Microsoft ASP.NET 2.0 Providers

Microsoft Corporation

March 2006

Introduction

Microsoft ASP.NET 2.0 includes a number of services that store state in databases and other storage media. For example, the session state service manages per-user session state by storing it in-process (in memory in the application domain of the host application), in memory in an external process (the "state server process"), or in a Microsoft SQL Server database, whereas the membership service stores user names, passwords, and other membership data in Microsoft SQL Server or Microsoft Active Directory.

These and other state management services in ASP.NET 2.0 use the provider model pictured in Figure 1 to maximize storage flexibility. Providers abstract storage media in much the same way that device drivers abstract hardware devices. The membership service is equally at home using SQL Server or Active Directory, because ASP.NET 2.0 includes providers for each. Moreover, ASP.NET 2.0 can be extended with custom providers to add support for Web services, Oracle databases, SQL Server databases with custom schemas, and other media not supported by the built-in providers.

Figure 1. The ASP.NET 2.0 provider model

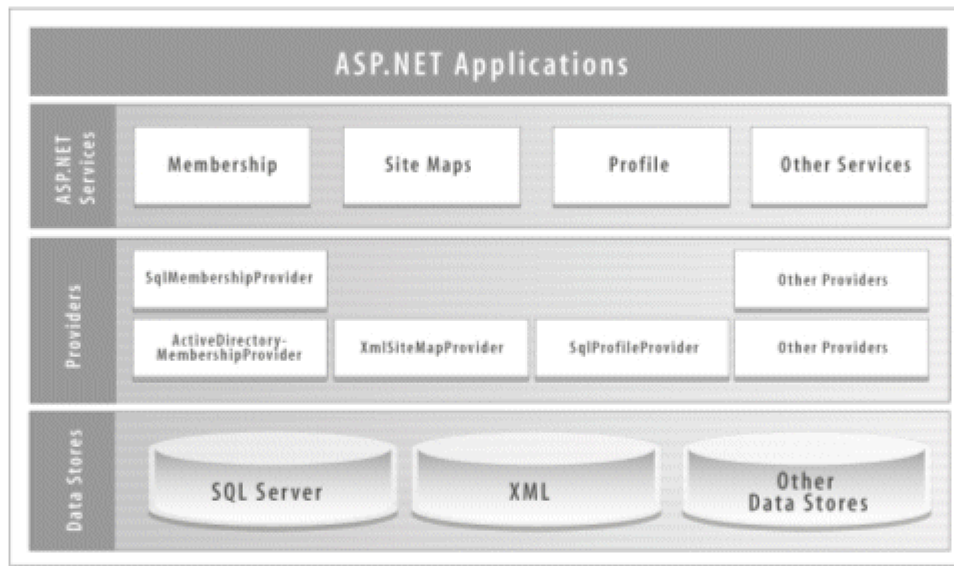


Table 1 lists the providers that are included with ASP.NET 2.0.

Table 1. ASP.NET 2.0 providers

Provider Type	Built-In Provider(s)
Membership	*System.Web.Security.ActiveDirectoryMembershipProvider* *System.Web.Security.SqlMembershipProvider*
Role management	*System.Web.Security.AuthorizationStoreRoleProvider* *System.Web.Security.SqlRoleProvider* *System.Web.Security.WindowsTokenRoleProvider*
Site map	*System.Web.XmlSiteMapProvider*

Profile	*System.Web.Profile.SqlProfileProvider*
Session state	*System.Web.SessionState.InProcSessionStateStore* *System.Web.SessionState.OutOfProcSessionStateStore* *System.Web.SessionState.SqlSessionStateStore*
Web events	*System.Web.Management.EventLogWebEventProvider* *System.Web.Management.SimpleMailWebEventProvider* *System.Web.Management.TemplatedMailWebEventProvider* *System.Web.Management.SqlWebEventProvider* *System.Web.Management.TraceWebEventProvider* *System.Web.Management.WmiWebEventProvider*
Web Parts personalization	*System.Web.UI.WebControls.WebParts.SqlPersonalizationProvider*
Protected configuration	*System.Configuration.DPAPIProtectedConfigurationProvider* *System.Configuration.RSAProtectedConfigurationProvider*

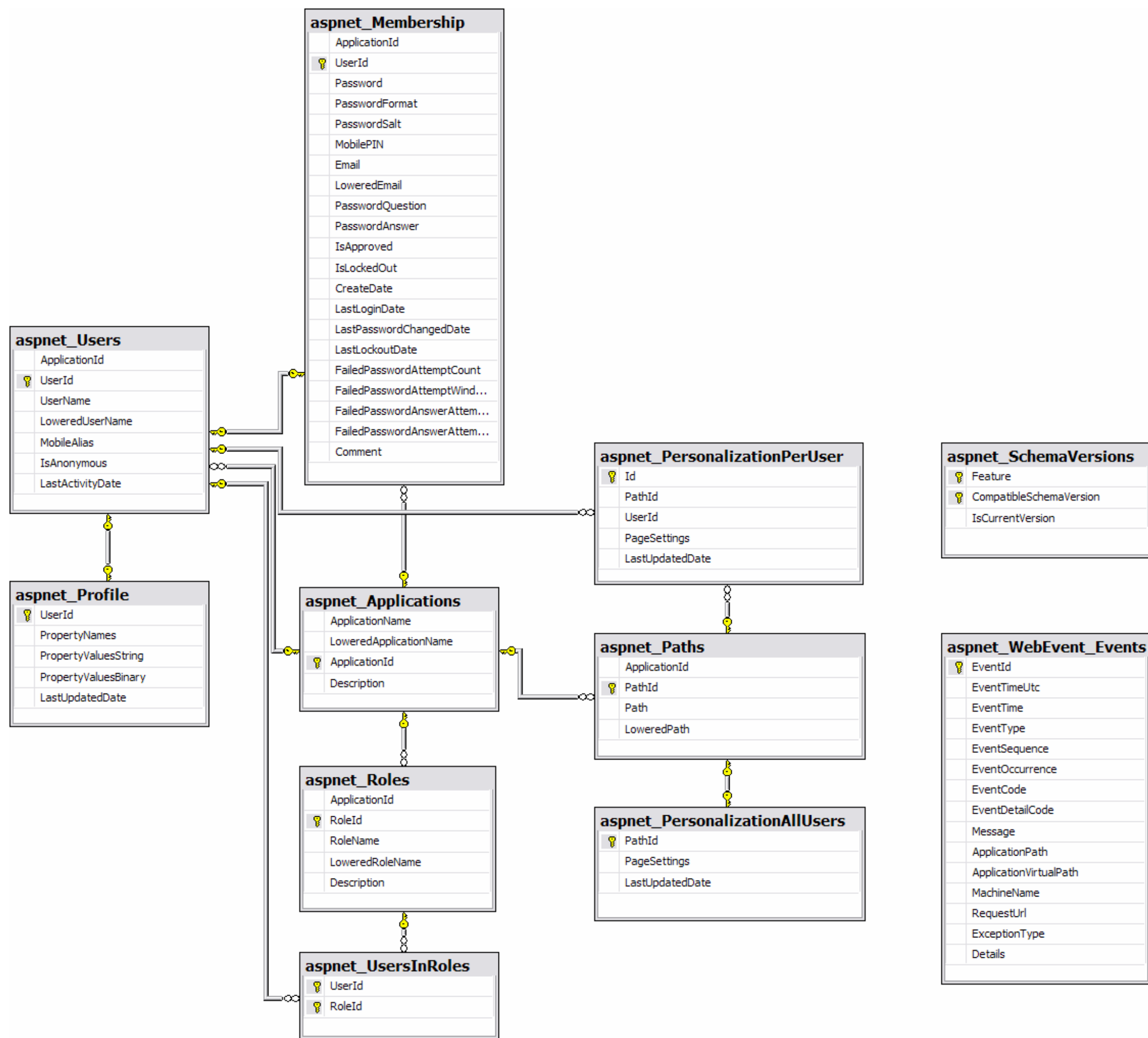
This whitepaper documents the design and operation of many of the built-in providers. It supplements the providers' source code and contains helpful insights for developers writing custom providers of their own.

The SQL Provider Database

Many of the Microsoft ASP.NET 2.0 providers are SQL providersproviders that persist state in SQL Server (or SQL Server Express) databases. The SQL providers include *SqlMembershipProvider*, *SqlRoleProvider*, *SqlProfileProvider*, *SqlSessionStateStore*, *SqlWebEventProvider*, and *SqlPersonalizationProvider*. Each stores data using a predefined schema. The Aspnet_regsql.exe tool that comes with ASP.NET 2.0 creates a SQL Server database with a compatible schema. That database, which is named aspnetdb by default, will hereafter be referred to as the *SQL provider database* or simply the *provider database*.

Figure 2 shows the structure of the SQL provider database. Some of the tables are provider-specific. The aspnet_Membership table, for example, is used exclusively by *SqlMembershipProvider*, whereas the aspnet_Roles and aspnet_UsersInRoles tables are used exclusively by *SqlRoleProvider*.

Figure 2. The SQL provider database



Other tables are not provider-specific, but instead exist for the benefit of multiple SQL providers. The aspnet_Applications table is a great example. Many SQL providers support scoping of data through the *ApplicationName* property, which is initialized from the *applicationName* configuration attribute supported by many providers . For example, websites that register membership providers with identical *applicationName* attributes share membership data, whereas websites that register membership providers with unique *applicationName*s do not. SQL providers that support *ApplicationName* scoping do so by storing application IDs associated with the records that they create, and by including those application IDs in queries performed on the SQL provider database. Application IDs stored in aspnet_Membership, aspnet_Paths, and other provider-specific tables refer to the aspnet_Applications table, which contains a list of extant application IDs and the corresponding application names. Table 2 documents the schema of the aspnet_Applications table. The provider database contains a stored procedure named aspnet_Applications_CreateApplication that providers (or stored procedures) can call to retrieve an application ID from the aspnet_Applications table, or to create a new one if the specified application doesn't exist.

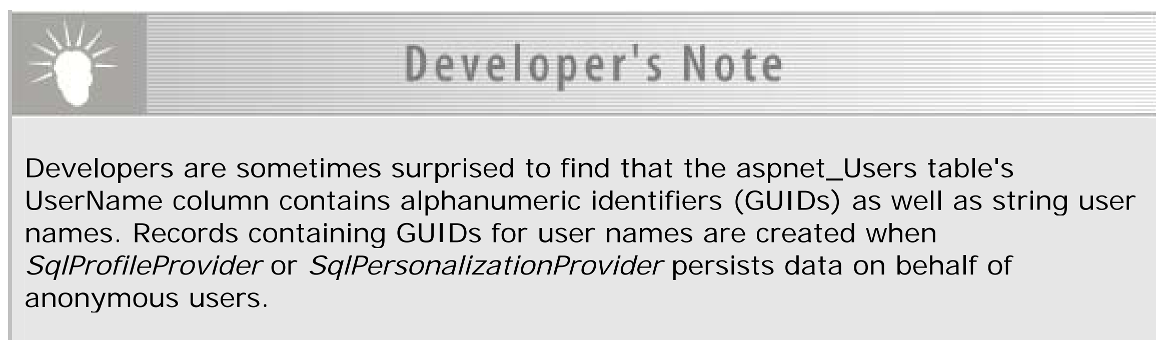
Table 2. The aspnet_Applications table

Column Name	Column Type	Description
ApplicationId	uniqueidentifier	Application ID
ApplicationName	nvarchar(256)	Application name
LoweredApplicationName	nvarchar(256)	Application name (lowercase)
Description	nvarchar(256)	Application description

aspnet_Users is another example of a table that's shared by SQL providers. It stores core provider-agnostic information regarding users, including user names and user IDs. *SqlMembershipProvider* stores membership-user data in the aspnet_Membership table, but that table contains a UserId column that refers to the column of the same name in aspnet_Users. Similarly, *SqlRoleProvider* stores data mapping users to roles in the aspnet_UsersInRoles table, and that table contains both a UserId column referring to the column of the same name in aspnet_Users, and a RoleId column referring to the column of the same name in aspnet_Roles. Table 3 documents the schema of the aspnet_Users table.

Table 3. The aspnet_Users table

Column Name	Column Type	Description
ApplicationId	uniqueidentifier	Application ID
UserId	uniqueidentifier	User ID
UserName	nvarchar(256)	User name
LoweredUserName	nvarchar(256)	User name (lowercase)
MobileAlias	nvarchar(16)	User's mobile alias (currently not used)
IsAnonymous	bit	*1*=Anonymous user, *0*=Not an anonymous user
LastActivityDate	datetime	Date and time of last activity by this user



Developers are sometimes surprised to find that the aspnet_Users table's UserName column contains alphanumeric identifiers (GUIDs) as well as string user names. Records containing GUIDs for user names are created when *SqlProfileProvider* or *SqlPersonalizationProvider* persists data on behalf of anonymous users.

The SQL providers never access tables in the provider database directly. Instead, they use stored procedures. When *SqlMembershipProvider*'s *CreateUser* method is called, for example, it calls a stored procedure named aspnet_Membership_CreateUser to add a new membership user to the provider database. aspnet_Membership_CreateUser adds a

record representing that user to the aspnet_Membership table, another record representing that user to the aspnet_Users table, and, if necessary, a record denoting a new application to the aspnet_Applications table. The use of stored procedures hides the database schema from the provider, which simplifies porting SQL providers to other database types (for example, Oracle databases), and to SQL Server databases that utilize custom schemas. Stored procedures that perform multistep updates typically use database transactions to roll back changes if an error occurs before the last step is completed. (There are a few cases in which providers manage transactions themselves in order to support batch deletes.)

SQL Server Express

Rather than use a pre-existing SQL provider database, the Microsoft SQL providers are equally happy to use a database managed by SQL Server Express, herafter referred to as the *express database*.

Internally, the express database has the same schema as the SQL provider database. The difference between the databases lies in how they're created. The SQL provider database is created externally when you run Aspnet_regsql.exe or an equivalent tool. The express database is created automatically the first time it's needed.

Each Microsoft SQL provider (with the exception of *SqlSessionStateStore*, which doesn't support express databases) has logic built in to automatically create the express database. The logic lives in a helper class named *SqlConnectionHelper*. Rather than create *SqlConnections* from raw connection strings, Microsoft SQL providers pass connection strings to *SqlConnectionHelper.GetConnection*, as follows:

```
SqlConnectionHolder holder = SqlConnectionHelper.GetConnection(
_sqlConnectionString, true );
```

SqlConnectionHelper.GetConnection parses the connection string and automatically creates the express database if the connection string meets certain criteria, and if the database doesn't already exist.

Developer's Note

When a Microsoft SQL provider needs an actual *SqlConnection*, it extracts it from the *Connection* property of the *SqlConnectionHolder*, as follows:

```
SqlCommand cmd = new SqlCommand("dbo.aspnet_Membership_CreateUser",
holder.Connection);
```

Similarly, it closes the connection by calling *SqlConnectionHolder.Close*.

The main purpose of the *SqlConnectionHolder* class is to simplify the security model when SQL providers are used in a website with client impersonation

enabled. *SqlConnectionHolder* encapsulates logic that temporarily reverts the thread identity to that of the current process identity or application impersonation identity when connecting to SQL Server. As a result, SQL providers run with a trusted subsystem model that doesn't require individual users to have access rights to the provider database.

The default LocalSqlServer connection string in Machine.config is an excellent example of a connection string that results in automatic creation of the express database:

```
data source=.\SQLEXPRESS;Integrated
Security=SSPI;AttachDBFilename=|DataDirectory|aspnetdb.mdf;User
Instance=true
```

The presence of *User Instance=true* and *AttachDBFilename=|DataDirectory|* cause *SqlConnectionHelper* to conclude that the connection string targets SQL Server Express and triggers the database's creation. (The presence of *data source=.\SQLEXPRESS* in the connection string does not factor into the decision, because *SqlConnectionHelper* supports non-default as well as default instances of SQL Server Express.) The *|DataDirectory|* portion of the connection string specifies that the MDF file is located inthe App_Data directory. *SqlConnectionHelper* derives the database name from the MDF file name. It also creates an App_Data folder to hold the MDF if the folder doesn't already exist.

When *SqlConnectionHelper* creates the express database, it sets *SIZE* to *10* (10 MB) and *FILEGROWTH* to *50%*. The database's sort order, case sensitivity, accent sensitivity, and other locale-dependent settings are inherited from the default SQL Server Express instance, which ensures that the database locale is consistent with that of the host computer.

Developer's Note

The extra overhead incurred by checking for the existence of the express database before using it means that Microsoft SQL providers run marginally slower against SQL Server Express than SQL Server. Hopefully, the small performance loss is offset by the added convenience of automatically created express databases.

Membership Providers

Membership providers provide the interface between Microsoft ASP.NET's membership service and membership data sources. ASP.NET 2.0 ships with two membership providers:

- *SqlMembershipProvider*, which stores membership data in Microsoft SQL Server and SQL Server Express databases
- *ActiveDirectoryMembershipProvider*, which retrieves membership data from Microsoft Active Directory

The fundamental job of a membership provider is to manage the data regarding a site's registered users, and to provide methods for creating users, deleting users, verifying login credentials, changing passwords, and so on. The Microsoft .NET Framework's *System.Web.Security* namespace includes a class named *MembershipUser* that defines the basic attributes of a membership user, and that a membership provider uses to represent individual users. It also includes a base class named *MembershipProvider* that defines the basic characteristics of a membership provider. *MembershipProvider* is prototyped as follows:

```
public abstract class MembershipProvider : ProviderBase
{
    // Abstract properties
    public abstract bool EnablePasswordRetrieval { get; }
    public abstract bool EnablePasswordReset { get; }
    public abstract bool RequiresQuestionAndAnswer { get; }
    public abstract string ApplicationName { get; set; }
    public abstract int MaxInvalidPasswordAttempts { get; }
    public abstract int PasswordAttemptWindow { get; }
    public abstract bool RequiresUniqueEmail { get; }
    public abstract MembershipPasswordFormat PasswordFormat { get; }
    public abstract int MinRequiredPasswordLength { get; }
    public abstract int MinRequiredNonAlphanumericCharacters { get; }
    public abstract string PasswordStrengthRegularExpression { get; }

    // Abstract methods
    public abstract MembershipUser CreateUser (string username,
        string password, string email, string passwordQuestion,
        string passwordAnswer, bool isApproved, object providerUserKey,
        out MembershipCreateStatus status);

    public abstract bool ChangePasswordQuestionAndAnswer
        (string username, string password,
        string newPasswordQuestion, string newPasswordAnswer);

    public abstract string GetPassword (string username,
```

```
  string answer);

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

public abstract string ResetPassword (string username,
    string answer);

public abstract void UpdateUser (MembershipUser user);

public abstract bool ValidateUser (string username,
    string password);

public abstract bool UnlockUser (string userName);

public abstract MembershipUser GetUser (object providerUserKey,
    bool userIsOnline);

public abstract MembershipUser GetUser (string username,
    bool userIsOnline);

public abstract string GetUserNameByEmail (string email);

public abstract bool DeleteUser (string username,
    bool deleteAllRelatedData);

public abstract MembershipUserCollection GetAllUsers
    (int pageIndex, int pageSize, out int totalRecords);

public abstract int GetNumberOfUsersOnline ();

public abstract MembershipUserCollection FindUsersByName
    (string usernameToMatch, int pageIndex, int pageSize,
    out int totalRecords);

public abstract MembershipUserCollection FindUsersByEmail
    (string emailToMatch, int pageIndex, int pageSize,
    out int totalRecords);

// Virtual methods
protected virtual byte[] EncryptPassword (byte[] password);
protected virtual byte[] DecryptPassword (byte[] encodedPassword);
protected virtual void OnValidatingPassword
    (ValidatePasswordEventArgs e);
```

```
    // Events
    public event MembershipValidatePasswordEventHandler
        ValidatingPassword;
}
```

The following sections document the implementation of *SqlMembershipProvider*, which derives from *MembershipProvider*.

SqlMembershipProvider

SqlMembershipProvider is the Microsoft membership provider for SQL Server databases. It stores membership data using the schema documented in "Data Schema," and it uses the stored procedures documented in "Data Access." All knowledge of the database schema is hidden in the stored procedures, so porting *SqlMembershipProvider* to other database types requires little more than modifying the stored procedures. (Depending on the targeted database type, the ADO.NET code used to call the stored procedures might have to change, too. The Microsoft Oracle .NET provider, for example, uses a different syntax for named parameters.)

The ultimate reference for *SqlMembershipProvider* is the *SqlMembershipProvider* source code, which is found in SqlMembershipProvider.cs. The sections that follow highlight key aspects of *SqlMembershipProvider*'s design and operation.

Provider Initialization

Initialization occurs in *SqlMembershipProvider.Initialize*, which is called one timewhen the provider is loadedby ASP.NET. *SqlMembershipProvider.Initialize*'s duties include:

- Initializing the various *SqlMembershipProvider* properties such as *EnablePasswordRetrieval* and *EnablePasswordReset* from the corresponding configuration attributes (*enablePasswordRetrieval*, *enablePasswordReset*, and so on).
- Performing common-sense checks on the property valuesfor example, throwing an exception if *PasswordFormat* is "hashed" (*MembershipPasswordFormat.Hashed*) but *EnablePasswordRetrieval* is true. (By definition, passwords can't be computed from password hashes.)
- Throwing an exception if unrecognized configuration attributes remain after all supported configuration attributes are processed.

SqlMembershipProvider.Initialize also reads the connection string identified by the *connectionStringName* attribute from the *<connectionStrings>* configuration section, and caches it in a private field. It throws a *ProviderException* if the attribute is empty or nonexistent, or if the attribute references a nonexistent connection string.

Data Schema

SqlMembershipProvider stores membership data in the aspnet_Membership table of the provider database. Each record in aspnet_Membership corresponds to one membership user. Table 4 documents the aspnet_Membership table's schema.

Table 4. The aspnet_Membership table

Column Name	Column Type	Description
ApplicationId	uniqueidentifier	Application ID
UserId	uniqueidentifier	User ID
Password	nvarchar(128)	Password (plaintext, hashed, or encrypted; base-64-encoded if hashed or encrypted)
PasswordFormat	int	Password format (*0*=Plaintext, *1*=Hashed, *2*=Encrypted)
PasswordSalt	nvarchar(128)	Randomly generated 128-bit value used to salt password hashes; stored in base-64-encoded form
MobilePIN	nvarchar(16)	User's mobile PIN (currently not used)
Email	nvarchar(256)	User's e-mail address
LoweredEmail	nvarchar(256)	User's e-mail address (lowercase)
PasswordQuestion	nvarchar(256)	Password question
PasswordAnswer	nvarchar(128)	Answer to password question
IsApproved	bit	*1*=Approved, *0*=Not approved
IsLockedOut	bit	*1*=Locked out, *0*=Not locked out
CreateDate	datetime	Date and time this account was created
LastLoginDate	datetime	Date and time of this user's last login
LastPasswordChangedDate	datetime	Date and time this user's password was last changed
LastLockoutDate	datetime	Date and time this user was last locked out
FailedPasswordAttemptCount	int	Number of consecutive failed login attempts
FailedPasswordAttempt-WindowStart	datetime	Date and time of first failed login if FailedPasswordAttemptCount is nonzero
FailedPasswordAnswer-AttemptCount	int	Number of consecutive failed password answer attempts
FailedPasswordAnswer-AttemptWindowStart	datetime	Date and time of first failed password answer if

		FailedPasswordAnswerAttemptCount is nonzero
Comment	ntext	Additional text

The aspnet_Membership table has foreign-key relationships with two other provider database tables: aspnet_Applications (see Table 2) and aspnet_Users (see Table 3). The aspnet_Membership table's ApplicationId column references the column of the same name in the aspnet_Applications table. (Although this column is not strictly necessary, because the UserId can be used to derive the ApplicationId, the ApplicationId column was added to the aspnet_Membership table to speed up queries and reduce the need to join through to the aspnet_Users table.) aspnet_Membership's UserId column references the column of the same name in the aspnet_Users table. A complete record for a given membership user consists of data corresponding to that user's user ID in the aspnet_Users table, and data corresponding to the same user ID in the aspnet_Membership table. Stored procedures such as aspnet_Membership_GetUserByName pull data from both tables to create *MembershipUser* objects representing individual users.

Scoping of Membership Data

Websites that register membership providers with identical *applicationName* attributes share membership data, whereas websites that register membership providers with unique *applicationNames* do not. To that end, *SqlMembershipProvider* records an application ID in the *ApplicationId* field of each record in the aspnet_Membership table. aspnet_Membership's *ApplicationId* field refers to the field of the same name in the aspnet_Applications table, and each unique *applicationName* has a corresponding *ApplicationId* in that table.

Data Access

SqlMembershipProvider performs all database accesses through stored procedures. Table 5 lists the stored procedures that it uses.

Table 5. Stored procedures used by SqlMembershipProvider

Stored Procedure	**Description**
aspnet_Membership_ChangePassword-QuestionAndAnswer	Changes the specified user's password question and answer.
aspnet_Membership_CreateUser	Adds a new membership user to the membership database. Records the user in the aspnet_Users and aspnet_Membership tables and, if necessary, adds a new application to the aspnet_Applications table.
aspnet_Membership_FindUsersByEmail	Retrieves records from aspnet_Membership table with e-mail addresses matching the

	specified pattern and with the specified application ID.
aspnet_Membership_FindUsersByName	Retrieves records from aspnet_Membership table with user names matching the specified pattern and with the specified application ID.
aspnet_Membership_GetAllUsers	Retrieves all users from the aspnet_Membership table with the specified application ID.
aspnet_Membership_GetNumberOfUsersOnline	Gets the number of users currently online (those whose last activity dates.
aspnet_Membership_GetPassword	Gets the specified user's password data from the database. Used for retrieving passwords with a user-supplied password answer.
aspnet_Membership_GetPasswordWithFormat	Gets the specified user's password from the database. Used by the provider to retrieve passwords for performing password comparisons (for example, when *ValidateUser* needs to validate a password).
aspnet_Membership_GetUserByEmail	Given an e-mail address and application ID, retrieves the corresponding record from the aspnet_Membership table.
aspnet_Membership_GetUserByName	Given a user name and application ID, retrieves the corresponding record from the aspnet_Membership table.
aspnet_Membership_GetUserByUserId	Given a user ID and application ID, retrieves the corresponding record from the aspnet_Membership table.
aspnet_Membership_ResetPassword	Resets the specified user's password based on a password answer.
aspnet_Membership_SetPassword	Sets the specified user's password to the password input to the stored procedure.
aspnet_Membership_UnlockUser	Restores login privileges for the specified user by setting the user's

	IsLockedOut bit to *0*.
aspnet_Membership_UpdateUser	Updates the user's last activity date in the aspnet_Users table and e-mail address, comment, is-approved status, and last login date in the aspnet_Membership table.
aspnet_Membership_UpdateUserInfo	Updates account locking data for the specified user in the aspnet_Users and aspnet_Membership tables. Used in conjunction with provider methods that track bad password and bad password-answer attempts.
aspnet_Users_CreateUser	Adds a user to the aspnet_Users table. Called by aspnet_Membership_CreateUser.
aspnet_Users_DeleteUser	Deletes a user from the aspnet_Membership table and optionally from other SQL provider tables, including aspnet_Users.

Stored procedure names are generally indicative of the *SqlMembershipProvider* methods that call them. For example, applications call the membership service's *Membership.CreateUser* method to register new users. *Membership.CreateUser*, in turn, delegates to the *CreateUser* method of the default membership provider, which, in the case of *SqlMembershipProvider*, validates the input parameters and calls aspnet_Membership_CreateUser to register a new user.

Creating Membership Users

SqlMembershipProvider.CreateUser calls the stored procedure aspnet_Membership_CreateUser to create new membership users. Before calling the stored procedure, *SqlMembershipProvider.CreateUser* validates the input parameters, encodes the password (and, if present, the password answer) provided to it, and fires an *OnValidatingPassword* event. Then aspnet_Membership_CreateUser performs the following tasks:

1. Calls the stored procedure aspnet_Applications_CreateApplication to convert the application name passed to it (which comes from the provider's *ApplicationName* property) into an application ID. If the application name already appears in the aspnet_Applications table, aspnet_Applications_CreateApplication returns the existing application ID. If the application name is *not* already present in the aspnet_Applications table, aspnet_Applications_CreateApplication adds a new application to aspnet_Applications and returns the application ID.

2. Calls aspnet_Users_CreateUser to insert a record representing the new user into the aspnet_Users table.
3. Performs an optional check to ensure that the new user's e-mail address is unique with respect to other registered e-mail addresses.
4. Updates the *LastActivityDate* field in the aspnet_Users table with the current time and date.
5. Inserts a record representing the new user into the aspnet_Membership table.

aspnet_Membership_CreateUser performs all these steps within a transaction to ensure that changes are committed as a group or not at all.

Deleting Membership Users

Applications call the membership service's *Membership.DeleteUser* method to delete membership users. *Membership.DeleteUser* calls the default membership provider's *DeleteUser* method, which takes a user name as input and also accepts a bool named *deleteAllRelatedData* that specifies whether other data associated with the specified user should be deleted in addition to membership data. "Other data" includes role data, profile data (including anonymous profile datamore on this later), and Web Parts personalization data.

SqlMembershipProvider.DeleteUser calls the stored procedure aspnet_Users_DeleteUser to delete membership users. In addition to accepting a user name, aspnet_Users_DeleteUser accepts a bit mask named @TablesToDeleteFrom that specifies which provider database tables the user should be deleted from. If *deleteAllRelatedData* is *false*, *SqlMembershipProvider.DeleteUser* passes a bit mask of *1*, prompting aspnet_Users_DeleteUser to delete the user only from the aspnet_Membership table. However, if *deleteAllRelatedData* is *true*, *SqlMembershipProvider.DeleteUser* passes a bit mask of *15* (binary 1111), prompting aspnet_Users_DeleteUser to delete the specified user from the aspnet_Membership, aspnet_UsersInRoles, aspnet_Profile, aspnet_PersonalizationPerUser, and aspnet_Users tables. aspnet_Users_DeleteUser uses a database transaction to ensure that the deletions are performed in whole or not at all.

Another little known fact is that *Membership.DeleteUser* can be used to clean up the records that accrue in the aspnet_Users and aspnet_Profile tables when using the anonymous identification feature to store profile data on behalf of anonymous users. Simply call *Membership.DeleteUser* with *deleteAllRelatedData* set to *true*, and *username* set to *Request.AnonymousID*. This deletes the anonymous user's data from the aspnet_Profile table, *and* it deletes the base user record from aspnet_Users.

Validating Membership Users

Applications call the membership service's *Membership.ValidateUser* method to validate membership usersthat is, to verify that a given user name and password corresponds to a registered membership user. *Membership.ValidateUser* calls the default membership provider's *ValidateUser* method, which returns *true* or *false*, indicating whether the user name and password are valid.

SqlMembershipProvider.ValidateUser performs the following tasks:

1. Calls the stored procedure aspnet_Membership_GetPasswordWithFormat to retrieve the user's password from the database. If the password is hashed or encrypted, it is returned in encoded ("formatted") form; otherwise, it's returned as plaintext.
2. Encodes the password input to *ValidateUser* using the same encoding, if any, used to encode the password retrieved in the previous step.
3. Compares the password retrieved from the database to the encoded input password.
4. If the passwords match, *ValidateUser* raises an *AuditMembershipAuthenticationSuccess* Web event, increments a performance counter that tracks successful logins, and returns true.
5. If the passwords don't match, *ValidateUser* raises an *AuditMembershipAuthenticationFailure* Web event, increments a performance counter that tracks failed logins, and returns false. It also calls aspnet_Membership_UpdateUserInfo to update the aspnet_Membership table with information about the failed login, so that the account can be locked if too many failed logins occur within the time span indicated by the provider's *PasswordAttemptWindow* property.

Account locking is a feature of *SqlMembershipProvider* that provides a safeguard against password guessing. It is described in "Account Locking."

Password Protection

Applications that store user names, passwords, and other authentication information in a database should never store passwords in plaintext, lest the database be stolen or compromised. To that end, *SqlMembershipProvider* supports three storage formats ("encodings") for passwords and password answers. The provider's *PasswordFormat* property, which is initialized from the *passwordFormat* configuration attribute, determines which format is used:

- *MembershipPasswordFormat.Clear*, which stores passwords and password answers in plaintext.
- *MembershipPasswordFormat.Hashed* (the default), which stores salted hashes generated from passwords and password answers. The salt is a random 128-bit value generated by the .NET Framework's *RNGCryptoServiceProvider* class. Each password/password answer pair is salted with this unique value, and the salt is stored in the aspnet_Membership table's *PasswordSalt* field. The result of hashing the password and the salt is stored in the *Password* field. Similarly, the result of hashing the password answer and the salt is stored in the *PasswordAnswer* field.
- *MembershipPasswordFormat.Encrypted*, which stores encrypted passwords and password answers. *SqlMembershipProvider* encrypts passwords and password answers using the symmetric encryption/decryption key specified in the <machineKey> configuration section's *decryptionKey* attribute, and the encryption algorithm specified in the <machineKey> configuration section's *decryption* attribute. *SqlMembershipProvider* throws an exception if it is asked to encrypt passwords and password answers, and if *decryptionKey* is set to *Autogenerate*. This prevents a membership database containing encrypted passwords and password answers from becoming invalid if moved to another server or another application.

Developer's Note

Storing unsalted password hashes leaves password databases vulnerable to dictionary attacks. *SqlMembershipProvider* salts password hashes as a hedge against such attacks. However, the fact that *SqlMembershipProvider* stores salts in the database alongside the password hashes means that the salt's effective key space is a function not of the length of the salt, but of the number of salts in the database. In other words, the more records the aspnet_Membership table contains, the more secure the password hashes.

If desired, a custom membership provider could use an altogether different strategy for storing saltsone whose security did not depend on the volume of membership data. It could, for example, use the same randomly generated salt for every hash, but protect the salt by storing it outside the databaseperhaps in an ACLed registry key.

To provide additional protection against password hacking, *SqlMembershipProvider* also supports user-configurable password strengths. *CreateUser* and other methods that modify passwords (*ChangePassword* and *ResetPassword*) validate the passwords against the provider's *MinRequiredPasswordLength* and *MinRequiredNonAlphanumericCharacters* properties. *SqlMembershipProvider* defaults these properties to *7* and *1*, respectively. In addition, *SqlMembershipProvider* validates passwords against the regular expression, if any, stored in the *PasswordStrengthRegularExpression* property, as follows:

```
if( PasswordStrengthRegularExpression.Length > 0 )
{
    if( !Regex.IsMatch( password, PasswordStrengthRegularExpression ) )
    {
        status = MembershipCreateStatus.InvalidPassword;
        return null;
    }
}
```

The combination of non-plaintext password storage formats and user-configurable password strengths enables *SqlMembershipProvider* to store passwords as securely as is practically possible.

Account Locking

To guard against password guessing, *SqlMembershipProvider* supports the automatic locking of accounts that incur suspicious activity. If, for a given user, the number of consecutive invalid passwords or password answers submitted to methods such as *ValidateUser* and *GetPassword* exceeds the value stored in the provider's *MaxInvalidPasswordAttempts* property, and if the consecutive attempts occur within the

time period specified by the *PasswordAttemptWindow* property, *SqlMembershipProvider* sets the corresponding *IsLockedOut* field in the aspnet_Membership table to *1*. Further logins by the affected user are disallowed until *IsLockedOut* is reset to *0* by calling the provider's *UnlockUser* method. *SqlMembershipProvider* defaults *MaxInvalidPasswordAttempts* to *5* and *PasswordAttemptWindow* to *10* (that is, 10 minutes).

As an example of how account locking is implemented, suppose a user submits a password from a login page that uses a *Login* control to validate logins. *Login* controls call *Membership.ValidateUser*. *Membership.ValidateUser* calls *SqlMembershipProvider.ValidateUser* (assuming *SqlMembershipProvider* is the default membership provider), which calls the private *SqlMembershipProvider.CheckPassword* method to validate passwords. *CheckPassword* uses aspnet_Membership_GetPasswordWithFormat to retrieve an encoded password. The stored procedure checks the IsLockedOut bit of the record it retrieves from the database, and returns an error code of *99* if *IsLockedOut* is set. The error code causes *CheckPassword* to return *false*. That causes *ValidateUser* to return *false*, which in turn prevents the user from logging ineven if the password he or she typed was valid.

How does an account become locked in the first place? Suppose the user types an incorrect password into the login page. After ascertaining that the password is invalid, *CheckPassword* calls the stored procedure aspnet_Membership_UpdateUserInfo to update the corresponding record in the aspnet_Membership table. It passes in a bit flag indicating an invalid password was submitted. Seeing the flag, the stored procedure increments the failed password attempt count. If the count exceeds the maximum specified by *MaxInvalidPasswordAttempts*, and if all the password failures occurred within the time window specified by *PasswordAttemptWindow*, the stored procedure sets *IsLockedOut* to *1*, effectively locking the account until further notice. Thus, locking is handled primarily at the database level, and it is largely opaque to the provider itself.

Differences Between the Published Source Code and the .NET Framework's SqlMembershipProvider

The published source code for *SqlMembershipProvider* differs from the .NET Framework version in the following respects:

- Declarative and imperative CAS demands were commented out. Because the source code can be compiled standalone, and thus will run as user code rather than trusted code in the global assembly cache, the CAS demands are not strictly necessary. For reference, however, the original demands from the .NET Framework version of the provider have been retained as comments.
- The performance counter and Web event code in *ValidateUser* has been commented out, because the .NET Framework provider relies on internal helper classes to manipulate these counters. For reference, the original code has been retained as comments.
- The internal helper methods *EncodePassword* and *UnEncodePassword* have been included in the accompanying source code. In the .NET Framework, these are actually internal helper methods implemented by the base *MembershipProvider* type.

Role Providers

Role providers provide the interface between Microsoft ASP.NET's role management service (the "role manager") and role data sources. ASP.NET 2.0 ships with three role providers:

- **SqlRoleProvider**, which stores role data in Microsoft SQL Server and Microsoft SQL Server Express databases
- **AuthorizationStoreRoleProvider**, which retrieves role information from Microsoft Authorization Manager ("AzMan")
- **WindowTokenRoleProvider**, which retrieves role information from each user's Microsoft Windows authentication token, and returns his or her group membership(s).

The fundamental job of a role provider is to interface with data sources containing containing role data mapping users to roles, and to provide methods for creating roles, deleting roles, adding users to roles, and so on. The Microsoft .NET Framework's *System.Web.Security* namespace includes a class named *RoleProvider* that defines the basic characteristics of a role provider. *RoleProvider* is prototyped as follows:

```
public abstract class RoleProvider : ProviderBase
{
    // Abstract properties
    public abstract string ApplicationName { get; set; }

    // Abstract methods
    public abstract bool IsUserInRole (string username,
        string roleName);
    public abstract string[] GetRolesForUser (string username);
    public abstract void CreateRole (string roleName);
    public abstract bool DeleteRole (string roleName,
        bool throwOnPopulatedRole);
    public abstract bool RoleExists (string roleName);
    public abstract void AddUsersToRoles (string[] usernames,
        string[] roleNames);
    public abstract void RemoveUsersFromRoles (string[] usernames,
        string[] roleNames);
    public abstract string[] GetUsersInRole (string roleName);
    public abstract string[] GetAllRoles ();
    public abstract string[] FindUsersInRole (string roleName,
        string usernameToMatch);
}
```

The following sections document the implementation of *SqlRoleProvider* and *AuthorizationStoreRoleProvider*, both of which derive from *RoleProvider*.

SqlRoleProvider

SqlRoleProvider is the Microsoft role provider for SQL Server databases. It stores role data, using the schema documented in "Data Schema," and it uses the stored procedures documented in "Data Access." All knowledge of the database schema is hidden in the stored procedures, so that porting *SqlRoleProvider* to other database types requires little more than modifying the stored procedures. (Depending on the targeted database type, the ADO.NET code used to call the stored procedures might have to change, too. The Microsoft Oracle .NET provider, for example, uses a different syntax for named parameters.)

The ultimate reference for *SqlRoleProvider* is the *SqlRoleProvider* source code, which is found in SqlRoleProvider.cs. The sections that follow highlight key aspects of *SqlRoleProvider*'s design and operation.

Provider Initialization

Initialization occurs in *SqlRoleProvider.Initialize*, which is called one timewhen the provider is loadedby ASP.NET. *SqlRoleProvider.Initialize* processes the configuration attributes *applicationName*, *connectionStringName*, and *commandTimeout*, and throws a *ProviderException* exception if unrecognized configuration attributes remain.

SqlRoleProvider.Initialize also reads the connection string identified by the *connectionStringName* attribute from the *<connectionStrings>* configuration section, and caches it in a private field. It throws a *ProviderException* if the attribute is empty or nonexistent, or if the attribute references a nonexistent connection string.

Data Schema

SqlRoleProvider persists roles in the aspnet_Roles table of the provider database. Each record in aspnet_Roles corresponds to one role. The ApplicationId column refers to the column of the same name in the aspnet_Applications table, and it is used to scope roles by application. Table 6 documents the aspnet_Roles table's schema.

Table 6. The aspnet_Roles table

Column Name	Column Type	Description
ApplicationId	uniqueidentifier	Application ID
RoleId	uniqueidentifier	Role ID
RoleName	nvarchar(256)	Role name
LoweredRoleName	nvarchar(256)	Role name (lowercase)
Description	nvarchar(256)	Role description (currently unused)

SqlRoleProvider uses a separate table named aspnet_UsersInRoles to map roles to users. The UserId column identifies a user in the aspnet_Users table, whereas the RoleId column identifies a role in the aspnet_Roles table. The structure of this table, which is documented in Table 7, lends itself to the types of queries performed by a role provider. With a single query, for example, a role provider could select all the users belonging to a given role, or all the roles assigned to a given user.

Table 7. The aspnet_UsersInRoles table

Column Name	Column Type	Description
UserId	uniqueidentifier	User ID
RoleId	uniqueidentifier	Role ID

Scoping of Role Data

Websites that register role providers with identical *applicationName* attributes share role data, whereas websites that register role providers with unique *applicationNames* do not. To that end, *SqlRoleProvider* records an application ID in the *ApplicationId* field of each record in the aspnet_Roles table. aspnet_Roles' *ApplicationId* field refers to the field of the same name in the aspnet_Applications table, and each unique *applicationName* has a corresponding *ApplicationId* in that table.

Data Access

SqlRoleProvider performs all database accesses through stored procedures. Table 8 lists the stored procedures that it uses.

Table 8. Stored procedures used by SqlRoleProvider

Stored Procedure	Description
aspnet_Roles_CreateRole	Adds a role to the aspnet_Roles table and, if necessary, adds a new application to the aspnet_Applications table.
aspnet_Roles_DeleteRole	Removes a role from the aspnet_Roles table. Optionally deletes records referencing the deleted role from the aspnet_UsersInRoles table.
aspnet_Roles_GetAllRoles	Retrieves all roles with the specified application ID from the aspnet_Roles table.
aspnet_Roles_RoleExists	Checks the aspnet_Roles table to determine whether the specified role exists.
aspnet_UsersInRoles_AddUsersToRoles	Adds the specified users to the specified roles by adding them to the aspnet_UsersInRoles table.
aspnet_UsersInRoles_FindUsersInRole	Queries the aspnet_UsersInRoles table for all users belonging to the specified role whose user

	names match the specified pattern.
aspnet_UsersInRoles_GetRolesForUser	Queries the aspnet_UsersInRoles table for all roles assigned to a specified user.
aspnet_UsersInRoles_GetUsersInRoles	Queries the aspnet_UsersInRoles table for all users belonging to the specified role.
aspnet_UsersInRoles_IsUserInRole	Checks the aspnet_UsersInRoles table to determine whether the specified user belongs to the specified role.
aspnet_UsersInRoles_RemoveUsersFromRoles	Removes the specified users from the specified roles by deleting the corresponding records from the aspnet_UsersInRoles table.

Stored procedure names are generally indicative of the *SqlRoleProvider* methods that call them. For example, applications call the role manager's *Roles.CreateRole* method to create new roles. *Roles.CreateRole*, in turn, delegates to the *CreateRole* method of the default role provider, which, in the case of *SqlRoleProvider*, validates the input parameters, and calls aspnet_Roles_CreateRole to create a new role.

Creating Roles

SqlRoleProvider.CreateRole calls the stored procedure aspnet_Roles_CreateRole, which performs the following tasks:

1. Calls aspnet_Applications_CreateApplication to retrieve an application ID (or create a new one).
2. Verifies that the specified role doesn't already existthat is, that it's not already defined in the aspnet_Roles table.
3. Inserts a record representing the new role into the aspnet_Roles table.

aspnet_Roles_CreateRole performs all these steps within a transaction to ensure that changes are committed as a group or not at all.

Deleting Roles

Applications call the role manager's *Roles.DeleteRole* method to delete roles. *Roles.DeleteRole* calls the default role provider's *DeleteRole* method, and passes in a flag named *throwOnPopulatedRole* that indicates whether *DeleteRole* should throw an exception if the role being deleted isn't emptythat is, if one or more users are assigned to it.

SqlRoleProvider.DeleteRole calls the stored procedure aspnet_Roles_DeleteRole. The stored procedure performs the following tasks:

1. Verifies that the role to be deleted exists.
2. If *throwOnPopulatedRole* is true, checks the aspnet_UsersInRoles table for records containing the specified role ID, and returns an error code if the query turns up one or more records.
3. Deletes all records containing the specified role ID from the aspnet_UsersInRole table.
4. Deletes all records containing the specified role ID from the aspnet_Roles table.

aspnet_Roles_DeleteRole performs all these steps within a transaction to ensure that changes are committed as a group or not at all.

Adding Users to Roles

Applications call the role manager's *Roles.AddUserToRole*, *Roles.AddUserToRoles*, *Roles.AddUsersToRole*, or *Roles.AddUsersToRoles* method to add users to roles. These methods, in turn, call the default role provider's *AddUsersToRoles* method. *SqlRoleProvider.AddUsersToRoles* converts the arrays of user names and role names in the parameter list into comma-delimited lists, and passes them to the stored procedure aspnet_UsersInRoles_AddUsersToRoles.

aspnet_UsersInRoles_AddUsersToRoles validates the user names and role names passed to it, by verifying their presence in the aspnet_Users and aspnet_Roles tables. Then, it adds the specified users to the specified roles, by inserting one record into the aspnet_UsersInRoles table for each user name/role name pair passed to it.

Removing Users from Roles

Applications call the role manager's *Roles.RemoveUserFromRole*, *Roles.RemoveUserFromRoles*, *Roles.RemoveUsersFromRole*, or *Roles.RemoveUsersFromRoles* method to remove users from roles. These methods, in turn, call the default role provider's *RemoveUsersFromRoles* method. *SqlRoleProvider.RemoveUsersFromRoles* converts the arrays of user names and role names in the parameter list into comma-delimited lists, and passes them to the stored procedure aspnet_UsersInRoles_RemoveUsersFromRoles.

aspnet_UsersInRoles_RemoveUsersFromRoles validates the user names and role names passed to it, by verifying their presence in the aspnet_Users and aspnet_Roles tables. Then, it removes the specified users from the specified roles, by deleting the corresponding records from the aspnet_UsersInRoles table.

Differences Between the Published Source Code and the .NET Framework's SqlRoleProvider

The published source code for *SqlRoleProvider* differs from the .NET Framework version in one respect: Declarative and imperative CAS demands were commented out. Because the source code can be compiled standalone, and thus will run as user code rather than trusted code in the global assembly cache, the CAS demands are not strictly necessary. For reference, however, the original demands from the .NET Framework version of the provider have been retained as comments.

AuthorizationStoreRoleProvider

AuthorizationStoreRoleProvider is the Microsoft role provider for Microsoft Authorization Manager data stores. Authorization Manager, also known as "AzMan," is a role-based access control framework for Microsoft Windows applications. *AuthorizationStoreRoleProvider* maps AzMan roles to the ASP.NET role manager, and it is an alternative to *SqlRoleProvider* for organizations that don't wish to store role data in databases. Because AzMan integrates with Active Directory, *AuthorizationStoreRoleProvider* can be combined with *ActiveDirectoryMembershipProvider* to employ Active Directory as the source for both membership data and role data in ASP.NET applications. *AuthorizationStoreRoleProvider* can also leverage AzMan's ability to store policy data in XML files and Active Directory Application Mode (ADAM), a service introduced in Microsoft Windows Server 2003 that supports application-specific views of Active Directory. For an overview of AzMan's features and capabilities, and how to use them from managed code, see "Use Role-Based Security in Your Middle Tier .NET Apps with Authorization Manager."

AuthorizationStoreRoleProvider doesn't support the full range of Authorization Manager features, opting instead to support the subset of features that map directly to the capabilities of the ASP.NET role manager. For example, AzMan allows tasks such as "Approve purchase order" and "View salary history" to be associated with roles, and it allows tasks to be subdivided into operations such as "View purchase order" and "Sign purchase order." *AuthorizationStoreRoleProvider* exposes AzMan roles as role-manager roles, but it does not expose (or otherwise use) information regarding tasks and operations.

The ultimate reference for *AuthorizationStoreRoleProvider* is the *AuthorizationStoreRoleProvider* source code, which is found in AuthStoreRoleProvider.cs. The sections that follow highlight key aspects of *AuthorizationStoreRoleProvider*'s design and operation.

AzMan Data Stores

AuthorizationStoreRoleProvider doesn't maintain data stores of its own, instead relying on AzMan data stores, which can be administered with tools such as the AzMan MMC snap-in or, if *AuthorizationStoreRoleProvider* is the default role provider, the Web Site Administration Tool that comes with ASP.NET. However, you can use the Web Site Adminstration Tool for role management only if you use a membership provider (for example, *ActiveDirectoryMembershipProvider*) as well. The Web Site Administration Tool has no user interface for manipulating Windows user accounts directly.

Authorization Manager can store data in XML files, Active Directory, or ADAM containers. *AuthorizationStoreRoleProvider* supports all three types of data stores. The *connectionStringName* configuration attribute tells *AuthorizationStoreRoleProvider* where AzMan data is stored. For example, the following connection string points *AuthorizationStoreRoleProvider* to an XML policy file:

```
<connectionStrings>
  <add name="AzManConnectionString"
    connectionString="msxml://c:/websites/App_Data/roles/roles.xml" />
</connectionStrings>
```

By contrast, the following connection string points it to an ADAM container named Contoso, on a remote server named ORION:

```
<connectionStrings>
  <add name="AzManConnectionString"
    connectionString="msldap://ORION/…
      CN=Contoso,OU=ContosoPartition,O=Contoso,C=US" />
</connectionStrings>
```

AuthorizationStoreRoleProvider is agnostic to the type of data store, because it uses the Authorization Manager API to read and write role data.

Scoping of Role Data

AzMan data stores can be partitioned into applications and scopes, enabling one data store to hold authorization data for multiple applications, and one application to hold multiple sets of authorization settings. *AuthorizationStoreRoleProvider* supports both forms of scoping through its *ApplicatioName* and *ScopeName* properties.

When *AuthorizationStoreRoleProvider* calls AzMan to open a data store, it passes in the application name and scope name stored in the *ApplicationName* and *ScopeName* properties, respectively. Thus, if the roles used by *AuthorizationStoreRoleProvider* are defined in an AzMan application named Contoso, and a scope named Roles, then matching *applicationName* and *scopeName* attributes should be included in the provider's configuration. If no application name is specified, *AuthorizationStoreRoleProvider* uses a default application name, which for a Web application is the application's virtual directory. Because AzMan doesn't allow applications with names like /, you should always explicitly set the *applicationName* for *AuthorizationStoreRoleProvider*. If no scope name is specified, *AuthorizationStoreRoleProvider* doesn't use a scope name when opening the data store.

AzMan Data Store Access

AzMan's features are exposed to applications through unmanaged COM interfaces such as *IAzAuthorizationStore*, which represents AzMan data stores; *IAzApplication*, which represents AzMan applications; *IAzScope*, which represents AzMan scopes; and *IAzRole*, which represents AzMan roles. *AuthorizationStoreRoleManager* uses COM interop and late binding to invoke AzMan methods exposed through these interfaces. Invocation code is wrapped in helper methods named *CallMethod* (reproduced in Figure 3) and *CallProperty*, which are used extensively by other *AuthorizationStoreRoleProvider* methods. For example, *AuthorizationStoreRoleProvider.CreateRole* uses the following code to call *IAzScope.CreateRole* or *IAzApplication.CreateRole* to create an AzMan role:

```
object[] args = new object[2];
args[0] = roleName;
args[1] = null;
object role = CallMethod(_ObjAzScope != null ?
```

```
    _ObjAzScope : _ObjAzApplication, "CreateRole", args);
```

_objAzScope and *_objAzApplication* contain references to AzMan scope and application objects created when the provider was initialized. For details, refer to "Provider Initialization."

Figure 3. AuthorizationStoreRoleProvider's CallMethod method

```
private object CallMethod(object objectToCallOn,
    string methodName, object[] args)
{
    if( HostingEnvironment.IsHosted && _XmlFileName != null) {
        InternalSecurityPermissions.Unrestricted.Assert();
    }

    try {
        using (new ApplicationImpersonationContext()) {
            return objectToCallOn.GetType().InvokeMember(methodName,
                BindingFlags.InvokeMethod | BindingFlags.Public |
                BindingFlags.Instance, null, objectToCallOn, args,
                CultureInfo.InvariantCulture);
        }
    } catch {
        throw;
    }
}
```

In Figure 3, note the provider's use of the internal *ApplicationImpersonationContext* type. This ensures that the provider connects to the directory using either the current process credentials, or the application impersonation credentials if an explicit username and password were specified in the *<identity>* element. The provider never connects to the AzMan policy store by using the credentials of an end user, even if user impersonation is enabled.

Also, a quick note on the unrestricted assert: The provider normally will not work in partially trusted ASP.NET applications, because of its reliance on COM interop to call unmanaged code. However, if your policy store is in an XML file, then the provider relies on file I/O CAS permissions as a surrogate security policy. This means you can use the provider in partially trusted ASP.NET applications, provided that those applications have CAS permissions to read the configured file path.

Provider Initialization

AuthorizationStoreRoleProvider initialization occurs in two stages.

Stage 1 initialization occurs in *AuthorizationStoreRoleProvider.Initialize*, which is called one timewhen the provider is loadedby ASP.NET. *AuthorizationStoreRoleProvider.Initialize* processes the configuration attributes

applicationName, *scopeName*, *connectionStringName*, and *cacheRefreshInterval*, and throws an exception if unrecognized configuration attributes remain.

Stage 2 initialization is performed on a lazy, as-needed basis by a private *AuthorizationStoreRoleProvider* helper method named *InitApp*, which is called by *CreateRole*, *DeleteRole*, and other *AuthorizationStoreRoleProvider* methods prior to carrying out the operations they're tasked with. *InitApp* performs the following tasks:

1. If the connection string referenced by *connectionStringName* contains an msxml:// prefix, *InitApp* converts the path into a fully qualified file-system path, verifies that the file exists, and verifies that the provider has permission to access it. Then, it caches the path name in a private field.
2. Uses reflection (through *Activator.CreateInstance*) to instantiate one of two versions of an internal class named *Microsoft.Interop.Security.AzRoles.AzAuthorizationStoreClass* representing AzMan. It instantiates version 1.2 of that class if it exists, or version 1.0 if it does not.
3. Calls AzMan's *IAzAuthorizationStore.Initialize* method, passing in the connection string retrieved from the configuration.
4. Calls AzMan's *IAzAuthorizationStore.OpenApplication* or *IAzAuthorizationStore.OpenApplication2* method (depending on which version of *AzAuthorizationStoreClass* was loaded) with the application name stored in *ApplicationName* to open an AzMan application (and create an application object).
5. If *ScopeName* is neither null nor empty, passes *ScopeName* to the *IAzApplication.OpenScope* method of the application object created in the previous step to open the specified scope (and create a scope object).

After *InitApp* has executed, *AuthorizationStoreRoleProvider* is fully initialized and ready to do business. It caches references to the AzAuthorizationStoreClass application, and to scope objects that it created in private fields for use in subsequent method calls. If any of its initialization steps fail, *InitApp* responds by throwing a *ProviderException*.

Although *AuthorizationStoreRoleProvider* initializes AzMan only once, key methods such as *IsUserInRole* and *GetRolesForUser*, which retrieve information regarding specific users, call a private helper method named *GetClientContext* to initialize AzMan's client context on every call. If the application employs Windows authentication, *GetClientContext* initializes the client context from the user's Windows token. If forms authentication is used instead, *GetClientContext* initializes the client context from the forms-authentication user name. Initializing the client context from a Windows token is faster, but the fact that *GetClientContext* abstracts the authentication type means that *AuthorizationStoreRoleProvider* works equally well with Windows authentication and forms authentication.

Cache Refresh

InitApp uses a Boolean flag named *_InitAppDone* to avoid redundant execution. Successful execution of *InitApp* sets *_InitAppDone* to *true*. The next time *InitApp* is called, it returns without doing anything, *unless* AzMan's *IAzAuthorizationStore.UpdateCache* method hasn't been called recentlythat is, within the time window specified through the provider's *CacheRefreshInterval* property. In that case, *InitApp* refreshes the AzMan cache from the underlying data store, by calling

UpdateCache on AzMan. *CacheRefreshInterval* defaults to *60* (minutes), meaning that, by default, it could be up to an hour before changes made to role definitions and assignments in the data store propagate to AzMan (and therefore to *AuthorizationStoreRoleProvider*). If desired, you can change the cache refresh interval, by using the *cacheRefreshInterval* configuration attribute.

The set accessors for *AuthorizationStoreRoleProvider*'s *ApplicationName* and *ScopeName* properties set *_InitAppDone* to *false*. Therefore, changing these property values at run-time refreshes the cache, regardless of the value of *CacheRefreshInterval* (or how much time has elapsed since the last call to AzMan's *UpdateCache* method).

Creating Roles

Applications call the role manager's *Roles.CreateRole* method to create new roles. *Roles.CreateRole* calls the default role provider's *CreateRole* method. *AuthorizationStoreRoleProvider.CreateRole* uses the helper method *CallMethod* to call AzMan's *CreateRole* method, allowing the .NET run-time to marshal the managed string containing the role name into a COM-compatible string, as follows:

```
object[] args = new object[2];
args[0] = roleName;
args[1] = null;
object role = CallMethod(_ObjAzScope != null ?
    _ObjAzScope : _ObjAzApplication, "CreateRole", args);
```

In this example (and many others like it), *CallMethod* calls the specified method on the application object created by *InitApp* if the provider lacks a *ScopeName*, or on the scope object created by *InitApp* if the provider has a *ScopeName*.

Following a successful call to AzMan's *CreateRole* method, *AuthorizationStoreRoleProvider.CreateRole* calls *CallMethod* again to call *Submit* on the AzMan role object returned by the previous call, and to commit the new role to the data store. Then, it calls *Marshal.FinalReleaseComObject* to release the role object.

Deleting Roles

Applications call the role manager's *Roles.DeleteRole* method to delete roles. *Roles.DeleteRole*, in turn, calls the default role provider's *DeleteRole* method. If the third parameter (*throwOnPopulatedRole*) passed to *DeleteRole* is *true*, *AuthorizationStoreRoleProvider.DeleteRole* calls *AuthorizationStoreRoleProvider.GetUsersInRole* to determine whether the role is empty, and throws a *ProviderException* if it's not. Then it calls AzMan's *DeleteRole* and *Submit* methods to delete the role from the data store.

Adding Users to Roles and More

Other *AuthorizationStoreRoleProvider* methods do as *CreateRole* and *DeleteRole*, using *CallMethod* and *CallProperty* to delegate to AzMan methods. For example, *AuthorizationStoreRoleProvider.AddUsersToRoles* first iterates through the array of role names passed to it, calling AzMan's *OpenRole* method to validate each role and convert

it into an AzMan role object. Then, it iterates through all the user names input to it, calling AzMan's *AddMemberName* method repeatedly to add the users to the roles, and finishes up by calling *Submit* on each AzMan role object.

Differences Between the Published Source Code and the .NET Framework's AuthorizationStoreRoleProvider

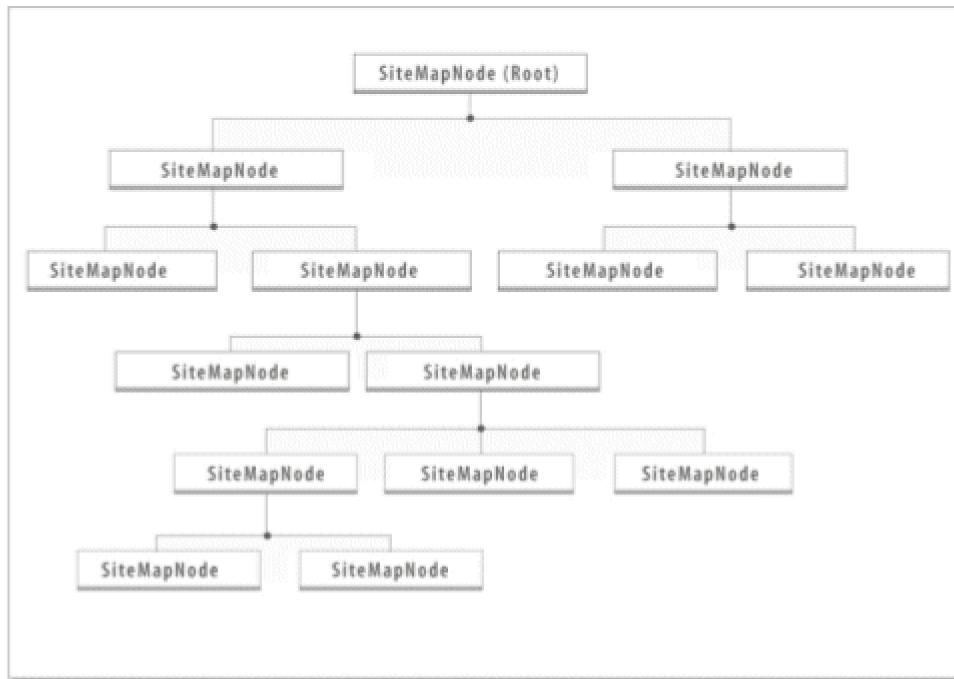
The source code for the *AuthorizationStoreRoleProvider* is being released unchanged. This means you will not be able to compile it in its current state, because it contains calls to internal helper methods. However, you can reference the source code to see exactly how the provider maps role manager calls to AzMan.

Site Map Providers

Site map providers provide the interface between Microsoft ASP.NET's data-driven site-navigation features and site map data sources. ASP.NET 2.0 ships with one site map provider: *XmlSiteMapProvider*, which reads site maps from XML site map files.

The fundamental job of a site map provider is to read site map data from a data source and build an upside-down tree of *SiteMapNode* objects (see Figure 4), and to provide methods for retrieving nodes from the site map. Each *SiteMapNode* in the tree represents one node in the site map. *SiteMapNode* properties such as *Title*, *Url*, *ParentNode*, and *ChildNodes* define the characteristics of each node, and allow the tree to be navigated up, down, and sideways. A single site map can be managed by one or several providers. Site map providers can form a tree of their own, linked together by their *ParentProvider* properties, with each provider in the tree claiming responsibility for a subset of the site map. A *SiteMapNode*'s *Provider* property identifies the provider that "owns" that node.

Figure 4. Site map structure



The Microsoft .NET Framework's *System.Web* namespace includes a class named *SiteMapProvider* that defines the basic characteristics of a site map provider. It also contains a *SiteMapProvider*-derivative named *StaticSiteMapProvider* that provides default implementations of most of *SiteMapProvider*'s abstract methods, and that overrides key virtuals to provide functional, even optimized, implementations. Providers that derive from *StaticSiteMapProvider* require considerably less code than providers derived from *SiteMapProvider*. *StaticSiteMapProvider* is prototyped as follows:

```
public abstract class StaticSiteMapProvider : SiteMapProvider
{
```

```
    public abstract SiteMapNode BuildSiteMap();
    protected virtual void Clear() {}
    protected internal override void AddNode(SiteMapNode node,
        SiteMapNode parentNode) {}
    protected internal override void RemoveNode(SiteMapNode node) {}
    public override SiteMapNode FindSiteMapNode(string rawUrl) {}
    public override SiteMapNode FindSiteMapNodeFromKey(string key) {}
    public override SiteMapNodeCollection
        GetChildNodes(SiteMapNode node) {}
    public override SiteMapNode GetParentNode(SiteMapNode node) {}
}
```

One of the key features of a site map provider is security trimming, which restricts the visibility of site map nodes, based on users' role memberships. The *SiteMapProvider* class contains built-in support for security trimming. *SiteMapProvider* implements a Boolean read-only property named *SecurityTrimmingEnabled*, which indicates whether security trimming is enabled. Furthermore, *SiteMapProvider*'s *Initialize* method initializes this property from the provider's *securityTrimmingEnabled* configuration attribute. Internally, *SiteMapProvider* methods that retrieve nodes from the site map call the provider's virtual *IsAccessibleToUser* method to determine whether nodes can be retrieved. All a derived class has to do to support security trimming is initialize each *SiteMapNode*'s *Roles* property with an array of role names identifying users that are permitted to access that node, or with *** if everyone (including unauthenticated users and users who enjoy no role memberships) is permitted.

Developer's Note

The default implementation of *SiteMapProvider.IsAccessibleToUser* makes use of the URL and file authorization checks built into ASP.NET. Consequently, a site map provider with security trimming enabled frequently needs only ***** in the *roles* attribute for the site map's root <siteMapNode>. Explicitly specifying the *roles* attribute for other nodes is necessary only for nodes that don't have a URL (or whose URL targets a destination outside the application's directory hierarchy), and for nodes whose visibility you wish to expand beyond the boundaries that URL and file authorization would normally allow.

The following section documents the implementation of *XmlSiteMapProvider*, which derives from *StaticSiteMapProvider*.

XmlSiteMapProvider

XmlSiteMapProvider is the Microsoft XML site map provider. It reads site map data from XML files that utilize the schema documented in "Data Schema." It includes localization support that enables localized node titles and descriptions to be loaded from resources (see "Localization"), and it supports security trimming. (Other than flowing its own

SecurityTrimmingEnabled property down to child instances of *XmlSiteMapProvider*, and including logic to parse comma-delimited or semicolon-delimited lists of role names into string arrays, *XmlSiteMapProvider* doesn't do anything special to support security trimming; that support comes by inheritance from *StaticSiteMapProvider* and *SiteMapProvider*.) It also employs a mechanism for automatically refreshing the site map if the site map file changes (see "Refreshing the Site Map").

The ultimate reference for *XmlSiteMapProvider* is the *XmlSiteMapProvider* source code, which is found in XmlSiteMapProvider.cs. The sections that follow highlight key aspects of *XmlSiteMapProvider*'s design and operation.

Provider Initialization

Initialization occurs in *XmlSiteMapProvider.Initialize*, which is called one timewhen the provider is loadedby ASP.NET. *XmlSiteMapProvider.Initialize* processes the *siteMapFile* configuration attribute (if present), and converts it into an object representing the virtual path to the site map file. Then, after deferring to the base class's *Initialize* method to process other attributes such as *securityTrimmingEnabled*, *XmlSiteMapProvider.Initialize* throws a *ProviderException* if unrecognized configuration attributes remain.

Data Schema

XmlSiteMapProvider reads site map data from XML files structured like the one in Figure 5. Each *<siteMapNode>* element defines one node in the site map, and can include the following attributes:

- *title*, which specifies the text displayed for the node in a navigation UI
- *description*, which provides descriptive text that may be shown in a navigation UI (for example, when the cursor hovers a node)
- *url*, which identifies the target of the link
- *roles*, which specifies which roles the node is visible to in a navigation UI when security trimming is enabled
- *resourceKey*, which specifies a resource key used to load localized text from localization resources

Site map files must contain a root element named *<siteMap>*, and the *<siteMap>* element can contain only one root *<siteMapNode>*.

Figure 5. Sample site map file

```
<siteMap>
  <siteMapNode title="Home" description="Home" url="~/default.aspx">
    <siteMapNode title="Products" description="Our products"
      url="~/Products.aspx" roles="*">
      <siteMapNode title="Hardware" description="Hardware choices"
        url="~/Hardware.aspx" />
      <siteMapNode title="Software" description="Software choices"
        url="~/Software.aspx" />
```

```
    </siteMapNode>
    <siteMapNode title="Services" description="Services we offer"
        url="~/Services.aspx" roles="*">
        <siteMapNode title="Training" description="Training classes"
          url="~/Training.aspx" />
        <siteMapNode title="Consulting"
          description="Consulting services"
          url="~/Consulting.aspx" />
        <siteMapNode title="Support" description="Supports plans"
          url="~/Support.aspx" />
    </siteMapNode>
    <siteMapNode title="Members Only" description="Premium content"
        url="~/Members.aspx" roles="Members,Administrators">
        <siteMapNode title="Account Management"
          description="Manage your account"
          url="~/MembersOnly/Accounts.aspx" />
        <siteMapNode title="Discussion Forums"
          description="Converse with other members"
          url="~/MembersOnly/Forums.aspx" />
    </siteMapNode>
  </siteMapNode>
</siteMap>
```

A *<siteMapNode>* element may also include a *provider* attribute that delegates responsibility for that node and its children to another provider. The *provider* attribute is not valid if the *<siteMapNode>* element contains other attributes. In addition, a *<siteMapNode>* element can contain a *siteMapFile* attribute pointing to another site map file. That attribute, too, is valid only in the absence of other attributes.

XmlSiteMapProvider doesn't validate XML site map files against an XML schema. Instead, the schema is implicit in *XmlSiteMapProvider*'s reading and handling of site map data. For example, the following code checks for a root *<siteMap>* element, and throws an exception if the element doesn't exist:

```
XmlNode node = null;
foreach (XmlNode siteMapNode in document.ChildNodes) {
    if (String.Equals(siteMapNode.Name, "siteMap",
        StringComparison.Ordinal)) {
        node = siteMapNode;
        break;
    }
}

if (node == null)
    throw new ConfigurationErrorsException(SR.GetString(
```

```
        SR.XmlSiteMapProvider_Top_Element_Must_Be_SiteMap), document);
```

XmlSiteMapProvider uses an XmlTextReader to read the site map file, and then wraps the nodes in an XmlDocument for easy navigation as it builds the site map in memory.

Building the Site Map

The heart of *XmlSiteMapProvider* is its *BuildSiteMap* method, which is called by ASP.NET when it needs the site map from the provider. Because *BuildSiteMap* is called many times over the provider's lifetime, *XmlSiteMapProvider.BuildSiteMap* contains logic for building the site map from the XML site map file the first time *BuildSiteMap* is called, and for returning the existing site map in subsequent calls.

XmlSiteMapProvider.BuildSiteMap reads the site map from the file specified through the *siteMapFile* configuration attribute, and builds an in-memory tree of *SiteMapNode* objects. Before building the site map, *BuildSiteMap* performs a series of validation checks on the site map data, making sure, for example, that it contains a root *<siteMap>* element, and that *<siteMap>* contains one (and only one) *<siteMapNode>* element. *BuildSiteMap* also checks the *<siteMap>* element for an *enableLocalization* attribute and, if present, initializes its own *EnableLocalization* property (inherited from *SiteMapProvider*) accordingly.

To convert XML nodes into *SiteMapNode*s, and to build the *SiteMapNode* tree, *BuildSiteMap* creates an instance of *System.Collections.Queue*, adds the root XML *<siteMapNode>* to the queue, and calls a method named *ConvertFromXmlNode*. With help from helper methods such as *GetNodeFromProvider*, *GetNodeFromSiteMapFile*, *GetNodeFromXmlNode*, and *AddNodeInternal*, *ConvertFromXmlNode* walks the XML site map from top to bottom, converting XML nodes into *SiteMapNode*s, and adding them to the tree. The *SiteMapNode* reference returned by *ConvertFromXmlNode* represents the *SiteMapNode* at the top of the tree (the root *SiteMapNode*). That reference is returned by *BuildSiteMap* to hand the site map off to ASP.NET. The *GetNodeFromProvider* and *GetNodeFromSiteMapFile* methods enable *XmlSiteMapProvider* to link one site map to other site maps managed by separate instances of *SiteMapProviders.*

As it converts XML nodes into *SiteMapNodes*, *XmlSiteMapProvider* verifies that each *SiteMapNode*'s *Url* and *Key* properties are unique with respect to other *SiteMapNode*s. For nodes that have URLs assigned to them, *XmlSiteMapProvider* sets *Key* equal to *Url.* For nodes that do not have URLs assigned to them, *XmlSiteMapProvider* sets *Key* to a randomly generated GUID. *XmlSiteMapProvider* also verifies that node URLs are application-relative, and that they don't contain URL-encoded charactersa sign of malformed (and potentially dangerous) URLs.

Refreshing the Site Map

Early in its lifetime, *BuildSiteMap* calls a helper method named *GetConfigDocument* that registers an event handler named *OnConfigFileChange* to be called if, and when, the site map file changes:

```
_handler = new FileChangeEventHandler(this.OnConfigFileChange);
HttpRuntime.FileChangesMonitor.StartMonitoringFile
```

```
    (_filename, _handler);
```

If the site map file changes, *OnConfigFileChange* notifies the parent provider (if any), and then calls the *Clear* method inherited from *StaticSiteMapProvider* to clear the site map, as follows:

```
private void OnConfigFileChange(Object sender, FileChangeEvent e) {
    // Notifiy the parent for the change.
    XmlSiteMapProvider parentProvider =
        ParentProvider as XmlSiteMapProvider;
    if (parentProvider != null) {
        parentProvider.OnConfigFileChange(sender, e);
    }
    Clear();
}
```

The next time ASP.NET calls *BuildSiteMap*, *XmlSiteMapProvider* rebuilds the site map. Consequently, changing the site map file at run-time causes *XmlSiteMapProvider* to refresh the site map. The change is visible in *TreeView*s, *Menu*s, or other controls that render navigation UIs from the site map.

XmlSiteMapProvider cancels the file-change monitor at dispose time, as follows:

```
protected virtual void Dispose(bool disposing) {
    if (_handler != null) {
        Debug.Assert(_filename != null);
        HttpRuntime.FileChangesMonitor.StopMonitoringFile
            (_filename, _handler);
    }
}
```

This simple bit of housekeeping prevents an active file-change monitor from referencing an event handler that no longer exists.

Localization

As an aid in localizing navigation UIs for users around the world, *XmlSiteMapProvider* offers built-in support for loading node titles and descriptions from string resources. For an excellent overview of how to localize site maps, see "How to Localize Site Map Data."

XmlSiteMapProvider supports two types of localization expressions: implicit and explicit. Implicit expressions use *resourceKey* attributes to specify the root names of localization resources. The provider is responsible for parsing the localization expressions, whereas *SiteMapNode* performs the actual resource lookups at runtime (more on this in a moment). Root names are combined with attribute names to generate fully qualified resource names. The following node definition loads values for *title* and *description* from

string resources named HomePage.title and HomePage.description in RESX files named Web.sitemap.*culture*.resx, where *culture* is a culture identifier such as fr or en-us:

```
<siteMapNode resourceKey="HomePage" title="Home"
  description="My home page" url="~/Default.aspx">
```

The default *title* and *description* are used if the resource manager can't find a RESX with the appropriate culturefor example, if the requested culture is fr, but the application contains RESX files only for de and us. The current culturethe one that determines which RESX resources are loaded fromis determined by the *CurrentThread.CurrentUICulture* property. The value of that property can be set programmatically; or, it can be declaratively initialized to the culture specified in each HTTP request's *Accept-Language* header, by setting *UICulture* to *auto* in an *@ Page* directive, or by setting *uiCulture* to *auto* in a *<globalization>* configuration element.

Explicit expressions don't rely on *resourceKey* attributes; instead, they use explicit resource names. In the following example, the node title comes from the resource named HomePageTitle in NavResources.*culture*.resx, whereas the node description comes from the resource named HomePageDesc (also in NavResources.*culture*.resx):

```
<siteMapNode title="$resources:NavResources,HomePageTitle,Home"
  description="$resources:NavResources,HomePageDesc,My home page"
  url="~/Default.aspx">
```

Once more, the current culture is defined by the value of *CurrentThread.CurrentUICulture*.

SiteMapNode performs the bulk of the work in loading node titles and descriptions from resources. All *XmlSiteMapProvider* has to do is parse out the resource keysexplicit and implicitand pass them to *SiteMapNode*'s constructor, as follows:

```
node = new SiteMapNode(this, key, url, title, description, roleList,
    attributeCollection, resourceKeyCollection, resourceKey);
```

For a node that uses an implicit expression, *XmlSiteMapProvider* passes the resource key in *resourceKey*. For a node that uses explicit expressions, *XmlSiteMapProvider* passes a collection of resource keys in *resourceKeyCollection*. Each item in the collection is either an attribute name/class name pair (for example, *title* and *NavResources*) or an attribute name/key name pair (for example, *title* and *HomePageTitle*). The parsing of explicit resource keys is handled by the private *XmlSiteMapProvider.HandleResourceAttribute* method, which is called by *XmlSiteMapProvider.GetNodeFromXmlNode*.

After the resource keys are provided to *SiteMapNode*'s constructor, *SiteMapNode* handles the task of loading the resources. The key code is contained in the get accessors for *SiteMapNode*'s *Title* and *Description* properties, which include logic for loading

property values from string resources when implicit or explicit resource keys are provided.

Differences Between the Published Source Code and the .NET Framework's XmlSiteMapProvider

The published source code for *XmlSiteMapProvider* and *StaticSiteMapProvider* differs from the .NET Framework versions in the following respects:

- Declarative and imperative CAS demands were commented out. Because the source code can be compiled standalone, and thus will run as user code rather than trusted code in the global assembly cache, the CAS demands are not strictly necessary. For reference, however, the original demands from the .NET Framework version of the provider have been retained as comments.
- The automatic reloading of site map data based on file-change monitoring was commented out, because the implementation depends on some internal unmanaged code helpers.

The standalone version of *XmlSiteMapProvider* supports linking only to child providers that are instances of the standalone version of *XmlSiteMapProvider*. The version of the provider that ships in the .NET Framework can be linked to child providers of any arbitrary type that implements *SiteMapProvider*.

Session State Providers

Session state providers provide the interface between Microsoft ASP.NET's session state module and session state data sources. ASP.NET 2.0 ships with three session state providers:

- *InProcSessionStateStore*, which stores session state in memory in the ASP.NET worker process
- *OutOfProcSessionStateStore*, which stores session state in memory in an external state server process
- *SqlSessionStateStore*, which stores session state in Microsoft SQL Server and Microsoft SQL Server Express databases

Core ASP.NET session state services are provided by *System.Web.SessionState.SessionStateModule*, instances of which are referred to as *session state modules*. Session state modules encapsulate session state in instances of *System.Web.SessionState.SessionStateStoreData*, allocating one *SessionStateStoreData* per session (per user). The fundamental job of a session state provider is to serialize *SessionStateDataStore*s to session state data sources, and deserialize them on demand. *SessionStateDataStore* has three properties that must be serialized in order to hydrate class instances:

- *Items*, which encapsulates a session's non-static objects
- *StaticObjects*, which encapsulates a session's static objects
- *Timeout*, which specifies the session's timeout (in minutes)

Items and *StaticObjects* can be serialized and deserialized easily enough, by calling their *Serialize* and *Deserialize* methods, respectively. The *Timeout* property is a simple System.Int32, and it is therefore also easily serialized and deserialized.

The .NET Framework's *System.Web.SessionState* namespace includes a class named *SessionStateStoreProviderBase* that defines the basic characteristics of a session state provider. *SessionStateStoreProviderBase* is prototyped as follows:

```
public abstract class SessionStateStoreProviderBase : ProviderBase
{
    public abstract void Dispose();

    public abstract bool SetItemExpireCallback
        (SessionStateItemExpireCallback expireCallback);

    public abstract void InitializeRequest(HttpContext context);

    public abstract SessionStateStoreData GetItem
        (HttpContext context, String id, out bool locked,
        out TimeSpan lockAge, out object lockId,
        out SessionStateActions actions);
```

```
    public abstract SessionStateStoreData GetItemExclusive
        (HttpContext context, String id, out bool locked,
        out TimeSpan lockAge, out object lockId,
        out SessionStateActions actions);

    public abstract void ReleaseItemExclusive(HttpContext context,
        String id, object lockId);

    public abstract void SetAndReleaseItemExclusive
        (HttpContext context, String id, SessionStateStoreData item,
        object lockId, bool newItem);

    public abstract void RemoveItem(HttpContext context,
        String id, object lockId, SessionStateStoreData item);

    public abstract void ResetItemTimeout(HttpContext context,
        String id);

    public abstract SessionStateStoreData CreateNewStoreData
        (HttpContext context, int timeout);

    public abstract void CreateUninitializedItem
        (HttpContext context, String id, int timeout);

    public abstract void EndRequest(HttpContext context);
}
```

Three of the most important methods in a session state provider are *GetItem*, *GetItemExclusive*, and *SetAndReleaseItemExclusive*. The first two are called by *SessionStateModule* to retrieve a session from the data source. If the requested page implements the *IRequiresSessionState* interface (by default, all pages implement *IRequiresSessionState*), *SessionStateModule*'s *AcquireRequestState* event handler calls the session state provider's *GetItemExclusive* method. The word "Exclusive" in the method name means that the session should be retrieved only if it's not currently being used by another request. If, on the other hand, the requested page implements the *IReadOnlySessionState* interface (the most common way to achieve this is to include an *EnableSessionState="ReadOnly"* attribute in the page's *@ Page* directive), *SessionStateModule* calls the provider's *GetItem* method. No exclusivity is required here, because overlapping read accesses are permitted by *SessionStateModule*.

In order to provide the exclusivity required by *GetItemExclusive*, a session state provider must implement a locking mechanism that prevents a given session from being accessed by two or more concurrent requests requiring read/write access to session state. That mechanism ensures the consistency of session state, by preventing concurrent requests from overwriting each other's changes. The locking mechanism must work even if the session state data source is a remote resource shared by several Web servers.

SessionStateModule reads sessions from the data source at the outset of each request, by calling *GetItem* or *GetItemExclusive* from its *AcquireRequestState* handler. At the end of the request, *SessionStateModule*'s *ReleaseRequestState* handler calls the session state provider's *SetAndReleaseItemExclusive* method to commit changes to the data source, and release locks held by *GetItemExclusive*. A related method named *ReleaseItemExclusive* exists so that *SessionStateModule* can time out a locked session by commanding the session state provider to release the lock.

The following section documents the implementation of *SqlSessionStateStore*, which derives from *SessionStateStoreProviderBase*.

SqlSessionStateStore

SqlSessionStateStore is the Microsoft session state provider for SQL Server databases. It stores session data using the schema documented in "Data Schema," and it uses the stored procedures documented in "Data Access." All knowledge of the database schema is hidden in the stored procedures, so that porting *SqlSessionStateStore* to other database types requires little more than modifying the stored procedures. (Depending on the targeted database type, the ADO.NET code used to call the stored procedures might have to change, too. The Microsoft Oracle .NET provider, for example, uses a different syntax for named parameters.)

SqlSessionStateStore is alone among SQL providers, in that it doesn't use the provider database. Instead, it uses a separate session state database whose name (by default) is ASPState. The session state database can be created by running Aspnet_regsql.exe with an *-ssadd* switch. The session state database can be stored either in tempdb (which will not survive a server restart), or in a custom database (for example, ASPState) that will survive a server restart. Both persistence options can be exercised using Aspnet_regsql.exe's *–sstype* switch.

SqlSessionStateStore supports a key scalability feature of ASP.NET 2.0 known as *session state partitioning*. By default, all sessions for all applications are stored in a single SQL Server database. However, developers can implement custom *partition resolvers*classes that implement the *IPartitionResolver* interfaceto partition sessions into multiple databases. Partition resolvers convert session IDs into database connection strings; before accessing the session state database, *SqlSessionStateStore* calls into the active partition resolver to get the connection string it needs. One use for custom partition resolvers is to divide session state for one application into two or more databases. Session state partitioning helps ASP.NET applications scale out horizontally, by eliminating the bottleneck of a single session state database. For an excellent overview of how partitioning works, and how to write custom partition resolvers, see "Fast, Scalable, and Secure Session State Management for Your Web Applications" In the September 2005 issue of MSDN Magazine.

The ultimate reference for *SqlSessionStateStore* is the *SqlSessionStateStore* source code, which is found in SqlStateClientManager.cs. The sections that follow highlight key aspects of *SqlSessionStateStore*'s design and operation.

Provider Initialization

Initialization occurs in *SqlSessionStateStore.Initialize*, which is called when the provider is loaded by *SessionStateModule*. The version of *Initialize* that's called isn't the one

inherited from *ProviderBase*, but a special one defined in *SessionStateStoreProviderBase* that receives an *IPartitionResolver* parameter. If a custom partition resolver is registered (registration is accomplished by including a *partitionResolverType* attribute in the *<sessionState>* configuration element), that parameter references a custom partition resolver. Otherwise, it's set to null, indicating the absence of a custom partition resolver. *SqlSessionStateStore* caches the reference (null or not) in a private field named *_partitionResolver*.

The *Initialize* method called by *SessionStateModule* then calls the *Initialize* method inherited from *ProviderBase*. This method calls *base.Initialize*, and then delegates to a private helper method named *OneTimeInit*. If *_partitionResolver* is null, *OneTimeInit* creates a *SqlPartitionInfo* object, and caches a reference to it in a private static field. (*SqlPartitionInfo* essentially wraps a database connection string, and is used when the connection string used to access the session state database won't change over time.) However, if *_partitionResolver* is not null, then *OneTimeInit* sets a static private field named *s_usePartition* to *true*, creates a new *PartitionManager* object (note that this is an internal type) to encapsulate the partition resolver, and stores a reference to the *PartitionManager* in another private static field. Whenever *SqlSessionStateStore* needs to access the session state database, it calls the helper method *GetConnection*, which retrieves a connection targeting the proper session state database, using the partition resolver, if present, to acquire the connection. *GetConnection* also includes pooling logic, allowing session state database connections to be pooled when circumstances permit.

Initialize's final task is to register a handler for the *DomainUnload* event that fires when the host application domain unloads. The handler*OnAppDomainUnload*performs cleanup duties by calling *Dispose* on the *SqlPartitionInfo* or *PartitionManager* object.

Data Schema

SqlSessionStateStore stores session data in the ASPStateTempSessions table of the session state database. Each record in ASPStateTempSessions holds the serialized session state for one session, and the auxiliary data that accompanies that session. Table 9 documents the ASPStateTempSessions table's schema.

Table 9. The ASPStateTempSessions table

Column Name	Column Type	Description
SessionId	nvarchar(88)	Session ID + application ID
Created	datetime	Date and time session was created (UTC)
Expires	datetime	Date and time session expires (UTC)
LockDate	datetime	UTC date and time session was locked
LockDateLocal	datetime	Local date and time session was locked
LockCookie	int	Lock ID
Timeout	int	Session timeout in minutes
Locked	bit	*1*=Session locked, *0*=Session not locked
SessionItemShort	varbinary(7000)	Serialized session state (if <= 7,000 bytes)

SessionItemLong	image	Serialized session state (if > 7,000 bytes)
Flags	int	Session state flags (*1*=Uninitialized session)

Serialized session state is stored in binary form in the *SessionItemShort* and *SessionItemLong* fields. Sessions that serialize to a length of 7,000 bytes or less are stored in *SessionItemShort*, whereas sessions that serialize to a length of more than 7,000 bytes are stored in *SessionItemLong*. (Storing "short" sessions in a varbinary field offers a performance advantage, because the data can be stored in the table row, rather than externally in other data pages.) The *Expires* field is used to clean up expired sessions, as described in "Session Expiration." The *Locked*, *LockDate*, *LockDateLocal*, and *LockCookie* fields are used to lock concurrent accesses to a session. *SqlSessionStateStore*'s locking strategy is described in "Serializing Concurrent Accesses to a Session."

In addition to scoping data by user, *SqlSessionStateStore* scopes data by application, so that multiple applications can store sessions in one session state database. To that end, the session state database contains an ASPStateTempApplications table that records application names and application IDs. Application names are not explicitly specified as they are for other providers; instead, *SqlSessionStateStore* uses the website's IIS metabase path as the application name. Application IDs are hashes generated from application names by the stored procedure GetHashCode. (*SqlSessionStateStore* differs in this respect, too, from other SQL providers, which use randomly generated GUIDs as application IDs.) Table 10 documents the schema of the ASPStateTempApplications table.

Table 10. The ASPStateTempApplications table

Column Name	**Column Type**	**Description**
AppId	int	Application ID
AppName	char(280)	Application name

Curiously, the ASPStateTempSessions table lacks an AppId column linking it to ASPTempStateApplications. The linkage occurs in ASPStateTempSessions's *SessionId* field, which doesn't store just session IDs. It stores session IDs with application IDs appended to them. The statement

```
cmd.Parameters[0].Value = id + _partitionInfo.AppSuffix; // @id
```

in *SqlSessionStateStore.DoGet* (discussed in "Reading Sessions from the Database") is one example of how *SqlSessionStateStore* generates the session ID values input to database queries.

Data Access

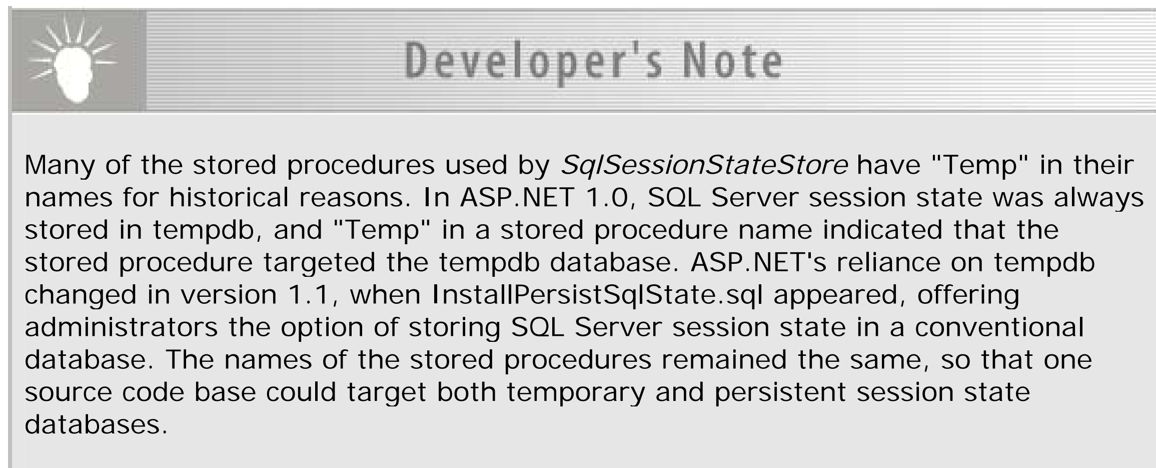
SqlSessionStateStore performs all database accesses through stored procedures. Table 11 lists the stored procedures that it uses.

Table 11. Stored procedures used by SqlSessionStateStore

Stored Procedure	Description
CreateTempTables	Creates the ASPStateTempSessions and ASPStateTempApplications tables; called during setup, but not called by *SqlSessionStateStore*.
DeleteExpiredSessions	Used by SQL Server Agent to remove expired sessions.
GetHashCode	Hashes an application name and returns the hash; called by TempGetAppID.
GetMajorVersion	Returns SQL Server's major version number.
TempGetAppID	Converts an application name into an application ID; queries the ASPStateTempApplications table and inserts a new record if necessary.
TempGetStateItem	Retrieves read-only session state from the database (ASP.NET 1.0; ASP.NET 1.1/SQL Server 7).
TempGetStateItem2	Retrieves read-only session state from the database (ASP.NET 1.1).
TempGetStateItem3	Retrieves read-only session state from the database (ASP.NET 2.0).
TempGetStateItemExclusive	Retrieves read/write session state from the database (ASP.NET 1.0; ASP.NET 1.1/SQL Server 7).
TempGetStateItemExclusive2	Retrieves read/write session state from the database (ASP.NET 1.1).
TempGetStateItemExclusive3	Retrieves read/write session state from the database (ASP.NET 2.0).
TempGetVersion	Marker whose presence indicates to ASP.NET 2.0 that the session state database is ASP.NET 2.0-compatible.
TempInsertStateItemLong	Adds a new session, whose size is > 7,000 bytes, to the database.
TempInsertStateItemShort	Adds a new session, whose size is <= 7,000 bytes, to the database.
TempInsertUninitializedItem	Adds a new uninitialized session to the database in support of cookieless sessions.
TempReleaseStateItemExclusive	Releases a lock on a session; called when ASP.NET determines that a request has

	timed out and calls the provider's *ReleaseItemExclusive* method.
TempRemoveStateItem	Removes a session from the database when the session is abandoned.
TempResetTimeout	Resets a session's timeout by writing the current date and time to the corresponding record's *Expires* field.
TempUpdateStateItemLong	Updates a session whose size is > 7,000 bytes.
TempUpdateStateItemLongNullShort	Updates a session whose old size is <= 7,000 bytes, but whose new size is > 7,000 bytes.
TempUpdateStateItemShort	Updates a session whose size is <= 7,000 bytes.
TempUpdateStateItemShortNullLong	Updates a session whose old size is > 7,000 bytes, but whose new size is <= 7,000 bytes.

Some of the stored procedures exist in different versions (for example, TempGetStateItem, TempGetStateItem2, and TempGetStateItem3), in order to support different versions of ASP.NET and different versions of SQL Server. ASP.NET 2.0 uses the "3" versions of these stored procedures, and it never calls the old versions. The older stored procedures are retained to allow ASP.NET 1.1 servers to use the same session state database as ASP.NET 2.0.


Developer's Note

Many of the stored procedures used by *SqlSessionStateStore* have "Temp" in their names for historical reasons. In ASP.NET 1.0, SQL Server session state was always stored in tempdb, and "Temp" in a stored procedure name indicated that the stored procedure targeted the tempdb database. ASP.NET's reliance on tempdb changed in version 1.1, when InstallPersistSqlState.sql appeared, offering administrators the option of storing SQL Server session state in a conventional database. The names of the stored procedures remained the same, so that one source code base could target both temporary and persistent session state databases.

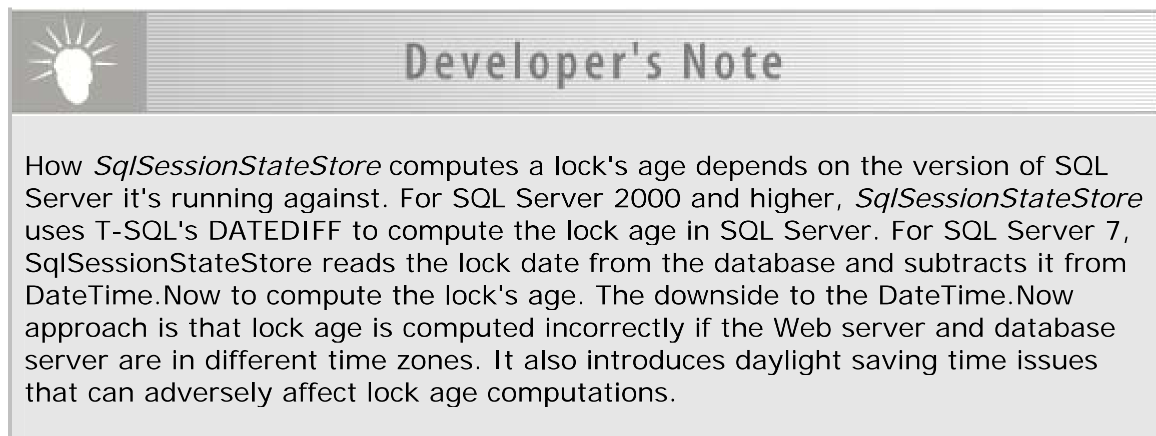
Reading Sessions from the Database

To retrieve a session from the session state data source, *SessionStateModule* calls the default session state provider's *GetItem* or *GetItemExclusive* method. The former is called for pages that implement the *IReadOnlySessionState* interface (pages that read

session state, but do not write it), whereas the latter is called for pages that implement *IRequiresSessionState* (indicating that they both read and write session state). Both *GetItem* and *GetItemExclusive* delegate to a private helper method named *DoGet*. *GetItem* passes *false* in *DoGet*'s third parameter, indicating that exclusivity is not required. *GetItemExclusive* passes in *true*.

SqlSessionStateStore.DoGet retrieves the session from the database, or returns locking information if the session is locked because it's being used by a concurrent request. It begins by calling *GetConnection* to get a connection to the session state database. Then, it calls one of two stored procedures: TempGetStateItem3 if the third parameter passed to *DoGet* is *false* (that is, if *DoGet* was called by *GetItem*), or TempGetStateItemExclusive3 if the third parameter is *true* (if *DoGet* was called by *GetItemExclusive*).

What happens when the stored procedure returns depends on whether the requested session is currently locked. If the session isn't locked, *DoGet* extracts the serialized session state from the *SessionItemShort* or *SessionItemLong* field, deserializes the session state into a *SessionStateStoreData* object, and returns it. However, if the session is locked, *DoGet* returns null, but uses the *out* parameters *locked* and *lockAge* to inform *SessionStateModule* that the session is locked and how long the lock has been active. Lock age is used by *SessionStateModule* to forcibly release a lock if the lock is held for too long.



Developer's Note

How *SqlSessionStateStore* computes a lock's age depends on the version of SQL Server it's running against. For SQL Server 2000 and higher, *SqlSessionStateStore* uses T-SQL's DATEDIFF to compute the lock age in SQL Server. For SQL Server 7, SqlSessionStateStore reads the lock date from the database and subtracts it from DateTime.Now to compute the lock's age. The downside to the DateTime.Now approach is that lock age is computed incorrectly if the Web server and database server are in different time zones. It also introduces daylight saving time issues that can adversely affect lock age computations.

Writing Sessions to the Database

To save a session to the session state data source, *SessionStateModule* calls the default session state provider's *SetAndReleaseItemExclusive* method. *SqlSessionStateStore.SetAndReleaseItemExclusive* serializes the *SessionStateStoreData* passed to it, taking care to call *SqlSessionStateStore.ReleaseItemExclusive* to release the lock (if any) on the session if the serialization attempt fails.

Next, *SqlSessionStateStore.SetAndReleaseItemExclusive* calls *GetConnection* to get a database connection. Then, it checks the *newItem* parameter passed to it, to determine whether the session being saved is a new session or an existing session. If *newItem* is *true*, indicating that the session has no corresponding row in the database, *SetAndReleaseItemExclusive* calls the stored procedure TempInsertStateItemShort or

TempInsertStateItemLong (depending on the size of the serialized session) to record the session in a new row in the session state database. If *newItem* is *false*, indicating that the database already contains a row representing the session, *SetAndReleaseItemExclusive* calls one of the following stored procedures to update that row:

- TempUpdateStateItemShort if the serialized session contains 7,000 or fewer bytes, and if it formerly contained 7,000 or fewer bytes also
- TempUpdateStateItemLong if the serialized session contains more than 7,000 bytes, and if it formerly contained more than 7,000 bytes also
- TempUpdateStateItemLongNullShort if the serialized session contains more than 7,000 bytes, but it formerly contained 7,000 or fewer bytes
- TempUpdateStateItemShortNullLong if the serialized session contains 7,000 or fewer bytes, but it formerly contained more than 7,000 bytes

The "Null" versions of these stored procedures nullify the field containing the old session data before recording new session data in *SessionItemShort* or *SessionItemLong*.

Serializing Concurrent Accesses to a Session

SqlSessionStateStore employs a locking strategy that relies on fields in the session state database and the stored procedures that access them. The following is a synopsis of how it works.

When *SessionStateModule* calls *GetItem* to retrieve a session from the database, *SqlSessionStateStore* calls the stored procedure TempGetStateItem3. The stored procedure does no locking of its own, but checks the *Locked* field of the corresponding record before deciding what to return. If *Locked* is *0*, indicating that the session isn't locked, TempGetStateItem3 returns the serialized session data through the *@itemShort* output parameter if the session is stored in the *SessionItemShort* field, or as a query result if the session is stored in the *SessionItemLong* field. If *Locked* is *not 0*, however, TempGetStateItem3 returns no session state. Instead, it uses the output parameters named *@locked*, *@lockAge*, and *@lockCookie*, to return a nonzero value indicating that the session is locked, the lock age, and the lock ID, respectively. *SessionStateModule* responds by retrying the call to *GetItem* at half-second intervals until the lock is removed.

How does a record become locked in the first place? That happens in TempGetStateItemExclusive3, which is called when *SessionStateModule* calls *SqlSessionStateStore*'s *GetItemExclusive* method. If called to retrieve a session that's already locked (*Locked=1*), TempGetStateItemExclusive3 behaves much like TempGetStateItem3. But, if called to retrieve a session that isn't locked (*Locked=0*), TempGetStateItemExclusive3 sets *LockDate* and *LockDateLocal* to the current time, returns *0* through the *@locked* parameterindicating that the session wasn't locked when the read occurredand returns the serialized session. It also sets the record's *Locked* field to *1*, effectively locking the session and preventing subsequent calls to TempGetStateItem3 or TempGetStateItemExclusive3 from returning sessions until *Locked* is reset to *0*.

Locked is reset to *0* by all of the stored procedures called by *SqlSessionStateStore*'s *SetAndReleaseItemExclusive* method to write a session to the database.

TempUpdateStateItemShort, reproduced in Figure 6, is one example. A session can also be unlocked with the stored procedure TempReleaseStateItemExclusive, which is called by *SqlSessionStateStore*'s *ReleaseItemExclusive* method. *SessionStateModule* calls that method to forcibly release a lock if repeated attempts to retrieve the session don't succeed. Figure 7 shows the relevant code in *SessionStateModule*.

Figure 6. TempUpdateStateItemShort

```
CREATE PROCEDURE [dbo].[TempUpdateStateItemShort]
    @id         tSessionId,
    @itemShort  tSessionItemShort,
    @timeout    int,
    @lockCookie int
AS
    UPDATE [ASPState].dbo.ASPStateTempSessions
    SET Expires = DATEADD(n, Timeout, GETUTCDATE()),
        SessionItemShort = @itemShort,
        Timeout = @timeout,
        Locked = 0 /* Unlock the session! */
    WHERE SessionId = @id AND LockCookie = @lockCookie
    RETURN 0
```

Figure 7. SessionStateModule code for timing out locks

```
if (_rqReadonly) {
    _rqItem = _store.GetItem(_rqContext, _rqId, out locked,
        out lockAge, out _rqLockId, out _rqActionFlags);
}
else {
   _rqItem = _store.GetItemExclusive(_rqContext, _rqId, out locked,
       out lockAge, out _rqLockId, out _rqActionFlags);
}

// We didn't get it because it's locked....
if (_rqItem == null && locked) {
    if (lockAge >= _rqExecutionTimeout) {
        /* Release the lock on the item, which is held
        by another thread*/
        _store.ReleaseItemExclusive(_rqContext, _rqId, _rqLockId);
    }
    isCompleted = false;
    PollLockedSession();
}
```

Supporting Cookieless Sessions

ASP.NET supports two different types of sessions: cookied and cookieless. The terms "cookied" and "cookieless" refer to the mechanism used to round-trip session IDs between clients and Web servers. Cookied sessions round-trip session IDs in HTTP cookies, whereas cookieless sessions embed session IDs in URLs using a technique known as "URL munging."

In order to support cookieless sessions, a session state provider must implement a *CreateUninitializedItem* method that creates an uninitialized session. When a request arrives, and session state is configured with the default settings for cookieless mode (for example, when the *<sessionState>* configuration element contains *cookieless="UseUri"* and *regenerateExpiredSessionId="true"* attributes), *SessionStateModule* creates a new session ID, munges it onto the URL, and passes it to *CreateUninitializedItem*. Afterwards, a redirect occurs, with the munged URL as the target. The purpose of calling *CreateUninitializedItem* is to allow the session ID to be recognized as a valid ID after the redirect. (Otherwise, *SessionStateModule* would think that the ID extracted from the URL after the redirect represents an expired session, in which case it would generate a new session ID, which would force another redirect and result in an endless loop.) If sessions are cookied rather than cookieless, the provider's *CreateUninitializedItem* method is never called.

SqlSessionStateStore supports cookied and cookieless sessions. Its *CreateUninitializedItem* method calls *TempInsertUninitializedItem*, which adds a row to the session state database, and flags it as an uninitialized session by setting the *Flags* field to *1*. The flag is reset to *0* when the session is retrieved from the database by TempGetStateItem3 or TempGetStateItemExclusive3, following a redirect.

Session Expiration

Each session created by ASP.NET has a timeout value (by default, *20* minutes) associated with it. If no accesses to the session occur within the session timeout, the session is deemed to be expired, and it is no longer valid.

SqlSessionStateStore uses the *Expires* field of the ASPStateTempSessions table to record the date and time that each session expires. All stored procedures that read or write a session set the *Expires* field equal to the current date and time plus the session timeout, effectively extending the session's lifetime for another full timeout period.

SqlSessionStateStore doesn't actively monitor the *Expires* field. Instead, it relies on an external agent to scavenge the database and delete expired sessionssessions whose *Expires* field holds a date and time less than the current date and time. The ASPState database includes a SQL Server Agent job that periodically (by default, every 60 seconds) calls the stored procedure DeleteExpiredSessions to remove expired sessions. DeleteExpiredSessions uses the following simple *DELETE* statement to delete all qualifying rows from the ASPStateTempSessions table:

```
DELETE [ASPState].dbo.ASPStateTempSessions WHERE Expires < @now
```

SessionStateModule doesn't fire *Session_End* events when *SqlSessionStateStore* is the default session state provider, because *SqlSessionStateStore* doesn't notify it when a session expires.

If an application abandons a session by calling *Session.Abandon*, *SessionStateModule* calls the provider's *RemoveItem* method. *SqlSessionStateStore.RemoveItem* calls the stored procedure TempRemoveStateItem to delete the session from the database.

Differences Between the Published Source Code and the .NET Framework's SqlSessionStateStore

The published source code for *SqlSessionStateStore* differs from the .NET Framework version in the following respects:

- Some imperative CAS checks were commented out. Because the source code can be compiled standalone, and thus will run as user code rather than trusted code in the global assembly cache, the CAS checks are not necessary.
- Calls to performance counters were commented out, because the .NET Framework provider relies on internal helper classes for manipulating these counters. For reference, the original code has been retained as comments.
- The published version does not support the use of multiple database partitions, because the .NET Framework provider uses a number of internal classes to implement this functionality. However, the published version contains commented code, so that you can see how the .NET Framework provider supports multiple database partitions.
- Some internal helper methods used by the .NET Framework provider for serializing and deserializing session data have been cloned in the published provider.

Profile Providers

Profile providers provide the interface between Microsoft ASP.NET's profile service and profile data sources. ASP.NET 2.0 ships with one profile provider: *SqlProfileProvider*, which stores profile data in Microsoft SQL Server and Microsoft SQL Server Express databases.

The fundamental job of a profile provider is to write profile property values supplied by ASP.NET to a persistent profile data source, and to read the property values back from the data source when requested by ASP.NET. The Microsoft .NET Framework's *System.Web.Profile* namespace includes a class named *ProfileProvider* that defines the basic characteristics of a profile provider. *ProfileProvider* is prototyped as follows:

```
public abstract class ProfileProvider : SettingsProvider
{
    public abstract int DeleteProfiles
        (ProfileInfoCollection profiles);

    public abstract int DeleteProfiles (string[] usernames);

    public abstract int DeleteInactiveProfiles
        (ProfileAuthenticationOption authenticationOption,
        DateTime userInactiveSinceDate);

    public abstract int GetNumberOfInactiveProfiles
        (ProfileAuthenticationOption authenticationOption,
        DateTime userInactiveSinceDate);

    public abstract ProfileInfoCollection GetAllProfiles
        (ProfileAuthenticationOption authenticationOption,
        int pageIndex, int pageSize, out int totalRecords);

    public abstract ProfileInfoCollection GetAllInactiveProfiles
        (ProfileAuthenticationOption authenticationOption,
        DateTime userInactiveSinceDate, int pageIndex,
        int pageSize, out int totalRecords);

    public abstract ProfileInfoCollection FindProfilesByUserName
        (ProfileAuthenticationOption authenticationOption,
        string usernameToMatch, int pageIndex, int pageSize,
        out int totalRecords);

    public abstract ProfileInfoCollection
        FindInactiveProfilesByUserName (ProfileAuthenticationOption
        authenticationOption, string usernameToMatch,
```

```
        DateTime userInactiveSinceDate, int pageIndex,
        int pageSize, out int totalRecords);
}
```

A *ProfileProvider*-derived class must also implement the abstract methods and properties defined in *System.Configuration.SettingsProvider*, which is prototyped as follows:

```
public abstract class SettingsProvider : ProviderBase
{
    // Properties
    public abstract string ApplicationName { get; set; }

    // Methods
    public abstract SettingsPropertyValueCollection
        GetPropertyValues (SettingsContext context,
        SettingsPropertyCollection properties);

    public abstract void SetPropertyValues(SettingsContext context,
        SettingsPropertyValueCollection properties);
}
```

The two most important methods in a profile provider are the *GetPropertyValues* and *SetPropertyValues* methods inherited from *SettingsProvider*. These methods are called by ASP.NET to read property values from the data source, and write them back. Other profile provider methods play a lesser role, by performing administrative functions such as enumerating and deleting profiles.

When code executing within a request reads a profile property, ASP.NET calls the default profile provider's *GetPropertyValues* method. The *context* parameter passed to *GetPropertyValues* is a dictionary of key/value pairs containing information about the context in which *GetPropertyValues* was called. It contains the following keys:

- *UserName* **-** User name or user ID of the profile to read.
- *IsAuthenticated* **-** Indicates whether the requestor is authenticated.

The *properties* parameter contains a collection of *SettingsProperty* objects representing the property values ASP.NET is requesting. Each object in the collection represents one of the properties defined in the *<profile>* configuration section. *GetPropertyValues*'s job is to return a *SettingsPropertyValuesCollection* supplying values for the properties in the *SettingsPropertyCollection*. If the property values have been persisted before, then *GetPropertyValues* can retrieve the values from the data source. Otherwise, it can return a *SettingsPropertyValuesCollection* that instructs ASP.NET to assign default values.

SetPropertyValues is the counterpart to *GetPropertyValues*. It's called by ASP.NET to persist property values in the profile data source. Like *GetPropertyValues*, it's passed a *SettingsContext* object containing a user name (or ID), and a Boolean indicating whether the user is authenticated. It's also passed a *SettingsPropertyValueCollection*

containing the property values to be persisted. The format in which the data is persistedand the physical storage medium that it's persisted inis up to the provider. Obviously, the format in which *SetPropertyValues* persists profile data must be understood by the provider's *GetProfileProperties* method.

Profile property values are inherently scoped by user. For authenticated users, each set of persisted profile property values is accompanied by a user ID that uniquely identifies an authenticated user. For anonymous users, each set of persisted profile property values is accompanied by an anonymous user ID assigned by ASP.NET's anonymous identification service.

The following section documents the implementation of *SqlProfileProvider*, which derives from *ProfileProvider*.

SqlProfileProvider

SqlProfileProvider is the Microsoft profile provider for SQL Server databases. It stores profile data, using the schema documented in "Data Schema," and it uses the stored procedures documented in "Data Access." All knowledge of the database schema is hidden in the stored procedures, so that porting *SqlProfileProvider* to other database types requires little more than modifying the stored procedures. (Depending on the targeted database type, the ADO.NET code used to call the stored procedures might have to change, too. The Microsoft Oracle .NET provider, for example, uses a different syntax for named parameters.)

The ultimate reference for *SqlProfileProvider* is the *SqlProfileProvider* source code, which is found in SqlProfileProvider.cs. The sections that follow highlight key aspects of *SqlProfileProvider*'s design and operation.

Provider Initialization

Initialization occurs in *SqlProfileProvider.Initialize*, which is called one timewhen the provider is loadedby ASP.NET. *SqlProfileProvider.Initialize* processes the *applicationName*, *connectionStringName*, and *commandTimeout* configuration attributes, and throws a *ProviderException* if unrecognized configuration attributes remain. It also reads the connection string identified by the *connectionStringName* attribute from the *<connectionStrings>* configuration section, and caches it in a private field, throwing a *ProviderException* if the attribute is empty or nonexistent, or if the attribute references a nonexistent connection string.

Data Schema

SqlProfileProvider stores profile data in the aspnet_Profile table of the provider database. Each record in aspnet_Profile corresponds to one user's persisted profile properties. Table 12 documents the aspnet_Profile table's schema.

Table 12. The aspnet_Profile table

Column Name	Column Type	Description
UserId	uniqueidentifier	ID of the user to which this profile data pertains
PropertyNames	ntext	Names of all property values stored in

		this profile
PropertyValuesString	ntext	Values of properties that could be persisted as text
PropertyValuesBinary	image	Values of properties that were configured to use binary serialization
LastUpdatedDate	datetime	Date and time this profile was last updated

The aspnet_Profile table has a foreign-key relationship with one other provider database table: aspnet_Users (see Table 1-3). The aspnet_Profile table's UserId column references the column of the same name in the aspnet_Users table, and it is used to scope profile data by user.

Additional Scoping of Profile Data

In addition to scoping profile data by user, *SqlProfileProvider* supports scoping by application name. Websites that register profile providers with identical *applicationName* attributes share profile data, whereas websites that register profile providers with unique *applicationName*s do not. Scoping by user name (or user ID) is facilitated by the user ID recorded with each set of persisted profile properties, whereas scoping by application name is facilitated by the application ID accompanying each user ID in the aspnet_Users table.

Data Access

SqlProfileProvider performs all database accesses through stored procedures. Table 13 lists the stored procedures that it uses.

Table 13. Stored procedures used by SqlProfileProvider

Stored Procedure	Description
aspnet_Profile_DeleteInactiveProfiles	Deletes profile data from the aspnet_Profile table for users whose last activity dates in the aspnet_Users table fall on or before the specified date.
aspnet_Profile_DeleteProfiles	Deletes profile data from the aspnet_Profile table for the specified users.
aspnet_Profile_GetNumberOfInactiveProfiles	Queries the aspnet_Profile table to get a count of profiles whose last activity dates (in the aspnet_Users table) fall on or before the specified date.
aspnet_Profile_GetProfiles	Retrieves profile data from the aspnet_Profile table for users who match the criteria input to

	the stored procedure.
aspnet_Profile_GetProperties	Retrieves profile data for the specified user.
aspnet_Profile_SetProperties	Saves profile data for the specified user.

Stored procedure names are generally indicative of the *SqlProfileProvider* methods that call them. For example, ASP.NET calls the default profile provider's *GetPropertyValues* and *SetPropertyValues* methods to read and write profile data, and these methods in turn call the stored procedures named aspnet_Profile_GetProperties and aspnet_Profile_SetProperties, respectively.

Saving Profile Property Values

ASP.NET calls the default profile provider's *SetPropertyValues* method to persist profile properties for a given user. *SqlProfileProvider.SetPropertyValues* performs the following actions:

1. Extracts the user name and a value indicating whether the user is authenticated, from the *SettingsContext* parameter passed to it.
2. Formats the property values for saving, by iterating through the items in the *SettingsPropertyValuesCollection* passed to it, and initializing three variables—*names*, *values*, and *buf*—with the values to be written to the database. (For more information about the format of these variables, see "Persistence Format.")
3. Calls the stored procedure aspnet_Profile_SetProperties to write *names*, *values*, and *buf* to the *PropertyNames*, *PropertyValuesString*, and *PropertyValuesBinary* fields of the provider database, respectively.

SqlProfileProvider.SetPropertyValues delegates the task of initializing *names*, *values*, and *buf* from the *SettingsPropertyValuesCollection* input to it, to a helper method named *PrepareDataForSaving*. That method employs the following logic:

1. Checks the property values in the collection, and returns without doing anything if none of the property values are dirty, or if the collection contains dirty property values, but the user is not authenticated and none of the dirty property values have *allowAnonymous* attributes equal to true. This optimization prevents round-tripping to the database in cases where no data has changed.
2. Iterates through the collection, skipping property values lacking *allowAnonymous* attributes equal to true if the user is not authenticated, as well as property values that are not dirty and whose *UsingDefaultValue* property is true. In the first case, it is the responsibility of the provider to enforce the distinction between anonymous and authenticated use of profile properties. *SqlProfileProvider* does so by simply ignoring any profile properties that have not been marked with the *allowAnonymous* attribute when the current user is anonymous. The second case is a bit more subtle. A profile property can be assigned a default value in the profile definition. Alternatively, a profile property value may have been fetched from the database but never modified in code. If the profile property is using the default value from the profile definition, then the provider avoids the overhead of serializing it, as well as

the storage overhead of storing the value in the database. After all, the value can always be reconstituted from the *defaultValue* attribute in the profile definition.

3. Processes the remaining property values in the *SettingsPropertyValueCollection*, by appending each property's *SerializedValue* property to any data already in *values* or *buf*. String *SerializedValues* are appended to *values*, whereas non-string *SerializedValues* are appended to *buf*. In either case, information denoting where *SerializedValue* was stored is written to *names*. As an optimization, if it finds that a property value's *Deserialized* property is true and it's *PropertyValue* property is null, the helper method doesn't record *SerializedValue* at all; instead, it simply records a length of -1 for that property value in *names*.

When called by *SqlProfileProvider.SetPropertyValues*, aspnet_Profile_SetProperties performs the following actions:

1. Calls the stored procedure aspnet_Applications_CreateApplication to convert the application name input to it into an application ID.
2. Queries the aspnet_Users table to convert the user name input to it into a user ID. If the query returns no records, aspnet_Profile_SetProperties calls aspnet_Users_CreateUser to add the user to the aspnet_Users table and return a user ID.
3. Updates the user's last activity date in the aspnet_Users table with the current date and time.
4. Either updates an existing record in the aspnet_Profile table if an entry for the specified user already exists, or inserts a new one.

aspnet_Profile_SetProperties performs all these steps within a transaction, to ensure that changes are committed as a group or not at all.

Loading Profile Property Values

ASP.NET calls the default profile provider's *GetPropertyValues* method to retrieve profile properties for a given user. *SqlProfileProvider.GetPropertyValues* performs the following actions:

1. Creates an empty *SettingsPropertyValueCollection* to hold the *SettingsPropertyValues* that will ultimately be returned to ASP.NET.
2. Extracts the user name from the *SettingsContext* parameter passed to it.
3. Iterates through the *SettingsPropertyCollection* passed to it and, for each *SettingsProperty* it finds, adds a corresponding *SettingsPropertyValue* to the *SettingsPropertyValueCollection* created in the first step. If a *SettingsProperty* lacks a *serializeAs* attribute, *SqlProfileProvider.GetPropertyValues* sets the *SerializeAs* property of the corresponding *SettingsPropertyValue* to *SettingsSerializeAs.String* if the property value is a string or primitive, or to *SettingsSerializeAs.Xml* if the property value is of any other type.
4. Calls the stored procedure aspnet_Profile_GetProperties to retrieve the user's profile data from the provider database. *SqlProfileProvider.GetPropertyValues* copies the data returned by the stored procedure into variables named *names*, *values*, and *buf*. (For more information about the format of these variables, see "Persistence Format.")

5. Parses the data stored in *names*, *values*, and *buf*, and uses it to initialize the *SettingsPropertyValues*.
6. Returns the *SettingsPropertyValuesCollection* containing the initialized *SettingsPropertyValues* to ASP.NET.

SqlProfileProvider.GetPropertyValues delegates the task of parsing the data retrieved from the database and using it to initialize the *SettingsPropertyValues*, to a helper method named *ParseDataFromDB*. That method decomposes the *names* array into items denoting the names and locations of profile properties. For each item in *names*, *ParseDataFromDB* retrieves the corresponding property value from *values* or *buf*, and writes it to the *SerializedValue* property of the corresponding *SettingsPropertyValue*. As an optimization, if the length recorded for a profile property in *names* is *−1*, and the property represents a reference type (as opposed to a value type), the helper method sets the corresponding *SettingsPropertyValue*'s *PropertyValue* property to null and the *Deserialized* property to *true*, effectively returning a null property value to ASP.NET.

When called by *SqlProfileProvider.GetPropertyValues*, aspnet_Profile_GetProperties performs the following actions:

1. Queries the aspnet_Applications table to convert the application name input to it into an application ID.
2. Queries the aspnet_Users table to convert the user name input to it into a user ID.
3. Queries the aspnet_Profile table for the *PropertyNames*, *PropertyValuesString*, and *PropertyValuesBinary* fields for the specified user.
4. Updates the user's last activity date in the aspnet_Users table with the current date and time.

If anything goes wrong along the wayif, for example, the application ID input to the stored procedure is invalidaspnet_Profile_GetProperties return no records, indicating that no profile data exists for the specified user.

Persistence Format

SqlProfileProvider persists profile properties in three fields of the aspnet_Profile table: *PropertyNames*, *PropertyValuesString*, and *PropertyValuesBinary*. The following is a synopsis of what's stored in each field:

- *PropertyNames* holds a string value containing information about the profile property values present in the *PropertyValuesString* and *PropertyValuesBinary* fields. The string holds a colon-delimited list of items; each item denotes one property value, and it is encoded in the following format:

  ```
  Name:B|S:StartPos:Length
  ```

  Name is the property value's name. The second parameter, which is either B (for "binary") or S (for "string"), indicates whether the corresponding property value is stored in the *PropertyValuesString* field (S) or the *PropertyValuesBinary* field (B). *StartPos* and *Length* indicate the starting position (0-based) of the corresponding property value within these fields, and the length of the data, respectively. A length of -1 indicates that the property is a reference type, and that its value is null.

- *PropertyValuesString* stores profile property values persisted as strings. This includes property values serialized by the .NET Framework's XML serializer, and property values serialized using string type converters. The "S" values in the *PropertyNames* field contain the offsets and lengths needed to decompose *PropertyValuesString* into individual property values.
- *PropertyValuesBinary* stores profile property values persisted in binary formatthat is, profile properties that were serialized using the .NET Framework's binary serializer. The "B" values in the *PropertyNames* field contain the offsets and lengths needed to decompose *PropertyValuesBinary* into individual property values.

Note that profile providers are *not* required to persist data in this format or any other format. The format in which profile data is stored is left to the discretion of the person or persons writing the provider.

Differences Between the Published Source Code and the .NET Framework's SqlProfileProvider

The published source code for *SqlProfileProvider* differs from the .NET Framework version in the following respects:

- Declarative and imperative CAS checks were commented out. Because the source code can be compiled standalone, and thus will run as user code rather than trusted code in the global assembly cache, the CAS checks are not necessary.
- Calls to internal helper methods that integrate with ETW tracing have been commented out in the published version.
- The internal helper methods for packaging data and unpacking data have been cloned into the published provider, so that you can see the logic that is used in the *ParseDataFromDB* and *PrepareDataForSaving* methods.
- Because the standalone provider compiles into a regular assembly, in partial trust applications you will be able to use only string serialization or XML serialization. Binary serialization requires a specific permission assertion (*SecurityPermission.SerializationFormatter*) that has been commented out in the code for the cloned versions of *ParseDataFromDB* and *PrepareDataForSaving*.

Web Event Providers

Web event providers provide the interface between Microsoft ASP.NET's health monitoring subsystem and data sources for the events ("Web events") fired by that subsystem. ASP.NET 2.0 ships with the following Web event providers:

- *EventLogWebEventProvider*, which logs Web events in the Windows event log.
- *SqlWebEventProvider*, which log Web events in Microsoft SQL Server and Microsoft SQL Server Express databases.
- *SimpleMailWebEventProvider* and *TemplatedMailWebEventProvider*, which respond to Web events by sending e-mail.
- *TraceWebEventProvider*, which forwards Web events to diagnostics trace.
- *WmiWebEventProvider*, which forwards Web events to the Microsoft Windows Management Instrumentation (WMI) subsystem.

The Microsoft .NET Framework's *System.Web.Management* namespace includes a class named *WebEventProvider* that defines the basic characteristics of a Web event provider. It also contains a *WebEventProvider*-derivative named *BufferedWebEventProvider* that adds buffering support. *SqlWebEventProvider* derives from *BufferedWebEventProvider* and improves performance by "batching" Web events and committing them to the database en masse. *BufferedWebEventProvider* is prototyped as follows:

```
public abstract class BufferedWebEventProvider : WebEventProvider
{
    // Properties
    public bool UseBuffering { get; }
    public string BufferMode { get; }

    // Virtual methods
    public override void Initialize (string name,
        NameValueCollection config);
    public override void ProcessEvent (WebBaseEvent raisedEvent);
    public override void Flush ();

    // Abstract methods
    public abstract void ProcessEventFlush (WebEventBufferFlushInfo
        flushInfo);
}
```

The following section documents the implementation of *SqlWebEventProvider*.

SqlWebEventProvider

SqlWebEventProvider is the Microsoft Web event provider for SQL Server databases. It logs Web events in the provider database, using the schema documented in "Data

Schema," and it uses the stored procedure documented in "Data Access." Knowledge of the database schema is hidden in the stored procedure, so that porting *SqlWebEventProvider* to other database types requires little more than modifying the stored procedure. (Depending on the targeted database type, the ADO.NET code used to call the stored procedure might have to change, too. The Microsoft Oracle .NET provider, for example, uses a different syntax for named parameters.)

The ultimate reference for *SqlWebEventProvider* is the *SqlWebEventProvider* source code, which is found in SqlWebEventProvider.cs. The sections that follow highlight key aspects of *SqlWebEventProvider*'s design and operation.

Provider Initialization

Initialization occurs in *SqlWebEventProvider.Initialize*, which is called one timewhen the provider is loadedby ASP.NET. *SqlWebEventProvider.Initialize* processes the *connectionStringName* and *maxDetailsEventLength* configuration attributes. Then, it calls *base.Initialize* to process other supported configuration attributes such, as *useBuffering*, and throw an exception if unrecognized configuration attributes remain.

SqlWebEventProvider.Initialize reads the connection string identified by the *connectionStringName* attribute from the *<connectionStrings>* configuration section, and caches it in a private field. It throws a *ConfigurationErrorsException* if the attribute is empty or nonexistent, if the attribute references a nonexistent connection string, or if the connection string doesn't use integrated security.

Data Schema

SqlWebEventProvider logs Web events in the aspnet_WebEvents_Events table of the provider database. Each record in aspnet_ WebEvents_Events corresponds to one Web event. Table 14 documents the aspnet_ WebEvents_Events table's schema.

Table 14. The aspnet_WebEvent_Events table

Column Name	Column Type	Description
EventId	char(32)	Event ID (from *WebBaseEvent.EventId*)
EventTimeUtc	datetime	UTC time at which the event was fired (from *WebBaseEvent.EventTimeUtc*)
EventTime	datetime	Local time at which the event was fired (from *WebBaseEvent.EventTime*)
EventType	nvarchar(256)	Event type (for example, *WebFailureAuditEvent*)
EventSequence	decimal(19,0)	Event sequence number (from *WebBaseEvent.EventSequence*)
EventOccurrence	decimal(19,0)	Event occurrence count (from *WebBaseEvent.EventOccurrence*)
EventCode	int	Event code (from *WebBaseEvent.EventCode*)

EventDetailCode	int	Event detail code (from *WebBaseEvent.EventDetailCode*)
Message	nvarchar(1024)	Event message (from *WebBaseEvent.EventMessage*)
ApplicationPath	nvarchar(256)	Physical path of the application that generated the Web event (for example, C:\Websites\MyApp)
ApplicationVirtualPath	nvarchar(256)	Virtual path of the application that generated the event (for example, /MyApp)
MachineName	nvarchar(256)	Name of the machine on which the event was generated
RequestUrl	nvarchar(1024)	URL of the request that generated the Web event
ExceptionType	nvarchar(256)	If the Web event is a *WebBaseErrorEvent*, type of exception recorded in the *ErrorException* property; otherwise, *DBNull*
Details	ntext	Text generated by calling *ToString* on the Web event

aspnet_WebEvents_Events is a stand-alone table that has no relationships with other tables in the provider database. Many of its fields are filled with values obtained from *WebBaseEvent* properties of the same name. *ApplicationPath*, *ApplicationVirtualPath*, and *MachineName* contain values obtained from the Web event's *ApplicationInformation* property. For details of how values are generated for all aspnet_WebEvents_Events fields, see the *SqlWebEventProvider.FillParams* method in SqlWebEventProvider.cs.

Data Access

SqlWebEventProvider performs all database accesses through the stored procedure named aspnet_WebEvent_LogEvent (Table 15). That stored procedure is a simple one consisting of a single *INSERT* statement that initializes the fields of the new record with the input parameters generated by *SqlWebEventProvider.FillParams*.

Table 15. Stored procedure used by SqlWebEventProvider

Stored Procedure	**Description**
aspnet_WebEvent_LogEvent	Records a Web event in the aspnet_WebEvents_Events table.

Processing Web Events

When a Web event is raised, the Web events subsystem calls the *ProcessEvent* method of each Web event provider mapped to that event type. If buffering is not enabled, *SqlWebEventProvider.ProcessEvent* records the Web event in the provider database, by

calling the helper method *WriteToSQL*. If buffering is enabled, *SqlWebEventProvider.ProcessEvent* buffers the Web event by calling the base class's *ProcessEvent* method. The following is the source code for *SqlWebEventProvider.ProcessEvent*, with diagnostic code removed for clarity:

```
public override void ProcessEvent(WebBaseEvent eventRaised)
{
    if (UseBuffering) {
        base.ProcessEvent(eventRaised);
    }
    else {
        WriteToSQL(new WebBaseEventCollection(eventRaised),
            0, new DateTime(0));
    }
}
```

If buffering is enabled, the Web events subsystem calls the provider's *ProcessEventFlush* method to flush buffered Web events. *ProcessEventFlush*'s job is to read buffered Web events from the event buffer, and commit them to the database. *SqlWebEventProvider.ProcessEventFlush* calls *WriteToSQL* to log the buffered events. as follows:

```
public override void ProcessEventFlush
    (WebEventBufferFlushInfo flushInfo)
{
    WriteToSQL(flushInfo.Events,
        flushInfo.EventsDiscardedSinceLastNotification,
        flushInfo.LastNotificationUtc);
}
```

SqlWebEventProvider.WriteToSQL contains the logic for recording Web events in the provider database. *WriteToSQL* calls *FillParams* to generate the parameters written to the database, and then calls the stored procedure aspnet_WebEvent_LogEvent to write the parameters to the provider database. *WriteToSql* is capable of writing one Web event or multiple Web events, and it contains built-in logic for delaying retries for at least 30 seconds (or the number of seconds specified by the *commandTimeout* configuration attribute) after a failed attempt to write to the database.

Differences Between the Published Source Code and the .NET Framework's SqlWebEventProvider

The published source code for *SqlWebEventProvider* differs from the .NET Framework version in the following respects:

- Declarative and imperative CAS checks were commented out. Because the source code can be compiled standalone, and thus will run as user code rather than trusted code in the global assembly cache, the CAS checks are not necessary.
- The published version includes a derived event type called *MyWebBaseEvent* that is used for manipulating events in the provider. The .NET Framework provider uses the base *WebBaseEvent* class directly, because the .NET Framework provider is able to call internal *WebBaseEvent* methods.

Web Parts Personalization Providers

Web Parts personalization providers provide the interface between Microsoft ASP.NET's Web Parts personalization service and personalization data sources. ASP.NET 2.0 ships with one Web Parts personalization provider: *SqlPersonalizationProvider*, which stores personalization data in Microsoft SQL Server and Microsoft SQL Server Express databases.

The fundamental job of a Web Parts personalization provider is to provide persistent storage for personalization statestate regarding the content and layout of Web Parts pagesgenerated by the Web Parts personalization service. Personalization state is represented by instances of *System.Web.UI.WebControls.WebParts.PersonalizationState*. The personalization service serializes and deserializes personalization state, and presents it to the provider as opaque byte arrays. The heart of a personalization provider is a set of methods that transfer these byte arrays to and from persistent storage.

The Microsoft .NET Framework's *System.Web.UI.WebControls.WebParts* namespace includes a class named *PersonalizationProvider* that defines the basic characteristics of a Web Parts personalization provider. *PersonalizationProvider* is prototyped as follows:

```
public abstract class PersonalizationProvider : ProviderBase
{
    // Properties
    public abstract string ApplicationName { get; set; }

    // Virtual methods
    protected virtual IList CreateSupportedUserCapabilities() {}
    public virtual PersonalizationScope DetermineInitialScope
        (WebPartManager webPartManager,
        PersonalizationState loadedState) {}
    public virtual IDictionary DetermineUserCapabilities
        (WebPartManager webPartManager) {}
    public virtual PersonalizationState LoadPersonalizationState
        (WebPartManager webPartManager, bool ignoreCurrentUser) {}
    public virtual void ResetPersonalizationState
        (WebPartManager webPartManager) {}
    public virtual void SavePersonalizationState
       (PersonalizationState state) {}

    // Abstract methods
    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);
    protected abstract void ResetPersonalizationBlob
        (WebPartManager webPartManager, string path, string userName);
    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);
}
```

The following section documents the implementation of *SqlPersonalizationProvider*, which derives from *PersonalizationProvider*.

SqlPersonalizationProvider

SqlPersonalizationProvider is the Microsoft Web Parts personalization provider for SQL Server databases. It stores personalization data, using the schema documented in "Data Schema," and it uses the stored procedures documented in "Data Access." All knowledge of the database schema is hidden in the stored procedures, so that porting *SqlPersonalizationProvider* to other database types requires little more than modifying the stored procedures. (Depending on the targeted database type, the ADO.NET code used to call the stored procedures might have to change, too. The Microsoft Oracle .NET provider, for example, uses a different syntax for named parameters.)

The ultimate reference for *SqlPersonalizationProvider* is the *SqlPersonalizationProvider* source code, which is found in SqlPersonalizationProvider.cs. The sections that follow highlight key aspects of *SqlPersonalizationProvider*'s design and operation.

Provider Initialization

Initialization occurs in *SqlPersonalizationProvider.Initialize*, which is called one timewhen the provider is loadedby ASP.NET. *SqlPersonalizationProvider.Initialize* processes the *description*, *applicationName*, *connectionStringName*, and *commandTimeout* configuration attributes, and throws a *ProviderException* if unrecognized configuration attributes remain. It also reads the connection string identified by the *connectionStringName* attribute from the *<connectionStrings>* configuration section, and caches it in a private field, throwing a *ProviderException* if the attribute is empty or nonexistent, or if the attribute references a nonexistent connection string.

Data Schema

Web Parts personalization state is inherently scoped by user name and request path. Scoping by user name allows personalization state to be maintained independently for each user. Scoping by path ensures that personalization settings for one page don't affect personalization settings for others. The Web Parts personalization service also supports *shared state*, which is scoped by request path, but not by user name. (When

the service passes shared state to a provider, it passes in a null user name.) When storing personalization state, a provider must take care to key the data by user name and request path, so that it can be retrieved using the same keys later. A provider must also ensure that it stores user-scoped data separately from shared state.

SqlPersonalizationProvider persists per-user personalization state in the aspnet_PersonalizationPerUser table of the provider database, and it persists shared personalization state in the aspnet_PersonalizationAllUsers table. Tables 16 and 17 document the schemas of these two tables, respectively. State is persisted as a serialized blob in the *PageSettings* field. The *PathId* and *UserId* fields store scoping data, while *LastUpdatedDate* stores time stamps.

Table 16. The aspnet_PersonalizationPerUser table

Column Name	Column Type	Description
Id	uniqueidentifier	ID of this record
PathId	uniqueidentifier	ID of the virtual path to which this state pertains
UserId	uniqueidentifier	ID of the user to which this state pertains
PageSettings	image	Serialized personalization state
LastUpdatedDate	datetime	Date and time state was saved

Table 17. The aspnet_PersonalizationAllUsers table

Column Name	Column Type	Description
PathId	uniqueidentifier	ID of the virtual path to which this state pertains
PageSettings	image	Serialized personalization state
LastUpdatedDate	datetime	Date and time state was saved

The aspnet_PersonalizationPerUser and aspnet_PersonalizationAllUsers tables contain columns named PathID that refer to the column of the same name in the aspnet_Paths table (see Table 18). Each entry in the aspnet_Paths table defines one path (for example, ~/MyPage.aspx) for which Web Parts personalization state has been saved. Paths are defined in a separate table, because, for a given path, a personalization provider may be asked to save two types of state: per-user and shared. In that case, both the aspnet_PersonalizationPerUser and aspnet_PersonalizationAllUsers tables will contain entries for the corresponding paths, and each entry will contain a PathId referring to the same entry in aspnet_Paths.

Table 18. The aspnet_Paths table

Column Name	Column Type	Description
ApplicationId	uniqueidentifier	Application ID
PathId	uniqueidentifier	Path ID
Path	nvarchar(256)	Path name

LoweredPath	nvarchar(256)	Path name (lowercase)

The provider database contains a stored procedure named aspnet_Paths_CreatePath that providers (or stored procedures) can call to retrieve a path ID from the aspnet_Paths table, or to create a new one if the specified path doesn't exist.

Additional Scoping of Personalization Data

In addition to scoping personalization state by user name and path, *SqlPersonalizationProvider* supports scoping by application name. Websites that register personalization providers with identical *applicationName* attributes share Web Parts personalization data, whereas websites that register personalization providers with unique *applicationNames* do not. Due to the page-specific and control-specific nature of personalization data, however, it usually doesn't make sense to use the same *applicationName* for Web Parts personalization data across different websites.

In support of application-name scoping, *SqlPersonalizationProvider* records an application ID in the *ApplicationId* field of each record in the aspnet_Paths table. aspnet_Paths' *ApplicationId* field refers to the field of the same name in the aspnet_Applications table, and each unique *applicationName* has a corresponding *ApplicationId* in that table.

Data Access

SqlPersonalizationProvider performs all database accesses through stored procedures. Table 19 lists the stored procedures that it uses.

Table 19. Stored procedures used by SqlPersonalizationProvider

Stored Procedure	Description
aspnet_PersonalizationAdministration_DeleteAllState	Deletes all records from aspnet_PersonalizationAllUsers or aspnet_PersonalizationPerUser corresponding to the specified application ID.
aspnet_PersonalizationAdministration_FindState	Retrieves profile data from aspnet_PersonalizationAllUsers or aspnet_PersonalizationPerUser meeting several input criteria.
aspnet_PersonalizationAdministration_GetCountOfState	Returns a count of records in the aspnet_PersonalizationAllUsers table with path names matching the specified pattern, or a count of records in the aspnet_PersonalizationPerUser table meeting several input criteria.
aspnet_PersonalizationAdministration_ResetSharedState	Resets shared state for the specified page, by deleting the corresponding record from the aspnet_PersonalizationAllUsers table.
aspnet_PersonalizationAdministration_ResetUserState	Resets per-user state for the specified user and the specified page, by deleting the corresponding record from the aspnet_PersonalizationPerUser table. Can also delete

	records, based on the user's last activity date if it falls on or before the specified date.
aspnet_PersonalizationAllUsers_GetPageSettings	Retrieves shared state for the specified page from the aspnet_PersonalizationAllUsers table.
aspnet_PersonalizationAllUsers_ResetPageSettings	Resets shared state for the specified page, by deleting the corresponding record from the aspnet_PersonalizationAllUsers table.
aspnet_PersonalizationAllUsers_SetPageSettings	Saves shared state for the specified page in the aspnet_PersonalizationAllUsers table.
aspnet_PersonalizationPerUser_GetPageSettings	Retrieves per-user state for the specified page and the specified user from the aspnet_PersonalizationPerUser table.
aspnet_PersonalizationPerUser_ResetPageSettings	Resets per-user state for the specified page and the specified user, by deleting the corresponding record from the aspnet_PersonalizationPerUser table.
aspnet_PersonalizationPerUser_SetPageSettings	Saves per-user state for the specified page and the specified user in the aspnet_PersonalizationPerUser table.

Stored procedure names are generally indicative of the *SqlPersonalizationProvider* methods that call them. For example, ASP.NET calls the default Web Parts personalization provider's *SavePersonalizationBlob* method to save personalization state, and *SavePersonalizationBlob*, in turn, calls either aspnet_PersonalizationPerUser_SetPageSettings to save per-user personalization state, or aspnet_PersonalizationAllUsers_SetPageSettings to save shared personalization state, depending on whether it's passed a user name.

Saving Per-User Personalization State

To save Web Parts personalization state for a given user and a given page, ASP.NET calls the *SavePersonalizationBlob* method of the default Web Parts personalization provider, passing in the user name, the path to the page, and a byte array containing the serialized personalization state. *SqlPersonalizationProvider.SavePersonalizationBlob* validates the input parameters and calls the stored procedure aspnet_PersonalizationPerUser_SetPageSettings to write the information to the provider database.

aspnet_PersonalizationPerUser_SetPageSettings performs the following actions:

1. Calls the stored procedure aspnet_Applications_CreateApplication to convert the application name into an application ID, and to create an application record in the aspnet_Applications table if one does not already exist.
2. Calls the stored procedure aspnet_Paths_CreatePath to convert the path into a path ID, and to create a path record in aspnet_Paths if one does not already exist.
3. If the user name input to aspnet_PersonalizationPerUser_SetPageSettings doesn't already exist in the aspnet_Users table, calls aspnet_Users_CreateUser to record a new user and return a user ID.

4. Updates the user's last activity date in the aspnet_Users table with the current date and time.
5. Either updates an existing record in the aspnet_PersonalizationPerUser table if an entry for the specified user and specified path already exists, or inserts a new one.

Currently, neither *SqlPersonalizationProvider.SavePersonalizationBlob* nor aspnet_PersonalizationPerUser_SetPageSettings uses transactions to ensure that all changes (that is, creating a new application record, a new path record, and a new user record) are committed to the database as a whole or not at all, leaving open the possibility that the database could be left in an inconsistent state when all these records need to be created for the very first time.

Loading Per-User Personalization State

To load Web Parts personalization state for a given user and a given page, ASP.NET calls the *LoadPersonalizationBlobs* method of the default Web Parts personalization provider, passing in the user name, the path to the page, and a reference to a byte array through which serialized personalization state is returned. *SqlPersonalizationProvider.LoadPersonalizationBlobs* validates the input parameters, and calls the stored procedure aspnet_PersonalizationPerUser_GetPageSettings to read the information from the provider database.

aspnet_PersonalizationPerUser_GetPageSettings performs the following actions:

1. Calls the stored procedure aspnet_Personalization_GetApplicationId to convert the application name input to it into an application ID.
2. Queries the aspnet_Paths table to convert the path name input to it into a path ID.
3. Queries the aspnet_Users table to convert the user name input to it into a user ID.
4. Updates the user's last activity date in the aspnet_Users table with the current date and time.
5. Queries the PageSettings column of the aspnet_PersonalizationPerUser table for the serialized personalization state.

Saving Shared Personalization State

To save shared Web Parts personalization state for a given page, ASP.NET calls the *SavePersonalizationBlob* method of the default Web Parts personalization provider, passing in a null user name, the path to the page, and a byte array containing the serialized personalization state. *SqlPersonalizationProvider.SavePersonalizationBlob* validates the input parameters and, seeing the null user name, calls the stored procedure aspnet_PersonalizationAllUsers_SetPageSettings to write the information to the provider database.

aspnet_PersonalizationAllUsers_SetPageSettings performs the following actions:

1. Calls the stored procedure aspnet_Applications_CreateApplication to convert the application name into an application ID.
2. Calls the stored procedure aspnet_Paths_CreatePath to convert the path into a path ID.

3. Either updates an existing record in the aspnet_PersonalizationAllUsers table if an entry for the specified path already exists, or inserts a new one.

Currently, neither *SqlPersonalizationProvider.SavePersonalizationBlob* nor aspnet_PersonalizationAllUsers_SetPageSettings uses transactions to ensure that all changes (that is, creating a new application record and a new path record) are committed to the database as a whole or not at all, leaving open the possibility that the database could be left in an inconsistent state when both of these records need to be created for the very first time.

Loading Shared Personalization State

To load shared Web Parts personalization state for a given page, ASP.NET calls the *LoadPersonalizationBlobs* method of the default Web Parts personalization provider, passing in the path to the page, and a reference to a byte array through which serialized personalization state is returned. *SqlPersonalizationProvider.LoadPersonalizationBlobs* validates the input parameters and calls the stored procedure aspnet_PersonalizationAllUsers_GetPageSettings to read the information from the provider database.

aspnet_PersonalizationAllUsers_GetPageSettings performs the following actions:

1. Calls the stored procedure aspnet_Personalization_GetApplicationId to convert the application name input to it into an application ID.
2. Queries the aspnet_Paths table to convert the path name input to it into a path ID.
3. Queries the PageSettings column of the aspnet_PersonalizationAllUsers table for the serialized personalization state.

Differences Between the Published Source Code and the .NET Framework's SqlPersonalizationProvider

The published source code for *SqlPersonalizationProvider* differs from the .NET Framework version in one respect: Declarative and imperative CAS checks were commented out. Because the source code can be compiled standalone, and thus will run as user code rather than trusted code in the global assembly cache, the CAS checks are not necessary.