

The ASP.NET 2.0 Provider Model

Microsoft Corporation

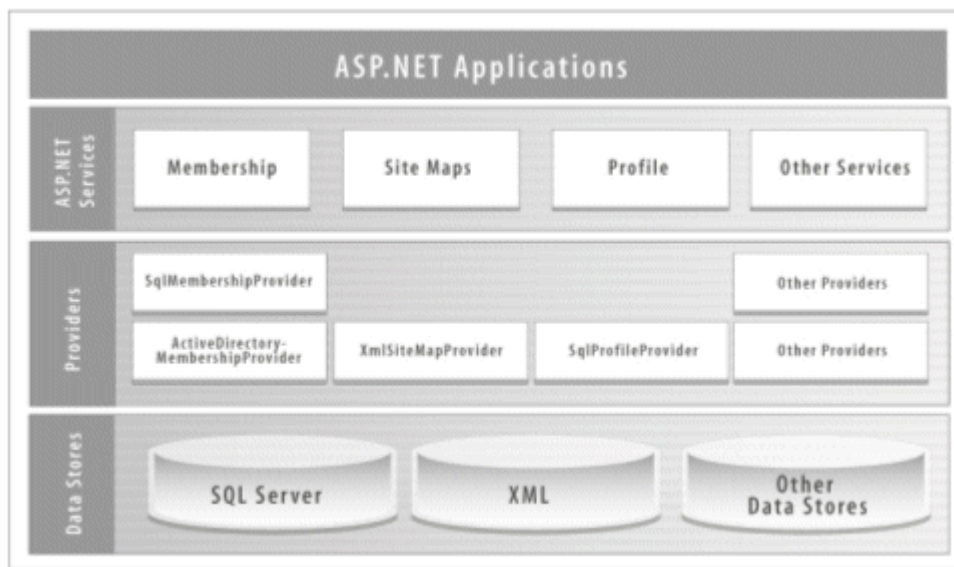
October 2005

Introduction

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, while the membership service stores user names, passwords, and other data in Microsoft SQL Server databases or Active Directory. For the majority of applications, the built-in storage options are sufficient. However, the need sometimes arises to store state in other media such as Oracle databases, DB2 databases, Microsoft SQL Server databases with custom schemas, XML files, or even data sources front-ended by Web services.

In ASP.NET 1.x, developers who wished to store state in alternative storage media were faced with the daunting prospect of rewriting large portions of ASP.NET. ASP.NET 2.0, by contrast, introduces extreme flexibility to state storage. ASP.NET 2.0 services that manage state do not interact directly with storage media; instead, they use providers as intermediaries, as pictured in Figure 1.

Figure 1. The ASP.NET 2.0 provider model



A *provider* is a software module that provides a uniform interface between a service and a data source. Providers abstract physical storage media, in much the same way that device drivers abstract physical hardware devices. Because virtually all ASP.NET 2.0 state-management services are provider-based, storing session state or membership state in an Oracle database rather than a Microsoft SQL Server database is as simple as plugging in Oracle session state and membership providers. Code outside the provider layer needn't be modified, and a simple configuration change, accomplished declaratively through Web.config, connects the relevant services to the Oracle providers.

Thanks to the provider model, ASP.NET 2.0 can be configured to store state virtually anywhere. Membership data, for example, could just as easily come from a Web service

as from a database. All that's required is a custom provider. Some companies will prefer to acquire custom providers from third parties. Others, however, will want to write their own, either because no suitable provider is available off the shelf, or because they wish to adapt ASP.NET to legacy storage media (for example, existing membership databases). This whitepaper documents the ASP.NET 2.0 provider model and provides the critical information that developers need to write robust, high-quality providers.

Goals of the Provider Model

The ASP.NET 2.0 provider model was designed with the following goals in mind:

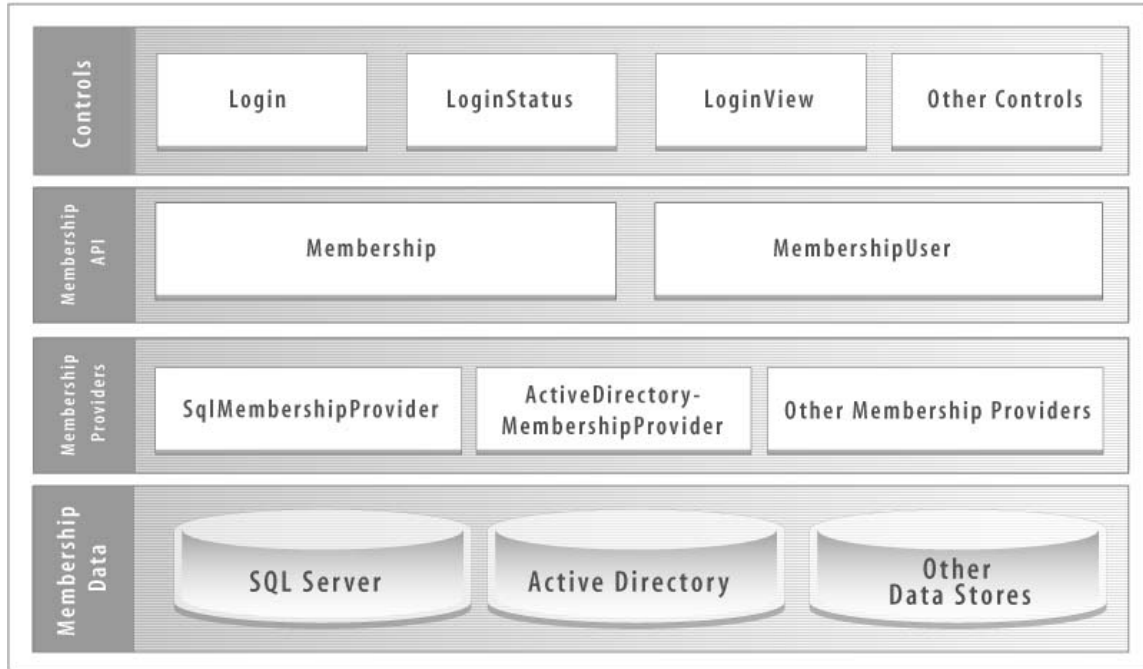
- To make ASP.NET state storage both flexible and extensible
- To insulate application-level code and code in the ASP.NET run-time from the physical storage media where state is stored, and to isolate the changes required to use alternative media types to a single well-defined layer with minimal surface area
- To make writing custom providers as simple as possible by providing a robust and well-documented set of base classes from which developers can derive provider classes of their own

It is expected that developers who wish to pair ASP.NET 2.0 with data sources for which off-the-shelf providers are not available can, with a reasonable amount of effort, write custom providers to do the job.

The Provider Model

Figure 2 depicts the provider model as it applies to the ASP.NET membership service. In the top layer are the login controls: *Login*, *LoginView*, and others that provide UIs for logging in users, recovering lost passwords, and more. Underneath the login controls sits the membership service, which provides the public API used by the controls and that can be used by application code as well. The membership service stores login credentials and other information in a membership data source, but rather than access the data source directly, it interacts with it through a membership provider. Thus, the login controls and the membership service itself can be adapted to different types of data sources (for example, Oracle databases) simply by adding new providers.

Figure 2. The membership provider model



Membership providers implement a well-defined interface consisting of methods and properties defined in an abstract base class named *MembershipProvider*. Because all membership providers are built to the same contract, the membership service can interact with them without knowing or caring how they choose to store the data.

Provider Types

Membership is one of several ASP.NET 2.0 services that use the provider architecture. Table 1 documents the features and services that are provider-based and the default providers that service them:

Table 1. Provider-based services

Feature or Service	Default Provider
Membership	System.Web.Security.SqlMembershipProvider
Role management	System.Web.Security.SqlRoleProvider
Site map	System.Web.XmlSiteMapProvider
Profile	System.Web.Profile.SqlProfileProvider
Session state	System.Web.SessionState.InProcSessionStateStore
Web events	N/A (see below)
Web Parts personalization	System.Web.UI.WebControls.WebParts.SqlPersonalizationProvider
Protected configuration	N/A (see below)

SQL providers such as *SqlMembershipProvider* and *SqlProfileProvider* use Microsoft SQL Server or SQL Server Express as their data source. *InProcSessionStateStore* stores session state in memory, while *XmlSiteMapProvider* uses XML files as its data source. N/A (Not Applicable) designates features and services for which providers must be explicitly identified. These include:

- Web events, which must be explicitly mapped to providers in the <healthMonitoring> configuration section. The master Web.config file maps certain Web events to *System.Web.Management.EventLogWebEventProvider*, causing them to be logged in the Windows event log without any setup.
- Protected configuration, which requires callers who invoke its encryption services to specify a provider. The *Aspnet_regiis.exe* tool that comes with ASP.NET uses protected configuration to encrypt and decrypt configuration sections. Unless instructed to do otherwise, *Aspnet_regiis.exe* calls upon *System.Configuration.RsaProtectedConfigurationProvider* to provide encryption and decryption services.

Built-In Providers

ASP.NET 2.0 ships with the providers listed in Table 2:

Table 2. ASP.NET 2.0 providers

Provider Type	Built-In Provider(s)
Membership	<i>System.Web.Security.ActiveDirectoryMembershipProvider</i> <i>System.Web.Security.SqlMembershipProvider</i>
Role management	<i>System.Web.Security.AuthorizationStoreRoleProvider</i> <i>System.Web.Security.SqlRoleProvider</i> <i>System.Web.Security.WindowsTokenRoleProvider</i>
Site map	<i>System.Web.XmlSiteMapProvider</i>
Profile	<i>System.Web.Profile.SqlProfileProvider</i>
Session state	<i>System.Web.SessionState.InProcSessionStateStore</i> <i>System.Web.SessionState.OutOfProcSessionStateStore</i> <i>System.Web.SessionState.SqlSessionStateStore</i>
Web events	<i>System.Web.Management.EventLogWebEventProvider</i> <i>System.Web.Management.SimpleMailWebEventProvider</i> <i>System.Web.Management.TemplatedMailWebEventProvider</i> <i>System.Web.Management.SqlWebEventProvider</i> <i>System.Web.Management.TraceWebEventProvider</i> <i>System.Web.Management.WmiWebEventProvider</i>
Web Parts personalization	<i>System.Web.UI.WebControls.WebParts.SqlPersonalizationProvider</i>
Protected configuration	<i>System.Configuration.DPAPIProtectedConfigurationProvider</i> <i>System.Configuration.RSAProtectedConfigurationProvider</i>

In addition, Microsoft intends to make a set of providers targeting Microsoft Access available as a free download. Developers are discouraged from using Microsoft Access on the back end of enterprise applications, but Microsoft realizes that Access may be appropriate for small Web sites that serve a very limited number of users.

Provider Base Classes

The *System.Configuration.Provider* namespace includes a class named *ProviderBase* that serves as the root class for all providers. *ProviderBase* is prototyped as follows:

```
public class ProviderBase
{
    public virtual string Name { get; }
    public virtual string Description { get; }
    public virtual void Initialize (string name,
        NameValueCollection config);
}
```

The *Name* property returns the provider's name (for example, "AspNetSqlMembershipProvider"), while *Description* returns a textual description. *Initialize* is called by ASP.NET when the provider is loaded, affording the provider the opportunity to initialize itself. The *name* parameter contains the provider's name; its value comes from the *name* attribute of the <add> element that registered the provider, as in

```
<add name="AspNetSqlMembershipProvider" ... />
```

The *config* parameter contains the remaining name/value pairs present in the <add> element.

The default implementation of *Initialize* ensures that *Initialize* hasn't been called before and initializes *Name* and *Description* from the configuration attributes of the same name. *ProviderBase*'s source code appears as follows.

```
namespace System.Configuration.Provider
{
    using System.Collections.Specialized;
    using System.Runtime.Serialization;

    public abstract class ProviderBase
    {
        private string _name;
        private string _Description;
        private bool _Initialized;
    }
}
```

```

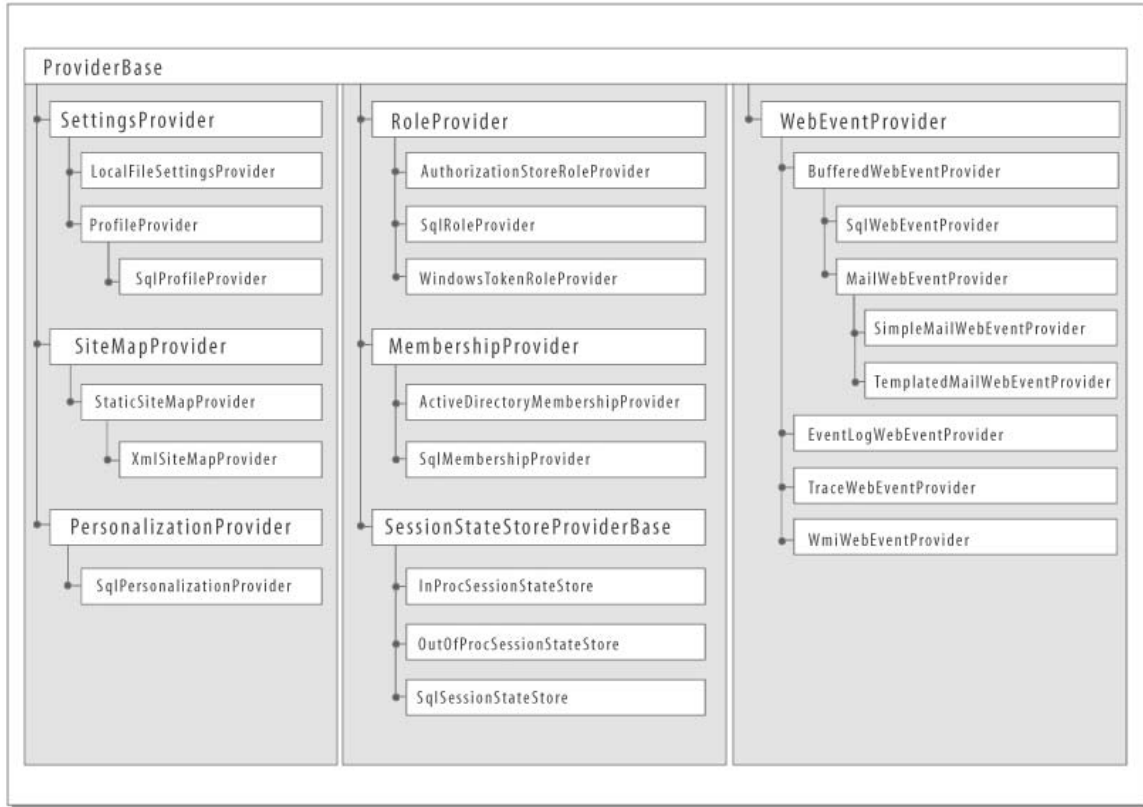
public virtual string Name { get { return _name; } }
public virtual string Description
{
    get { return string.IsNullOrEmpty(_Description) ?
        Name : _Description; }
}

public virtual void Initialize(string name,
    NameValueCollection config)
{
    lock (this) {
        if (_Initialized)
            throw new InvalidOperationException("...");
        _Initialized = true;
    }
    if (name == null)
        throw new ArgumentNullException("name");
    if (name.Length == 0)
        throw new ArgumentException("...", "name");
    _name = name;
    if (config != null) {
        _Description = config["description"];
        config.Remove("description");
    }
}
}
}
}

```

Developers typically derive from *ProviderBase* only if they're writing custom services that are provider-based (see "Custom Provider-Based Services"). The .NET Framework includes *ProviderBase* derivatives that define contracts between services and data sources. For example, the *MembershipProvider* class derives from *ProviderBase* and defines the interface between the membership service and membership data sources. Developers writing custom membership providers should derive from *MembershipProvider* rather than *ProviderBase*. Figure 3 shows the provider class hierarchy.

Figure 3. ASP.NET 2.0 provider classes



Provider Registration and Configuration

Providers are registered in <providers> configuration sections in the configuration sections of the features and services that they serve. For example, membership providers are registered this way:

```

<configuration>
  <system.web>
    <membership ...>
      <providers>
        <!-- Membership providers registered here -->
      </providers>
    </membership>
    ...
  </system.web>
</configuration>
  
```

While role providers are registered like this:

```

<configuration>
  <system.web>
    <roleManager ...>
  
```



```

    <providers>
      <!-- Role providers registered here -->
    </providers>
  </roleManager>

  ...
</system.web>
</configuration>

```

<add> elements within <providers> elements register providers and make them available for use. <add> elements support a common set of configuration attributes such as *name*, *type*, and *description*, plus provider-specific configuration attributes that are unique to each provider:

```

<configuration>
  <system.web>
    <membership ...>
      <providers>
        <add name="AspNetSqlMembershipProvider"
            type="[Type name]"
            description="SQL Server membership provider"
            connectionStringName="LocalSqlServer"

            ...
        />

        ...
      </providers>
    </membership>

    ...
  </system.web>
</configuration>

```

Once registered, a provider is usually designated as the default (active) provider using the *defaultProvider* attribute of the corresponding configuration element. For example, the following <membership> element designates *SqlMembershipProvider* as the default provider for the membership service:

```

<membership defaultProvider="AspNetSqlMembershipProvider">
  <providers>
    ...
  </providers>
</membership>

```

The *defaultProvider* attribute identifies by logical name (rather than type name) a provider registered in <providers>. Note that ASP.NET is not entirely consistent in its

use of the *defaultProvider* attribute. For example, the <sessionState> element uses a *customProvider* attribute to designate the default session state provider.

Any number of providers may be registered for a given service, but only one can be the default. All providers registered for a given service may be enumerated at run-time using the service's *Providers* property (for example, *Membership.Providers*).

General Considerations for Building Custom Providers

All providers have certain characteristics in common. All, for example, initialize themselves when the ASP.NET run-time calls the *Initialize* method that they inherit from *ProviderBase*, and all must be thread-safe. The sections that follow document the key principles and patterns that apply to all providers, regardless of type.

Provider Initialization

All provider classes derive, either directly or indirectly, from *System.Configuration.Provider.ProviderBase*. As such, they inherit a virtual method named *Initialize* that's called by ASP.NET when the provider is loaded. Derived classes should override *Initialize* and use it to perform provider-specific initializations. An overridden *Initialize* method should perform the following tasks:

1. Make sure the provider has the permissions it needs to run and throw an exception if it doesn't. (Alternatively, a provider may use declarative attributes such as *System.Security.Permissions.FileIOPermissionAttribute* to ensure that it has the necessary permissions.)
2. Verify that the *config* parameter passed to *Initialize* isn't null and throw an *ArgumentNullException* if it is.
3. Call the base class version of *Initialize*, ensuring that the *name* parameter passed to the base class is neither null nor an empty string (and assigning the provider a default name if it is), and adding a default *description* attribute to the *config* parameter passed to the base class if *config* currently lacks a *description* attribute or the attribute is empty.
4. Configure itself by reading and applying the configuration attributes encapsulated in *config*, making sure to call *Remove* on each recognized configuration attribute. Configuration attributes can be read using *NameValueCollection*'s string indexer.
5. Throw a *ProviderException* if *config.Count > 0*, which means that the element used to register the provider contains one or more unrecognized configuration attributes.
6. Do anything else the provider needs to do to get itself ready to run—for example, read state from an XML file so the file needn't be reparsed on each request. However, it's critical that *Initialize* not call any feature APIs in the service that the provider serves, because doing so may cause infinite recursion. For example, creating a *MembershipUser* object in a membership provider's *Initialize* method causes *Initialize* to be called again.

The code below shows a canonical *Initialize* method for a SQL Server provider that recognizes one provider-specific configuration attribute named *connectionStringName*. This is a great example to pattern your code after and it closely parallels the *Initialize* methods found in built-in SQL Server providers such as *SqlMembershipProvider*.

```

public override void Initialize(string name,
    NameValueCollection config)
{
    // Verify that the provider has sufficient trust to operate. In
    // this example, a SecurityException will be thrown if the provider
    // lacks permission to call out to SQL Server. The built-in
    // providers tend to be less stringent here, simply ensuring that
    // they're running with at least low trust.
    SqlClientPermission.Demand ();

    // Verify that config isn't null
    if (config == null)
        throw new ArgumentNullException ("config");

    // Assign "name" a default value if it currently has no value
    // or is an empty string
    if (String.IsNullOrEmpty (name))
        name = "SampleSqlProvider";

    // Add a default "description" attribute to config if the
    // attribute doesn't exist or is empty
    if (string.IsNullOrEmpty (config["description"])) {
        config.Remove ("description");
        config.Add ("description", "Sample SQL provider");
    }

    // Call the base class's Initialize method
    base.Initialize(name, config);

    // Initialize _connectionStringName from the connectionStringName
    // configuration attribute, or throw an exception if the attribute
    // doesn't exist or is an empty string, or if it designates a
    // nonexistent connection string
    string connect = config["connectionStringName"];

    if (String.IsNullOrEmpty (connect))
        throw new ProviderException
            ("Empty or missing connectionStringName");

    config.Remove ("connectionStringName");

    if (WebConfigurationManager.ConnectionStrings[connect] == null)
        throw new ProviderException ("Missing connection string");
}

```

```

    _connectionString = WebConfigurationManager.ConnectionStrings
        [connect].ConnectionString;

    if (String.IsNullOrEmpty (_connectionString))
        throw new ProviderException ("Empty connection string");

    // Throw an exception if unrecognized attributes remain
    if (config.Count > 0) {
        string attr = config.GetKey (0);
        if (!String.IsNullOrEmpty (attr))
            throw new ProviderException ("Unrecognized attribute: " +
                attr);
    }
}

```

In the above code, *connectionStringName* represents a required configuration attribute. Consequently, *Initialize* throws an exception if the attribute isn't present. Some attributes may be optional rather than required. In the case of an optional attribute, *Initialize* should assign a sensible default value to the corresponding field or property if the attribute isn't present.

Provider Lifetime

Providers are loaded when the application using them first accesses a feature of the corresponding service, and they're instanced just once per application (that is, per application domain). The lifetime of a provider roughly equals the lifetime of the application, so one can safely write "stateful" providers that store state in fields. This one-instance-per-application model is convenient for persisting data across requests, but it has a downside. That downside is described in the next section.

Thread Safety

In general, ASP.NET goes to great lengths to prevent developers from having to write thread-safe code. HTTP handlers and HTTP modules, for example, are instanced on a per-request (per-thread) basis, so they don't have to be thread-safe unless they access shared state.

Providers are an exception to the one-instance-per-thread rule. ASP.NET 2.0 providers are instanced one time during an application's lifetime and are shared among all requests. Because each request is processed on a different thread drawn from a thread pool that serves ASP.NET, providers can (and probably will be) accessed by two or more threads at the same time. This means providers must be thread-safe. Providers containing non-thread-safe code may seem to work at first (and may work just fine under light loads), but are likely to suffer spurious, hard-to-reproduce data-corruption errors when the load-and consequently, the number of concurrently executing request-processing threads-increases.

The only provider method that doesn't have to be thread-safe is the *Initialize* method inherited from *ProviderBase*. ASP.NET ensures that *Initialize* is never executed on two or more threads concurrently.

A comprehensive discussion of thread-safe code is beyond the scope of this document, but most threading-related concurrency errors result when two or more threads access the same data at the same time and at least one of those threads is writing rather than reading. Consider, for example, the following class definition:

```
public class Foo
{
    private int _count;

    public int Count
    {
        get { return _count; }
        set { _count = value; }
    }
}
```

Suppose two threads each hold a reference to an instance of *Foo* (that is, there are two threads but only one *Foo* object), and one thread reads the object's *Count* property at the same time that the other thread writes it. If the read and write occur at precisely the same time, the thread that does the reading might receive a bogus value. One solution is to use the Framework's *System.Threading.Monitor* class (or its equivalent, C#'s *lock* keyword) to serialize access to *_count*. Here's a revised version of *Foo* that is thread-safe:

```
public class Foo
{
    private int _count;
    private object _syncLock = new object ();

    public int Count
    {
        get { lock (_syncLock) { return _count; } }
        set { lock (_syncLock) { _count = value; } }
    }
}
```

The *System.Threading* namespace includes other synchronization classes as well, including a *ReaderWriterLock* class that allows any number of threads to read shared data concurrently but prevents overlapping reads and writes as well as overlapping writes. (By contrast, the *Monitor* class restricts access to one thread at a time, even if all threads are readers.) In addition, the

System.Runtime.CompilerServices.MethodImplOptions class can be used to serialize access to entire methods:

```
[MethodImpl (MethodImplOptions.Synchronized)]
public void SomeMethod ()
{
    // Only one thread at a time may execute this method
}
```

However, synchronizing with *MethodImpl* is inferior to synchronizing with *Monitor*, *ReaderWriterLock*, and other *System.Threading* classes due to its coarseness in locking out threads. At best, *MethodImpl* (*MethodImplOptions.Synchronized*) locks at the object level—the equivalent of *lock* (*this*). Under certain circumstances, it locks at the type level or even at the application domain level, either of which adversely affects performance and scalability.

Here are some key points regarding thread safety to keep in mind as you write and test custom providers:

- Outside of the *Initialize* method, a thread-safe provider serializes read/write accesses to all instance data, including fields. Accesses need not be serialized if they are read-only. For example, a provider's *Initialize* method might read a connection string from *Web.config* and store it in (write it to) an instance field. The write doesn't have to be synchronized because it's performed in *Initialize*, which never executes on concurrent threads. If all accesses to the field outside of *Initialize* are reads rather than writes, then those accesses do not require synchronization, either
- A thread-safe provider does *not* have to serialize accesses to local variables or other stack-based data
- *Never* pass a value type (such as an *int* or a *struct*) to *lock* or *Monitor.Enter*. The compiler won't let you because if it did, you'd get no synchronization at all due to the fact that the value type would be boxed. That's why the thread-safe version of *Foo* uses *lock* (*_syncLock*) rather than *lock* (*_count*). *_syncLock* is a reference type; *_count* is not
- The best way to test the thread safety of a custom provider is to subject it to heavy loads on a multiprocessor machine

For more information on synchronizing concurrently executing threads in managed code, refer to the following resources:

- "Safe Thread Synchronization" by Jeffrey Richter. Article in the January 2003 issue of MSDN Magazine.
- Chapter 14 of the book "Programming Microsoft .NET" by Jeff Prosise (2002, Microsoft Press).

Localization

For simplicity, the provider code above (and elsewhere in this document) hard-codes error messages, provider descriptions, and other text strings. Providers intended for

general consumption should avoid hard-coded strings and use localized string resources instead.

Atomicity

Some provider operations involve multiple updates to the data source. For example, a role provider's *AddUsersToRoles* method is capable of adding multiple users to multiple roles in a single operation and therefore may require multiple updates to the data source. When the data source supports transactions (for example, when the data source is SQL Server), it is recommended that you use transactions to ensure the atomicity of updates—that is, to roll back already-completed updates if a subsequent update fails. If the data source doesn't support transactions, the provider author is responsible for ensuring that updates are performed either in whole or not at all.

Exceptions and Exception Types

The .NET Framework's *System.Configuration.Provider* namespace includes a general-purpose exception class named *ProviderException* that providers can throw when errors occur. In general, exception types should be as specific as possible. For example, when a null reference is passed to a method that requires a non-null reference, the provider should throw an *ArgumentNullException*. Similarly, when an empty string is passed to a method that requires a non-empty string, the provider should throw an *ArgumentException*.

More general errors, such as asking a role provider to delete a role that doesn't exist, can be handled by throwing *ProviderExceptions*. If desired, you can define your own error-specific exception classes and use them in lieu of *ProviderExceptions*. The providers included with ASP.NET 2.0 use *ProviderExceptions* extensively to flag errors.

Implementation Completeness

A provider is not required to implement the full contract defined by the base class it derives from. For example, a membership provider may choose not to implement the *GetNumberOfUsersOnline* method inherited from *MembershipProvider*. (The method must be overridden in the derived class to satisfy the compiler, but its implementation might simply throw a *NotSupportedException*.) Obviously, the more complete the implementation the better, and developers should take care to document any features that aren't supported.

Membership Providers

Membership providers provide the interface between ASP.NET's membership service and membership data sources. The two most common reasons for writing a custom membership provider are:

- You wish to store membership information in a data source that is not supported by the membership providers included with the .NET Framework, such as an Oracle database or a Web service.
- You wish to store membership information in a SQL Server database whose schema differs from that of the database used by *System.Web.Security.SqlMembershipProvider*, for example, you need to integrate ASP.NET's membership service with an existing membership database.

The fundamental job of a membership provider is to interface with data sources containing 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 .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.

The MembershipProvider Class

Developers writing custom membership providers begin by deriving from *System.Web.Security.MembershipProvider*, which derives from *ProviderBase* and adds abstract methods and properties (as well as a handful of virtuals) defining 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 table describes *MembershipProvider*'s methods and properties and provides helpful notes regarding their implementation:

Table 3. MembershipProvider methods and properties

Method or Property	Description
<i>EnablePasswordRetrieval</i>	Indicates whether passwords can be retrieved using the provider's <i>GetPassword</i> method. This property is read-only.
<i>EnablePasswordReset</i>	Indicates whether passwords can be reset using the provider's <i>ResetPassword</i> method. This property is read-only.
<i>RequiresQuestionAndAnswer</i>	Indicates whether a password answer must be supplied when calling the provider's <i>GetPassword</i> and <i>ResetPassword</i> methods. This property is read-only.
<i>ApplicationName</i>	The name of the application using the membership provider. <i>ApplicationName</i> is used to scope membership data so that applications can choose whether to share membership data with other applications. This property can be read and written.
<i>MaxInvalidPasswordAttempts</i>	Works in conjunction with <i>PasswordAttemptWindow</i> to provide a safeguard against password guessing. If the number of consecutive invalid passwords or password questions ("invalid attempts") submitted to the provider for a given user reaches <i>MaxInvalidPasswordAttempts</i> within the number of minutes specified by <i>PasswordAttemptWindow</i> , the user is locked out of the system. The user remains locked out until the provider's <i>UnlockUser</i>

	<p>method is called to remove the lock.</p> <p>The count of consecutive invalid attempts is incremented when an invalid password or password answer is submitted to the provider's <i>ValidateUser</i>, <i>ChangePassword</i>, <i>ChangePasswordQuestionAndAnswer</i>, <i>GetPassword</i>, and <i>ResetPassword</i> methods.</p> <p>If a valid password or password answer is supplied before the <i>MaxInvalidPasswordAttempts</i> is reached, the count of consecutive invalid attempts is reset to zero. If the <i>RequiresQuestionAndAnswer</i> property is false, invalid password answer attempts are not tracked.</p> <p>This property is read-only.</p>
<i>PasswordAttemptWindow</i>	For a description, see <i>MaxInvalidPasswordAttempts</i> . This property is read-only.
<i>RequiresUniqueEmail</i>	Indicates whether each registered user must have a unique e-mail address. This property is read-only.
<i>PasswordFormat</i>	Indicates what format that passwords are stored in: clear (plaintext), encrypted, or hashed. Clear and encrypted passwords can be retrieved; hashed passwords cannot. This property is read-only.
<i>MinRequiredPasswordLength</i>	The minimum number of characters required in a password. This property is read-only.
<i>MinRequiredNonAlphanumericCharacters</i>	The minimum number of non-alphanumeric characters required in a password. This property is read-only.
<i>PasswordStrengthRegularExpression</i>	A regular expression specifying a pattern to which passwords must conform. This property is read-only.
<i>CreateUser</i>	Takes, as input, a user name, password, e-mail address, and other information and adds a new user to the membership data source. <i>CreateUser</i> returns a <i>MembershipUser</i> object representing the newly created user. It also accepts an out parameter (in Visual Basic, <i>ByRef</i>) that returns a <i>MembershipCreateStatus</i> value

indicating whether the user was successfully created or, if the user was not created, the reason why. If the user was not created, CreateUser returns null.

Before creating a new user, CreateUser calls the provider's virtual OnValidatingPassword method to validate the supplied password. It then creates the user or cancels the action based on the outcome of the call.

ChangePasswordQuestionAndAnswer

Takes, as input, a user name, password, password question, and password answer and updates the password question and answer in the data source if the user name and password are valid. This method returns true if the password question and answer are successfully updated. Otherwise, it returns false.

ChangePasswordQuestionAndAnswer returns false if either the user name or password is invalid.

GetPassword

Takes, as input, a user name and a password answer and returns that user's password. If the user name is not valid, GetPassword throws a ProviderException.

Before retrieving a password, GetPassword verifies that EnablePasswordRetrieval is true. If EnablePasswordRetrieval is false, GetPassword throws a NotSupportedException. If EnablePasswordRetrieval is true but the password format is hashed, GetPassword throws a ProviderException since hashed passwords cannot, by definition, be retrieved. A membership provider should also throw a ProviderException from Initialize if EnablePasswordRetrieval is true but the password format is hashed.

GetPassword also checks the value of the RequiresQuestionAndAnswer property before retrieving a password. If RequiresQuestionAndAnswer is true, GetPassword compares the supplied password answer to the stored password answer and throws a MembershipPasswordException if the two

don't match. GetPassword also throws a MembershipPasswordException if the user whose password is being retrieved is currently locked out.

ChangePassword

Takes, as input, a user name, a password (the user's current password), and a new password and updates the password in the membership data source. ChangePassword returns true if the password was updated successfully. Otherwise, it returns false.

Before changing a password, ChangePassword calls the provider's virtual OnValidatingPassword method to validate the new password. It then changes the password or cancels the action based on the outcome of the call.

If the user name, password, new password, or password answer is not valid, ChangePassword does not throw an exception; it simply returns false.

Following a successful password change, ChangePassword updates the user's LastPasswordChangedDate.

ResetPassword

Takes, as input, a user name and a password answer and replaces the user's current password with a new, random password. ResetPassword then returns the new password. A convenient mechanism for generating a random password is the Membership.GeneratePassword method.

If the user name is not valid, ResetPassword throws a ProviderException. ResetPassword also checks the value of the RequiresQuestionAndAnswer property before resetting a password. If RequiresQuestionAndAnswer is true, ResetPassword compares the supplied password answer to the stored password answer and throws a MembershipPasswordException if the two don't match.

Before resetting a password, ResetPassword verifies that EnablePasswordReset is true. If EnablePasswordReset is false,

ResetPassword throws a `NotSupportedException`. If the user whose password is being changed is currently locked out, ResetPassword throws a `MembershipPasswordException`.

Before resetting a password, ResetPassword calls the provider's virtual `OnValidatingPassword` method to validate the new password. It then resets the password or cancels the action based on the outcome of the call. If the new password is invalid, ResetPassword throws a `ProviderException`.

Following a successful password reset, ResetPassword updates the user's `LastPasswordChangedDate`.

UpdateUser

Takes, as input, a `MembershipUser` object representing a registered user and updates the information stored for that user in the membership data source. If any of the input submitted in the `MembershipUser` object is not valid, UpdateUser throws a `ProviderException`.

Note that UpdateUser is not obligated to allow all the data that can be encapsulated in a `MembershipUser` object to be updated in the data source.

ValidateUser

Takes, as input, a user name and a password and verifies that they are valid—that is, that the membership data source contains a matching user name and password. ValidateUser returns true if the user name and password are valid, if the user is approved (that is, if `MembershipUser.IsApproved` is true), and if the user isn't currently locked out. Otherwise, it returns false.

Following a successful validation, ValidateUser updates the user's `LastLoginDate` and fires an `AuditMembershipAuthenticationSuccess` Web event. Following a failed validation, it fires an `AuditMembershipAuthenticationFailure` Web event.

<i>UnlockUser</i>	<p>Unlocks (that is, restores login privileges for) the specified user. <code>UnlockUser</code> returns true if the user is successfully unlocked. Otherwise, it returns false. If the user is already unlocked, <code>UnlockUser</code> simply returns true.</p>
<i>GetUser</i>	<p>Takes, as input, a user name or user ID (the method is overloaded) and a Boolean value indicating whether to update the user's <code>LastActivityDate</code> to show that the user is currently online. <code>GetUser</code> returns a <code>MembershipUser</code> object representing the specified user. If the user name or user ID is invalid (that is, if it doesn't represent a registered user) <code>GetUser</code> returns null (Nothing in Visual Basic).</p>
<i>GetUserNameByEmail</i>	<p>Takes, as input, an e-mail address and returns the first registered user name whose e-mail address matches the one supplied.</p> <p>If it doesn't find a user with a matching e-mail address, <code>GetUserNameByEmail</code> returns an empty string.</p>
<i>DeleteUser</i>	<p>Takes, as input, a user name and deletes that user from the membership data source. <code>DeleteUser</code> returns true if the user was successfully deleted. Otherwise, it returns false.</p> <p><code>DeleteUser</code> takes a third parameter—a Boolean named <code>deleteAllRelatedData</code>—that specifies whether related data for that user should be deleted also. If <code>deleteAllRelatedData</code> is true, <code>DeleteUser</code> should delete role data, profile data, and all other data associated with that user.</p>
<i>GetAllUsers</i>	<p>Returns a <code>MembershipUserCollection</code> containing <code>MembershipUser</code> objects representing all registered users. If there are no registered users, <code>GetAllUsers</code> returns an empty <code>MembershipUserCollection</code>.</p> <p>The results returned by <code>GetAllUsers</code> are constrained by the <code>pageIndex</code> and <code>pageSize</code> input parameters. <code>pageSize</code> specifies the maximum number of</p>

MembershipUser objects to return. pageIndex identifies which page of results to return. Page indexes are 0-based.

GetAllUsers also takes an out parameter (in Visual Basic, ByRef) named totalRecords that, on return, holds a count of all registered users.

GetNumberOfUsersOnline

Returns a count of users that are currently online-that is, whose LastActivityDate is greater than the current date and time minus the value of the membership service's UserIsOnlineTimeWindow property, which can be read from Membership.UserIsOnlineTimeWindow. UserIsOnlineTimeWindow specifies a time in minutes and is set using the <membership> element's userIsOnlineTimeWindow attribute.

FindUsersByName

Returns a MembershipUserCollection containing MembershipUser objects representing users whose user names match the usernameToMatch input parameter. Wildcard syntax is data source-dependent. MembershipUser objects in the MembershipUserCollection are sorted by user name. If FindUsersByName finds no matching users, it returns an empty MembershipUserCollection.

For an explanation of the pageIndex, pageSize, and totalRecords parameters, see the GetAllUsers method.

FindUsersByEmail

Returns a MembershipUserCollection containing MembershipUser objects representing users whose e-mail addresses match the emailToMatch input parameter. Wildcard syntax is data source-dependent. MembershipUser objects in the MembershipUserCollection are sorted by e-mail address. If FindUsersByEmail finds no matching users, it returns an empty MembershipUserCollection.

For an explanation of the pageIndex, pageSize, and totalRecords parameters, see the GetAllUsers method.

EncryptPassword

Takes, as input, a byte array containing a plaintext password and returns a byte array containing the password in encrypted form. The default implementation in *MembershipProvider* encrypts the password using *<machineKey>*'s *decryptionKey*, but throws an exception if the decryption key is autogenerated.

Override only if you want to customize the encryption process. Do not call the base class's *EncryptPassword* method if you override this method.

DecryptPassword

Takes, as input, a byte array containing an encrypted password and returns a byte array containing the password in plaintext form. The default implementation in *MembershipProvider* decrypts the password using *<machineKey>*'s *decryptionKey*, but throws an exception if the decryption key is autogenerated.

Override only if you want to customize the decryption process. Do not call the base class's *DecryptPassword* method if you override this method.

OnValidatingPassword

Virtual method called when a password is created. The default implementation in *MembershipProvider* fires a *ValidatingPassword* event, so be sure to call the base class's *OnValidatingPassword* method if you override this method. The *ValidatingPassword* event allows applications to apply additional tests to passwords by registering event handlers.

A custom provider's *CreateUser*, *ChangePassword*, and *ResetPassword* methods (in short, all methods that record new passwords) should call this method.

Your job in implementing a custom membership provider in a derived class is to override and provide implementations of *MembershipProvider*'s abstract members, and optionally to override key virtuals such as *Initialize*.



Inside the ASP.NET Team

Do you wonder why *MembershipProvider's EncryptPassword* and *DecryptPassword* methods throw exceptions if `<machineKey>`'s decryption key (which also happens to be an encryption key) is autogenerated? Here's how one member of the ASP.NET team explained it:

"Back in the alpha we kept running across developers that worked a little bit on one machine and then picked up their MDB and copied it to another machine. At which point surprise! none of the passwords could be decrypted any more. So we decided to disallow autogenerated keys when using encrypted passwords. The reality is that autogenerated keys are really fragile. It's just way too easy to get yourself in a situation where these keys change. And once that happens, you are left with a useless membership database."

Scoping of Membership Data

All membership providers inherit from *MembershipProvider* a property named *ApplicationName* whose purpose it to scope the data managed by the provider. Applications that specify the same *ApplicationName* when configuring the membership service share membership data; applications that specify unique *ApplicationNames* do not. Membership provider implementations must therefore associate users with application names so operations performed on membership data sources can be scoped accordingly.

As an example, a provider that stores membership data in a SQL database might use a command similar to the following to determine whether a specified user name and password are valid for the application named "Contoso:"

```
SELECT COUNT (*) FROM Users
WHERE UserName='Jeff' AND Password='imbatman'
AND ApplicationName='Contoso'
```

The final AND in the WHERE clause ensures that other applications containing identical user names and passwords don't produce false positives in the "Contoso" application.

Strong Password Policies

A full-featured membership provider supports strong password policies. Before creating a password, a membership provider should verify that the password contains at least the number of characters specified by the *MinRequiredPasswordLength* property, at least the number of non-alphanumeric characters specified by *MinRequiredNonAlphanumericCharacters*, and, if *PasswordStrengthRegularExpression* is neither null nor empty, that the password conforms to the pattern specified by the regular expression stored in that property. A membership consumer can then enact strong password policies in either of two ways:

- Set the provider's `PasswordStrengthRegularExpression` property to null and use `MinRequiredPasswordLength` and `MinRequiredNonAlphanumericCharacters` to specify minimum character counts
- Set the provider's `MinRequiredPasswordLength` property to 1 and `MinRequiredNonAlphanumericCharacters` to 0 and use `PasswordStrengthRegularExpression` to specify a regular expression defining acceptable password formats

Password-validation logic should be applied in all provider methods that create or accept new passwords, including `CreateUser`, `ChangePassword`, and `ResetPassword`. This logic is not automatically supplied by `MembershipProvider`.

Account Locking

A full-featured membership provider also supports the locking out of users after a consecutive number of failed login attempts within a specified time period. A consumer uses the provider's `MaxInvalidPasswordAttempts` and `PasswordAttemptsWindow` properties to configure this feature. Once locked out, a user may not log in again until his or her account is unlocked. `MembershipProvider` defines an `Unlock` method for unlocking locked-out users, and the `System.Web.Security.MembershipUser` class, which represents individual users managed by the membership service, includes an `IsLockedOut` property that indicates whether the corresponding user is currently locked out of the system.

ReadOnlyXmlMembershipProvider

Figure 4 contains the source code for a membership provider named `ReadOnlyXmlMembershipProvider` that demonstrates the minimum functionality required of a membership provider—the provider equivalent of "hello, world." Despite its simplicity, `ReadOnlyXmlMembershipProvider` is capable of supporting applications that authenticate users using `Login` controls or direct calls to `Membership.ValidateUser`. It also provides data regarding membership users to applications that request it using `Membership` methods such as `GetUser` and `GetAllUsers`. It does not support `Membership` methods such as `CreateUser` and `ChangePassword` that modify the data source, hence the "ReadOnly" in the class name. `ReadOnlyXmlMembershipProvider` methods that write to the data source throw `NotSupportedException`s if called.

Figure 4. ReadOnlyXmlMembershipProvider

```
using System;
using System.Xml;
using System.Collections.Generic;
using System.Collections.Specialized;
using System.Configuration.Provider;
using System.Web.Security;
using System.Web.Hosting;
using System.Web.Management;
using System.Security.Permissions;
using System.Web;

public class ReadOnlyXmlMembershipProvider : MembershipProvider
```

```
{
    private Dictionary<string, MembershipUser> _Users;
    private string _XmlFileName;

    // MembershipProvider Properties
    public override string ApplicationName
    {
        get { throw new NotSupportedException(); }
        set { throw new NotSupportedException(); }
    }

    public override bool EnablePasswordRetrieval
    {
        get { return false; }
    }

    public override bool EnablePasswordReset
    {
        get { return false; }
    }

    public override int MaxInvalidPasswordAttempts
    {
        get { throw new NotSupportedException(); }
    }

    public override int MinRequiredNonAlphanumericCharacters
    {
        get { throw new NotSupportedException(); }
    }

    public override int MinRequiredPasswordLength
    {
        get { throw new NotSupportedException(); }
    }

    public override int PasswordAttemptWindow
    {
        get { throw new NotSupportedException(); }
    }

    public override MembershipPasswordFormat PasswordFormat
    {
        get { throw new NotSupportedException(); }
    }
}
```

```

public override string PasswordStrengthRegularExpression
{
    get { throw new NotSupportedException(); }
}

public override bool RequiresQuestionAndAnswer
{
    get { throw new NotSupportedException(); }
}

public override bool RequiresUniqueEmail
{
    get { throw new NotSupportedException(); }
}

// MembershipProvider Methods
public override void Initialize (string name,
    NameValueCollection config)
{
    // Verify that config isn't null
    if (config == null)
        throw new ArgumentNullException ("config");

    // Assign the provider a default name if it doesn't have one
    if (String.IsNullOrEmpty (name))
        name = "ReadOnlyXmlMembershipProvider";

    // Add a default "description" attribute to config if the
    // attribute doesn't exist or is empty
    if (string.IsNullOrEmpty (config["description"])) {
        config.Remove ("description");
        config.Add ("description",
            "Read-only XML membership provider");
    }

    // Call the base class's Initialize method
    base.Initialize(name, config);

    // Initialize _XmlFileName and make sure the path
    // is app-relative
    string path = config["xmlFileName"];

    if (String.IsNullOrEmpty (path))
        path = "~/App_Data/Users.xml";
}

```

```

if (!VirtualPathUtility.IsAppRelative(path))
    throw new ArgumentException
        ("xmlFileName must be app-relative");

string fullyQualifiedPath = VirtualPathUtility.Combine
    (VirtualPathUtility.AppendTrailingSlash
    (HttpRuntime.AppDomainAppVirtualPath), path);

_XmlFileName = HostingEnvironment.MapPath(fullyQualifiedPath);
config.Remove ("xmlFileName");

// Make sure we have permission to read the XML data source and
// throw an exception if we don't
FileIOPermission permission =
    new FileIOPermission(FileIOPermissionAccess.Read,
        _XmlFileName);
permission.Demand();

// Throw an exception if unrecognized attributes remain
if (config.Count > 0) {
    string attr = config.GetKey (0);
    if (!String.IsNullOrEmpty (attr))
        throw new ProviderException
            ("Unrecognized attribute: " + attr);
}
}

public override bool ValidateUser(string username, string password)
{
    // Validate input parameters
    if (String.IsNullOrEmpty(username) ||
        String.IsNullOrEmpty(password))
        return false;

    try
    {
        // Make sure the data source has been loaded
        ReadMembershipDataStore();

        // Validate the user name and password
        MembershipUser user;
        if (_Users.TryGetValue (username, out user))
        {
            if (user.Comment == password) // Case-sensitive

```

```

        {
            // NOTE: A read/write membership provider
            // would update the user's LastLoginDate here.
            // A fully featured provider would also fire
            // an AuditMembershipAuthenticationSuccess
            // Web event
            return true;
        }
    }

    // NOTE: A fully featured membership provider would
    // fire an AuditMembershipAuthenticationFailure
    // Web event here
    return false;
}
catch (Exception)
{
    return false;
}
}

public override MembershipUser GetUser(string username,
    bool userIsOnline)
{
    // Note: This implementation ignores userIsOnline

    // Validate input parameters
    if (String.IsNullOrEmpty(username))
        return null;

    // Make sure the data source has been loaded
    ReadMembershipDataStore();

    // Retrieve the user from the data source
    MembershipUser user;
    if (_Users.TryGetValue (username, out user))
        return user;

    return null;
}

public override MembershipUserCollection GetAllUsers(int pageIndex,
    int pageSize, out int totalRecords)
{
    // Note: This implementation ignores pageIndex and pageSize,

```

```
// and it doesn't sort the MembershipUser objects returned

// Make sure the data source has been loaded
ReadMembershipDataStore();

MembershipUserCollection users =
    new MembershipUserCollection();

foreach (KeyValuePair<string, MembershipUser> pair in _Users)
    users.Add(pair.Value);

totalRecords = users.Count;
return users;
}

public override int GetNumberOfUsersOnline()
{
    throw new NotSupportedException();
}

public override bool ChangePassword(string username,
    string oldPassword, string newPassword)
{
    throw new NotSupportedException();
}

public override bool
    ChangePasswordQuestionAndAnswer(string username,
    string password, string newPasswordQuestion,
    string newPasswordAnswer)
{
    throw new NotSupportedException();
}

public override MembershipUser CreateUser(string username,
    string password, string email, string passwordQuestion,
    string passwordAnswer, bool isApproved, object providerUserKey,
    out MembershipCreateStatus status)
{
    throw new NotSupportedException();
}

public override bool DeleteUser(string username,
    bool deleteAllRelatedData)
{

```



```
        throw new NotSupportedException();
    }

    public override MembershipUserCollection
        FindUsersByEmail(string emailToMatch, int pageIndex,
            int pageSize, out int totalRecords)
    {
        throw new NotSupportedException();
    }

    public override MembershipUserCollection
        FindUsersByName(string usernameToMatch, int pageIndex,
            int pageSize, out int totalRecords)
    {
        throw new NotSupportedException();
    }

    public override string GetPassword(string username, string answer)
    {
        throw new NotSupportedException();
    }

    public override MembershipUser GetUser(object providerUserKey,
        bool userIsOnline)
    {
        throw new NotSupportedException();
    }

    public override string GetUserNameByEmail(string email)
    {
        throw new NotSupportedException();
    }

    public override string ResetPassword(string username,
        string answer)
    {
        throw new NotSupportedException();
    }

    public override bool UnlockUser(string userName)
    {
        throw new NotSupportedException();
    }

    public override void UpdateUser(MembershipUser user)
```

```

    {
        throw new NotSupportedException();
    }

    // Helper method
    private void ReadMembershipDataStore()
    {
        lock (this)
        {
            if (_Users == null)
            {
                _Users = new Dictionary<string, MembershipUser>
                    (16, StringComparer.InvariantCultureIgnoreCase);
                XmlDocument doc = new XmlDocument();
                doc.Load(_XmlFileName);
                XmlNodeList nodes = doc.GetElementsByTagName("User");

                foreach (XmlNode node in nodes)
                {
                    MembershipUser user = new MembershipUser(
                        Name, // Provider name
                        node["UserName"].InnerText, // Username
                        null, // providerUserKey
                        node["EMail"].InnerText, // Email
                        String.Empty, // passwordQuestion
                        node["Password"].InnerText, // Comment
                        true, // isApproved
                        false, // isLockedOut
                        DateTime.Now, // creationDate
                        DateTime.Now, // lastLoginDate
                        DateTime.Now, // lastActivityDate
                        DateTime.Now, // lastPasswordChangedDate
                        new DateTime(1980, 1, 1) // lastLockoutDate
                    );

                    _Users.Add(user.UserName, user);
                }
            }
        }
    }
}

```

ReadOnlyXmlMembershipProvider uses an XML file with a schema matching that of the below as its data source (see Figure 5). Each <User> element defines one membership user. To avoid redundant file I/O and XML parsing, the provider reads the XML file once

and stores the data in a dictionary of *MembershipUser* objects. Each object in the dictionary is keyed with a user name, making lookups fast and easy.

Figure 5. Sample ReadOnlyXmlMembershipProvider Data Source

```
<Users>
  <User>
    <UserName>Bob</UserName>
    <Password>contoso!</Password>
    <EMail>bob@contoso.com</EMail>
  </User>
  <User>
    <UserName>Alice</UserName>
    <Password>contoso!</Password>
    <EMail>alice@contoso.com</EMail>
  </User>
</Users>
```

ReadOnlyXmlMembershipProvider supports one custom configuration attribute: *xmlFileName*. The provider's *Initialize* method initializes a private field named *_XmlFileName* with the attribute value and defaults to *~/App_Data/Users.xml* if the attribute isn't present. The *Web.config* file in Figure 6 registers *ReadOnlyXmlMembershipProvider*, makes it the default membership provider, and points it to *MembershipUsers.xml* (located in the application root) as the data source. The type name specified in the *<add>* element assumes that the provider is deployed in an assembly named *CustomProviders*.

Figure 6. Web.config file making ReadOnlyXmlMembershipProvider the default membership provider

```
<configuration>
  <system.web>
    <membership defaultProvider="AspNetReadOnlyXmlMembershipProvider">
      <providers>
        <add name="AspNetReadOnlyXmlMembershipProvider"
            type="ReadOnlyXmlMembershipProvider, CustomProviders"
            description="Read-only XML membership provider"
            xmlFileName="~/App_Data/MembershipUsers.xml"
          />
      </providers>
    </membership>
  </system.web>
</configuration>
```

As you peruse *ReadOnlyXmlMembershipProvider*'s source code, here are a few key points to keep in mind regarding its implementation:

- For simplicity, *ReadOnlyXmlMembershipProvider* doesn't support encrypted or hashed passwords. Passwords are stored in plaintext, and they're stored in the

Comment properties of the corresponding *MembershipUser* objects since the *MembershipUser* class, by design, lacks a *Password* property. In practice, *MembershipUser* objects should *never* store passwords. Passwords should stay in the data source, and they should be stored in encrypted or hashed form in the absence of a compelling reason to do otherwise. (In fact, it's quite acceptable for membership providers to not support plaintext password storage as long as that fact is documented.)

- *ReadOnlyXmlMembershipProvider* doesn't read the XML data source in *Initialize*; rather, it loads it on demand, the first time the data is needed. This is done for a pragmatic reason. The very act of creating *MembershipUser* objects in *Initialize* would cause *Initialize* to be called again, resulting in an infinite loop and an eventual stack overflow. As stated in "Provider Initialization," a provider's *Initialize* method must avoid making calls into the service that the provider serves because doing so may cause deadly reentrancies.
- For simplicity, *ReadOnlyXmlMembershipProvider* doesn't scope membership data using the *ApplicationName* property. Instead, it assumes that different applications will target different membership data sources by specifying different XML file names.
- *ReadOnlyXmlMembershipProvider*'s *GetAllUsers* method doesn't honor the *pageIndex* and *PageSize* parameters, nor does it sort the *MembershipUser* objects that it returns by user name.
- *ReadOnlyXmlMembershipProvider* contains minimal thread synchronization code because most of its work involves reading, not writing. It does lock when reading the membership data source to ensure that two threads won't try to initialize the in-memory representation of that source (a *Dictionary* object) at the same time.
- *ReadOnlyXmlMembershipProvider.Initialize* calls *Demand* on a *FileIOPermission* object to verify that it can read the XML data source. It delays making the call until after processing the *xmlFileName* configuration attribute so it knows the data source's file name.

ReadOnlyXmlMembershipProvider is a good starting point for understanding membership providers, but a full-featured provider must implement methods that write to the data source as well as methods that read from them. A full-featured provider must also support non-plaintext password storage and scoping by *ApplicationName*.

Role Providers

Role providers provide the interface between ASP.NET's role management service (the "role manager") and role data sources. The two most common reasons for writing a custom role provider are:

- You wish to store role information in a data source that is not supported by the role providers included with the .NET Framework, such as an Oracle database or a Web service.
- You wish to store role information in a SQL Server database whose schema differs from that of the database used by *System.Web.Security.SqlRoleProvider*-if, for example, you need to integrate ASP.NET's role manager with an existing role database.

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. Given a user name, the role manager relies on the role provider to determine whether what role or roles the user belongs to. The role manager also implements administrative methods such as *Roles.CreateRole* and *Roles.AddUserToRole* by calling the underlying methods in the provider.

The RoleProvider Class

Developers writing custom role providers begin by deriving from *System.Web.Security.RoleProvider*, which derives from *ProviderBase* and adds abstract methods and properties defining 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 table describes *RoleProvider*'s members and provides helpful notes regarding their implementation. Unless otherwise noted, *RoleProvider* methods that accept user names, role names, and other strings as input consider null (*Nothing* in Visual Basic) or empty strings to be errors and throw *ArgumentNullExceptions* or *ArgumentExceptions* in response.

Table 4. RoleProvider methods and properties

Method or Property	Description
<i>ApplicationName</i>	The name of the application using the role provider. <i>ApplicationName</i> is used to scope role data so that applications can choose whether to share role data with other applications. This property can be read and written.
<i>IsUserInRole</i>	Takes, as input, a user name and a role name and determines whether the specified user is associated with the specified role. If the user or role does not exist, <i>IsUserInRole</i> throws a <i>ProviderException</i> .
<i>GetRolesForUser</i>	Takes, as input, a user name and returns the names of the roles to which the user belongs. If the user is not assigned to any roles, <i>GetRolesForUser</i> returns an empty string array (a string array with no elements). If the user name does not exist, <i>GetRolesForUser</i> throws a <i>ProviderException</i> .
<i>CreateRole</i>	Takes, as input, a role name and creates the specified role. <i>CreateRole</i> throws a <i>ProviderException</i> if the role already exists, the role name contains a comma, or the role name exceeds the maximum length allowed by the data source.
<i>DeleteRole</i>	Takes, as input, a role name and a Boolean value that indicates whether to throw an exception if there are users currently associated with the role, and then deletes the specified role. If the <i>throwOnPopulatedRole</i> input parameter is true and the specified role has one or more members, <i>DeleteRole</i> throws a <i>ProviderException</i> and does not delete the role. If <i>throwOnPopulatedRole</i> is false, <i>DeleteRole</i> deletes the

	<p>role whether it is empty or not.</p> <p>When <i>DeleteRole</i> deletes a role and there are users assigned to that role, it also removes users from the role.</p>
<i>RoleExists</i>	<p>Takes, as input, a role name and determines whether the role exists.</p>
<i>AddUsersToRoles</i>	<p>Takes, as input, a list of user names and a list of role names and adds the specified users to the specified roles.</p> <p><i>AddUsersToRoles</i> throws a <i>ProviderException</i> if any of the user names or role names do not exist. If any user name or role name is null (Nothing in Visual Basic), <i>AddUsersToRoles</i> throws an <i>ArgumentNullException</i>. If any user name or role name is an empty string, <i>AddUsersToRoles</i> throws an <i>ArgumentException</i>.</p>
<i>RemoveUsersFromRoles</i>	<p>Takes, as input, a list of user names and a list of role names and removes the specified users from the specified roles.</p> <p><i>RemoveUsersFromRoles</i> throws a <i>ProviderException</i> if any of the users or roles do not exist, or if any user specified in the call does not belong to the role from which he or she is being removed.</p>
<i>GetUsersInRole</i>	<p>Takes, as input, a role name and returns the names of all users assigned to that role.</p> <p>If no users are associated with the specified role, <i>GetUserInRole</i> returns an empty string array (a string array with no elements). If the role does not exist, <i>GetUsersInRole</i> throws a <i>ProviderException</i>.</p>
<i>GetAllRoles</i>	<p>Returns the names of all existing roles. If no roles exist, <i>GetAllRoles</i> returns an empty string array (a string array with no elements).</p>
<i>FindUsersInRole</i>	<p>Takes, as input, a search pattern and a role name and returns a list of users belonging to the specified role whose user names match the pattern. Wildcard syntax is data-source-dependent and may vary from provider to provider. User names are returned in alphabetical order.</p> <p>If the search finds no matches, <i>FindUsersInRole</i> returns an empty string array (a string array with no elements). If the role does not exist, <i>FindUsersInRole</i> throws a <i>ProviderException</i>.</p>

Your job in implementing a custom role provider in a derived class is to override and provide implementations of *RoleProvider*'s abstract members, and optionally to override key virtuals such as *Initialize*.

Scoping of Role Data

All role providers inherit from *RoleProvider* a property named *ApplicationName* whose purpose it to scope the data managed by the provider. Applications that specify the same *ApplicationName* when configuring the role provider share role data; applications that specify unique *ApplicationNames* do not. Role provider implementations must therefore associate role names with application names so operations performed on role data sources can be scoped accordingly.

As an example, a role provider that stores role data in a SQL database might use a command similar to the following to delete the role named "Administrators" from the application named "Contoso:"

```
DELETE FROM Roles
WHERE Role='Administrators' AND ApplicationName='Contoso'
```

The AND in the WHERE clause ensures that other applications containing roles named "Administrators" are not affected.

ReadOnlyXmlRoleProvider

Figure 7 contains the source code for a rudimentary role provider named *ReadOnlyXmlRoleProvider*-the counterpart to the *ReadOnlyXmlMembershipProvider* class presented in "Membership Providers." *ReadOnlyXmlRoleProvider* supports applications that use the ASP.NET role manager to restrict access to resources based on role memberships. It implements *RoleProvider* methods that read from the data source, but it doesn't implement methods that write to the data source. Roles methods such as *IsUserInRole* and *RoleExists* will work when *ReadOnlyXmlRoleProvider* is acting as the role provider; methods such as *CreateRole* and *AddUserToRole* will not.

Figure 7. ReadOnlyXmlRoleProvider

```
using System;
using System.Web.Security;
using System.Collections.Generic;
using System.Collections.Specialized;
using System.Configuration.Provider;
using System.Web.Hosting;
using System.Xml;
using System.Security.Permissions;
using System.Web;

public class ReadOnlyXmlRoleProvider : RoleProvider
{
    private Dictionary<string, string[]> _UsersAndRoles =
```



```

        new Dictionary<string, string[]>(16,
        StringComparer.InvariantCultureIgnoreCase);

private Dictionary<string, string[]> _RolesAndUsers =
    new Dictionary<string, string[]>(16,
    StringComparer.InvariantCultureIgnoreCase);

private string _XmlFileName;

// RoleProvider properties
public override string ApplicationName
{
    get { throw new NotSupportedException(); }
    set { throw new NotSupportedException(); }
}

// RoleProvider methods
public override void Initialize(string name,
    NameValueCollection config)
{
    // Verify that config isn't null
    if (config == null)
        throw new ArgumentNullException("config");

    // Assign the provider a default name if it doesn't have one
    if (String.IsNullOrEmpty(name))
        name = "ReadOnlyXmlRoleProvider";

    // Add a default "description" attribute to config if the
    // attribute doesn't exist or is empty
    if (string.IsNullOrEmpty(config["description"]))
    {
        config.Remove("description");
        config.Add("description", "Read-only XML role provider");
    }

    // Call the base class's Initialize method
    base.Initialize(name, config);

    // Initialize _XmlFileName and make sure the path
    // is app-relative
    string path = config["xmlFileName"];

    if (String.IsNullOrEmpty (path))
        path = "~/App_Data/Users.xml";
}

```

```

if (!VirtualPathUtility.IsAppRelative(path))
    throw new ArgumentException
        ("xmlFileName must be app-relative");

string fullyQualifiedPath = VirtualPathUtility.Combine
    (VirtualPathUtility.AppendTrailingSlash
    (HttpRuntime.AppDomainAppVirtualPath), path);

_XmlFileName = HostingEnvironment.MapPath(fullyQualifiedPath);
config.Remove ("xmlFileName");

// Make sure we have permission to read the XML data source and
// throw an exception if we don't
FileIOPermission permission =
    new FileIOPermission(FileIOPermissionAccess.Read,
        _XmlFileName);
permission.Demand();

// Throw an exception if unrecognized attributes remain
if (config.Count > 0)
{
    string attr = config.GetKey(0);
    if (!String.IsNullOrEmpty(attr))
        throw new ProviderException
            ("Unrecognized attribute: " + attr);
}

// Read the role data source. NOTE: Unlike
// ReadOnlyXmlMembershipProvider, this provider can
// read the data source at this point because Read-
// RoleDataStore doesn't call into the role manager
ReadRoleDataStore();
}

public override bool IsUserInRole(string username, string roleName)
{
    // Validate input parameters
    if (username == null || roleName == null)
        throw new ArgumentNullException();
    if (username == String.Empty || roleName == string.Empty)
        throw new ArgumentException();

    // Make sure the user name and role name are valid
    if (!_UsersAndRoles.ContainsKey (username))

```

```

        throw new ProviderException("Invalid user name");
    if (!_RolesAndUsers.ContainsKey(roleName))
        throw new ProviderException("Invalid role name");

    // Determine whether the user is in the specified role
    string[] roles = _UsersAndRoles[username];
    foreach (string role in roles)
    {
        if (String.Compare(role, roleName, true) == 0)
            return true;
    }

    return false;
}

public override string[] GetRolesForUser(string username)
{
    // Validate input parameters
    if (username == null)
        throw new ArgumentNullException();
    if (username == string.Empty)
        throw new ArgumentException();

    // Make sure the user name is valid
    string[] roles;
    if (!_UsersAndRoles.TryGetValue(username, out roles))
        throw new ProviderException("Invalid user name");

    // Return role names
    return roles;
}

public override string[] GetUsersInRole(string roleName)
{
    // Validate input parameters
    if (roleName == null)
        throw new ArgumentNullException();
    if (roleName == string.Empty)
        throw new ArgumentException();

    // Make sure the role name is valid
    string[] users;
    if (!_RolesAndUsers.TryGetValue (roleName, out users))
        throw new ProviderException("Invalid role name");
}

```

```
        // Return user names
        return users;
    }

    public override string[] GetAllRoles()
    {
        int i = 0;
        string[] roles = new string[_RolesAndUsers.Count];
        foreach (KeyValuePair<string, string[]> pair in _RolesAndUsers)
            roles[i++] = pair.Key;
        return roles;
    }

    public override bool RoleExists(string roleName)
    {
        // Validate input parameters
        if (roleName == null)
            throw new ArgumentNullException();
        if (roleName == string.Empty)
            throw new ArgumentException();

        // Determine whether the role exists
        return _RolesAndUsers.ContainsKey(roleName);
    }

    public override void CreateRole(string roleName)
    {
        throw new NotSupportedException();
    }

    public override bool DeleteRole(string roleName,
        bool throwOnPopulatedRole)
    {
        throw new NotSupportedException();
    }

    public override void AddUsersToRoles(string[] usernames,
        string[] roleNames)
    {
        throw new NotSupportedException();
    }

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

```

```

        throw new NotSupportedException();
    }

    public override void RemoveUsersFromRoles(string[] usernames,
        string[] roleNames)
    {
        throw new NotSupportedException();
    }

    // Helper method
    private void ReadRoleDataStore()
    {
        XmlDocument doc = new XmlDocument();
        doc.Load(_XmlFileName);
        XmlNodeList nodes = doc.GetElementsByTagName("User");

        foreach (XmlNode node in nodes)
        {
            if (node["UserName"] == null)
                throw new ProviderException
                    ("Missing UserName element");

            string user = node["UserName"].InnerText;
            if (String.IsNullOrEmpty(user))
                throw new ProviderException("Empty UserName element");

            if (node["Roles"] == null ||
                String.IsNullOrEmpty (node["Roles"].InnerText))
                _UsersAndRoles.Add(user, new string[0]);
            else
            {
                string[] roles = node["Roles"].InnerText.Split(',');

                // Add the role names to _UsersAndRoles and
                // key them by user name
                _UsersAndRoles.Add(user, roles);

                foreach (string role in roles)
                {
                    // Add the user name to _RolesAndUsers and
                    // key it by role names
                    string[] users1;

                    if (_RolesAndUsers.TryGetValue(role, out users1))
                    {

```


ReadOnlyXmlRoleProvider supports one custom configuration attribute: *xmlFileName*. The provider's *Initialize* method initializes a private field named *_XmlFileName* with the attribute value and defaults to *Roles.xml* if the attribute isn't present. The *Web.config* file in Figure 9 registers *ReadOnlyXmlRoleProvider*, makes it the default role provider, and designates *~/App_Data/UserRoles.xml* as the data source. The type name specified in the *<add>* element assumes that the provider is deployed in an assembly named *CustomProviders*.

Figure 9. Web.config file making *ReadOnlyXmlRoleProvider* the default role provider

```
<configuration>
  <system.web>
    <roleManager enabled="true"
      defaultProvider="AspNetReadOnlyXmlRoleProvider">
      <providers>
        <add name="AspNetReadOnlyXmlRoleProvider"
          type="ReadOnlyXmlRoleProvider, CustomProviders"
          description="Read-only XML role provider"
          xmlFileName="~/App_Data/UserRoles.xml"
        />
      </providers>
    </roleManager>
  </system.web>
</configuration>
```

For simplicity, *ReadOnlyXmlRoleProvider* doesn't scope role data using the *ApplicationName* property. Instead, it assumes that different applications will target different role data sources by specifying different XML file names.

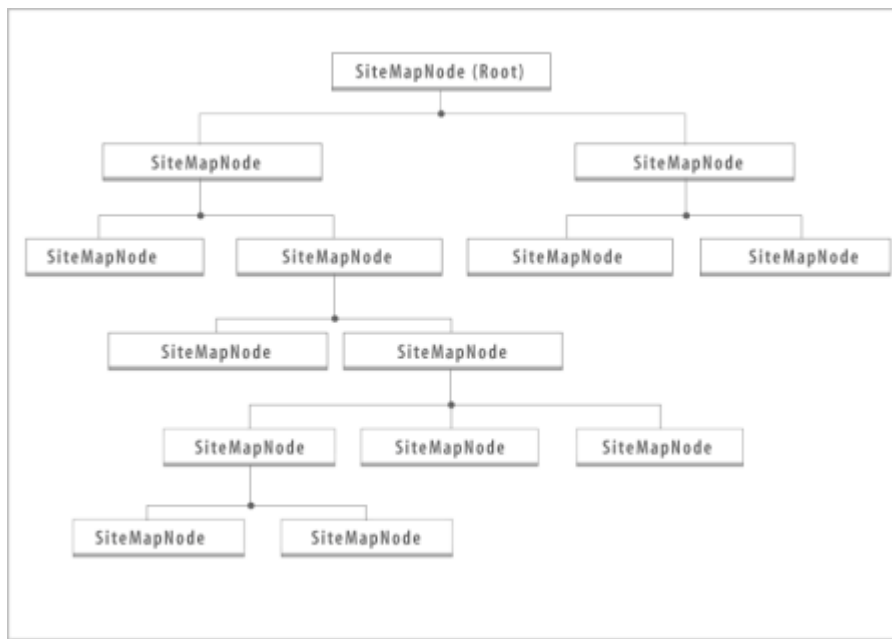
Site Map Providers

Site map providers provide the interface between ASP.NET's data-driven site-navigation features and site map data sources. The two most common reasons for writing a custom site map provider are:

- You wish to store site maps in a data source that is not supported by the site map providers included with the .NET Framework, such as a Microsoft SQL Server database.
- You wish to store site map data in an XML file whose schema differs from that of the one used by *System.Web.XmlSiteMapProvider*.

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 (Figure 10), 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 10. Site map structure



ASP.NET assumes that each site map node's URL is unique with respect to other URLs in the site map. However, site maps do support nodes without URLs. Every *SiteMapNode* has a *Key* property that the provider initializes with a value that uniquely identifies the node. ASP.NET's *XmlSiteMapProvider* class sets *SiteMapNode.Key* equal to the node's URL if *SiteMapNode.Url* is neither null nor empty, or to a randomly generated GUID otherwise. Site map providers include methods for finding nodes by URL or by key.

Developers writing custom site map providers typically begin by deriving from *System.Web.StaticSiteMapProvider*, which derives from *System.Web.SiteMapProvider*. However, developers also have the option of deriving from *SiteMapProvider* directly. *SiteMapProvider* defines the basic contract between ASP.NET and site map providers. *StaticSiteMapProvider* goes much further, providing default implementations of most of *SiteMapProvider*'s abstract methods and overriding key virtuals to provide functional, even optimized, implementations.

Deriving from *StaticSiteMapProvider* is appropriate for custom providers that read node data once (or infrequently) and then cache the information for the lifetime of the provider. Deriving from *SiteMapProvider* is appropriate for custom providers that query a database or other underlying data source in every method call.

The SiteMapProvider Class

System.Web.SiteMapProvider is prototyped as follows:

```
public abstract class SiteMapProvider : ProviderBase
{
    // Public events
    public event SiteMapResolveEventHandler SiteMapResolve;

    // Public properties
    public virtual SiteMapNode CurrentNode { get; }
    public bool EnableLocalization { get; set; }
    public virtual SiteMapProvider ParentProvider { get; set; }
    public string ResourceKey { get; set; }
    public virtual SiteMapProvider RootProvider { get; }
    public virtual SiteMapNode RootNode { get; }
    public bool SecurityTrimmingEnabled { get; }

    // Non-virtual methods
    protected SiteMapNode ResolveSiteMapNode(HttpContext context) {}
    protected static SiteMapNode GetRootNodeCoreFromProvider
        (SiteMapProvider provider) {}

    // Virtual methods
    public override void Initialize(string name,
        NameValueCollection attributes) {}
    protected virtual void AddNode(SiteMapNode node) {}
    protected internal virtual void AddNode(SiteMapNode node,
        SiteMapNode parentNode) {}
    protected internal virtual void RemoveNode(SiteMapNode node) {}
    public virtual SiteMapNode FindSiteMapNode(HttpContext context) {}
    public virtual SiteMapNode FindSiteMapNodeFromKey(string key) {}
    public virtual SiteMapNode GetCurrentNodeAndHintAncestorNodes
        (int upLevel) {}
}
```

```

public virtual SiteMapNode GetCurrentNodeAndHintNeighborhoodNodes
    (int upLevel, int downLevel) {}
public virtual SiteMapNode
    GetParentNodeRelativeToCurrentNodeAndHintDownFromParent
    (int walkupLevels, int relativeDepthFromWalkup) {}
public virtual SiteMapNode
    GetParentNodeRelativeToNodeAndHintDownFromParent
    (SiteMapNode node, int walkupLevels,
    int relativeDepthFromWalkup) {}
public virtual void HintAncestorNodes(SiteMapNode node,
    int upLevel) {}
public virtual void HintNeighborhoodNodes(SiteMapNode node,
    int upLevel, int downLevel) {}
public virtual bool IsAccessibleToUser(HttpContext context,
    SiteMapNode node) {}

// Abstract methods
public abstract GetChildNodes(SiteMapNode node);
public abstract SiteMapNode FindSiteMapNode(string rawUrl);
public abstract SiteMapNode GetParentNode(SiteMapNode node);
protected internal abstract SiteMapNode GetRootNodeCore();
}

```

The following table describes *SiteMapProvider's* methods and properties and provides helpful notes regarding their implementation:


Table 5. SiteMapProvider methods and properties

Method or Property	Description
<i>CurrentNode</i>	Returns a SiteMapNode reference to the site map node representing the current page (the page targeted by the current request). This property is read-only.
<i>EnableLocalization</i>	Indicates whether localization is enabled for this provider. This property can be read or written.
<i>ParentProvider</i>	Gets or sets a SiteMapProvider reference to this provider's parent provider, if any. Get accessor returns null (Nothing in Visual Basic) if this provider doesn't have a parent. This property can be read or written.
<i>ResourceKey</i>	Gets or sets the provider's resource key-the root name of the resource files from which SiteMapNodes extract values for properties such as Title and Description. This property can be read or written.

<i>RootProvider</i>	Returns a SiteMapProvider reference to the root site map provider. This property is read-only.
<i>RootNode</i>	Returns a SiteMapNode reference to the site map's root node. This property is read-only.
<i>SecurityTrimmingEnabled</i>	Returns a Boolean indicating whether security trimming is enabled. SiteMapProvider initializes this property from the securityTrimmingEnabled configuration attribute. This property is read-only.
<i>Initialize</i>	Called to initialize the provider. The default implementation in SiteMapProvider initializes the provider's Description property, calls base.Initialize, and initializes the SecurityTrimmingEnabled property from the securityTrimmingEnabled configuration attribute.
<i>AddNode (SiteMapNode)</i>	Adds a root SiteMapNode to the site map. The default implementation in SiteMapProvider calls AddNode (node, null), which throws a NotImplementedException.
<i>AddNode (SiteMapNode, SiteMapNode)</i>	Adds a SiteMapNode to the site map as a child of the specified SiteMapNode, or as the root node if the specified SiteMapNode is null (Nothing in Visual Basic). The default implementation in SiteMapProvider throws a NotImplementedException, so this method should be overridden in a derived class.
<i>RemoveNode</i>	Removes the specified SiteMapNode from the site map. The default implementation in SiteMapProvider throws a NotImplementedException, so this method should be overridden in a derived class.
<i>FindSiteMapNode (HttpContext)</i>	Retrieves a SiteMapNode representing the current page. The default implementation in SiteMapProvider calls the abstract FindSiteMapNode method to return the SiteMapNode that corresponds to HttpContext.Request.RawUrl.
<i>FindSiteMapNode (string)</i>	Returns a SiteMapNode representing the page at the specified URL. Returns null (Nothing in Visual Basic) if the specified node isn't found. This method is abstract (MustOverride in Visual Basic) and must be overridden in a derived class.
<i>FindSiteMapNodeFromKey</i>	Retrieves a SiteMapNode keyed by the specified key—that is, a SiteMapNode whose Key property matches the input key. Node lookups are normally performed based on URLs, but this method is provided so nodes that lack URLs can be retrieved from the site map. Returns null (Nothing in Visual Basic) if the specified

	node isn't found. The default implementation in SiteMapProvider always returns null, so this method should be overridden in a derived class.
<i>GetCurrentNodeAndHintAncestorNodes</i>	Retrieves a SiteMapNode representing the current page. The default implementation in SiteMapProvider simply returns CurrentNode, but derived classes that don't store entire site maps in memory can override this method and return a SiteMapNode along with the number of generations of ancestor nodes specified in the upLevel parameter. Returns null (Nothing in Visual Basic) if the specified node isn't found.
<i>GetCurrentNodeAndHintNeighborhoodNodes</i>	Retrieves a SiteMapNode representing the current page. The default implementation in SiteMapProvider simply returns CurrentNode, but derived classes that don't store entire site maps in memory can override this method and return a SiteMapNode along with the number of generations of ancestor nodes specified in the upLevel parameter and the number of generations of descendant nodes specified in the downLevel parameter. Returns null (Nothing in Visual Basic) if the specified node isn't found.
<i>GetParentNodeRelativeToCurrentNodeAndHintDownFromParent</i>	Retrieves a SiteMapNode representing an ancestor of the current node the specified number of generations higher in the hierarchy. Derived classes that don't store entire site maps in memory can override this method and return a SiteMapNode along with the number of generations of descendant nodes specified in the relativeDepthFromWalkup parameter. Returns null (Nothing in Visual Basic) if the specified node isn't found.
<i>GetParentNodeRelativeToNodeAndHintDownFromParent</i>	Same as GetParentNodeRelativeToCurrentNodeAndHintDownFromParent, but takes a SiteMapNode as input and performs lookup relative to that node rather than the current node.
<i>HintAncestorNodes</i>	Takes the specified SiteMapNode and fills its ancestor hierarchy with the number of generations specified in the UpLevel parameter. The default implementation in SiteMapProvider does nothing.
<i>HintNeighborhoodNodes</i>	Takes the specified SiteMapNode and fills its ancestor and descendant hierarchies with the number of generations specified in the UpLevel parameter and downLevel parameters. The default implementation in SiteMapProvider does nothing.
<i>IsAccessibleToUser</i>	Returns a Boolean indicating whether the current user

	<p>has permission to access the specified SiteMapNode. The default implementation in SiteMapProvider returns false if security trimming is enabled and the user does not belong to any of the roles associated with the SiteMapNode and the user does not have access to the corresponding URL according to any URL or file authorization rules currently in effect. SiteMapProvider also recognizes the role name "*" as an indication that everyone is permitted to access this node. Security trimming is described more fully in "Security Trimming."</p>
<i>GetChildNodes</i>	<p>Returns a SiteMapNodeCollection representing the specified SiteMapNode's children. Returns an empty SiteMapNodeCollection if the node has no children.</p> <p>This method is abstract (MustOverride in Visual Basic) and must be overridden in a derived class.</p>
<i>GetParentNode</i>	<p>Returns a SiteMapNode representing the specified SiteMapNode's parent. Returns null (in Visual Basic, Nothing) if the node has no parent.</p> <p>This method is abstract (MustOverride in Visual Basic) and must be overridden in a derived class.</p>
<i>GetRootNodeCore</i>	<p>Returns a SiteMapNode representing the root node of the site map managed by this provider. If this is the only site map provider, or if it is the top provider in a hierarchy of providers, GetRootNodeCore returns the same SiteMapNode as SiteMap.RootNode. Otherwise, it may return a SiteMapNode representing a node elsewhere in the site map hierarchy.</p> <p>This method is abstract (MustOverride in Visual Basic) and must be overridden in a derived class.</p>



Inside the ASP.NET Team

You may think it curious that in addition to supporting methods such as *GetChildNodes* and *GetParent* for retrieving nodes from a site map provider, the *SiteMapProvider* class also has "Hint" methods such as *GetCurrentNodeAndHintAncestorNodes* that can be called to say "give me a node and make sure that the hierarchy of nodes above and below it are populated to a specified depth." After all, isn't a *SiteMapNode* implicitly wired to its parent and child nodes without explicitly demanding that it be so? Here's some background from the ASP.NET team:

"We added the Hint methods prior to Beta 1 after meeting with the Sharepoint*

and Content Management teams. Since for Sharepoint their custom site map providers run against a database, they wanted ways to be able to tell the provider that certain pieces of additional data above and beyond a specific requested node may be needed. Some of the methods incorporate both the hint and return a node (e.g., *GetCurrentNodeAndHint-AncestorNodes*), while others are just straight hint methods (e.g., *HintAncestorNodes*). "Either way, we would still want providers to implement methods such as *GetChildNodes* and *GetParent* to return nodes that can navigate into their children or up to their parent. A provider could just choose to return the set of relevant nodes based on the method that is called. Inside of *SiteMapNode.Parent* and *SiteMapNode.ChildNodes*, the provider is called to fetch more nodes. This is where the hint methods are useful. A developer who understands the usage patterns on their site can intelligently hint the provider that parent or child nodes "around" a specific node may be requested, and thus a custom provider can proactively fetch and cache the information."

A site map provider that stores entire site maps in memory can simply use the default implementations of the "Hint" methods. Site map providers that don't store entire site maps in memory, however, might choose to override these methods and use them to optimize *SiteMapNode* lookups.


Security Trimming

The *SiteMapProvider* class contains built-in support for security trimming, which restricts the visibility of site map nodes based on users' role memberships. *SiteMapProvider* implements a Boolean read-only property named *SecurityTrimmingEnabled*, which indicates whether security trimming is enabled. Furthermore, *SiteMapProvider.Initialize* 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 custom provider 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 "*" if everyone (including unauthenticated users and users who enjoy no role memberships) is permitted.

Security trimming doesn't necessarily prevent *SiteMapProvider.IsAccessibleToUser* from returning true if the user doesn't belong to one of the roles specified in a node's *Roles* property. Here's a synopsis of how *IsAccessibleToUser* uses *SiteMapNode.Roles* to authorize access to a node—that is, to determine whether to return true or false:

1. If the current user is in a role specified in the node's *Roles* property, or if *Roles* is "*", the node is returned.
2. If the current user is not in a role specified in the node's *Roles* property, then a URL authorization check is performed to determine whether the user has access to the node's URL. If the answer is yes, the node is returned.
3. If Windows authentication is being used on the site, and if (1) and (2) failed, then a file authorization (ACL) check is performed against the node's URL using the current user's security token. If the ACL check succeeds, the node is returned.

Nodes lower in the hierarchy implicitly inherit the *Roles* properties of their parents, but only to a point. Refer to the sidebar below for a more detailed explanation.



Inside the ASP.NET Team

At first glance, it might appear that as a custom site map creates a tree of *SiteMapNodes*, it must explicitly flow the *Roles* properties of parents down to their children. However, that's not the case. Here's an explanation from the team:

*"In terms of role inheritance-there is not direct inheritance. Instead, when you iterate through a set of nodes, if you encounter a node that the current user is not authorized to access, you should stop. As an example, suppose some piece of code is recursing down through the nodes, and when it reaches a certain node, *IsAccessibleToUser* returns false. The code should stop iterating at that point and go no lower. However, if you were to call *FindSiteMapNode* to get one of the child nodes directly, that would work."*

In other words, a provider need not copy a parent node's *Roles* property to child nodes. Nor is *FindSiteMapNode* or *IsAccessibleToUser* obligated to walk up the *SiteMapNode* tree to determine whether a specified node is accessible. Node accessibility is primarily a mechanism allowing controls such as *TreeView*s and *Menus*, which iterate through trees of *SiteMapNodes* from top to bottom in order to render site maps into HTML, to stop iterating when they encounter a node that the current user lacks permission to view.

Content Localization

The combination of *SiteMapProvider*'s *EnableLocalization* and *ResourceKey* properties and *SiteMapNode*'s *ResourceKey* property enables site map providers to support content localization. If localization is enabled (as indicated by the provider's *EnableLocalization* property), a provider may use the *ResourceKey* properties to load text for *SiteMapNode*'s *Title* and *Description* properties, and even for custom properties, from resources (for example, compiled RESX files).

Resource keys can be implicit or explicit. Here's an example of an *XmlSiteMapProvider*-compatible site map that uses implicit resource keys to load node titles and descriptions:

```
<siteMap enableLocalization="true">
  <siteMapNode description="Home" url="~/Default.aspx">
    <siteMapNode title="Autos" description="Autos"
      url="~/Autos.aspx" resourceKey="Autos" />
    <siteMapNode title="Games" description="Games"
      url="~/Games.aspx" resourceKey="Games" />
    <siteMapNode title="Health" description="Health"
```

```
    url="~/Health.aspx" resourceKey="Health" />
  <siteMapNode title="News" description="News"
    url="~/News.aspx" resourceKey="News" />
</siteMapNode>
</siteMap>
```

In this example, the "Autos" site map node's *Title* and *Description* properties are taken from resources named *Autos.Title* and *Autos.Description* (and default to "Autos" and "Autos" if those resources don't exist or localization isn't enabled). *SiteMapNode* formulates the resource names by combining *resourceKey* values and property names; it also handles the chore of loading the resources.

The root name of the file containing the resources is specified using the provider's *ResourceKey* property. For example, suppose *ResourceKey*= "SiteMapResources", and that localization resources are defined in RESX files deployed in the application's *App_GlobalResources* directory for automatic compilation. *SiteMapNode* will therefore extract resources from *SiteMapResources.culture.resx*, where *culture* is the culture expressed in *Thread.CurrentThread.CurrentUICulture*. Resources for the *fr* culture would come from *SiteMapResources.fr.resx*, resources for *en-us* would come from *SiteMapResources.en-us.resx*, and so on. Requests lacking a culture specifier would default to *SiteMapResources.resx*.

A site map provider that supports localization via implicit resource keys should do the following:

- Initialize its *ResourceKey* property with a root resource file name. This value could come from a custom configuration attribute, or it could be based on something else entirely. ASP.NET's *XmlSiteMapProvider*, for example, sets the provider's *ResourceKey* property equal to the site map file name specified with the *siteMapFile* configuration attribute (which defaults to *Web.sitemap*).
- Pass resource key values specified for site map nodes for example, the *resourceKey* attribute values in *<siteMapNode>* elements to *SiteMapNode*'s constructor as the *implicitResourceKey* parameter.
- Set its own *EnableLocalization* property to true.


Although *SiteMapProvider* implements the *EnableLocalization* property, neither *SiteMapProvider* nor *StaticSiteMapProvider* initializes that property from a configuration attribute. If you want to support *enableLocalization* in your provider's configuration, you should do as *XmlSiteMapProvider* does and initialize the provider's *EnableLocalization* property from the *enableLocalization* attribute of the *<siteMap>* element in the site map file. It's important to set the provider's *EnableLocalization* property to true if you wish to localize site map nodes, because the *SiteMapNode* class checks that property before deciding whether to load content from resources.

The StaticSiteMapProvider Class

Whereas *SiteMapProvider* defines the basic contract between ASP.NET and site map providers, *StaticSiteMapProvider* aids developers in implementing that contract. *StaticSiteMapProvider* is the base class for ASP.NET's *XmlSiteMapProvider*. It can also be used as the base class for custom site map providers. Provider classes that derive from

StaticSiteMapProvider require considerably less code than providers derived from *SiteMapProvider*.

The word "Static" in *StaticSiteMapProvider* refers to the fact that the site map data source is static. Nonetheless, while the data source may be static, the site map itself does not have to be. Developers can add, remove, and change site map nodes on the fly by responding to a site map provider's *SiteMapResolve* events.



Inside the ASP.NET Team

ASP.NET's *XmlSiteMapProvider* goes to the extra trouble of monitoring the site map file and reloading it if it changes. While not required of a custom site map provider, that behavior will certainly be appreciated by administrators who want to modify a site map without having to restart the application. The *System.IO.FileSystemWatcher* class provides an efficient means for monitoring files for changes. If you use it, don't forget to implement *IDisposable* in the derived class and close the *FileSystemWatcher* in the *Dispose* method. If site map data is stored in a Microsoft SQL Server database, consider using ASP.NET 2.0's *SqlCacheDependency* class to monitor the database for changes.

System.Web.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) {}
}
```

The following table describes *StaticSiteMapProvider*'s methods and provides helpful notes regarding their implementation:

Table 6. StaticSiteMapProvider methods and properties

Method	Description
--------	-------------

<i>BuildSiteMap</i>	Called to read the site map from the data source and return a reference to the root SiteMapNode. Since this method may be called more than once by ASP.NET, the method implementation should include an internal check that refrains from reloading the site map if it has already been loaded. This method is abstract (MustOverride in Visual Basic) and must be overridden in a derived class.
<i>Clear</i>	Clears the site map by removing all SiteMapNodes.
<i>FindSiteMapNode</i>	Returns a SiteMapNode representing the page at the specified URL. Returns null (Nothing in Visual Basic) if the specified node isn't found.
<i>AddNode</i>	Adds a SiteMapNode to the site map as a child of the specified SiteMapNode, or as the root node if the specified SiteMapNode is null (Nothing in Visual Basic). The default implementation in StaticSiteMapProvider performs several important checks on the node before adding it to the site map, including a check for duplicate Url or Key properties.
<i>RemoveNode</i>	Removes the specified SiteMapNode from the site map.
<i>FindSiteMapNodeFromKey</i>	Retrieves a SiteMapNode keyed by the specified key that is, a SiteMapNode whose Key property matches the input key. Node lookups are normally performed based on URLs, but this method is provided so nodes that lack URLs can be retrieved from the site map. Returns null (Nothing in Visual Basic) if the specified node isn't found.
<i>GetChildNodes</i>	Returns a SiteMapNodeCollection representing the specified SiteMapNode's children. Returns an empty SiteMapNodeCollection if the node has no children.
<i>GetParentNode</i>	Returns a SiteMapNode representing the specified SiteMapNode's parent. Returns null (Nothing in Visual Basic) if the node has no parent.

StaticSiteMapProvider provides default implementations of most of *SiteMapProvider's* abstract methods (the notable exception being *GetRootNodeCore*), and internally it uses a set of *Hashtables* to make lookups performed by *FindSiteMapNode*, *GetParentNode*, and *GetChildNodes* fast and efficient. It also defines an abstract method of its own, *BuildSiteMap*, which signals the provider to read the data source and build the site map. *BuildSiteMap* can be (and in practice, is) called many times throughout the provider's lifetime, so it's crucial to build in an internal check to make sure the site map is loaded just once. (Subsequent calls to *BuildSiteMap* can simply return a reference to the

existing root node.) In addition, *BuildSiteMap* should generally not call other site map provider methods or properties, because many of the default implementations of those methods and properties call *BuildSiteMap*. For example, the simple act of reading *RootNode* in *BuildSiteMap* causes a recursive condition that terminates in a stack overflow.

SqlSiteMapProvider

SqlSiteMapProvider is a *StaticSiteMapProvider*-derivative that demonstrates the key ingredients that go into a custom site map provider. Unlike *XmlSiteMapProvider*, which reads site map data from an XML file, *SqlSiteMapProvider* reads site maps from a SQL Server database. It doesn't support localization, but it does support other significant features found in *XmlSiteMapProvider*, including security trimming and site maps of unlimited depth. *SqlSiteMapProvider*'s source code appears in Figure 11.

Figure 11. SqlSiteMapProvider

```
using System;
using System.Web;
using System.Data.SqlClient;
using System.Collections.Specialized;
using System.Configuration;
using System.Web.Configuration;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using System.Configuration.Provider;
using System.Security.Permissions;
using System.Data.Common;

[SqlClientPermission (SecurityAction.Demand, Unrestricted=true)]
public class SqlSiteMapProvider : StaticSiteMapProvider
{
    private const string _errmsg1 = "Missing node ID";
    private const string _errmsg2 = "Duplicate node ID";
    private const string _errmsg3 = "Missing parent ID";
    private const string _errmsg4 = "Invalid parent ID";
    private const string _errmsg5 =
        "Empty or missing connectionStringName";
    private const string _errmsg6 = "Missing connection string";
    private const string _errmsg7 = "Empty connection string";

    private string _connect;
    private int _indexID, _indexTitle, _indexUrl,
        _indexDesc, _indexRoles, _indexParent;
    private Dictionary<int, SiteMapNode> _nodes =
        new Dictionary<int, SiteMapNode>(16);
    private SiteMapNode _root;
```

```

public override void Initialize (string name,
    NameValueCollection config)
{
    // Verify that config isn't null
    if (config == null)
        throw new ArgumentNullException("config");

    // Assign the provider a default name if it doesn't have one
    if (String.IsNullOrEmpty(name))
        name = "SqlSiteMapProvider";

    // Add a default "description" attribute to config if the
    // attribute doesn't exist or is empty
    if (string.IsNullOrEmpty(config["description"]))
    {
        config.Remove("description");
        config.Add("description", "SQL site map provider");
    }

    // Call the base class's Initialize method
    base.Initialize(name, config);

    // Initialize _connect
    string connect = config["connectionStringName"];

    if (String.IsNullOrEmpty (connect))
        throw new ProviderException (_errmsg5);

    config.Remove ("connectionStringName");

    if (WebConfigurationManager.ConnectionStrings[connect] == null)
        throw new ProviderException (_errmsg6);

    _connect = WebConfigurationManager.ConnectionStrings
        [connect].ConnectionString;

    if (String.IsNullOrEmpty (_connect))
        throw new ProviderException (_errmsg7);

    // In beta 2, SiteMapProvider processes the
    // securityTrimmingEnabled attribute but fails to remove it.
    // Remove it now so we can check for unrecognized
    // configuration attributes.

    if (config["securityTrimmingEnabled"] != null)

```

```

        config.Remove("securityTrimmingEnabled");

        // Throw an exception if unrecognized attributes remain
        if (config.Count > 0)
        {
            string attr = config.GetKey(0);
            if (!String.IsNullOrEmpty(attr))
                throw new ProviderException
                    ("Unrecognized attribute: " + attr);
        }
    }

    public override SiteMapNode BuildSiteMap()
    {
        lock (this)
        {
            // Return immediately if this method has been called before
            if (_root != null)
                return _root;

            // Query the database for site map nodes
            SqlConnection connection = new SqlConnection(_connect);

            try
            {
                connection.Open();
                SqlCommand command =
                    new SqlCommand("proc_GetSiteMap", connection);
                command.CommandType = CommandType.StoredProcedure;
                SqlDataReader reader = command.ExecuteReader();
                _indexID = reader.GetOrdinal("ID");
                _indexUrl = reader.GetOrdinal("Url");
                _indexTitle = reader.GetOrdinal("Title");
                _indexDesc = reader.GetOrdinal("Description");
                _indexRoles = reader.GetOrdinal("Roles");
                _indexParent = reader.GetOrdinal("Parent");

                if (reader.Read())
                {
                    // Create the root SiteMapNode and add it to
                    // the site map
                    _root = CreateSiteMapNodeFromDataReader(reader);
                    AddNode(_root, null);

                    // Build a tree of SiteMapNodes underneath

```

```

        // the root node
        while (reader.Read())
        {
            // Create another site map node and
            // add it to the site map
            SiteMapNode node =
                CreateSiteMapNodeFromDataReader(reader);
            AddNode(node,
                GetParentNodeFromDataReader (reader));
        }
    }
    finally
    {
        connection.Close();
    }

    // Return the root SiteMapNode
    return _root;
}
}

protected override SiteMapNode GetRootNodeCore ()
{
    BuildSiteMap ();
    return _root;
}

// Helper methods
private SiteMapNode
    CreateSiteMapNodeFromDataReader (DbDataReader reader)
{
    // Make sure the node ID is present
    if (reader.IsDBNull (_indexID))
        throw new ProviderException (_errmsg1);

    // Get the node ID from the DataReader
    int id = reader.GetInt32 (_indexID);

    // Make sure the node ID is unique
    if (_nodes.ContainsKey(id))
        throw new ProviderException(_errmsg2);

    // Get title, URL, description, and roles from the DataReader
    string title = reader.IsDBNull (_indexTitle) ?

```

```

        null : reader.GetString (_indexTitle).Trim ();
string url = reader.IsDBNull (_indexUrl) ?
        null : reader.GetString (_indexUrl).Trim ();
string description = reader.IsDBNull (_indexDesc) ?
        null : reader.GetString (_indexDesc).Trim ();
string roles = reader.IsDBNull(_indexRoles) ?
        null : reader.GetString(_indexRoles).Trim();

// If roles were specified, turn the list into a string array
string[] rolelist = null;
if (!String.IsNullOrEmpty(roles))
    rolelist = roles.Split(new char[] { ',', ';' }, 512);

// Create a SiteMapNode
SiteMapNode node = new SiteMapNode(this, id.ToString(), url,
    title, description, rolelist, null, null, null);

// Record the node in the _nodes dictionary
_nodes.Add(id, node);

// Return the node
return node;
}

private SiteMapNode
    GetParentNodeFromDataReader(DbDataReader reader)
{
    // Make sure the parent ID is present
    if (reader.IsDBNull (_indexParent))
        throw new ProviderException (_errmsg3);

    // Get the parent ID from the DataReader
    int pid = reader.GetInt32(_indexParent);

    // Make sure the parent ID is valid
    if (!_nodes.ContainsKey(pid))
        throw new ProviderException(_errmsg4);

    // Return the parent SiteMapNode
    return _nodes[pid];
}
}

```

The heart of *SqlSiteMapProvider* is its *BuildSiteMap* method, which queries the database for site map node data and constructs *SiteMapNodes* from the query results. The query is performed by calling the stored procedure named *proc_GetSiteMap*, which is defined as follows:

```
CREATE PROCEDURE proc_GetSiteMap AS SELECT [ID], [Title],
[Description], [Url], [Roles], [Parent] FROM [SiteMap] ORDER BY [ID]
```

The helper method *CreateSiteMapNodeFromDataReader* does the node construction, checking for errors such as missing or non-unique node IDs as well. *CreateSiteMapNodeFromDataReader* also records each *SiteMapNode* that it creates in an internal dictionary used by the other helper method, *GetParentNodeFromDataReader*, to retrieve a reference to a node's parent before adding the node to the site map with *AddNode*.

SqlSiteMapProvider inherits support for the *securityTrimmingEnabled* configuration attribute from *SiteMapProvider*. It also supports one configuration attribute of its own: *connectionStringName*. The provider's *Initialize* method reads *connectionStringName* and *BuildSiteMap* uses it to read a database connection string from the <connectionStrings> configuration section. This is the connection string used to connect to the SQL Server database containing the site map. The Web.config file in Figure 12 makes *SqlSiteMapProvider* the default provider and provides it with a connection string.

Figure 12. Web.config file making *SqlSiteMapProvider* the default site map provider and enabling security trimming

```
<configuration>
  <connectionStrings>
    <add name="SiteMapConnectionString" connectionString="..." />
  </connectionStrings>
  <system.web>
    <siteMap enabled="true" defaultProvider="AspNetSqlSiteMapProvider">
      <providers>
        <add name="AspNetSqlSiteMapProvider"
          type="SqlSiteMapProvider, CustomProviders"
          description="SQL Server site map provider"
          securityTrimmingEnabled="true"
          connectionStringName="SiteMapConnectionString"
        />
      </providers>
    </siteMap>
  </system.web>
</configuration>
```

The SQL script in Figure 13 creates a *SqlSiteMapProvider*-compatible *SiteMap* table. The Title, Description, Url, and Roles columns correspond to the *SiteMapNode* properties of the same names. The ID and Parent columns serve to uniquely identify nodes in the site

map and form parent-child relationships between them. Every node must have a unique ID in the ID column, and every node except the root node must contain an ID in the Parent column identifying the node's parent. The one constraint to be aware of is that a child node's ID must be greater than its parent's ID. In other words, a node with an ID of 100 can be the child of a node with an ID of 99, but it can't be the child of a node with an ID of 101. That's a consequence of the manner in which *SqlSiteMapProvider* builds the site map in memory as it reads nodes from the database.

Figure 13. SQL script for creating a *SqlSiteMapProvider*-compatible SiteMap table

```
CREATE TABLE [dbo].[SiteMap] (  
    [ID]          [int] NOT NULL,  
    [Title]       [varchar] (32),  
    [Description] [varchar] (512),  
    [Url]         [varchar] (512),  
    [Roles]       [varchar] (512),  
    [Parent]      [int]  
) ON [PRIMARY]  
GO  
  
ALTER TABLE [dbo].[SiteMap] ADD  
    CONSTRAINT [PK_SiteMap] PRIMARY KEY CLUSTERED  
    (  
        [ID]  
    ) ON [PRIMARY]  
GO
```

Figure 14 shows a view in SQL Server Enterprise Manager of a SiteMap table. Observe that each node has a unique ID and a unique URL. (The latter is a requirement of site maps managed by site map providers that derive from *StaticSiteMapProvider*.) All nodes but the root node also have a parent ID that refers to another node in the table, and the parent's ID is always less than the child's ID. Finally, because the "Entertainment" node has a Roles value of "Members,Administrators," a navigational control such as a *TreeView* or *Menu* will only render that node and its children if the user viewing the page has been authenticated and is a member of the Members or Administrators group. That's assuming, of course, that security trimming is enabled. If security trimming is not enabled, all nodes are visible to all users. *SqlSiteMapProvider* contains no special code to make security trimming work; that logic is inherited from *SiteMapProvider* and *StaticSiteMapProvider*.

Figure 14. Sample SiteMap table

SQL Server Enterprise Manager - [Data in Table 'SiteMap' in 'Times' on '(local)']

File Window Help

SQL

ID	Title	Description	Url	Roles	Parent
1	Home	<NULL >	~/Default.aspx	<NULL >	<NULL >
10	News	<NULL >	<NULL >	*	1
11	Local	News from greater Seattle	~/Summary.aspx?CategoryID=0	<NULL >	10
12	National	News from the United States	~/Summary.aspx?CategoryID=1	<NULL >	10
13	World	News from around the world	~/Summary.aspx?CategoryID=2	<NULL >	10
20	Sports	<NULL >	<NULL >	*	1
21	Baseball	What's happening in baseball	~/Summary.aspx?CategoryID=3	<NULL >	20
22	Basketball	What's happening in basketball	~/Summary.aspx?CategoryID=4	<NULL >	20
23	Football	What's happening in football	~/Summary.aspx?CategoryID=5	<NULL >	20
30	Members Only	<NULL >	<NULL >	Members,Administrators	1
31	Travel	Travel news	~/Summary.aspx?CategoryID=6	<NULL >	30
32	Entertainment	Entertainment news	~/Summary.aspx?CategoryID=7	<NULL >	30
33	Technology	Technology news	~/Summary.aspx?CategoryID=8	<NULL >	30
*					

Session State Providers

Session state providers provide the interface between ASP.NET session state and session state data sources. The two most common reasons for writing a custom session state provider are:

- You wish to store session state in a data source that is not supported by the session state providers included with the .NET Framework, such as an Oracle database.
- You wish to store session state in a SQL Server database whose schema differs from that of the database used by *System.Web.SessionState.SqlSessionStateStore*.

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 *SessionStateDataStores* 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 time-out (in minutes)

Items and *StaticObjects* can be serialized and deserialized easily enough by calling their *Serialize* and *Deserialize* methods. The *Timeout* property is a simple *System.Int32* and is therefore also easily serialized and deserialized. Thus, if *sssd* is a reference to an instance of *SessionStateStoreData*, the core logic to write the contents of a session to a session state data source is often no more complicated than this:

```
stream1 = new MemoryStream ();
stream2 = new MemoryStream ();
writer1 = new BinaryWriter (stream1);
writer2 = new BinaryWriter (stream2);

// Serialize Items and StaticObjects
((SessionStateItemCollection) sssd.Items).Serialize (writer1);
sssd.StaticObjects.Serialize (writer2);

// Convert serialized items into byte arrays
byte[] items = stream1.ToArray ();
byte[] statics = stream2.ToArray ();
int timeout = sssd.Timeout;

// TODO: Write items, statics, and timeout to the data source
```

One of the challenges to writing a session state provider is implementing a locking mechanism that prevents a given session from being accessed by two or more concurrent requests. That mechanism ensures the consistency of session state data by preventing one request from reading a session at the same time that another request is writing it. The locking mechanism must work even if the session state data source is a remote resource shared by several Web servers. Locking behavior is discussed in "Synchronizing Concurrent Accesses to a Session."

Another consideration to take into account when designing a session state provider is whether to support expiration callbacks notifying *SessionStateModule* when sessions time out. Expiration callbacks are discussed in "Expiration Callbacks and Session_End Events."

The SessionStateStoreProviderBase Class

Developers writing custom session state providers begin by deriving from *System.Web.SessionState.SessionStateStoreProviderBase*, which derives from *ProviderBase* and adds abstract methods defining 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);
}

```

The following table describes *SessionStateStoreProviderBase*'s methods and provides helpful notes regarding their implementation:

Table 7. SessionStateStoreProviderBase methods and properties

Method	Description
<i>CreateNewStoreData</i>	Called to create a <i>SessionStateStoreData</i> for a new session.
<i>CreateUninitializedItem</i>	Called to create a new, uninitialized session in the data source. Called by <i>SessionStateModule</i> when session state is cookieless to prevent the session from being unrecognized following a redirect.
<i>Dispose</i>	Called when the provider is disposed of to afford it the opportunity to perform any last-chance cleanup.
<i>EndRequest</i>	Called in response to <i>EndRequest</i> events to afford the provider the opportunity to perform any per-request cleanup.
<i>GetItem</i>	<p>If session state is read-only (that is, if the requested page implements <i>IReadOnlySessionState</i>), called to load the <i>SessionStateStoreData</i> corresponding to the specified session ID and apply a read lock so other requests can read the <i>SessionStateDataStore</i> but not modify it until the lock is released.</p> <p>If the specified session doesn't exist in the data source, this method returns null (Nothing in Visual Basic) and sets the out parameter named <i>locked</i> to false. <i>SessionStateModule</i> then calls <i>CreateNewStoreData</i> to create a new</p>

	<p>SessionStateStoreData to serve this session.</p> <p>If the specified session is currently locked, this method returns null (Nothing in Visual Basic) and sets the out parameter named locked to true, causing SessionStateModule to retry GetItem at half-second intervals until the lock comes free or times out. In addition to setting locked to true, this method also returns the lock age and lock ID using the lockAge and lockId parameters.</p>
<i>GetItemExclusive</i>	<p>If session state is read-write (that is, if the requested page implements IRequiresSessionState), called to load the SessionStateStoreData corresponding to the specified session ID and apply a write lock so other requests can neither read nor write the SessionStateStoreData until the lock is released.</p> <p>If the specified session doesn't exist in the data source, this method returns null (Nothing in Visual Basic) and sets the out parameter named locked to false. SessionStateModule then calls CreateNewStoreData to create a new SessionStateStoreData to serve this session.</p> <p>If the specified session is currently locked, this method returns null (Nothing in Visual Basic) and sets the out parameter named locked to true, causing SessionStateModule to retry GetItemExclusive at half-second intervals until the lock comes free or times out. In addition to setting locked to true, this method also returns the lock age and lock ID using the lockAge and lockId parameters.</p>
<i>InitializeRequest</i>	Called in response to AcquireRequestState events to afford the provider the opportunity to perform any per-request initialization.
<i>ReleaseItemExclusive</i>	Called to unlock the specified session if a request times out waiting for the lock to come free.
<i>RemoveItem</i>	Called to remove the specified session from the data source.
<i>ResetItemTimeout</i>	Called to reset the expiration time of the specified session.
<i>SetAndReleaseItemExclusive</i>	Called to write modified session state to the data source. The newItem parameter indicates whether the supplied SessionStateStoreData corresponds to an existing session in the data source or a new one. If newItem is true,

	SetAndReleaseItemExclusive adds a new session to the data source. Otherwise, it updates an existing one.
<i>SetItemExpireCallback</i>	Called to supply the provider with a callback method for notifying SessionStateModule that a session has expired. If the provider supports session expiration, it should return true from this method and notify ASP.NET when sessions expire by calling the supplied callback method. If the provider does not support session expiration, it should return false from this method.

Your job in implementing a custom session state provider in a derived class is to override and provide implementations of *SessionStateStoreProviderBase*'s abstract members, and optionally to override key virtuals such as *Initialize*.

Synchronizing Concurrent Accesses to a Session

ASP.NET applications are inherently multithreaded. Because requests that arrive in parallel are processed on concurrent threads drawn from a thread pool, it's possible that two or more requests targeting the same session will execute at the same time. (The classic example is when a page contains two frames, each targeting a different ASPX in the same application, causing the browser to submit overlapping requests for the two pages.) To avoid data collisions and erratic behavior, the provider "locks" the session when it begins processing the first request, causing other requests targeting the same session to wait for the lock to come free.

Because there's no harm in allowing concurrent requests to perform overlapping reads, the lock is typically implemented as a reader/writer lock—that is, one that allows any number of threads to read a session but that prevents overlapping reads and writes as well as overlapping writes.

Which brings up two very important questions:

1. How does a session state provider know when to apply a lock?
2. How does the provider know whether to treat a request as a reader or a writer?

If the requested page implements the *IRequiresSessionState* interface (by default, all pages implement *IRequiresSessionState*), ASP.NET assumes that the page requires read/write access to session state. In response to the *AcquireRequestState* event fired from the pipeline, *SessionStateModule* calls the session state provider's *GetItemExclusive* method. If the targeted session isn't already locked, *GetItemExclusive* applies a write lock and returns the requested data along with a lock ID (a value that uniquely identifies the lock). However, if the session is locked when *GetItemExclusive* is called, indicating that another request targeting the same session is currently executing, *GetItemExclusive* returns null and uses the *out* parameters passed to it to return the lock ID and the lock's age (how long, in seconds, the session has been locked).

If the requested page implements the *IReadOnlySessionState* interface instead, ASP.NET assumes that the page reads but does not write session state. (The most common way to implement *IReadOnlySessionState* is to include an *EnableSessionState="ReadOnly"*

attribute in the page's @ Page directive.) Rather than call the provider's *GetItemExclusive* method to retrieve the requestor's session state, ASP.NET calls *GetItem* instead. If the targeted session isn't locked by a writer when *GetItem* is called, *GetItem* applies a read lock and returns the requested data. (The read lock ensures that if a read/write request arrives while the current request is executing, it waits for the lock to come free. This prevents read/write requests from overlapping with read requests that are already executing.) Otherwise, *GetItem* returns null and uses the *out* parameters passed to it to return the lock ID and the lock's age—just like *GetItemExclusive*.

The third possibility that the page implements neither *IRequiresSessionState* nor *IReadOnlySessionState* tells ASP.NET that the page doesn't use session state, in which case *SessionStateModule* calls neither *GetItem* nor *GetItemExclusive*. The most common way to indicate that a page should implement neither interface is to include an *EnableSessionState="false"* attribute in the page's @ Page directive.

If *SessionStateModule* encounters a locked session when it calls *GetItem* or *GetItemExclusive* (that is, if either method returns null), it rerequests the data at half-second intervals until the lock is released or the request times out. If a time-out occurs, *SessionStateModule* calls the provider's *ReleaseItemExclusive* method to release the lock and allow the session to be accessed.

SessionStateModule identifies locks using the lock IDs returned by *GetItem* and *GetItemExclusive*. When *SessionStateModule* calls *SetAndReleaseItemExclusive* or *ReleaseItemExclusive*, it passes in a lock ID. Due to the possibility that a call to *ReleaseItemExclusive* on one thread could free a lock just before another thread calls *SetAndReleaseItemExclusive*, the *SetAndReleaseItemExclusive* method of a provider that supports multiple locks IDs per session should only write the session to the data source if the lock ID input to it matches the lock ID in the data source.

Expiration Callbacks and Session_End Events

After loading a session state provider, *SessionStateModule* calls the provider's *SetItemExpireCallback* method, passing in a *SessionStateItemExpireCallback* delegate that enables the provider to notify *SessionStateModule* when a session times out. If the provider supports expiration callbacks, it should save the delegate and return true from *SetItemExpireCallback*. Then, whenever a session times out, the provider should notify *SessionStateModule* that a session has expired by calling the callback method encapsulated in the delegate. This enables *SessionStateModule* to fire *Session_End* events when sessions expire.

Session state providers aren't required to support expiration callbacks. A provider that doesn't support them should return false from *SetItemExpireCallback*. In that case, *SessionStateModule* will not be notified when a session expires and will not fire *Session_End* events.



Inside the ASP.NET Team

How do the built-in session state providers handle expiration callbacks? The in-

process session state provider, *InProcSessionStateStore*, stores session content in the ASP.NET application cache and takes advantage of the cache's sliding-expiration feature to expire sessions when they go for a specified period of time without being accessed. When the provider is notified via a cache removal callback that a session expired from the cache, it notifies *SessionStateModule*, and *SessionStateModule* fires a *Session_End* event.

The other two built-in providers *OutOfProcSessionStateStore* and *SqlSessionStateStore* don't support expiration callbacks. Both return false from *SetItemExpireCallback*. *OutOfProcSessionStateStore* uses the application cache to store sessions, but since session data is stored in a remote process (the "state server" process), the provider doesn't attempt to notify *SessionStateModule* when a session expires. *SqlSessionStateStore* relies on a SQL Server agent to "scavenge" the session state database and clean up expired sessions. Having the agent notify the provider about expired sessions so the provider could, in turn, notify *SessionStateModule* would be a tricky endeavor indeed-especially in a Web farm.

Cookieless Sessions

ASP.NET supports two different types of sessions: cookieless and cookieless. The term actually refers to the mechanism used to round-trip session IDs between clients and Web servers and does not imply any difference in the sessions themselves. Cookieless sessions round-trip session IDs in HTTP cookies, while 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 following the redirect. (Otherwise, *SessionStateModule* would think that the ID extracted from the URL following 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 cookieless rather than cookieless, the provider's *CreateUninitializedItem* method is never called. When testing a custom session state provider, be certain to test it in both cookieless and cookieless mode.

A *CreateUninitializedItem* implementation can use any technique it desires to ensure that the session ID passed to it is recognized as a valid ID following a redirect. ASP.NET's *InProcSessionStateStore* provider, for example, inserts an empty *SessionStateStoreData* (that is, a *SessionStateStoreData* object with null *Items* and *StaticObjects* properties) into the application cache accompanied by a flag marking it as an uninitialized session. *SqlSessionStateStore* acts similarly, adding a row representing the session to the session state database and flagging it as an uninitialized session.

When a session state provider's *GetItem* or *GetItemExclusive* method is called, it returns a *SessionStateActions* value through the *out* parameter named *actions*. The value returned depends on the state of the session identified by the supplied ID. If the data corresponding to the session doesn't exist in the data source or if it exists and is already initialized (that is, if the session was *not* created by *CreateUninitializedItem*), *GetItem* and *GetItemExclusive* should return *SessionStateActions.None* through the *actions* parameter. However, if the session data exists in the data source but is not initialized (indicating the session was created by *CreateUninitializedItem*), *GetItem* and *GetItemExclusive* should return *SessionStateActions.InitializeItem*. *SessionStateModule* responds to a *SessionStateActions.InitializeItem* flag by firing a *Session_Start* event signifying the start of a new session. It also raises a *Session_Start* event if *GetItem* or *GetItemExclusive* returns *SessionStateActions.None* following the creation of a new session.

TextFileSessionStateProvider

Figure 15 contains the source code for a *SessionStateStoreProviderBase*-derivative named *TextFileSessionStateProvider* that demonstrates the basics of custom session state providers. *TextFileSessionStateProvider* stores session state in text files named *SessionID_Session.txt* in the application's *~/App_Data/Session_Data* directory, where *SessionID* is the ID of the corresponding session. Each file contains the state for a specific session and consists of either two or four lines of text:

- A "0" or "1" indicating whether the session is initialized, where "1" means the session is initialized, and "0" means it is not
- If line 1 contains a "1", a base-64 string containing the session's serialized non-static objects
- If line 1 contains a "1", a base-64 string containing the session's serialized static objects
- A numeric string specifying the session's time-out in minutes

Thus, a file containing an initialized session contains four lines of text, and a file containing an uninitialized session—that is, a session created by *CreateUninitializedItem* in support of cookieless session state—contains two. You must create the *~/App_Data/Session_Data* directory before using the provider; the provider doesn't attempt to create the directory if it doesn't exist. In addition, the provider must have read/write access to the *~/App_Data/Session_Data* directory.

Figure 15. TextFileSessionStateProvider

```
using System;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.SessionState;
using System.Collections.Specialized;
using System.Collections.Generic;
using System.Configuration.Provider;
using System.Security.Permissions;
```

```

using System.Web.Hosting;
using System.IO;

public class TextFileSessionStateProvider :
    SessionStateStoreProviderBase
{
    private Dictionary<string, FileStream> _sessions =
        new Dictionary<string, FileStream> ();

    public override void Initialize(string name,
        NameValueCollection config)
    {
        // Verify that config isn't null
        if (config == null)
            throw new ArgumentNullException("config");

        // Assign the provider a default name if it doesn't have one
        if (String.IsNullOrEmpty(name))
            name = "TextFileSessionStateProvider";

        // Add a default "description" attribute to config if the
        // attribute doesn't exist or is empty
        if (string.IsNullOrEmpty(config["description"]))
        {
            config.Remove("description");
            config.Add("description",
                "Text file session state provider");
        }

        // Call the base class's Initialize method
        base.Initialize(name, config);

        // Throw an exception if unrecognized attributes remain
        if (config.Count > 0)
        {
            string attr = config.GetKey(0);
            if (!String.IsNullOrEmpty(attr))
                throw new ProviderException
                    ("Unrecognized attribute: " + attr);
        }

        // Make sure we can read and write files in the
        // ~/App_Data/Session_Data directory
        FileIOPermission permission =
            new FileIOPermission(FileIOPermissionAccess.AllAccess,

```

```

        HttpContext.Current.Server.MapPath(
            "~/App_Data/Session_Data"));
    permission.Demand();
}

public override SessionStateStoreData
    CreateNewStoreData(HttpContext context, int timeout)
{
    return new SessionStateStoreData(
        new SessionStateItemCollection(),
        SessionStateUtility.GetSessionStaticObjects(context),
        timeout
    );
}

public override void CreateUninitializedItem(HttpContext context,
    string id, int timeout)
{
    // Create a file containing an uninitialized flag
    // and a time-out
    StreamWriter writer = null;

    try
    {
        writer = new StreamWriter(context.Server.MapPath(
            GetSessionFileName(id)));
        writer.WriteLine("0");
        writer.WriteLine(timeout.ToString());
    }
    finally
    {
        if (writer != null)
            writer.Close();
    }
}

public override SessionStateStoreData GetItem(HttpContext context,
    string id, out bool locked, out TimeSpan lockAge,
    out object lockId, out SessionStateActions actions)
{
    return GetSession(context, id, out locked, out lockAge,
        out lockId, out actions, false);
}

public override SessionStateStoreData

```

```

    GetItemExclusive(HttpContext context, string id,
        out bool locked, out TimeSpan lockAge, out object lockId,
        out SessionStateActions actions)
    {
        return GetSession(context, id, out locked, out lockAge,
            out lockId, out actions, true);
    }

public override void
    SetAndReleaseItemExclusive(HttpContext context, string id,
        SessionStateStoreData item, object lockId, bool newItem)
    {
        // Serialize the session
        byte[] items, statics;
        SerializeSession(item, out items, out statics);
        string serializedItems = Convert.ToBase64String(items);
        string serializedStatics = Convert.ToBase64String(statics);

        // Get a FileStream representing the session state file
        FileStream stream = null;

        try
        {
            if (newItem)
                stream = File.Create(context.Server.MapPath
                    (GetSessionFileName (id)));
            else
            {
                stream = _sessions[id];
                stream.SetLength(0);
                stream.Seek(0, SeekOrigin.Begin);
            }

            // Write session state to the file
            StreamWriter writer = null;

            try
            {
                writer = new StreamWriter(stream);
                writer.WriteLine("1"); // Initialized flag
                writer.WriteLine(serializedItems);
                writer.WriteLine(serializedStatics);
                writer.WriteLine(item.Timeout.ToString());
            }
            finally

```

```

        {
            if (writer != null)
                writer.Close();
        }
    }
    finally
    {
        if (newItem && stream != null)
            stream.Close();
    }

    // Unlock the session
    ReleaseItemExclusive(context, id, lockId);
}

public override void ReleaseItemExclusive(HttpContext context,
    string id, object lockId)
{
    // Release the specified session by closing the corresponding
    // FileStream and deleting the lock file
    FileStream stream;

    if (_sessions.TryGetValue (id, out stream))
    {
        _sessions.Remove(id);
        ReleaseLock(context, (string) lockId);
        stream.Close();
    }
}

public override void ResetItemTimeout(HttpContext context,
    string id)
{
    // Update the time stamp on the session state file
    string path = context.Server.MapPath(GetSessionFileName(id));
    File.SetCreationTime(path, DateTime.Now);
}

public override void RemoveItem(HttpContext context, string id,
    object lockId, SessionStateStoreData item)
{
    // Make sure the session is unlocked
    ReleaseItemExclusive(context, id, lockId);

    // Delete the session state file

```

```

        File.Delete(context.Server.MapPath(GetSessionFileName(id)));
    }

    public override bool SetItemExpireCallback
        (SessionStateItemExpireCallback expireCallback)
    {
        // This provider doesn't support expiration callbacks,
        // so simply return false here
        return false;
    }

    public override void InitializeRequest(HttpContext context)
    {
    }

    public override void EndRequest(HttpContext context)
    {
    }

    public override void Dispose()
    {
        // Make sure no session state files are left open
        foreach (KeyValuePair<string, FileStream> pair in _sessions)
        {
            pair.Value.Close();
            _sessions.Remove(pair.Key);
        }

        // Delete session files and lock files
        File.Delete (HostingEnvironment.MapPath
            ("~/App_Data/Session_Data/*_Session.txt"));
        File.Delete (HostingEnvironment.MapPath
            ("~/App_Data/Session_Data/*_Lock.txt"));    }

    // Helper methods
    private SessionStateStoreData GetSession(HttpContext context,
        string id, out bool locked, out TimeSpan lockAge,
        out object lockId, out SessionStateActions actions,
        bool exclusive)
    {
        // Assign default values to out parameters
        locked = false;
        lockId = null;
        lockAge = TimeSpan.Zero;
        actions = SessionStateActions.None;
    }

```

```

FileStream stream = null;

try
{
    // Attempt to open the session state file
    string path =
        context.Server.MapPath(GetSessionFileName(id));
    FileAccess access = exclusive ?
        FileAccess.ReadWrite : FileAccess.Read;
    FileShare share = exclusive ?
        FileShare.None : FileShare.Read;
    stream = File.Open(path, FileMode.Open, access, share);
}
catch (FileNotFoundException)
{
    // Not an error if file doesn't exist
    return null;
}
catch (IOException)
{
    // If we come here, the session is locked because
    // the file couldn't be opened
    locked = true;
    lockId = id;
    lockAge = GetLockAge(context, id);
    return null;
}

// Place a lock on the session
CreateLock(context, id);
locked = true;
lockId = id;

// Save the FileStream reference so it can be used later
_sessions.Add(id, stream);

// Find out whether the session is initialized
StreamReader reader = new StreamReader(stream);
string flag = reader.ReadLine ();
bool initialized = (flag == "1");

if (!initialized)
{
    // Return an empty SessionStateStoreData

```



```

        actions = SessionStateActions.InitializeItem;
        int timeout = Convert.ToInt32(reader.ReadLine ());

        return new SessionStateStoreData(
            new SessionStateItemCollection (),
            SessionStateUtility.GetSessionStaticObjects (context),
            timeout
        );
    }
    else
    {
        // Read Items, StaticObjects, and Timeout from the file
        // (NOTE: Don't close the StreamReader, because doing so
        // will close the file)
        byte[] items = Convert.FromBase64String(reader.ReadLine());
        byte[] statics =
            Convert.FromBase64String(reader.ReadLine());
        int timeout = Convert.ToInt32(reader.ReadLine());

        // Deserialize the session
        return DeserializeSession(items, statics, timeout);
    }
}

private void CreateLock(HttpContext context, string id)
{
    // Create a lock file so the lock's age can be determined
    File.Create(context.Server.MapPath
        (GetLockFileName(id))).Close();
}

private void ReleaseLock(HttpContext context, string id)
{
    // Delete the lock file
    string path = context.Server.MapPath(GetLockFileName(id));
    if (File.Exists (path))
        File.Delete(path);
}

private TimeSpan GetLockAge(HttpContext context, string id)
{
    try
    {
        return DateTime.Now
            File.GetCreationTime(context.Server.MapPath

```

```

        (GetLockFileName(id)));
    }
    catch (FileNotFoundException)
    {
        // This is important, because it's possible that
        // a lock is active but the lock file hasn't been
        // created yet if another thread owns the lock
        return TimeSpan.Zero;
    }
}

string GetSessionFileName(string id)
{
    return String.Format("~/App_Data/Session_Data/{0}_Session.txt",
        id);
}

string GetLockFileName(string id)
{
    return String.Format("~/App_Data/Session_Data/{0}_Lock.txt",
        id);
}

private void SerializeSession(SessionStateStoreData store,
    out byte[] items, out byte[] statics)
{
    MemoryStream stream1 = null, stream2 = null;
    BinaryWriter writer1 = null, writer2 = null;

    try
    {
        stream1 = new MemoryStream();
        stream2 = new MemoryStream();
        writer1 = new BinaryWriter(stream1);
        writer2 = new BinaryWriter(stream2);

        ((SessionStateItemCollection)
            store.Items).Serialize(writer1);
        store.StaticObjects.Serialize(writer2);

        items = stream1.ToArray();
        statics = stream2.ToArray();
    }
    finally

```

```

        {
            if (writer2 != null)
                writer2.Close();
            if (writer1 != null)
                writer1.Close();
            if (stream2 != null)
                stream2.Close();
            if (stream1 != null)
                stream1.Close();
        }
    }

private SessionStateStoreData DeserializeSession(byte[] items,
    byte[] statics, int timeout)
{
    MemoryStream stream1 = null, stream2 = null;
    BinaryReader reader1 = null, reader2 = null;

    try
    {
        stream1 = new MemoryStream(items);
        stream2 = new MemoryStream(statics);
        reader1 = new BinaryReader(stream1);
        reader2 = new BinaryReader(stream2);

        return new SessionStateStoreData(
            SessionStateItemCollection.Deserialize(reader1),
            HttpStaticObjectsCollection.Deserialize(reader2),
            timeout
        );
    }
    finally
    {
        if (reader2 != null)
            reader2.Close();
        if (reader1 != null)
            reader1.Close();
        if (stream2 != null)
            stream2.Close();
        if (stream1 != null)
            stream1.Close();
    }
}
}

```

TextFileSessionStateProvider's locking strategy is built around file-sharing modes. *GetItemExclusive* attempts to open the session state file with a *FileShare.None* flag, giving it exclusive access to the file. *GetItem*, however, attempts to open the session state file with a *FileShare.Read* flag, allowing other readers to access the file, but not writers. So that *GetItem* and *GetItemExclusive* can return the lock's age if the session state file is locked when they're called, a 0-byte "lock file" named *SessionID_Lock.txt* is created in the `~/App_Data/Session_Data` directory when a session state file is successfully opened. *GetItem* and *GetItemExclusive* compute a lock's age by subtracting the lock file's creation time from the current time. *ReleaseItemExclusive* and *SetAndReleaseItemExclusive* release a lock by closing the session state file and deleting the corresponding lock file.

TextFileSessionStateProvider takes a simple approach to lock IDs. When it creates a lock, it assigns the lock a lock ID that equals the session ID. That's sufficient because the nature of the locks managed by *TextFileSessionStateProvider* is such that a given session never has more than one lock applied to it. This behavior is consistent with that of *SqlSessionStateStore*, which also uses one lock ID per given session.

Figure 16 demonstrates how to make *TextFileSessionStateProvider* the default session state provider. It assumes that *TextFileSessionStateProvider* is implemented in an assembly named *CustomProviders*. Note the syntactical differences between session state providers and other provider types. The `<sessionState>` element uses a *customProvider* attribute rather than a *defaultProvider* attribute to designate the default provider, and the *customProvider* attribute is ignored unless a *mode="Custom"* attribute is included, too.

Figure 16. Web.config file making *TextFileSessionStateProvider* the default session state provider

```
<configuration>
  <system.web>
    <sessionState mode="Custom"
      customProvider="TextFileSessionStateProvider">
      <providers>
        <add name="TextFileSessionStateProvider"
          type="TextFileSessionStateProvider" />
      </providers>
    </sessionState>
  </system.web>
</configuration>
```

TextFileSessionStateProvider is fully capable of reading and writing any session state generated by application code. It does not, however, support expiration callbacks. In fact, the session state files that it generates don't get cleaned up until the provider's *Dispose* method is called, which normally occurs when the application is shut down and the *AppDomain* is unloaded.

Profile Providers

Profile providers provide the interface between ASP.NET's profile service and profile data sources. The two most common reasons for writing a custom profile provider are:

- You wish to store profile data in a data source that is not supported by the profile providers included with the .NET Framework, such as an Oracle database.
- You wish to store profile data in a SQL Server database whose schema differs from that of the database used by *System.Web.Profile.SqlProfileProvider*.

The fundamental job of a profile provider is to write profile property values supplied by ASP.NET to persistent profile data sources, and to read the property values back from the data source when requested by ASP.NET. Profile providers also implement methods that allows consumers to manage profile data sources-for example, to delete profiles that haven't been accessed since a specified date.

The ProfileProvider Class

Developers writing custom profile providers begin by deriving from *System.Web.Profile.ProfileProvider*. *ProfileProvider* derives from *System.Configuration.SettingsProvider*, which in turn derives from *ProviderBase*. Together, *SettingsProvider* and *ProfileProvider* define the abstract class methods and properties that a derived class must implement in order to serve as an intermediary between the profile service and profile data sources. *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 following table describes *ProfileProvider*'s methods and properties and provides helpful notes regarding their implementation:

Table 8. ProfileProvider methods and properties

Method or Property	Description
<i>ApplicationName</i>	The name of the application using the profile provider. <i>ApplicationName</i> is used to scope profile data so that applications can choose whether to share profile data with other applications. This property can be read and

	written.
<i>GetPropertyValues</i>	Reads profile property values from the data source and returns them in a SettingsPropertyValueCollection. See "GetPropertyValues" for details.
<i>SetPropertyValues</i>	Writes profile property values to the data source. The values are provided by ASP.NET in a SettingsPropertyValueCollection. See "SetPropertyValues" for details.
<i>DeleteProfiles (ProfileInfoCollection)</i>	Deletes the specified profiles from the data source.
<i>DeleteProfiles (string[])</i>	Deletes the specified users' profiles from the data source.
<i>DeleteInactiveProfiles</i>	Deletes all inactive profiles-profiles that haven't been accessed since the specified date-from the data source.
<i>GetNumberOfInactiveProfiles</i>	Returns the number of profiles that haven't been accessed since the specified date.
<i>GetAllProfiles</i>	Returns a collection of ProfileInfo objects containing administrative information about all profiles, including user names and last activity dates.
<i>GetAllInactiveProfiles</i>	Returns a collection of ProfileInfo objects containing administrative information regarding profiles that haven't been accessed since the specified date.
<i>FindProfilesByUserName</i>	Returns a collection of ProfileInfo objects containing administrative information regarding profiles whose user names match a specified pattern.
<i>FindInactiveProfilesByUserName</i>	Returns a collection of ProfileInfo objects containing administrative information regarding profiles whose user names match a specified pattern and that haven't been accessed since the specified date.

Scoping of Profile Data

Profile data is inherently scoped by user name so that profile data can be maintained independently for each user. When storing profile data, a provider must take care to key the data by user name so it can be retrieved using the same key later. For anonymous users, profile providers use anonymous user IDs rather than user names to key profile properties. The user names passed to profile provider methods are in fact anonymous user IDs for users who are not authenticated.

In addition, all profile providers inherit from *SettingsProvider* a property named *ApplicationName* whose purpose it to scope the data managed by the provider. Applications that specify the same *ApplicationName* when configuring the profile service share profile data; applications that specify unique *ApplicationNames* do not. In addition to associating profiles with user names or anonymous user IDs (profiles are, after all, a means for storing per-user data), profile-provider implementations must associate profiles with application names so operations performed on profile data sources can be scoped accordingly.

As an example, a provider that stores profile data in a SQL database might use a command similar to the following to retrieve profile data for the user named "Jeff" and the application named "Contoso:"

```
SELECT * FROM Profiles
WHERE UserName='Jeff' AND ApplicationName='Contoso'
```

The AND in the WHERE clause ensures that other applications containing profiles keyed by the same user name don't conflict with the "Contoso" application.

GetPropertyValues

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 that executes 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'* 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.

As an example, suppose the <profile> configuration section is defined this way:

```
<profile>
  <properties>
    <add name="Greeting" type="String" />
    <add name="Count" type="Int32" defaultValue="0" />
```



```
</properties>  
</profile>
```

Each time *GetPropertyValues* is called, the *SettingsPropertyCollection* passed to it contains two *SettingsProperty* objects: one representing the *Greeting* property, the other representing the *Count* property. The first time *GetPropertyValues* is called, the provider can simply do this since the property values haven't yet been persisted in the data source:

```
SettingsPropertyValueCollection settings =  
    new SettingsPropertyValueCollection ();  
  
foreach (SettingsProperty property in properties)  
    settings.Add (new SettingsPropertyValue (property));  
  
return settings;
```

The returned *SettingsPropertyValueCollection* contains two *SettingsPropertyValues*: one representing the *Greeting* property's property value, and the other representing the *Count* property's property value. Moreover, because *PropertyValue* and *SerializedValue* are set to null in the *SettingsPropertyValue* objects and *Deserialized* is set to false, ASP.NET assigns each property a default value (which come from the properties' *defaultValue* attributes if present.)

The second time *GetPropertyValues* is called, it retrieves the property values from the data source (assuming the properties were persisted there in the call to *SetPropertyValues* that followed the previous call to *GetPropertyValues*). Once more, its job is to return a *SettingsPropertyValueCollection* containing property values. This time, however, *GetPropertyValues* has a choice of ways to communicate property values to ASP.NET:

- It can set the corresponding *SettingsPropertyValue* object's *PropertyValue* property equal to the actual property value and the object's *Deserialized* property to true. ASP.NET will retrieve the property value from *PropertyValue*. This is useful for primitive types that do not require serialization. It's also useful for explicitly assigning null values to reference types by setting *PropertyValue* to null and *Deserialized* to true.
- It can set the corresponding *SettingsPropertyValue* object's *SerializedValue* property equal to the serialized property value and the object's *Deserialized* property to false. ASP.NET will deserialize *SerializedValue* to obtain the actual property value, using the serialization type specified in the *SettingsProperty* object's *SerializeAs* property. This is useful for complex types that require serialization. The provider typically doesn't do the serialization itself; rather, it reads the serialized property value that was persisted in the data source by *SetPropertyValues*.



Inside the ASP.NET Team

Another reason for providing serialized data to ASP.NET via the *SerializedValue* property is that there is no guarantee the calling code that triggered the call to *GetPropertyValues* is actually interested in all the profile properties. Providing data through *SerializedValue* allows for lazy deserialization by *SettingsBase*. If you have ten properties being retrieved by the profile provider, and the calling code on a page only uses one of these properties, then nine of properties don't have to be deserialized, resulting in a potentially significant performance win.

Thus, *GetPropertyValues* might perform its duties this way the second time around:

```
SettingsPropertyValueCollection settings =
    new SettingsPropertyValueCollection ();

foreach (SettingsProperty property in properties)
{
    // Create a SettingsPropertyValue
    SettingsPropertyValue pp = new SettingsPropertyValue (property);

    // Read a persisted property value from the data source
    object val = GetPropertyValueFromDataSource (property.Name);

    // If val is null, set the property value to null
    if (val == null)
    {
        pp.PropertyValue = null;
        pp.Deserialized = true;
        pp.IsDirty = false;
    }

    // If val is not null, set the property value to a non-null value
    else
    {
        // TODO: Set pp.PropertyValue to the property value and
        // pp.Deserialized to true, or set pp.SerializedValue to
        // the serialized property value and Deserialized to false.
        // Which strategy you choose depends on which was written
        // to the data source: PropertyValue or SerializedValue.
    }

    // Add the SettingsPropertyValue to the collection
}
```

```
        settings.Add (pp);
    }

    // Return the collection
    return settings;
```

SetPropertyValues

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 persisted-and the physical storage medium that it's persisted in-is up to the provider. Obviously, the format in which *SetPropertyValues* persists profile data must be understood by the provider's *GetProfileProperties* method.

SetPropertyValues' job is to iterate through the supplied *SettingsPropertyValue* objects and write each property value to the data source where *GetPropertyValues* can retrieve it later on. Where *SetPropertyValues* obtains the property values from depends on the *Deserialized* properties of the corresponding *SettingsPropertyValue* objects:

- If *Deserialized* is true, *SetPropertyValues* can obtain the property value directly from the *SettingsPropertyValue* object's *PropertyValue* property.
- If *Deserialized* is false, *SetPropertyValues* can obtain the property value, in serialized form, from the *SettingsPropertyValue* object's *SerializedValue* property. There's no need for the provider to attempt to deserialize the serialized property value; it can treat the serialized property value as an opaque entity and write it to the data source. Later, *GetPropertyValues* can fetch the serialized property value from the data source and return it to ASP.NET in a *SettingsPropertyValue* object whose *SerializedValue* property holds the serialized property value and whose *Deserialized* property is false.

A profile provider's *SetPropertyValues* method might therefore be structured like this:

```
foreach (SettingsPropertyValue property in properties)
{
    // Get information about the user who owns the profile
    string username = (string) context["UserName"];
    bool authenticated = (bool) context["IsAuthenticated"];

    // Ignore this property if the user is anonymous and
    // the property's AllowAnonymous property is false
    if (!authenticated &&
        !(bool) property.Property.Attributes["AllowAnonymous"])
        continue;
```

```

// Otherwise persist the property value
if (property.Deserialized)
{
    // TODO: Write property.PropertyValue to the data source
}
else
{
    // TODO: Write property.SerializedValue to the data source
}
}

```

Alternatively, *SetPropertyValues* could ignore *PropertyValue* and simply write *SerializedValue* to the data source, regardless of whether *Deserialized* is true or false. The *GetPropertyValues* implementation would read *SerializedValue* from the data source and return it in a *SettingsPropertyValue* object's *SerializedValue* property with *Deserialized* set to false. ASP.NET would then compute the actual property value. This is the approach taken by ASP.NET's *SqlProfileProvider* provider, which only stores serialized property values in the profile database.

The example above doesn't persist a property value if the user isn't authenticated and the property isn't attributed to allow anonymous users. It assumes that if the property appears in a *SettingsPropertyCollection* passed to *GetPropertyValues*, *GetPropertyValues* will see that the property value isn't in the data source and allow ASP.NET to assign a default value. Similarly, *SetPropertyValues* may choose not to write to the data source properties whose *UsingDefaultValue* property is true, because such values are easily recreated when *GetPropertyValues* is called. If a profile provider only persists property values that have changed since they were loaded, it could even ignore properties whose *IsDirty* property is false.



Inside the ASP.NET Team

ASP.NET's *SqlProfileProvider* writes property values to the database even if *IsDirty* is false. (The *TextFileProfileProvider* class presented in the next section does the same.) That's because each time *SqlProfileProvider* records profile property values in the database, it overwrites existing values. It does, however, refrain from saving values whose *AllowAnonymous* property is false if the user is unauthenticated, and properties with *IsDirty* equal to false and *UsingDefaultValue* equal to true. A custom profile provider that stores property values in individual fields in the data source—fields that can be individually updated without affecting other fields—could be more efficient in its *SetPropertyValues* method by checking the properties' *IsDirty* values and only updating the ones that are dirty.

TextFileProfileProvider

Figure 17 contains the source code for a *ProfileProvider*-derivative named *TextFileProfileProvider* that demonstrates the minimum functionality required of a profile provider. It implements the two key *ProfileProvider* methods-*GetPropertyValues* and *SetPropertyValues*-but provides trivial implementations of the others. Despite its simplicity, *TextFileProfileProvider* is fully capable of reading and writing data generated from any profile defined in the <profile> configuration section.

TextFileProfileProvider stores profile data in text files named *Username_Profile.txt* in the application's *~/App_Data/Profile_Data* directory. Each file contains the profile data for a specific user and consists of a set of three strings (described later in this section). You must create the *~/App_Data/Profile_Data* directory before using the provider; the provider doesn't attempt to create the directory if it doesn't exist. In addition, the provider must have read/write access to the *~/App_Data/Profile_Data* directory.

Figure 17. TextFileProfileProvider

```
using System;
using System.Configuration;
using System.Configuration.Provider;
using System.Collections.Specialized;
using System.Security.Permissions;
using System.Web;
using System.Web.Profile;
using System.Web.Hosting;
using System.Globalization;
using System.IO;
using System.Text;

[SecurityPermission(SecurityAction.Assert,
    Flags=SecurityPermissionFlag.SerializationFormatter)]
public class TextFileProfileProvider : ProfileProvider
{
    public override string ApplicationName
    {
        get { throw new NotSupportedException(); }
        set { throw new NotSupportedException(); }
    }

    public override void Initialize(string name,
        NameValueCollection config)
    {
        // Verify that config isn't null
        if (config == null)
            throw new ArgumentNullException("config");

        // Assign the provider a default name if it doesn't have one
        if (String.IsNullOrEmpty(name))
```

```

        name = "TextFileProfileProvider";

// Add a default "description" attribute to config if the
// attribute doesn't exist or is empty
if (string.IsNullOrEmpty(config["description"]))
{
    config.Remove("description");
    config.Add("description", "Text file profile provider");
}

// Call the base class's Initialize method
base.Initialize(name, config);

// Throw an exception if unrecognized attributes remain
if (config.Count > 0)
{
    string attr = config.GetKey(0);
    if (!string.IsNullOrEmpty(attr))
        throw new ProviderException
            ("Unrecognized attribute: " + attr);
}

// Make sure we can read and write files
// in the ~/App_Data/Profile_Data directory
FileIOPermission permission =
    new FileIOPermission (FileIOPermissionAccess.AllAccess,
        HttpContext.Current.Server.MapPath(
            "~/App_Data/Profile_Data"));
permission.Demand();
}

public override SettingsPropertyValueCollection
    GetPropertyValues(SettingsContext context,
        SettingsPropertyCollection properties)
{
    SettingsPropertyValueCollection settings =
        new SettingsPropertyValueCollection();

// Do nothing if there are no properties to retrieve
if (properties.Count == 0)
    return settings;

// For properties lacking an explicit SerializeAs setting, set
// SerializeAs to String for strings and primitives, and XML
// for everything else

```

```

foreach (SettingsProperty property in properties)
{
    if (property.SerializeAs ==
        SettingsSerializeAs.ProviderSpecific)
        if (property.PropertyType.IsPrimitive ||
            property.PropertyType == typeof(String))
            property.SerializeAs = SettingsSerializeAs.String;
        else
            property.SerializeAs = SettingsSerializeAs.Xml;

    settings.Add(new SettingsPropertyValue(property));
}

// Get the user name or anonymous user ID
string username = (string)context["UserName"];

// NOTE: Consider validating the user name here to prevent
// malicious user names such as "../Foo" from targeting
// directories other than ~/App_Data/Profile_Data

// Load the profile
if (!String.IsNullOrEmpty(username))
{
    StreamReader reader = null;
    string[] names;
    string values;
    byte[] buf = null;

    try
    {
        // Open the file containing the profile data
        try
        {
            string path =
                String.Format(
                    "~/App_Data/Profile_Data/{0}_Profile.txt",
                    username.Replace('\\', '_'));
            reader = new StreamReader
                (HttpContext.Current.Server.MapPath(path));
        }
        catch (IOException)
        {
            // Not an error if file doesn't exist
            return settings;
        }
    }
}

```

```

        // Read names, values, and buf from the file
        names = reader.ReadLine().Split (':');

        values = reader.ReadLine();
        if (!string.IsNullOrEmpty(values))
        {
            UnicodeEncoding encoding = new UnicodeEncoding();
            values = encoding.GetString
                (Convert.FromBase64String(values));
        }

        string temp = reader.ReadLine();
        if (!String.IsNullOrEmpty(temp))
        {
            buf = Convert.FromBase64String(temp);
        }
        else
            buf = new byte[0];
    }
    finally
    {
        if (reader != null)
            reader.Close();
    }

    // Decode names, values, and buf and initialize the
    // SettingsPropertyValueCollection returned to the caller
    DecodeProfileData(names, values, buf, settings);
}

return settings;
}

public override void SetPropertyValues(SettingsContext context,
SettingsPropertyValueCollection properties)
{
    // Get information about the user who owns the profile
    string username = (string) context["UserName"];
    bool authenticated = (bool) context["IsAuthenticated"];

    // NOTE: Consider validating the user name here to prevent
    // malicious user names such as "../Foo" from targeting
    // directories other than ~/App_Data/Profile_Data

```



```

// Do nothing if there is no user name or no properties
if (String.IsNullOrEmpty (username) || properties.Count == 0)
    return;

// Format the profile data for saving
string names = String.Empty;
string values = String.Empty;
byte[] buf = null;

EncodeProfileData(ref names, ref values, ref buf,
    properties, authenticated);

// Do nothing if no properties need saving
if (names == String.Empty)
    return;

// Save the profile data
StreamWriter writer = null;

try
{
    string path =
        String.Format(
            "~/App_Data/Profile_Data/{0}_Profile.txt",
            username.Replace('\\', '_'));
    writer = new StreamWriter
        (HttpContext.Current.Server.MapPath(path), false);

    writer.WriteLine(names);

    if (!String.IsNullOrEmpty(values))
    {
        UnicodeEncoding encoding = new UnicodeEncoding();
        writer.WriteLine(Convert.ToBase64String
            (encoding.GetBytes(values)));
    }
    else
        writer.WriteLine();

    if (buf != null && buf.Length > 0)
        writer.WriteLine(Convert.ToBase64String(buf));
    else
        writer.WriteLine();
}
finally

```

```
        {
            if (writer != null)
                writer.Close();
        }
    }

    public override int DeleteInactiveProfiles
        (ProfileAuthenticationOption authenticationOption,
         DateTime userInactiveSinceDate)
    {
        throw new NotSupportedException();
    }

    public override int DeleteProfiles(string[] usernames)
    {
        throw new NotSupportedException();
    }

    public override int DeleteProfiles(ProfileInfoCollection profiles)
    {
        throw new NotSupportedException();
    }

    public override ProfileInfoCollection
        FindInactiveProfilesByUserName(ProfileAuthenticationOption
        authenticationOption, string usernameToMatch, DateTime
        userInactiveSinceDate, int pageIndex, int pageSize, out int
        totalRecords)
    {
        throw new NotSupportedException();
    }

    public override ProfileInfoCollection FindProfilesByUserName
        (ProfileAuthenticationOption authenticationOption,
         string usernameToMatch, int pageIndex, int pageSize,
         out int totalRecords)
    {
        throw new NotSupportedException();
    }

    public override ProfileInfoCollection GetAllInactiveProfiles
        (ProfileAuthenticationOption authenticationOption,
         DateTime userInactiveSinceDate, int pageIndex, int pageSize,
         out int totalRecords)
    {
```

```

        throw new NotSupportedException();
    }

    public override ProfileInfoCollection GetAllProfiles
        (ProfileAuthenticationOption authenticationOption,
         int pageIndex, int pageSize, out int totalRecords)
    {
        throw new NotSupportedException();
    }

    public override int GetNumberOfInactiveProfiles
        (ProfileAuthenticationOption authenticationOption,
         DateTime userInactiveSinceDate)
    {
        throw new NotSupportedException();
    }

    // Helper methods
    private void DecodeProfileData(string[] names, string values,
        byte[] buf, SettingsPropertyValueCollection properties)
    {
        if (names == null || values == null || buf == null ||
            properties == null)
            return;

        for (int i=0; i<names.Length; i+=4)
        {
            // Read the next property name from "names" and retrieve
            // the corresponding SettingsPropertyValue from
            // "properties"
            string name = names[i];
            SettingsPropertyValue pp = properties[name];

            if (pp == null)
                continue;

            // Get the length and index of the persisted property value
            int pos = Int32.Parse(names[i + 2],
                CultureInfo.InvariantCulture);
            int len = Int32.Parse(names[i + 3],
                CultureInfo.InvariantCulture);

            // If the length is -1 and the property is a reference
            // type, then the property value is null
            if (len == -1 && !pp.Property.PropertyType.IsValueType)

```

```

        {
            pp.PropertyValue = null;
            pp.IsDirty = false;
            pp.Deserialized = true;
        }

        // If the property value was persisted as a string,
        // restore it from "values"
        else if (names[i + 1] == "S" && pos >= 0 && len > 0 &&
            values.Length >= pos + len)
            pp.SerializedValue = values.Substring(pos, len);

        // If the property value was persisted as a byte array,
        // restore it from "buf"
        else if (names[i + 1] == "B" && pos >= 0 && len > 0 &&
            buf.Length >= pos + len)
        {
            byte[] buf2 = new byte[len];
            Buffer.BlockCopy(buf, pos, buf2, 0, len);
            pp.SerializedValue = buf2;
        }
    }
}

private void EncodeProfileData(ref string allNames,
    ref string allValues, ref byte[] buf,
    SettingsPropertyValueCollection properties,
    bool userIsAuthenticated)
{
    StringBuilder names = new StringBuilder();
    StringBuilder values = new StringBuilder();
    MemoryStream stream = new MemoryStream();

    try
    {
        foreach (SettingsPropertyValue pp in properties)
        {
            // Ignore this property if the user is anonymous and
            // the property's AllowAnonymous property is false
            if (!userIsAuthenticated &&
                !(bool)pp.Property.Attributes["AllowAnonymous"])
                continue;

            // Ignore this property if it's not dirty and is
            // currently assigned its default value

```

```

if (!pp.IsDirty && pp.UsingDefaultValue)
    continue;

int len = 0, pos = 0;
string propValue = null;

// If Deserialized is true and PropertyValue is null,
// then the property's current value is null (which
// we'll represent by setting len to -1)
if (pp.Deserialized && pp.PropertyValue == null)
    len = -1;

// Otherwise get the property value from
// SerializedValue
else
{
    object sVal = pp.SerializedValue;

    // If SerializedValue is null, then the property's
    // current value is null
    if (sVal == null)
        len = -1;

    // If sVal is a string, then encode it as a string
    else if (sVal is string)
    {
        propValue = (string)sVal;
        len = propValue.Length;
        pos = values.Length;
    }

    // If sVal is binary, then encode it as a byte
    // array
    else
    {
        byte[] b2 = (byte[])sVal;
        pos = (int)stream.Position;
        stream.Write(b2, 0, b2.Length);
        stream.Position = pos + b2.Length;
        len = b2.Length;
    }
}

// Add a string conforming to the following format
// to "names:"

```

```

//
// "name:B|S:pos:len"
//   ^   ^   ^   ^
//   |   |   |   |
//   |   |   |   +--- Length of data
//   |   |   +----- Offset of data
//   |   +----- Location (B="buf", S="values")
//   +----- Property name

names.Append(pp.Name + ":" + ((propValue != null) ?
    "S" : "B") + ":" +
    pos.ToString(CultureInfo.InvariantCulture) + ":" +
    len.ToString(CultureInfo.InvariantCulture) + ":");

// If the property value is encoded as a string, add the
// string to "values"
if (propValue != null)
    values.Append(propValue);
}

// Copy the binary property values written to the
// stream to "buf"
buf = stream.ToArray();
}
finally
{
    if (stream != null)
        stream.Close();
}

allNames = names.ToString();
allValues = values.ToString();
}
}

```

TextFileProfileProvider stores profile data in exactly the same format as ASP.NET's *SqlProfileProvider*, with some extra base-64 encoding thrown in to allow binary data and XML data to be stored in a single line of text. Its *EncodeProfileData* and *DecodeProfileData* methods, which do the encoding and decoding, are based on similar methods—methods which are internal and therefore can't be called from user code—in ASP.NET's *ProfileModule* class.

EncodeProfileData packs all the property values passed to it into three values:

- A string variable named *names* that encodes each property value 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 string variable named *values* (S) or the byte[] variable named *buf* (B). *StartPos* and *Length* indicate the starting position (0-based) within *values* or *buf* 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.

- A string variable named *values* that stores string and XML property values. Before writing *values* to a text file, *TextFileProfileProvider* base-64 encodes it so that XML data spanning multiple lines can be packed into a single line of text.
- A byte[] variable named *buf* that stores binary property values. Before writing *buf* to a text file, *TextFileProfileProvider* base-64 encodes it so that binary data can be packed into a line of text.

DecodeProfileData reverses the encoding, converting *names*, *values*, and *buf* back into property values and applying them to the members of the supplied *SettingsPropertyValueCollection*. 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 implementor.

Figure 18 demonstrates how to make *TextFileProfileProvider* the default profile provider. It assumes that *TextFileProfileProvider* is implemented in an assembly named *CustomProviders*.

Figure 18. Web.config file making TextFileProfileProvider the default profile provider

```
<configuration>
  <system.web>
    <profile defaultProvider="TextFileProfileProvider">
      <properties>
        ...
      </properties>
      <providers>
        <add name="TextFileProfileProvider"
            type="TextFileProfileProvider, CustomProviders"
            description="Text file profile provider"
        />
      </providers>
    </profile>
  </system.web>
</configuration>
```

For simplicity, *TextFileProfileProvider* does not honor the *ApplicationName* property. Because *TextFileProfileProvider* stores profile data in text files in a subdirectory of the application root, all data that it manages is inherently application-scoped. A full-featured profile provider must support *ApplicationName* so profile consumers can choose whether to keep profile data private or share it with other applications.

Web Event Providers

Web event providers provide the interface between ASP.NET's health monitoring subsystem and data sources that log or further process the events ("Web events") fired by that subsystem. The most common reason for writing a custom Web event provider is to enable administrators to log Web events in media not supported by the built-in Web event providers. ASP.NET 2.0 comes with Web event providers for logging Web events in the Windows event log (*EventLogWebEventProvider*) and in Microsoft SQL Server databases (*SqlWebEventProvider*). It also includes Web event providers that respond to Web events by sending e-mail (*SimpleMailWebEventProvider* and *TemplatedMailWebEventProvider*) and by forwarding them to the WMI subsystem (*WmiWebEventProvider*) and to diagnostics trace (*TraceWebEventProvider*).

Developers writing custom Web event providers generally begin by deriving from *System.Web.Management.WebEventProvider*, which derives from *ProviderBase* and adds abstract methods and properties defining the basic characteristics of a Web event provider, or from *System.Web.Management.BufferedWebEventProvider*, which derives from *WebEventProvider* and adds buffering support. (*SqlWebEventProvider*, for example, derives from *BufferedWebEventProvider* so events can be "batched" and committed to the database en masse.) Developers writing Web event providers that send e-mail may also derive from *System.Web.Management.MailWebEventProvider*, which is the base class for *SimpleMailWebEventProvider* and *TemplatedMailWebEventProvider*.

The WebEventProvider Class

System.Web.Management.WebEventProvider is prototyped as follows:

```
public abstract class WebEventProvider : ProviderBase
{
    public abstract void ProcessEvent (WebBaseEvent raisedEvent);
    public abstract void Flush ();
    public abstract void Shutdown ();
}
```

The following table describes *WebEventProvider*'s members and provides helpful notes regarding their implementation:

Table 9. WebEventProvider methods and properties

Method	Description
<i>ProcessEvent</i>	Called by ASP.NET when a Web event mapped to this provider fires. The <i>raisedEvent</i> parameter encapsulates information about the Web event, including the event type, event code, and a message describing the event.
<i>Flush</i>	Notifies providers that buffer Web events to flush their buffers. Called by ASP.NET when

	<i>System.Web.Management.WebEventManager.Flush</i> is called to flush buffered events.
<i>Shutdown</i>	Called by ASP.NET when the provider is unloaded. Use it to release any unmanaged resources held by the provider or to perform other clean-up operations.

Your job in implementing a custom Web event provider in a derived class is to override and provide implementations of *WebEventProvider*'s abstract methods, and optionally to override key virtuals such as *Initialize*.

TextFileWebEventProvider

Figure 19 contains the source code for a sample Web event provider named *TextFileWebEventProvider* that logs Web events in a text file. The text file's name is specified using the provider's *logFileName* attribute, and the text file is automatically created by the provider if it doesn't already exist. A new entry is written to the log each time *TextFileWebEventProvider*'s *ProcessEvent* method is called notifying the provider that a Web event has been fired.

Figure 19. TextFileWebEventProvider

```
using System;
using System.Web.Management;
using System.Configuration.Provider;
using System.Collections.Specialized;
using System.Web.Hosting;
using System.IO;
using System.Security.Permissions;
using System.Web;

public class TextFileWebEventProvider : WebEventProvider
{
    private string _LogFileName;

    public override void Initialize(string name,
        NameValueCollection config)
    {
        // Verify that config isn't null
        if (config == null)
            throw new ArgumentNullException("config");

        // Assign the provider a default name if it doesn't have one
        if (String.IsNullOrEmpty(name))
            name = "TextFileWebEventProvider";

        // Add a default "description" attribute to config if the
        // attribute doesn't exist or is empty
        if (string.IsNullOrEmpty(config["description"]))
```

```

    {
        config.Remove("description");
        config.Add("description", "Text file Web event provider");
    }

    // Call the base class's Initialize method
    base.Initialize(name, config);

    // Initialize _LogFileName and make sure the path
    // is app-relative
    string path = config["logFileName"];

    if (String.IsNullOrEmpty(path))
        throw new ProviderException
            ("Missing logFileName attribute");

    if (!VirtualPathUtility.IsAppRelative(path))
        throw new ArgumentException
            ("logFileName must be app-relative");

    string fullyQualifiedPath = VirtualPathUtility.Combine
        (VirtualPathUtility.AppendTrailingSlash
        (HttpRequest.AppDomainAppVirtualPath), path);

    _LogFileName = HostingEnvironment.MapPath(fullyQualifiedPath);
    config.Remove("logFileName");

    // Make sure we have permission to write to the log file
    // throw an exception if we don't
    FileIOPermission permission =
        new FileIOPermission(FileIOPermissionAccess.Write |
        FileIOPermissionAccess.Append, _LogFileName);
    permission.Demand();

    // Throw an exception if unrecognized attributes remain
    if (config.Count > 0)
    {
        string attr = config.GetKey(0);
        if (!String.IsNullOrEmpty(attr))
            throw new ProviderException
                ("Unrecognized attribute: " + attr);
    }
}

public override void ProcessEvent(WebBaseEvent raisedEvent)

```

```

    {
        // Write an entry to the log file
        LogEntry (FormatEntry(raisedEvent));
    }

public override void Flush() {}
public override void Shutdown() {}

// Helper methods
private string FormatEntry(WebBaseEvent e)
{
    return String.Format("{0}\t{1}\t{2} (Event Code: {3})",
        e.EventTime, e.GetType ().ToString (), e.Message,
        e.EventCode);
}

private void LogEntry(string entry)
{
    StreamWriter writer = null;

    try
    {
        writer = new StreamWriter(_LogFileName, true);
        writer.WriteLine(entry);
    }
    finally
    {
        if (writer != null)
            writer.Close();
    }
}
}
}

```

The Web.config file in Figure 20 registers *TextFileWebEventProvider* as a Web event provider and maps it to Application Lifetime events—one of several predefined Web event types fired by ASP.NET. Application Lifetime events fire at key junctures during an application's lifetime, including when the application starts and stops.

Figure 20. Web.config file mapping Application Lifetime events to TextFileWebEventProvider

```

<configuration>
  <system.web>
    <healthMonitoring enabled="true">
      <providers>
        <add name="AspNetTextFileWebEventProvider"

```

```

        type="TextFileWebEventProvider"
        logFileName="~/App_Data/Contosolog.txt"
    />
</providers>
<rules>
    <add name="Contoso Application Lifetime Events"
        eventName="Application Lifetime Events"
        provider="AspNetTextFileWebEventProvider"
        minInterval="00:00:01" minInstances="1"
        maxLimit="Infinite"
    />
</rules>
</healthMonitoring>
</system.web>
</configuration>

```

Figure 21 shows a log file generated by *TextFileWebEventProvider*, with tabs transformed into line breaks for formatting purposes. (In an actual log file, each entry comprises a single line.) An administrator using this log file now has a written record of application starts and stops.

Figure 21: Sample log produced by TextFileWebEventProvider

```

5/12/2005 5:56:05 PM
System.Web.Management.WebApplicationLifetimeEvent
Application is starting. (Event Code: 1001)

5/12/2005 5:56:16 PM
System.Web.Management.WebApplicationLifetimeEvent
Application is shutting down. Reason: Configuration changed. (Event Code:
1002)

5/12/2005 5:56:16 PM
System.Web.Management.WebApplicationLifetimeEvent
Application is shutting down. Reason: Configuration changed. (Event Code:
1002)

5/12/2005 5:56:19 PM
System.Web.Management.WebApplicationLifetimeEvent
Application is starting. (Event Code: 1001)

5/12/2005 5:56:23 PM
System.Web.Management.WebApplicationLifetimeEvent
Application is shutting down. Reason: Configuration changed. (Event Code:
1002)

5/12/2005 5:56:23 PM

```

```
System.Web.Management.WebApplicationLifetimeEvent
Application is shutting down. Reason: Configuration changed. (Event Code:
1002)
```

```
5/12/2005 5:56:26 PM
```

```
System.Web.Management.WebApplicationLifetimeEvent
Application is starting. (Event Code: 1001)
```

The BufferedWebEventProvider Class

One downside to *TextFileWebEventProvider* is that it opens, writes to, and closes a text file each time a Web event mapped to it fires. That might not be bad for Application Lifetime events, which fire relatively infrequently, but it could adversely impact the performance of the application as a whole if used to log events that fire more frequently (for example, in every request).

The solution is to do as *SqlWebEventProvider* does and derive from *BufferedWebEventProvider* rather than *WebEventProvider*. *BufferedWebEventProvider* adds buffering support to *WebEventProvider*. It provides default implementations of some of *WebEventProvider*'s abstract methods, most notably a default implementation of *ProcessEvent* that buffers Web events in a *WebEventBuffer* object if buffering is enabled. It also adds an abstract method named *ProcessEventFlush* that's called when buffered Web events need to be unbuffered. And it adds properties named *UseBuffering* and *BufferMode* (complete with implementations) that let the provider determine at run-time whether buffering is enabled and, if it is, what the buffering parameters are.

`System.Web.Management.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 table describes *BufferedWebEventProvider*'s members and provides helpful notes regarding their implementation:

Table 10. BufferedWebEventProvider methods and properties

Method or Property	Description
<i>UseBuffering</i>	Boolean property that specifies whether buffering is enabled. <i>BufferedWebEventProvider.Initialize</i> initializes this property from the <i>buffer</i> attribute of the <add> element that registers the provider. <i>UseBuffering</i> defaults to true.
<i>BufferMode</i>	String property that specifies the buffer mode. <i>BufferedWebEventProvider.Initialize</i> initializes this property from the <i>bufferMode</i> attribute of the <add> element that registers the provider. <i>bufferMode</i> values are defined in the <bufferModes> section of the <healthMonitoring> configuration section. This property has no default value. <i>BufferedWebEventProvider.Initialize</i> throws an exception if <i>UseBuffering</i> is true but the <i>bufferMode</i> attribute is missing.
<i>Initialize</i>	Overridden by <i>BufferedWebEventProvider</i> . The default implementation initializes the provider's <i>UseBuffering</i> and <i>BufferMode</i> properties, calls <i>base.Initialize</i> , and then throws an exception if unprocessed configuration attributes remain in the <i>config</i> parameter's <i>NameValueCollection</i> .
<i>ProcessEvent</i>	Overridden by <i>BufferedWebEventProvider</i> . The default implementation calls <i>ProcessEventFlush</i> if buffering is disabled (that is, if <i>UseBuffering</i> is false) or adds the event to an internal <i>WebEventBuffer</i> if buffering is enabled.
<i>Flush</i>	Overridden by <i>BufferedWebEventProvider</i> . The default implementation calls <i>Flush</i> on the <i>WebEventBuffer</i> holding buffered Web events. <i>WebEventBuffer.Flush</i> , in turn, conditionally calls <i>ProcessEventFlush</i> using internal logic that takes into account, among other things, the current buffer mode and elapsed time.
<i>ProcessEventFlush</i>	Called by ASP.NET to flush buffered Web events. The <i>WebEventBufferFlushInfo</i> parameter passed to this method includes, among other things, an <i>Event</i> property containing a collection of buffered Web events. This method is abstract (MustOverride in Visual Basic) and must be overridden in a derived class.

Your job in implementing a custom buffered Web event provider in a derived class is to override and provide implementations of *BufferedWebEventProvider*'s abstract methods, including the *Shutdown* method, which is inherited from *WebEventProvider* but not overridden by *BufferedWebEventProvider*, and *ProcessEventFlush*, which exposes a

collection of buffered Web events that you can iterate over. Of course, you can also override key virtuals such as *Initialize*.

BufferedTextFileWebEventProvider

Figure 22 contains the source code for a sample buffered Web event provider named *BufferedTextFileWebEventProvider*, which logs Web events in a text file just like *TextFileWebEventProvider*. However, unlike *TextFileWebEventProvider*, *BufferedTextFileWebEventProvider* doesn't write to the log file every time it receives a Web event. Instead, it uses the buffering support built into *BufferedWebEventProvider* to cache Web events. If *UseBuffering* is true, *BufferedTextFileWebEventProvider* commits Web events to the log file only when its *ProcessEventFlush* method is called.

Figure 22. BufferedTextFileWebEventProvider

```
using System;
using System.Web.Management;
using System.Configuration.Provider;
using System.Collections.Specialized;
using System.Web.Hosting;
using System.IO;
using System.Security.Permissions;
using System.Web;

public class BufferedTextFileWebEventProvider :
    BufferedWebEventProvider
{
    private string _LogFileName;

    public override void Initialize(string name,
        NameValueCollection config)
    {
        // Verify that config isn't null
        if (config == null)
            throw new ArgumentNullException("config");

        // Assign the provider a default name if it doesn't have one
        if (String.IsNullOrEmpty(name))
            name = "BufferedTextFileWebEventProvider";

        // Add a default "description" attribute to config if the
        // attribute doesn't exist or is empty
        if (string.IsNullOrEmpty(config["description"]))
        {
            config.Remove("description");
            config.Add("description",
                "Buffered text file Web event provider");
        }
    }
}
```

```

// Initialize _LogFileName. NOTE: Do this BEFORE calling the
// base class's Initialize method. BufferedWebEventProvider's
// Initialize method checks for unrecognized attributes and
// throws an exception if it finds any. If we don't process
// logFileName and remove it from config, base.Initialize will
// throw an exception.

string path = config["logFileName"];

if (String.IsNullOrEmpty(path))
    throw new ProviderException
        ("Missing logFileName attribute");

if (!VirtualPathUtility.IsAppRelative(path))
    throw new ArgumentException
        ("logFileName must be app-relative");

string fullyQualifiedPath = VirtualPathUtility.Combine
    (VirtualPathUtility.AppendTrailingSlash
    (HttpRuntime.AppDomainAppVirtualPath), path);

_LogFileName = HostingEnvironment.MapPath(fullyQualifiedPath);
config.Remove("logFileName");

// Make sure we have permission to write to the log file
// throw an exception if we don't
FileIOPermission permission =
    new FileIOPermission(FileIOPermissionAccess.Write |
    FileIOPermissionAccess.Append, _LogFileName);
permission.Demand();

// Call the base class's Initialize method
base.Initialize(name, config);

// NOTE: No need to check for unrecognized attributes
// here because base.Initialize has already done it
}

public override void ProcessEvent(WebBaseEvent raisedEvent)
{
    if (UseBuffering)
    {
        // If buffering is enabled, call the base class's
        // ProcessEvent method to buffer the event
    }
}

```



```

        base.ProcessEvent(raisedEvent);
    }
    else
    {
        // If buffering is not enabled, log the Web event now
        LogEntry(FormatEntry(raisedEvent));
    }
}

public override void ProcessEventFlush (WebEventBufferFlushInfo
flushInfo)
{
    // Log the events buffered in flushInfo.Events
    foreach (WebBaseEvent raisedEvent in flushInfo.Events)
        LogEntry (FormatEntry(raisedEvent));
}

public override void Shutdown()
{
    Flush();
}

// Helper methods
private string FormatEntry(WebBaseEvent e)
{
    return String.Format("{0}\t{1}\t{2} (Event Code: {3})",
        e.EventTime, e.GetType ().ToString (), e.Message,
        e.EventCode);
}


private void LogEntry(string entry)
{
    StreamWriter writer = null;

    try
    {
        writer = new StreamWriter(_LogFileName, true);
        writer.WriteLine(entry);
    }
    finally
    {
        if (writer != null)
            writer.Close();
    }
}

```

```
}
```

One notable aspect of *BufferedTextFileWebEventProvider*'s implementation is that its *Initialize* method processes the *logFileName* configuration attribute before calling *base.Initialize*, not after. The reason why is important. *BufferedWebEventProvider*'s *Initialize* method throws an exception if it doesn't recognize one or more of the configuration attributes in the *config* parameter. Therefore, custom attributes such as *logFileName* must be processed and removed from *config* before the base class's *Initialize* method is called. In addition, there's no need for *BufferedTextFileWebEventProvider*'s own *Initialize* method to check for unrecognized configuration attributes since that check is performed by the base class.



Inside the ASP.NET Team

BufferedWebEventProvider's *Initialize* method is inconsistent with other providers' *Initialize* implementations in its handling of configuration attributes. The difference isn't critical, but it is something that provider developers should be aware of. The reason for the inconsistency is simple and was summed up this way by an ASP.NET dev lead:

"Different devs wrote different providers and we didn't always manage to herd the cats."

That's something any dev who has worked as part of a large team can appreciate.

The *Web.config* file in Figure 23 registers *BufferedTextFileWebEventProvider* as a Web event provider and maps it to Application Lifetime events. Note the *bufferMode* attribute setting the buffer mode to "Logging." "Logging" is one of a handful of predefined buffer modes; you can examine them all in ASP.NET's default configuration files. If desired, additional buffer modes may be defined in the `<bufferModes>` section of the `<healthMonitoring>` configuration section. If no buffer mode is specified, the provider throws an exception. That behavior isn't coded into *BufferedTextFileWebEventProvider*, but instead is inherited from *BufferedWebEventProvider*.

Figure 23. Web.config file mapping Application Lifetime events to BufferedTextFileWebEventProvider

```
<configuration>
  <system.web>
    <healthMonitoring enabled="true">
      <providers>
        <add name="AspNetBufferedTextFileWebEventProvider"
            type="BufferedTextFileWebEventProvider"
            logFileName="~/App_Data/Contosolog.txt"
            bufferMode="Logging">>
      </providers>
    </healthMonitoring>
  </system.web>
</configuration>
```

```
    />
  </providers>
  <rules>
    <add name="Contoso Application Lifetime Events"
      eventName="Application Lifetime Events"
      provider="AspNetBufferedTextFileWebEventProvider"
      minInterval="00:00:01" minInstances="1"
      maxLimit="Infinite"
    />
  </rules>
</healthMonitoring>
</system.web>
</configuration>
```

Web Parts Personalization Providers

Web Parts personalization providers provide the interface between ASP.NET's Web Parts personalization service and personalization data sources. The two most common reasons for writing a custom Web Parts personalization provider are:

- You wish to store personalization data in a data source that is not supported by the Web Parts personalization providers included with the .NET Framework, such as an Oracle database.
- You wish to store personalization data in a SQL Server database whose schema differs from that of the database used by *System.Web.UI.WebControls.WebParts.SqlPersonalizationProvider*.

The fundamental job of a Web Parts personalization provider is to provide persistent storage for personalization state-state regarding the content and layout of Web Parts pages-generated 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 PersonalizationProvider Base Class

Developers writing custom personalization providers begin by deriving from *System.Web.UI.WebControls.WebParts.PersonalizationProvider*, which derives from *ProviderBase* and adds several methods and properties defining the characteristics of a Web Parts personalization provider. Because many *PersonalizationProvider* methods come with default implementations (that is, are virtual rather than abstract), a functional provider can be written by overriding as little as three or four members in a derived class. *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 table describes *PersonalizationProvider*'s members and provides helpful notes regarding their implementation:

Table 11. PersonalizationProvider methods and properties

Method or Property	Description
<i>ApplicationName</i>	The name of the application using the personalization provider. <i>ApplicationName</i> is used to scope personalization state so that applications can choose whether to share personalization state with other applications. This property can be read and written.
<i>CreateSupportedUserCapabilities</i>	Returns a list of <i>WebPartUserCapability</i> objects indicating which users can access shared personalization state and which are permitted to save personalization state. User capabilities can be specified in an <authorization> element in the <personalization> section of the <webParts> configuration section. By default, all users can read shared personalization data, and all users can read and write user-scoped personalization data.

<i>DetermineInitialScope</i>	Used by <i>WebPartPersonalization</i> to determine whether the initial scope of previously loaded personalization state is shared or per-user.
<i>DetermineUserCapabilities</i>	Returns a dictionary of <i>WebPartUserCapability</i> objects indicating whether the current user can access shared personalization state and save personalization state. User capabilities can be specified in an <authorization> element in the <personalization> section of the <webParts> configuration section. By default, all users can access shared state and all can save personalization settings.
<i>LoadPersonalizationState</i>	Retrieves raw personalization data from the data source and converts it into a <i>PersonalizationState</i> object. The default implementation retrieves the current user name (unless instructed not to with the <i>ignoreCurrentUser</i> parameter) and path from the specified <i>WebPartManager</i> . Then it calls <i>LoadPersonalizationBlobs</i> to get the raw data as a byte array and deserializes the byte array into a <i>PersonalizationState</i> object.
<i>ResetPersonalizationState</i>	Deletes personalization state from the data source. The default implementation retrieves the current user name and path from the specified <i>WebPartManager</i> and calls <i>ResetPersonalizationBlob</i> to delete the corresponding data.
<i>SavePersonalizationState</i>	Writes a <i>PersonalizationState</i> object to the data source. The default implementation serializes the <i>PersonalizationState</i> object into a byte array and calls <i>SavePersonalizationBlob</i> to write the byte array to the data source.
<i>FindState</i>	Returns a collection of <i>PersonalizationStateInfo</i> objects containing administrative information regarding records in the data source that match the specified criteria-for example, records corresponding to users named Jeff* that have been modified since January 1, 2005. Wildcard support is provider-dependent.
<i>GetCountOfState</i>	Returns the number of records in the data source that match the specified criteria-for example, records corresponding to users named Jeff* that haven't been modified since January 1, 2005. Wildcard support is provider-dependent.
<i>LoadPersonalizationBlobs</i>	Retrieves personalization state as opaque blobs

	from the data source. Retrieves both shared and user personalization state corresponding to a specified user and a specified page.
<i>ResetPersonalizationBlob</i>	Deletes personalization state corresponding to a specified user and a specified page from the data source.
<i>ResetState</i>	Deletes personalization state corresponding to the specified users and specified pages from the data source.
<i>ResetUserState</i>	Deletes user personalization state corresponding to the specified pages and that hasn't been updated since a specified date from the data source.
<i>SavePersonalizationBlob</i>	Writes personalization state corresponding to a specified user and a specified page as an opaque blob to the data source. If <i>userName</i> is null (Nothing in Visual Basic), then the personalization state is shared state and is not keyed by user name.

Your job in implementing a custom Web Parts personalization provider is to provide implementations of *PersonalizationProvider*'s abstract methods, and optionally to override key virtuals such as *Initialize*. In most cases, the default implementations of *PersonalizationProvider*'s *LoadPersonalizationState*, *ResetPersonalizationState*, and *SavePersonalizationState* methods will suffice. However, you may override these methods if you wish to modify the binary format in which personalization state is stored—though doing so also requires deriving from *PersonalizationState* to support custom serialization and deserialization.

You can build a provider that's capable of persisting the personalization state generated as users modify the content and layout of Web Parts pages by implementing three key *PersonalizationProvider* methods: *LoadPersonalizationBlobs*, *ResetPersonalizationBlob*, and *SavePersonalizationBlob*. It is highly recommended that you implement *GetCountOfState*, too, because that method is called by *WebPartPersonalization.HasPersonalizationState*, which in turn is called by the *WebPartPageMenu* control (which was present in beta 1 and is still available as a sample). Other abstract methods inherited from *PersonalizationProvider* are administrative in nature and should be implemented by a fully featured provider but are not strictly required.

Scoping of Personalization Data

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 it can be retrieved using the same keys later.

In addition, all Web Parts personalization providers inherit from *PersonalizationProvider* a property named *ApplicationName* whose purpose it to scope the data managed by the provider. Applications that specify the same *ApplicationName* when configuring the Web Parts personalization service share personalization state; applications that specify unique *ApplicationNames* do not. Web Parts personalization providers that support *ApplicationName* must associate personalization state with application names so operations performed on personalization data sources can be scoped accordingly.

As an example, a provider that stores Web Parts personalization data in a SQL database might use a command similar to the following to retrieve personalization state for the user named "Jeff" and the application named "Contoso:"

```
SELECT * FROM PersonalizationState
WHERE UserName='Jeff' AND Path='~/Default.aspx'
AND ApplicationName='Contoso'
```

The final AND in the WHERE clause ensures that other applications containing personalization state keyed by the same user name and path don't conflict with the "Contoso" application.

TextFilePersonalizationProvider

Figure 24 contains the source code for a *PersonalizationProvider*-derivative named *TextFilePersonalizationProvider* that demonstrates the minimum functionality required of a Web Parts personalization provider. It implements the three key abstract *PersonalizationProvider* methods-*LoadPersonalizationBlobs*, *ResetPersonalizationBlob*, and *SavePersonalizationBlob*-but provides trivial implementations of the others. Despite its simplicity, *TextFilePersonalizationProvider* is fully capable of reading and writing the personalization state generated as users customize the layout and content of Web Parts pages.

TextFilePersonalizationProvider stores personalization state in text files in the application's ~/App_Data/Personalization_Data directory. Each file contains the personalization state for a specific user and a specific page and consists of a single base-64 string generated from the personalization blob (byte array) passed to *SavePersonalizationBlob*. The file name, which is generated by combining the user name and a hash of the request path, indicates which user and which path the state corresponds to and is the key used to perform lookups. (Shared personalization state is stored in a file whose name contains a hash of the request path but no user name.) You must create the ~/App_Data/Personalization_Data directory before using the provider; the provider doesn't attempt to create the directory if it doesn't exist. In addition, the provider must have read/write access to the ~/App_Data/Personalization_Data directory.

Figure 24. TextFilePersonalizationProvider

```
using System;
using System.Configuration.Provider;
```



```

using System.Security.Permissions;
using System.Web;
using System.Web.UI.WebControls.WebParts;
using System.Collections.Specialized;
using System.Security.Cryptography;
using System.Text;
using System.IO;

public class TextFilePersonalizationProvider : PersonalizationProvider
{
    public override string ApplicationName
    {
        get { throw new NotSupportedException(); }
        set { throw new NotSupportedException(); }
    }

    public override void Initialize(string name,
        NameValueCollection config)
    {
        // Verify that config isn't null
        if (config == null)
            throw new ArgumentNullException("config");

        // Assign the provider a default name if it doesn't have one
        if (String.IsNullOrEmpty(name))
            name = "TextFilePersonalizationProvider";

        // Add a default "description" attribute to config if the
        // attribute doesn't exist or is empty
        if (string.IsNullOrEmpty(config["description"]))
        {
            config.Remove("description");
            config.Add("description",
                "Text file personalization provider");
        }

        // Call the base class's Initialize method
        base.Initialize(name, config);

        // Throw an exception if unrecognized attributes remain
        if (config.Count > 0)
        {
            string attr = config.GetKey(0);
            if (!String.IsNullOrEmpty(attr))
                throw new ProviderException

```

```

        ("Unrecognized attribute: " + attr);
    }

    // Make sure we can read and write files in the
    // ~/App_Data/Personalization_Data directory
    FileIOPermission permission = new FileIOPermission
        (FileIOPermissionAccess.AllAccess,
        HttpContext.Current.Server.MapPath
        ("~/App_Data/Personalization_Data"));
    permission.Demand();
}

protected override void LoadPersonalizationBlobs
    (WebPartManager webPartManager, string path, string userName,
    ref byte[] sharedDataBlob, ref byte[] userDataBlob)
{
    // Load shared state
    StreamReader reader1 = null;
    sharedDataBlob = null;

    try
    {
        reader1 = new StreamReader(GetPath(null, path));
        sharedDataBlob =
            Convert.FromBase64String(reader1.ReadLine());
    }
    catch (FileNotFoundException)
    {
        // Not an error if file doesn't exist
    }
    finally
    {
        if (reader1 != null)
            reader1.Close();
    }

    // Load private state if userName holds a user name
    if (!String.IsNullOrEmpty (userName))
    {
        StreamReader reader2 = null;
        userDataBlob = null;

        try
        {
            reader2 = new StreamReader(GetPath(userName, path));

```

```

        userDataBlob =
            Convert.FromBase64String(reader2.ReadLine());
    }
    catch (FileNotFoundException)
    {
        // Not an error if file doesn't exist
    }
    finally
    {
        if (reader2 != null)
            reader2.Close();
    }
}

protected override void ResetPersonalizationBlob
    (WebPartManager webPartManager, string path, string userName)
{
    // Delete the specified personalization file
    try
    {
        File.Delete(GetPath(userName, path));
    }
    catch (FileNotFoundException) {}
}

protected override void SavePersonalizationBlob
    (WebPartManager webPartManager, string path, string userName,
    byte[] dataBlob)
{
    StreamWriter writer = null;

    try
    {
        writer = new StreamWriter(GetPath (userName, path), false);
        writer.WriteLine(Convert.ToBase64String(dataBlob));
    }
    finally
    {
        if (writer != null)
            writer.Close();
    }
}

```

```
public override PersonalizationStateInfoCollection FindState
```

```

        (PersonalizationScope scope, PersonalizationStateQuery query,
        int pageIndex, int pageSize, out int totalRecords)
    {
        throw new NotSupportedException();
    }

    public override int GetCountOfState(PersonalizationScope scope,
        PersonalizationStateQuery query)
    {
        throw new NotSupportedException();
    }

    public override int ResetState(PersonalizationScope scope,
        string[] paths, string[] usernames)
    {
        throw new NotSupportedException();
    }

    public override int ResetUserState(string path,
        DateTime userInactiveSinceDate)
    {
        throw new NotSupportedException();
    }

    private string GetPath(string userName, string path)
    {
        SHA1CryptoServiceProvider sha =
            new SHA1CryptoServiceProvider();
        UnicodeEncoding encoding = new UnicodeEncoding ();
        string hash = Convert.ToBase64String(sha.ComputeHash
            (encoding.GetBytes (path))).Replace ('/', '_');

        if (String.IsNullOrEmpty(userName))
            return HttpContext.Current.Server.MapPath
(
String.Format("~/App_Data/Personalization_Data/{0}_Personalization.txt",
            hash));
        else
        {
            // NOTE: Consider validating the user name here to prevent
            // malicious user names such as "../Foo" from targeting
            // directories other than ~/App_Data/Personalization_Data

            return HttpContext.Current.Server.MapPath
(
String.Format("~/App_Data/Personalization_Data/{0}_{1}_Personalization.tx
t",

```

```
        userName.Replace('\\', '_'), hash));  
    }  
}  
}
```

Figure 25 demonstrates how to make *TextFilePersonalizationProvider* the default Web Parts personalization provider. It assumes that *TextFilePersonalizationProvider* is implemented in an assembly named *CustomProviders*.

Figure 25. Web.config file making TextFilePersonalizationProvider the default Web Parts personalization provider

```
<configuration>  
  <system.web>  
    <webParts>  
      <personalization  
        defaultProvider="AspNetTextFilePersonalizationProvider">  
        <providers>  
          <add name="AspNetTextFilePersonalizationProvider"  
            type="TextFilePersonalizationProvider, CustomProviders"/>  
        </providers>  
      </personalization>  
    </webParts>  
  </configuration>
```

For simplicity, *TextFilePersonalizationProvider* does not honor the *ApplicationName* property. Because *TextFilePersonalizationProvider* stores personalization state in text files in a subdirectory of the application root, all data that it manages is inherently application-scoped. A full-featured profile provider must support *ApplicationName* so Web Parts consumers can choose whether to keep personalization data private or share it with other applications.

Custom Provider-Based Services

ASP.NET 2.0 includes a number of provider-based services for reading, writing, and managing state maintained by applications and by the run-time itself. Developers can write services of their own to augment those provided with the system. And they can make those services provider-based to provide the same degree of flexibility in data storage as the built-in ASP.NET providers.

There are three issues that must be addressed when designing and implementing custom provider-based services:

- How to architect custom provider-based services
- How to expose configuration data for custom provider-based services
- How to load and initialize providers in custom provider-based services

The sections that follow discuss these issues and present code samples to serve as a guide for writing provider-based services of your own.

Architecting Custom Provider-Based Services

A classic example of a custom provider-based service is an image storage and retrieval service. Suppose an application relies heavily on images, and the application's architects would like to be able to target different image repositories by altering the application's configuration settings. Making the image service provider-based would afford them this freedom.

The first step in architecting such a service is to derive a class from *ProviderBase* and add abstract methods that define the calling interface for an image provider, as shown in Figure 26. While you're at it, derive a class from *ProviderCollection* to encapsulate collections of image providers. In Figure 26, that class is called *ImageProviderCollection*.

Figure 26. Abstract base class for image providers

```
public abstract class ImageProvider : ProviderBase
{
    // Properties
    public abstract string ApplicationName { get; set; }
    public abstract bool CanSaveImages { get; }

    // Methods
    public abstract Image RetrieveImage (string id);
    public abstract void SaveImage (string id, Image image);
}

public class ImageProviderCollection : ProviderCollection
{
    public new ImageProvider this[string name]
    {
        get { return (ImageProvider) base[name]; }
    }
}
```

```

public override void Add(ProviderBase provider)
{
    if (provider == null)
        throw new ArgumentNullException("provider");

    if (!(provider is ImageProvider))
        throw new ArgumentException
            ("Invalid provider type", "provider");

    base.Add(provider);
}
}

```

The next step is to build a concrete image provider class by deriving from *ImageProvider*. Figure 27 contains the skeleton for a SQL Server image provider that fetches images from a SQL Server database. *SqlImageProvider* supports image retrieval, but does not support image storage. Note the false return from *CanSaveImages*, and the *SaveImage* implementation that throws a *NotSupportedException*.

Figure 27. SQL Server image provider

```

[SqlClientPermission (SecurityAction.Demand, Unrestricted=true)]
public class SqlImageProvider : ImageProvider
{
    private string _applicationName;
    private string _connectionString;

    public override string ApplicationName
    {
        get { return _applicationName; }
        set { _applicationName = value; }
    }

    public override bool CanSaveImages
    {
        get { return false; }
    }

    public string ConnectionStringName
    {
        get { return _connectionStringName; }
        set { _connectionStringName = value; }
    }

    public override void Initialize (string name,

```

```

NameValueCollection config)
{
    // Verify that config isn't null
    if (config == null)
        throw new ArgumentNullException ("config");

    // Assign the provider a default name if it doesn't have one
    if (String.IsNullOrEmpty (name))
        name = "SqlImageProvider";

    // Add a default "description" attribute to config if the
    // attribute doesn't exist or is empty
    if (string.IsNullOrEmpty (config["description"])) {
        config.Remove ("description");
        config.Add ("description",
            "SQL image provider");
    }

    // Call the base class's Initialize method
    base.Initialize(name, config);

    // Initialize _applicationName
    _applicationName = config["applicationName"];

    if (string.IsNullOrEmpty(_applicationName))
        _applicationName = "/";

    config.Remove["applicationName"];

    // Initialize _connectionString
    string connect = config["connectionStringName"];

    if (String.IsNullOrEmpty (connect))
        throw new ProviderException
            ("Empty or missing connectionStringName");

    config.Remove ("connectionStringName");

    if (WebConfigurationManager.ConnectionStrings[connect] == null)
        throw new ProviderException ("Missing connection string");

    _connectionString = WebConfigurationManager.ConnectionStrings
        [connect].ConnectionString;

    if (String.IsNullOrEmpty (_connectionString))

```



```

        throw new ProviderException ("Empty connection string");

        // Throw an exception if unrecognized attributes remain
        if (config.Count > 0) {
            string attr = config.GetKey (0);
            if (!String.IsNullOrEmpty (attr))
                throw new ProviderException
                    ("Unrecognized attribute: " + attr);
        }
    }

    public override Image RetrieveImage (string id)
    {
        // TODO: Retrieve an image from the database using
        // _connectionString to open a database connection
    }

    public override void SaveImage (string id, Image image)
    {
        throw new NotSupportedException ();
    }
}

```

SqlImageProvider's Initialize method expects to find a configuration attribute named *connectionStringName* identifying a connection string in the <connectionStrings> configuration section. This connection string is used by the *RetrieveImage* method to connect to the database in preparation for retrieving an image. *SqlImageProvider* also accepts an *applicationName* attribute if provided but assigns *ApplicationName* a sensible default if no such attribute is present.

Configuring Custom Provider-Based Services

In order to use a provider-based service, consumers must be able to configure the service, register providers for it, and designate which provider is the default. The Web.config file in Figure 28 registers *SqlImageProvider* as a provider for the image service and makes it the default provider. The next challenge is to provide the infrastructure that allows such configuration directives to work.

Figure 28. Web.config file configuring the image service

```

<configuration >
    ...
    <connectionStrings>
        <add name="ImageServiceConnectionString" connectionString="..." />
    </connectionStrings>
    <system.web>
        <imageService defaultProvider="SqlImageProvider">
            <providers>

```

```

        <add name="SqlImageProvider" type="SqlImageProvider"
            connectionStringName="ImageServiceConnectionString"/>
    </providers>
</imageService>
</system.web>
</configuration>

```

Since `<imageService>` is not a stock configuration section, you must write a custom configuration section that derives from `System.Configuration.ConfigurationSection`. Figure 29 shows how. `ImageServiceSection` derives from `ConfigurationSection` and adds two properties: `Providers` and `DefaultProvider`. The `[ConfigurationProperty]` attributes decorating the property definitions map `ImageServiceSection` properties to `<imageService>` attributes. For example, the `[ConfigurationProperty]` attribute decorating the `DefaultProvider` property tells ASP.NET to initialize `DefaultProvider` with the value of the `<imageService>` element's `defaultProvider` attribute, if present.

Figure 29. Class representing the `<imageService>` configuration section

```

using System;
using System.Configuration;

public class ImageServiceSection : ConfigurationSection
{
    [ConfigurationProperty("providers")]
    public ProviderSettingsCollection Providers
    {
        get { return (ProviderSettingsCollection) base["providers"]; }
    }

    [StringValidator(MinLength = 1)]
    [ConfigurationProperty("defaultProvider",
        DefaultValue = "SqlImageProvider")]
    public string DefaultProvider
    {
        get { return (string) base["defaultProvider"]; }
        set { base["defaultProvider"] = value; }
    }
}

```

Next, you must register the `<imageService>` configuration section and designate `ImageServiceSection` as the handler for it. The `Web.config` file in Figure 30, which is identical to the one in Figure 28 except for the changes highlighted in bold, makes `<imageService>` a valid configuration section and maps it to `ImageServiceSection`. It assumes that `ImageServiceSection` lives in an assembly named `CustomSections`. If you use a different assembly name, you'll need to modify the `<section>` element's `type` attribute accordingly.

Figure 30. Making <imageService> a valid configuration section and registering ImageServiceSections as the configuration section handler

```
<configuration >
  <configSections>
    <sectionGroup name="system.web">
      <section name="imageService"
        type="ImageServiceSection, CustomSections"
        allowDefinition="MachineToApplication"
        restartOnExternalChanges="true" />
    </sectionGroup>
  </configSections>
  <connectionStrings>
    <add name="ImageServiceConnectionString" connectionString="..." />
  </connectionStrings>
  <system.web>
    <imageService defaultProvider="SqlImageProvider">
      <providers>
        <add name="SqlImageProvider" type="SqlImageProvider"
          connectionStringName="ImageServiceConnectionString" />
      </providers>
    </imageService>
  </system.web>
</configuration>
```

Loading and Initializing Custom Providers

The final piece of the puzzle is implementing the image service and writing code that loads and initializes the providers registered in <imageService>'s <providers> element.

Figure 31 contains the source code for a class named *ImageService* that provides a programmatic interface to the image service. Like ASP.NET's *Membership* class, which represents the membership service and contains static methods for performing membership-related tasks, *ImageService* represents the image service and contains static methods for loading and storing images. Those methods, *RetrieveImage* and *SaveImage*, call through to the provider methods of the same name.

Figure 31. ImageService class representing the image service

```
using System;
using System.Drawing;
using System.Configuration;
using System.Configuration.Provider;
using System.Web.Configuration;
using System.Web;

public class ImageService
{
```


Hands-on Custom Providers: The Contoso Times

Included with this whitepaper is a sample ASP.NET 2.0 application named Contoso Times (henceforth referred to as "Contoso") that models a newspaper-style content site and provides a hands-on medium for running several of the sample providers presented in this document. Specifically, Contoso is capable of using the following custom providers:

- *ReadOnlyXmlMembershipProvider*
- *ReadOnlyXmlRoleProvider*
- *SqlSiteMapProvider*
- *TextFileProfileProvider*
- *TextFileWebEventProvider*

The sections that follow provide instructions for getting Contoso up and running first using providers included with ASP.NET 2.0, and then using the custom providers listed above.

Installing Contoso

Perform the following steps to install Contoso on your Web server and configure it to use ASP.NET's *SqlMembershipProvider*, *SqlRoleProvider*, *XmlSiteMapProvider*, *SqlProfileProvider*, and *EventLogWebEventProvider* providers:

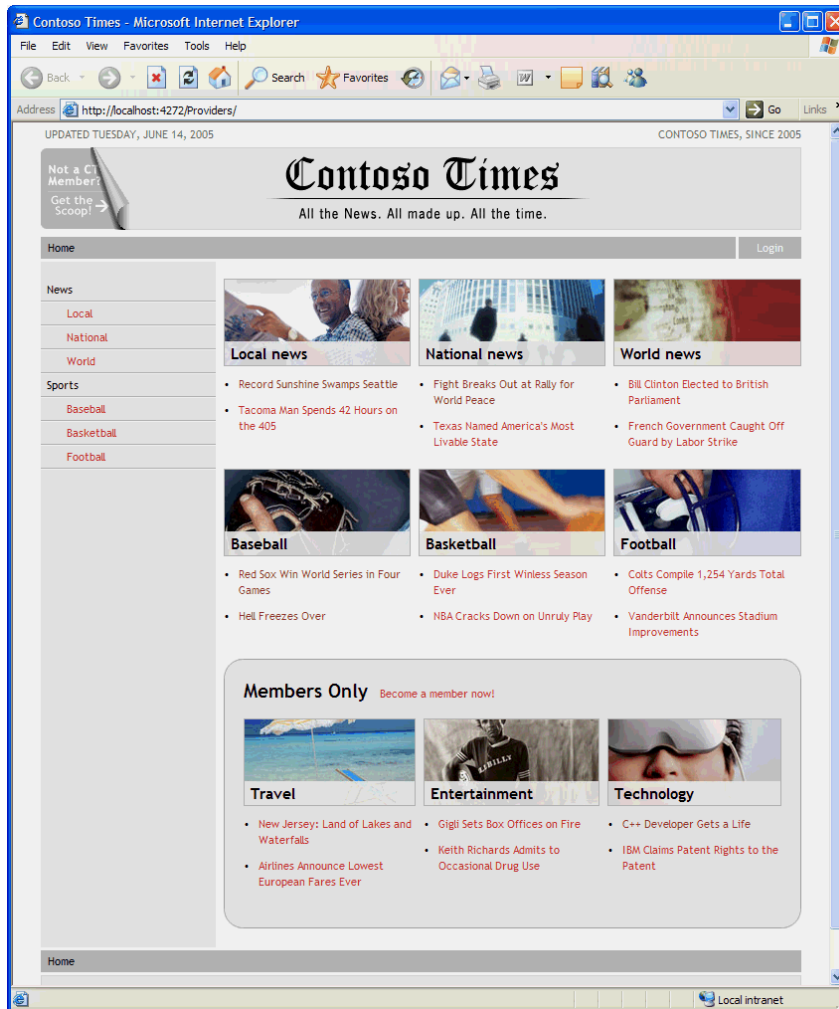
3. Create a directory named Contoso on your Web server.
4. Copy the Contoso files into the Contoso directory. These files include a configuration file named Custom.config that makes the providers listed above the default providers.
5. Ensure that Microsoft SQL Server is installed on your Web server. Then use the installation script in the supplied Contoso.sql file to create the Contoso database.
6. If you want ASP.NET's SQL providers to store state in SQL Server, use the Aspnet_regsql.exe utility that comes with ASP.NET 2.0 to create the Aspnetdb database. If you'd prefer that the SQL providers use SQL Server Express instead, delete the following statements from Web.config and Custom.config:

```
<remove name="LocalSqlServer" />
<add name="LocalSqlServer"
    connectionString="Server=localhost;Integrated Security=True;
    Database=aspnetdb;Persist Security Info=True" />
```

7. Open the Contoso site in Visual Studio 2005 and use the Website->ASP.NET Configuration command to run the Web Site Administration Tool.
8. Click the Web Site Administration Tool's Security tab. On the Security page, use the "Create or Manage Roles" link to create roles named "Members" and "Administrators."

9. Go back to the Security page and click "Create Users." Create a pair of users named Bob and Alice. Add Bob to the "Members" role, and Alice to both the "Members" and "Administrators" roles.
10. Close the Web Site Administration Tool and return to Visual Studio.
11. Use Visual Studio's Debug->Start Without Debugging command to launch Contoso. You should see the home page depicted in Figure 32.

Figure 32. The Contoso Times home page



Running Contoso with Default Providers

Now that the application is installed and configured to use the default providers, you can take it for a test drive. Here's how:

1. Click the "Login" link and log in as Bob. After you're redirected back to the home page, the text to the right of "Members Only" in the Members Only box at the bottom of the page should read "Welcome back, Bob," indicating that you're logged in as Bob and that *SqlMembershipProvider* is providing data to the membership service.

2. Log out by clicking the "Logout" link. Then log in again, but this time log in as "Alice." Observe that two new links appear to the left of the "Logout" link: "Admin" and "Recent Items." When you were logged in as Bob, the "Recent Items" link appeared but the "Admin" link did not. In Contoso, anonymous users see one link, users who belong to the "Members" role but not the "Administrators" role see two links, and users who belong to the "Administrators" role see three links. *SqlRoleProvider* is providing role data to the ASP.NET role manager. The appearing and disappearing links are managed by a *LoginView* control, which uses role memberships and login status to display content differently to different users.
3. The navigation bar on the left side of the home page is a *TreeView* control that obtains its data from a *SiteMapDataSource*. The *SiteMapDataSource*, in turn, gets its data from *XmlSiteMapProvider*, which reads the site map from Web.sitemap. *XmlSiteMapProvider* is currently configured to enable security trimming, which explains why anonymous users see two sets of links in the navigation bar, but logged-in users see three.
4. Open the Windows event log and observe the entry created there when the application started up. Logging occurred because of the <healthMonitoring> element in Web.config that maps to Application Lifetime events to *EventLogWebEventProvider*.
5. While logged in as Bob or Alice, use the links on the home page to view a few articles. Then click the "Recent Items" link to see a list of the articles you've viewed the most recently. Information regarding recently viewed articles is stored in the user profile, which is currently managed by *SqlProfileProvider*. Inspect the <profile> section of Web.config to see how the profile is defined.

Feel free to explore other parts of Contoso as well. For example, clicking the "Admin" link that's visible to administrators takes you to Contoso's secure back-end, which uses *GridView* and *DetailsView* controls to provide robust content-editing capabilities.

Running Contoso with Custom Providers

Now let's reconfigure Contoso to run with the custom providers listed at the beginning of this chapter. First rename Web.config to something else (for example, x-Web.config). Then rename Custom.config to Web.config. Custom.config contains configuration elements replacing *SqlMembershipProvider* with *ReadOnlyXmlMembershipProvider*, *SqlRoleProvider* with *ReadOnlyXmlRoleProvider*, *XmlSiteMapProvider* with *SqlSiteMapProvider*, *SqlProfileProvider* with *TextFileProfileProvider*, and *EventLogWebEventProvider* with *TextFileWebEventProvider*. Membership and role data now come from the file named Users.xml (where Bob and Alice are assigned the password "contoso!"); the site map now comes from the SiteMap table in the SQL Server database that contains Contoso's article content; profile data is now stored in text files in the ~/App_Data/Profile_Data directory; and Application Lifetime Web events are logged in ~/App_Data/Contosolog.txt.

If you exercise the application using the steps prescribed in "Running Contoso with Default Providers," you'll find that it works exactly as before. To the user, the application looks no different. But on the inside, it now relies on custom data sources serviced by custom providers—a great example of the provider model at work, and of the transparency it lends to ASP.NET 2.0 state management.