizable page. The framework includes components which perform the following tasks:

\* Define and handle customizable sections through zones

\* Pull Web Parts from a catalogue

\* Export and import Web Parts so that users can share them like gadgets

\* Save the user’s preferences to a permanent data store

By now it should be clear that a provider plays a role in this. Web Parts’ settings, locations and properties must be stored and retrieved when the user revisits the site. This implies that the user must be authenticated. There is an indirect relationship between the membership and role providers, and the Web Part personalization provider. There is no dependency on the profile provider, as explained in the previous section.

### Understanding the Web Parts Personalization Provider

Before you start working with the personalization provider and planning a custom implementation, you should understand the default provider and how it behaves. The default provider is System.Web.UI.WebControls.WebParts.SqlPersonalizationProvider, and it accesses the same SQL Server database used to support membership, roles and profiles internally. The prime job of the personalization provider is to store the state of Web Parts, which consists of Web Parts content and the layout of Web Parts pages. The state is held in a container of type System.Web.UI.WebControls.WebParts.PersonalizationState. The personalization service using the provider serializes the data and sends it to the provider as a stream of bytes to store. Conversely, it retrieves and deserializes data to restore the state of Web Parts.

Because the SqlPersonalizationProvider is a final implementation, there exists an abstract base class for such personalization providers—the PersonalizationProvider class.

public abstract class PersonalizationProvider : ProviderBase

{

protected PersonalizationProvider();

public abstract string ApplicationName { get; set; }

protected virtual IList CreateSupportedUserCapabilities();

public virtual PersonalizationScope DetermineInitialScope( ⮰

WebPartManager webPartManager, PersonalizationState loadedState);

public virtual IDictionary DetermineUserCapabilities( ⮰

WebPartManager webPartManager);

public abstract PersonalizationStateInfoCollection FindState( ⮰

PersonalizationScope scope, PersonalizationStateQuery query, ⮰

int pageIndex, int pageSize, out int totalRecords);

public abstract int GetCountOfState(PersonalizationScope scope, ⮰

PersonalizationStateQuery query);

protected abstract void LoadPersonalizationBlobs(WebPartManager webPartManager,⮰

string path, string userName, ⮰

ref byte[] sharedDataBlob, ref byte[] userDataBlob);

public virtual PersonalizationState LoadPersonalizationState( ⮰

WebPartManager webPartManager, bool ignoreCurrentUser);

protected abstract void ResetPersonalizationBlob(⮰

WebPartManager webPartManager, string path, string userName);

public virtual void ResetPersonalizationState(WebPartManager webPartManager);

public abstract int ResetState(PersonalizationScope scope, ⮰

string[] paths, ⮰

string[] usernames);

public abstract int ResetUserState(string path, DateTime userInactiveSinceDate);

protected abstract void SavePersonalizationBlob(WebPartManager webPartManager, ⮰

string path, string userName, byte[] dataBlob);

public virtual void SavePersonalizationState(PersonalizationState state);

}

The following table explains the methods and properties. The various abstract and virtual keywords indicate that only a few methods need to be implemented—the other methods are optional. As a result, this base class is a better starting point for a custom implementation than the ProviderBase base class.

Table 7-1. Members of the PersonalizationProvider base class

|  |  |
| --- | --- |
| Member | Description |
| ApplicationName | Gets or sets the name of the application. |
| CreateSupportedUserCapabilities | Returns a list of System.Web.UI.WebControls.WebParts.WebPartUserCapability objects that represent the set of known capabilities. |
| DetermineInitialScope | Determines whether the initial personalization scope should be Shared or User (from PersonalizationScope enum). Takes a WebPartManager object that manages the personalization information and the personalization state information. The method returns a PersonalizationScope enum value. |
| DetermineUserCapabilities | Returns a dictionary containing WebPartUserCapability instances that represent the personalization-related capabilities of the currently executing user account. |
| FindState | Returns a collection containing zero or more PersonalizationStateInfo objects based on scope and specific query parameters. Depends on scope, query, and paging information. |
| GetCountOfState | Returns the number of rows in the underlying data store that exist within the specified scope. Depends on query and scope. |
| LoadPersonalizationBlobs | Loads raw personalization data from the underlying data store. Takes a reference to the WebPartManager object managing the personalization data, a path as key, the user’s name and the blob data for shared and user part. |
| LoadPersonalizationState | Loads the raw data from the underlying data store and converts it into a PersonalizationState object. |
| ResetPersonalizationBlob | Deletes raw personalization data from the underlying data store. |
| ResetPersonalizationState | Resets personalization data to the underlying data store. |
| ResetState | Deletes personalization state from the underlying data store based on the specified parameters. Returns the number of rows deleted. |
| ResetUserState | Deletes Web Parts personalization data from the underlying data store based on the specified parameters. |
| SavePersonalizationBlob | Saves raw personalization data to the underlying data store. |
| SavePersonalizationState | Saves personalization data to a data store. |

Implementing this is not difficult. The Load, Save, and Reset methods are the core parts of this class.

One crucial aspect is the scope of the personalization data. Web Parts personalization has two scopes: by user and by path. This allows you to personalize the behavior either for each individual user or by each page’s path. Scoping by request path is called “shared” in the method names used by the provider. In these cases, the user name is set to null and all settings apply to all users. The provider must take care of such data and store it separately from the users’ storage.

The provider support is optional for multiple applications based on the ApplicationName property, but recommended. If you use a local store, such as an XML file (as shown in the example below), the application name can be ignored.

### Implementing a Custom Personalization Provider

A custom personalization provider has several prerequisites:

\* A portal page must exist that handles several Web Parts

\* The site must support Authentication and user membership

\* The custom Web Part personalization provider and its store must exist

\* The *web.config* file must be properly configured to support Web Part personalization

The next sections explain the required steps one by one.

#### Creating a Portal Page

Our Portal page to test the solution is a simple Web Part page as shown below.

Listing 7-1. The “portal” page to test the solution

<form id="form1" runat="server">

<div>

<asp:WebPartManager ID="WebPartManager1" runat="server">

<Personalization Enabled="true" ProviderName="XmlPersonalizationProvider" />

</asp:WebPartManager>

<table style="width: 100%">

<tr valign="middle" style="background: #dddddd">

<td colspan="2">

<h2>

Welcome to our Portal,

<asp:LoginName ID="LoginName1" runat="server" />

</h2>

</td>

<td>

<asp:Menu ID="Menu1" runat="server" ⮰

OnMenuItemClick="Menu1\_MenuItemClick">

</asp:Menu>

</td>

</tr>

<tr valign="top">

<td style="width: 20%">

<asp:CatalogZone ID="CatalogZone1" runat="server">

<ZoneTemplate>

<asp:PageCatalogPart ID="PageCatalogPart1" runat="server" />

</ZoneTemplate>

</asp:CatalogZone>

<asp:EditorZone ID="EditorZone1" runat="server">

</asp:EditorZone>

</td>

<td style="width: 60%">

<asp:WebPartZone ID="WebPartZone1" runat="server">

</asp:WebPartZone>

</td>

<td style="width: 20%">

<asp:WebPartZone ID="WebPartZone2" runat="server">

<TitleBarVerbStyle BackColor="ActiveBorder" />

<ZoneTemplate>

<asp:Calendar ID="Calendar1" runat="server"></asp:Calendar>

<asp:FileUpload ID="FileUpload1" runat="server" />

</ZoneTemplate>

</asp:WebPartZone>

</td>

</tr>

<tr>

<td colspan="3">

<asp:LoginStatus ID="LoginStatus1" runat="server" />

</td>

</tr>

</table>

</div>

</form>

The only dependency on other code parts is the selection of the current provider:

<Personalization Enabled="true" ProviderName="XmlPersonalizationProvider" />

As shown above, the personalization must be turned on in order to activate the provider using the Enabled attribute. The name of the provider is defined in *web.config*. (See the section “Configure the Provider” for more details.)

To force the page to change the settings and invoke the provider, the user must enter the Edit mode of the Web Part page. To do this easily, the following code-behind code is used:

public partial class \_Default : System.Web.UI.Page

{

protected void Page\_Load(object sender, EventArgs e)

{

if (!IsPostBack)

{

// Create a menu of web part modes

MenuItem rootItem = new MenuItem("Select Web Part Mode");

foreach (WebPartDisplayMode mode in WebPartManager1.DisplayModes)

{

rootItem.ChildItems.Add(new MenuItem(mode.Name));

}

Menu1.Items.Add(rootItem);

}

}

protected void Menu1\_MenuItemClick(object sender, MenuEventArgs e)

{

WebPartManager1.DisplayMode = WebPartManager1.DisplayModes[e.Item.Text];

}

}

The menu is filled with items from the collection of acceptable modes. In the MenuItemClick event handler, the same value is used to set the current mode.

Figure 7-1. The Web Part page

#### Prepare Authentication

For a simple scenario to login and logout users, I set up Forms authentication in *web.config*. Furthermore, I store the users’ credentials directly in *web.config* and configure the Membership provider to use that data instead of data from the default provider (SQL Server). This can be accomplished by handling the authentication event manually. The *web.config* section that defines the users looks like this:

Listing 7-2. Define some users quickly to test the application

<authorization>

<deny users="?"/>

</authorization>

<authentication mode="Forms">

<forms>

<credentials passwordFormat="Clear">

<user name="User1" password="User1"/>

<user name="User2" password="User2"/>

<user name="User3" password="User3"/>

</credentials>

</forms>

</authentication>

The application then requires at least one page in order to change and use the profile settings, as well as a login page. The login page consists of a single Login control and an event handler.

Listing 7-3. The login page in its simplest form

<body>

<form id="form1" runat="server">

<div>

<asp:Login ID="Login1" runat="server" onauthenticate="Login1\_Authenticate">

</asp:Login>

<br />

</div>

</form>

</body>

</html>

Listing 7-4. An event handler forces the login control to use the configured user names

public partial class Login : System.Web.UI.Page

{

protected void Page\_Load(object sender, EventArgs e)

{

}

protected void Login1\_Authenticate(object sender, AuthenticateEventArgs e)

{

e.Authenticated = FormsAuthentication.Authenticate(Login1.UserName, ⮰

Login1.Password);

}

}

The next page is also for testing purposes. It contains a simple form to save new profile settings, show current settings, and change accounts.

#### Create the Provider

The following implementation uses XML to store data. It renders the application independent of a SQL Server database. The implementation is as simple as possible—only the basic methods for loading and saving data are fully implemented.

Listing 7-5. A Web Part Personalization Provider using XML

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.UI.WebControls.WebParts;

using System.IO;

using System.Web.Hosting;

using System.Data;

using System.Xml.Linq;

using System.Xml.XPath;

namespace Apress.Extensibility.WebPartProvider

{

public class XmlPersonalizationProvider : PersonalizationProvider

{

private const string SETTINGSNAME = "WPSettings.xml";

private const string SETTINGSTAG = "WPSettings";

private string configFile;

public override void Initialize(string name, System.Collections.Specialized.NameValueCollection config)

{

base.Initialize(name, config);

configFile = HttpContext.Current.Request.MapPath(Path.Combine("~/App\_Data/", SETTINGSNAME));

// XML

if (!File.Exists(configFile))

{

XDocument cfgDoc = new XDocument(

new XElement("WebPartData", new XAttribute("Created", DateTime.Now.ToShortDateString()),

new XElement("UserScope"),

new XElement("SharedScope")));

cfgDoc.Declaration = new XDeclaration("1.0", "utf-8", "true");

cfgDoc.Save(configFile);

}

}

// NOTE: only Shared-scope personalization is loaded

protected override void LoadPersonalizationBlobs(

WebPartManager webPartManager,

string path,

string userName,

ref byte[] sharedDataBlob,

ref byte[] userDataBlob)

{

string fullPath = HttpContext.Current.Request.MapPath(path);

XDocument cfgDoc = XDocument.Load(configFile);

var root = cfgDoc.Element("WebPartData");

if (userName == null)

{

object cachedPageSettings = HttpContext.Current.Cache[SETTINGSTAG + ":" + path];

if (cachedPageSettings != null)

{

sharedDataBlob = (byte[])cachedPageSettings;

}

else

{

var shared = root.Element("SharedScope")

.Elements("Page")

.Single(n => n.Attribute("name").Value.Equals(userName));

if (shared == null)

{

sharedDataBlob = null;

}

else

{

sharedDataBlob = Convert.FromBase64String(shared.Value);

// cache shared settings

webPartManager.Page.Cache.Insert(SETTINGSTAG + ":" + path, sharedDataBlob, new System.Web.Caching.CacheDependency(configFile));

}

}

}

else

{

var pageElement = root.XPathSelectElement(

String.Format("//UserScope/User[@name='{0}']/Page[@name='{1}']",

userName,

path));

if (pageElement != null)

{

userDataBlob = Convert.FromBase64String(pageElement.Value);

}

}

}

protected override void ResetPersonalizationBlob(

WebPartManager webPartManager,

string path,

string userName)

{

}

protected override void SavePersonalizationBlob(

WebPartManager webPartManager,

string path,

string userName,

byte[] dataBlob)

{

string fullPath = HttpContext.Current.Request.MapPath(path);

string sBlob = Convert.ToBase64String(dataBlob);

lock (this)

{

XDocument cfgDoc = XDocument.Load(configFile);

var root = cfgDoc.Element("WebPartData");

if (!String.IsNullOrEmpty(userName))

{

// Scope: user

var userElement = root.XPathSelectElement(

String.Format("//UserScope/User[@name='{0}']",

userName));

if (userElement == null)

{

userElement = root.Element("UserScope");

// no user, add complete tree

userElement.Add(

new XElement("User", new XAttribute("name", userName),

new XElement("Page", new XAttribute("name", path),

sBlob)));

}

else

{

// with user, check page

var pageElement = userElement.Elements("Page")

.Single(n => n.Attribute("name").Value.Equals(path));

if (pageElement == null)

{

// no page

userElement.Add(new XElement("Page",

new XAttribute("name", path),

sBlob));

}

else

{

// new data for page

pageElement.Value = sBlob;

}

}

}

else

{

// Scope: Shared

var sharedElement = root.Elements("SharedScope")

.Elements("Page")

.Single(p => p.Attribute("Name").Value.Equals(path));

if (sharedElement == null)

{

sharedElement.Add(new XElement("Page",

new XAttribute("name", path),

sBlob));

}

else

{

sharedElement.Value = sBlob;

}

}

cfgDoc.Save(configFile);

}

}

public override string ApplicationName

{

get;

set;

}

public override int GetCountOfState(

PersonalizationScope scope,

PersonalizationStateQuery query)

{

return 0;

}

public override PersonalizationStateInfoCollection FindState(

PersonalizationScope scope,

PersonalizationStateQuery query,

int pageIndex,

int pageSize,

out int totalRecords)

{

totalRecords = 0;

return null;

}

public override int ResetState(

PersonalizationScope scope,

string[] paths,

string[] usernames)

{

return 0;

}

public override int ResetUserState(

string path,

DateTime userInactiveSinceDate)

{

return 0;

}

}

}

During the call to Initialize, a data store in the form of an XML file will be created, if there isn’t one already. When the user changes properties of Web Parts, the SavePersonalizationBlob method is called. The approach used is to search for the userName, and if it is null, meaning the user is not authenticated, assume that it’s a shared scope setting. Otherwise, it’s a personal setting, and the user element is retrieved. We also assume that data for this user does not already exist. (This is why XPath expressions are used instead of XLinq expression chains.) If the <User> element does not exist, it’s created using the current data. The data is passed to the provider as a byte array, and stored as a Base64 encoded string in the XML element. If the <User> element exists, the <Page> element is retrieved. Each personalized page has its own XML element. If this element does not exist, it’s created and the encoded string is used as its value. If it does exist, the current contents are replaced with the Value property.

A similar technique is used to retrieve the data. Every time the page refreshes, the provider is called to return the data as a byte array. If there is no data, we can safely return null. While the shared data is stored in a cache, the data for each user is retrieved directly from the XML file. There is plenty of scope for improving the file handling and caching, but that’s beyond the scope of this chapter. To locate the data again, an XPath expression is used. If a value is present, the FromBase64String method converts the string into a byte array.

This code produces an XML file in the App\_Data folder in this manner:

<?xml version="1.0" encoding="utf-8"?>

<WebPartData Created="5/16/2009">

<UserScope>

<User name="User1">

<Page name="~/Default.aspx">/wEUKwAKAgICARkqMVN5c3RlbS5XZWIuVUkuV2ViQ29udHJvbHMuV2ViUGFydHMuV2ViUGFydE1hbmFnZXIFBV9fd3BtZgIBHhBXZWJQYXJ0U3RhdGVVc2VyFCsACAUMZ3dwQ2FsZW5kYXIxBQxXZWJQYXJ0Wm9uZTJmaAUOZ3dwRmlsZVVwbG9hZDEFDFdlYlBhcnRab25lMWZoaGg=</Page>

</User>

<User name="User2">

<Page name="~/Default.aspx">/wEUKwAKAgICARkqMVN5c3RlbS5XZWIuVUkuV2ViQ29udHJvbHMuV2ViUGFydHMuV2ViUGFydE1hbmFnZXIFBV9fd3BtZgIBHhBXZWJQYXJ0U3RhdGVVc2VyFCsACAUMZ3dwQ2FsZW5kYXIxBQxXZWJQYXJ0Wm9uZTFmaAUOZ3dwRmlsZVVwbG9hZDEFDFdlYlBhcnRab25lMmZoaGg=</Page>

</User>

</UserScope>

<SharedScope />

</WebPartData>

Although its format is not impressive, this has everything we need in order to store data for any number of users and any number of personalized pages. The number of pages is usually low—for a customizable portal page it’s one—and the number of active users depends on your application. For an intranet with up to several hundred users, the XML file storage solution is easy to implement and sufficiently fast.

#### Configure the Provider

Having built the provider, it needs to be configured for use. I use Forms Authentication in the example and store the users’ data in *web.config*, as shown below. This is not meant to be used as production code, but it simplifies development by making user management very easy.

<system.web>

<webParts>

<personalization defaultProvider="XmlPersonalizationProvider">

<providers>

<add name="XmlPersonalizationProvider"

type="Apress.Extensibility.WebPartProvider.XmlPersonalizationProvider"/>

</providers>

</personalization>

</webParts>

<authentication mode="Forms">

<forms>

<credentials passwordFormat="Clear">

<user name="User1" password="user1"/>

<user name="User2" password="user2"/>

<user name="User3" password="user3"/>

</credentials>

</forms>

</authentication>

<authorization>

<deny users="?"/>

</authorization>

### Testing the Custom Web Part Personalization Provider

You can use the three prepared user accounts to test the provider. Simply launch the application and log on using one of the above credentials. Switch to “Edit” mode with the menu. Move the web parts around and re-arrange the page. The data is saved immediately. Logout and log back in as a different user. Create another page layout. Logout again and login with the account you used first. The page layout for the first user should re-appear. Other than the provider, there is no other custom code enabling this feature. The Web Part Personalization infrastructure is already available in ASP.NET and Web Parts used here are oblivious to the custom provider.

Caution: The sample provider lacks error handling and testing, and does not fully implement all the features of Web Parts providers. Before using as production code, more development is required.

# Summary

In this chapter, we looked at security and Profile features. The user-driven security model is covered by the Membership and Role services, both of which use providers to access a data store. Creating customized providers significantly extends the built-in role and membership models.

Once a user is authenticated, you can associate data with his or her Profile. The Profile service uses a provider that can be extended—or rewritten using a completely customized version. A Profile provider can also be reached by a client using pure AJAX, enabling you to save values to and retrieve values from the server without invoking server side code.

Both extension models provide a transparent and simple way of extending and customizing behavior, while allowing the existing controls to function as they would with default providers.

The Web Part Personalization Provider is an integrated technique which stores settings of Web Part pages for an authenticated user in a custom data store. In this chapter, you learned which base class to implement in order to change the default behavior and storage medium.