C0561900x.fm Page 109 Monday, September 15, 2003 11:01 AM

5 Managing Client State

One of the unique opportunities for members of the ASP.NET team is reviewing the architecture design for customers, which is beneficial not only for the customer but also for us. The customer gets validation and critical feedback about their design, and we get first-hand knowledge of usage scenarios that influence decisions for the technologies we build.

A common best practice that we advocate is to factor application and client state management into the solution early in the design. State management requires developers to plan for a Web server farm and to understand and adhere to specific design patterns. For example, out-of-process session state requires that the data stored be serializable by the binary serializer. In total, there are seven different techniques for managing state in ASP.NET, which are described in Table 5-1. In this chapter, we're going to examine managing state that is stored for clients. In Chapter 6, we'll examine application and request state.

Table 5-1 Techniques for Managing State in ASP.NET

Type of State	Applies to	Description
Session	Client	State stored within the application's memory or outside of the application's memory (out of process) and available only to the user who created the data.
ViewState	Client	State stored within embedded <*input type="hidden"*> HTML elements for pages that post back to themselves.
Cookie	Client	State stored in an HTTP cookie on the client's machine. Accessible until the cookie is expired or removed.

Table 5-1 Techniques for Managing State in ASP.NET

Type of State	Applies to	Description
static variables	Application	Static member variables are declared in global.asax or from an *HttpModule* and are available anywhere within the application.
Application	Application	State stored within application's memory and available anywhere within the application.
Cache	Application	State stored within application's memory and available anywhere within the application. Cache additionally supports dependencies and other features to expire items from memory.
HttpContext	Request	State stored within *HttpContext* is accessible only for the duration of the request.

In this chapter, we're going to examine managing state that is stored for clients. In Chapter 6, we'll examine application and request state.

Working with and understanding how client state is used in the application is critical to putting a good design into practice. The most common type of client state is session state. Before we look at how session state is used in ASP.NET, let's step back and review the history of session state, which will provide some context for understanding how it is used.

History of Session State

Microsoft Active Server Pages (ASP) first introduced the server-side programming model of session state. The *Session* object model has properties that are accessible within ASP and are used to store and manage user state data that needs to be persisted between browser requests. The programming model is a simple dictionary-style API in which all access is controlled through a known key. State data, such as the URL of the last page visited or the last 10 search requests, can be persisted using a key and retrieved using a key. For example, the following Visual Basic code works in both ASP and ASP.NET:

```
 ' Set a session value for last page visited
Session("LastPageVisited") = "http://www.asp.net/default.aspx"

' Get a session value for last page visited
lastPageVisited = Session("LastPageVisited")
```

Limitations with ASP Session State

Despite the advantages of ASP, ASP session state has three significant limitations: server affinity in Web farms, apartment model threading, and the HTTP cookie requirement. It was primarily for these three reasons that many developers avoided ASP session state.

Server Affinity in Web Farms

Prior to ASP.NET, server farms that supported session state required smart network hardware to ensure that a client was always redirected to the Web server initiating the first request. In ASP, the session data is stored within the Microsoft Internet Information Services (IIS) process that created it.

ASP runs as an ISAPI extension in the memory space of the IIS Web server. Data stored in memory by ASP is bound to that Web server process and can't be shared between servers. This limitation requires the client always to use the same Web server to guarantee consistency of its session state data. For example, if a user set data in session on server A and used server B on the next request, the session created on A would not be available data.

This connection of the client to a particular server is known as *IP affinity*, and it is a requirement for using session state. (There are other internal redirection solutions, but IP affinity is the most prevalent.) This solution usually requires complex networking hardware to load-balance the traffic. It relies on the client reusing the same IP address on multiple requests and the router maintaining a table of client IPs to server IPs. Routers then intelligently reroute requests based on the mapping of client IPs to server IPs, guaranteeing that the client goes back to the server it started with.

However, even with this IP affinity solution in place, many applications still failed to properly account for large Internet Service Providers (ISPs) such as AOL, MSN, and EarthLink. These three ISPs were the most well known for using reverse proxies for their clients, which meant that on each request the client could come from a different IP address.

Apartment Model Threading

Another limitation of ASP session state is that apartment model–threaded components are the de facto COM threading model. COM servers stored in session state cause multiple simultaneous requests to the same session value to be serialized. This problem is also common with ASP application state.

HTTP Cookies Requirement

ASP session state is also bound to an HTTP cookie. (You'll learn more about cookies later in this chapter.) When a new session starts, ASP assigns an HTTP cookie to the client; the cookie contains a unique key that the client and server share. This key, known as *SessionID*, is a unique value that the server generates and uses to associate session data with the client that posts the key. The IP address can't be used for this since the IP address can potentially change on each request.

Using an HTTP cookie works very well until a client decides not to accept cookies. In many cases this restriction breaks application functionality; since the client can't maintain a *SessionID*, the application can't rely on session state.

ASP.NET Session State

Session state still exists in ASP.NET, partly for backward compatibility, but also as a viable implementation that developers should no longer shy away from. ASP.NET session is free-threaded, but in some cases it can be accessed serially. Session state in ASP.NET still utilizes an HTTP cookie for managing the *SessionID*, however, ASP.NET also supports storing the *SessionID* in the URL if using cookies is not desirable. ASP.NET session state also supports two out-of-process modes to simplify deployment in Web server farms: out-of-process state server (*StateServer*), and out-of-process SQL Server (*SQLServer*).

In-Process Session State

ASP.NET defaults to what is known as in-process (*InProc*) session state. When in this mode, values stored within session state do not require serialization support and are stored within the memory space of the ASP.NET worker process. This behavior is identical to the way ASP stores its session data and has all the same shortcomings and limitations in a Web farm scenario. However, instead of the data being stored in the IIS process, the data is stored in managed memory within the ASP.NET worker process. (When ASP.NET is running on Microsoft Windows 2000, it defaults to the ASP.NET worker process aspnet_wp.exe. However, when ASP.NET is running on Microsoft Windows Server 2003, it will use the new IIS process model w3wp.exe.)

When stored in-process, session state data is lost whenever the process is recycled. In Microsoft Windows Server 2003 running IIS 6, the worker process automatically recycles every 29 hours, which is the default setting and is configurable. However, this does mean that every 29 hours the session data will be lost, whether it is 2:00 AM or 3:00 PM.

InProc is by far the fastest way to use session state. It doesn't support Web farm scenarios (unless you enforce client affinity). However, it also doesn't have the serialization and deserialization overhead associated with out-of-process modes. It's safe to assume that out-of-process session state is 15–30 percent slower (depending upon variables such as network speed and the size of the object or objects being serialized).

C0561900x.fm Page 113 Monday, September 15, 2003 11:01 AM

Important Use in-process session state (the default) if you have only a single server. In IIS 6, either use out-of-process or disable process recycling behavior to avoid data loss.

Code Listing 5-1 shows the configuration settings from machine.config that specify the default settings for session state. Values that apply to *InProc* appear in bold.

Code Listing 5-1 In-Process Session State Configuration

```
<configuration>
    <system.web>
        <sessionState mode="InProc"
                      stateConnectionString="tcpip=127.0.0.1:42424"
                      stateNetworkTimeout="10"
                      sqlConnectionString="..."
                      cookieless="false"
                      timeout="20" />
    </system.web>
</configuration>
```

The timeout value specifies the time, in minutes, after which a session is considered timed out and its values can be removed. Session state uses a sliding expiration: the timeout is reset each time the item is requested. A session could theoretically be kept alive indefinitely if a request was made just once before the value in the timeout is reached. We'll discuss the cookieless option later in the chapter.

The name of the HTTP cookie used to store the *SessionID* in ASP.NET is different from the cookie used to store the *SessionID* in ASP. There is no sharing of session data between ASP and ASP.NET. (See Chapter 11 for more details about migrating session state between ASP and ASP.NET.)

InProc session state allows any data type to be stored, and it participates in the global session events *Session_OnStart*, which is raised when a new session is created; and *Session_OnEnd*, which is raised when a session is abandoned. These events can be programmed in either global.asax or within an HTTP module.

Important Don't use the *Session_End* event; it can be called only for sessions created in the *InProc* mode. The event is not raised for sessions created in one of the out-of-process modes when sessions are abandoned.

Although the *InProc* session is the fastest, in some cases, you might want to trade performance for reliability or ease of management. For example, the out-of-process option is a good choice when you want to support multiple Web servers, or when you want to guarantee that session data can survive the Web server process.

Out-of-Process Session State

ASP.NET session state supports two out-of-process options, state server (*State-Server*) and SQL Server (*SQLServer*). Each has its own configuration settings and idiosyncrasies to contend with, such as managing stored types. The ASP.NET State Service is recommended for medium-size Web applications. For enterprise-size or highly-transactional Web applications, SQL Server is recommended.

Important It's important that the programming model is transparent. For example, we don't have to change how we access or use session state when we change the storage mode.

We recommend *SQLServer* for out-of-process session state because it is just as fast as *StateServer* and SQL Server is excellent at managing data. Furthermore, ASP.NET can communicate with SQL Server natively (meaning internally, using the *System.Data.SqlClient* libraries), and SQL Server can be configured to support data failover scenarios. In cases in which *SQLServer* is not available, *StateServer* works well, but it unfortunately does not support data replication or failover scenarios.

Managing Types for Out-of-Process Modes

If you're using an out-of-process mode, one of your major costs is the serialization and deserialization of items stored. Using an optimized internal method, ASP.NET performs the serialization and deserialization of certain "basic" types, including numeric types of all sizes, such as *Int*, *Byte*, and *Decimal*, as well as several non-numeric types, such as *String*, *DateTime*, *TimeSpan*, *Guid*, *IntPtr*, and *UintPtr*.

If you have a session variable that is not one of the basic types, ASP.NET will serialize and deserialize it using the *BinaryFormatter*, which is relatively slower than the internal method. If you've created a custom class, and you want to store it in session state, you must mark it with the *[Serializable]* meta-data attribute or implement the *ISerializable* interface. (*[Serializable]* is the C# meta-data attribute. <*Serializable()*> is the Microsoft Visual Basic .NET metadata attribute.) The *SerializableAttribute* class is defined in the mscorlib.dll assembly within the *System* namespace. The *ISerializable* interface is defined in the assembly mscorlib.dll and within the *System.Runtime.Serialization* namespace. When a class is marked with the *SerializableAttribute*, all public members will attempt to be serialized. If the class contains references to other objects, those objects must also be marked with the *SerializableAttribute* or implement *ISerializable*. Implementing *ISerializable* gives you more control over how the serialization and deserialization of your class takes place. For more details on the serialization of objects in Visual Basic .NET, visit *http://www.fawcette.com/reports/vsliveor/2002/09_18_02/hollis/.*

For the sake of performance, you're better off storing all session state data using only one of the basic data types (numeric and non-numeric types) listed earlier. For example, if you want to store a name and address in session state, you can store them using two *String* session variables, which is the most efficient method; or you can create a class with two *String* members and store that class object in a session variable, which is more costly.

Important Store only basic data types in session state; avoid storing complex types or custom classes. Storing basic data types will decrease the serialization and deserialization costs associated with out-of-process session as well as reduce the complexity of the system.

Now that you've had an overview of out-of-process session, let's discuss the two out-of-process modes, *StateServer* and *SQLServer*.

StateServer Mode

The *StateServer* out-of-process mode relies on a running Microsoft Windows NT Service as well as changes to the default configuration settings. Code Listing 5-2 shows machine.config with the necessary configuration settings (which appear in boldface) for *StateServer*. Note that the *mode* attribute is set to *StateServer*. The *stateConnectionString* and *stateNetworkTimeout* settings are required values for *StateServer* mode.

Code Listing 5-2 StateServer Session State Configuration

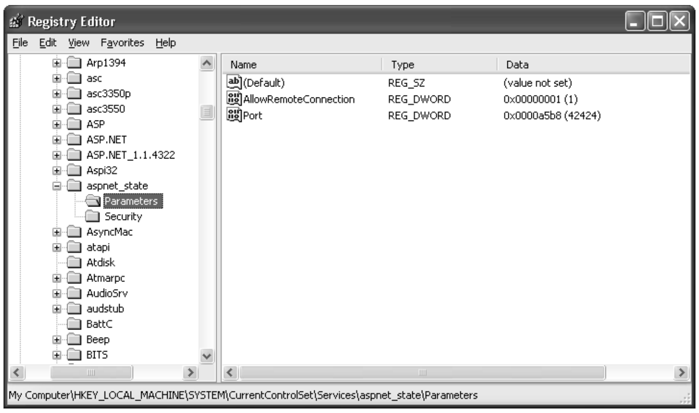
```
<configuration>
    <system.web>
        <sessionState mode="StateServer"
                      stateConnectionString="tcpip=127.0.0.1:42424"
                      stateNetworkTimeout="10"
                      cookieless="false"
                      timeout="20"/>
    </system.web>
</configuration>
```

When ASP.NET is configured to use state server for out-of-process session, it uses a TCP/IP address and port number to send HTTP requests to the state server (which is in fact a lightweight Web server running as a Microsoft Windows Service).

The IP address (in *stateConnectionString*) must be changed to the IP address of the machine running the ASP.NET State Service. The port (the default is 42424), should also be changed unless the state service is running behind a firewall (which it should be). The port number can be configured on the machine running the service by editing the registry and changing the value of the port setting found in the following:

```
HKLM\SYSTEM\CurrentControlSet\Services\aspnet_state\Parameters\
```

As seen in Figure 5-1, the default setting for the port is 0x0000A5B8 in hexadecimal, or 42424 in base 10.



Name	Type	Data
(Default)	REG_SZ	(value not set)
AllowRemoteConnection	REG_DWORD	0x00000001 (1)
Port	REG_DWORD	0x0000a5b8 (42424)

My Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\aspnet_state\Parameters

Figure 5-1 Changing the default port in the registry

Tip If the server running the state service is accessible outside the firewall, the port address of the state service should be changed to a value other than the default. In version 1.1 of ASP.NET, due to security reasons, only local machines can connect to the state server. To allow only non–local host requests in ASP.NET 1.1, open the same registry entry listed earlier for the port setting: *HKLM\SYSTEM\CurrentControlSet\Services\aspnet_state\Parameters\*. Change *AllowRemoteConnection* to 1.

The value of *stateNetworkTimeout* represents the number of seconds that may elapse between the time ASP.NET tries to connect to the state service and the time the request times out. Although the default value for *stateNetworkTimeout* does not need to be changed, you have the option to make the value higher or lower depending upon your requirements.

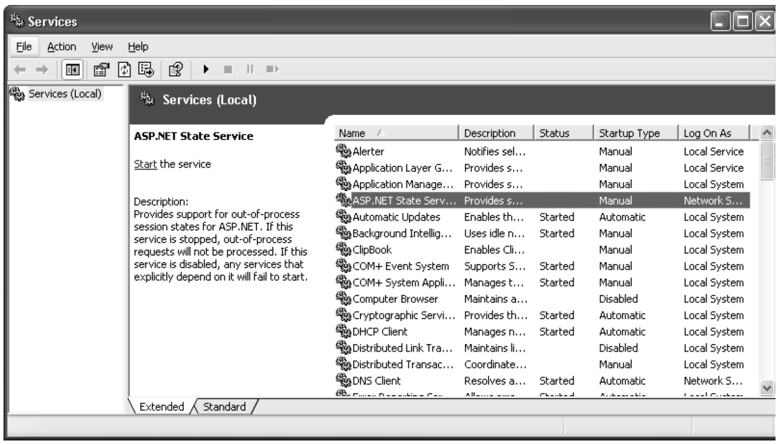
Once the server designated to run the state server has been properly configured, it is simply a matter of starting the Windows service. The service can be started from either the command line or the Microsoft Management Console (MMC) for Services.

Starting the state service from the command line is simple. Navigate to the .NET Framework installation directory. For version 1, this is [system drive]\WINDOWS\Microsoft.NET\Framework\v1.0.3705\. Start the server by executing a net start command:

```
net start aspnet_state
```

After starting the service, you should see the following text: "The ASP.NET State Service service is starting. The ASP.NET State Service service was started successfully."

The second option for starting the state service is through the Services MMC snap-in, which you open by navigating to Start\Administrative Tools\Services. Right-click on the ASP.NET State Service option in the list and select Start to start the service. Once the Services MMC is started, you should see a screen similar to Figure 5-2.



ASP.NET State Service

Start the service

Description:
Provides support for out-of-process session states for ASP.NET. If this service is stopped, out-of-process requests will not be processed. If this service is disabled, any services that explicitly depend on it will fail to start.

Name	Description	Status	Startup Type	Log On As
Alerter	Notifies sel...		Manual	Local Service
Application Layer G...	Provides s...		Manual	Local Service
Application Manage...	Provides s...		Manual	Local System
ASP.NET State Serv...	Provides s...		Manual	Network S...
Automatic Updates	Enables th...	Started	Automatic	Local System
Background Intellig...	Uses idle n...	Started	Manual	Local System
ClipBook	Enables Cli...		Manual	Local System
COM+ Event System	Supports S...	Started	Manual	Local System
COM+ System Appli...	Manages t...	Started	Manual	Local System
Computer Browser	Maintains a...		Disabled	Local System
Cryptographic Servi...	Provides th...	Started	Automatic	Local System
DHCP Client	Manages n...	Started	Automatic	Local System
Distributed Link Tra...	Maintains li...		Disabled	Local System
Distributed Transac...	Coordinate...		Manual	Local System
DNS Client	Resolves a...	Started	Automatic	Network S...

Figure 5-2 Services MMC with ASP.NET state service started

SQL Server Mode

SQL Server is an enterprise-class database solution optimized for managing, storing, and retrieving data quickly and efficiently. It is also capable of replication and clustering. In a clustered environment, SQL Server can be configured to failover. For example, when the clustered production SQL server fails, a backup can take over.

Note that clustered SQL Server scenarios are not supported out of the box for ASP.NET session state. To enable the clustering or replication features of SQL Server, session data must be stored in a non-tempDB table.

Again, you do not need to make any special changes to the code to use SQL Server as the session state store. Code Listing 5-3 shows the necessary configuration file machine.config for SQL Server. (Configuration changes are in boldface code.)

Code Listing 5-3 **SQL Server Session State Configuration**

```
<configuration>
    <system.web>
        <sessionState mode="SQLServer"
                      sqlConnectionString="database=[ServerName];
                                           Trusted_Connection=true"
                      cookieless="false"
                      timeout="20"/>
    </system.web>
</configuration>
```

The *mode* attribute needs to be *SQLServer*, and the *sqlConnectionString* attribute must point to a server running SQL Server that has already been configured for ASP.NET SQL session state.

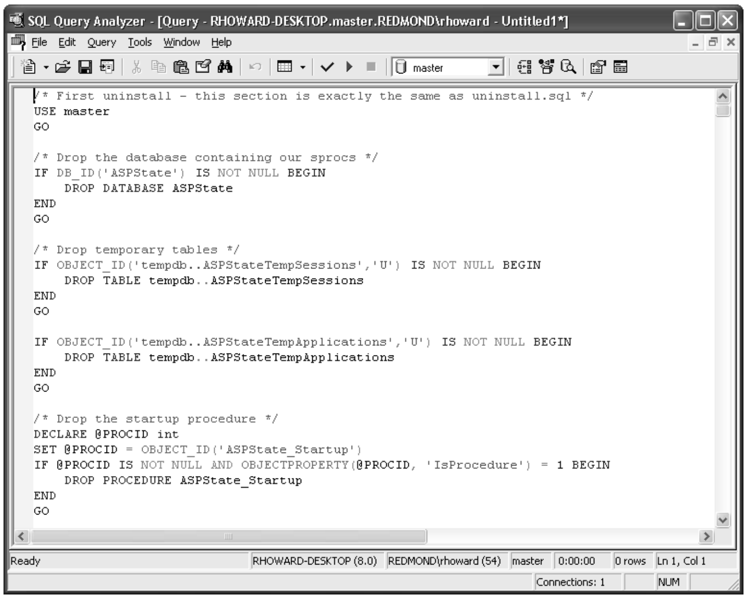
Tip For ASP.NET 1, configure SQL Server for mixed-mode authentication by adding the ASPNET account enabled for the necessary SQL Server permissions (EXECUTE) for ASP.NET session state. (The ASPNET account is the user that the ASP.NET worker process runs as.) For ASP.NET 1.1 running on IIS 6, configure SQL Server for mixed-mode authentication by adding the NT AUTHORITY\NETWORK SERVICE account.

If the account has the necessary permissions, *integrated authentication* should be used. This prevents the need to store a username and password in clear text within the configuration. When integrated authentication is used, ASP.NET accesses SQL Server using the credentials of the Windows user that the worker process runs as. By default, these credentials are ASPNET and NT AUTHORITY\NETWORK SERVICE on Windows Server 2003 running IIS 6.

Important Use integrated authentication rather than store SQL Server credentials within your configuration file. If you decide to use SQL Server user names and passwords, do not use the system administrator (sa) account. Instead use an account that has only the necessary access to the database object required for the operations (for session state, this account is EXECUTE only). If you must use SQL Server credentials, ASP.NET 1.1 supports storing credentials securely.

To configure SQL Server to support ASP.NET session state, either open the InstallSqlState.sql file in isqlw.exe (Microsoft SQL Server Query Analyzer), or use the command-line tool osql.exe. To use SQL Server Query Analyzer, from the Start menu, navigate to \All Programs\Microsoft SQL Server\Query Analyzer. The SQL Query Analyzer application appears in Figure 5-3.

Important Ensure SQL Server Agent is running before running the SQL Scripts. The agent runs a periodic job to purge expired sessions from the database.


SQL Query Analyzer - [Query - RHOWARD-DESKTOP.master.REDMOND\rhoward - Untitled1*]
File Edit Query Tools Window Help
master
Ready
RHOWARD-DESKTOP (8.0)
REDMOND\rhoward (54)
master
0:00:00
0 rows
Ln 1, Col 1
Connections: 1
NUM

```
/* First uninstall - this section is exactly the same as uninstall.sql */
USE master
GO

/* Drop the database containing our sprocs */
IF DB_ID('ASPState') IS NOT NULL BEGIN
    DROP DATABASE ASPState
END
GO

/* Drop temporary tables */
IF OBJECT_ID('tempdb..ASPStateTempSessions','U') IS NOT NULL BEGIN
    DROP TABLE tempdb..ASPStateTempSessions
END
GO

IF OBJECT_ID('tempdb..ASPStateTempApplications','U') IS NOT NULL BEGIN
    DROP TABLE tempdb..ASPStateTempApplications
END
GO

/* Drop the startup procedure */
DECLARE @PROCID int
SET @PROCID = OBJECT_ID('ASPState_Startup')
IF @PROCID IS NOT NULL AND OBJECTPROPERTY(@PROCID, 'IsProcedure') = 1 BEGIN
    DROP PROCEDURE ASPState_Startup
END
GO
```

Figure 5-3 SQL Query Analyzer

If you're running ASP.NET 1, from the toolbar select File | Open [system drive]\WINDOWS\Microsoft.NET\Framework\v1.0.3705\ InstallSqlState.sql. If you're running ASP.NET 1.1, navigate to [system drive]\WINDOWS\Microsoft.NET \Framework\v1.0.4322\ directory and open the same file. Execute the script, either by clicking the Play button or by pressing F5. Using the command-line tool (osql.exe), open a command window and then navigate to \[system drive]\WINDOWS\Microsoft.NET\Framework\[ASP.NET version]\.

If integrated authentication is enabled for SQL Server and the current Windows logged-on user has permissions to SQL Server, type the following: **osql -E < InstallSqlState.sql**.

If SQL Server mixed-mode authentication is enabled and the current logged-on user does not have permissions within SQL Server, specify a user name and password using **osql - U [sql user] - P [password] < InstallSqlState.sql**.

After running the SQL Script, SQL Server is configured to support ASP.NET session state. Two tables are created within the tempdb database: *ASPStateTempApplications* and *ASPStateTempSessions*.

C0561900x.fm Page 121 Monday, September 15, 2003 11:01 AM

Note Why do we use tempdb? We often get asked why we store session data using tempdb vs. a table. The original design goal was to ensure that SQL Server state was fast, and because tempdb is memory-based, SQL Server state would be very fast. However, since SQL Server is so optimized, storing the data in non-temporary tables proved to be nearly as efficient. (SQL Server is super aggressive about keeping frequently accessed data in memory.) In retrospect, using tempdb probably wasn't necessary.

Session Data in a Web Farm

The major benefit of out-of-process session is that it no longer requires client/server affinity. Servers in an ASP.NET Web server farm can be configured to share session data. (However, individual applications cannot share session data because a session is tied to a single application.)

To configure session support in a Web farm, you must take one additional step regardless of whether a session is even used: you must configure the *machineKey* settings. Machine-wide settings are configured in the machine.config file (\[Windows Directory]\Microsoft.NET\Framework\[Versions]\Config\). The *machineKey* settings store the *validationKey* and the *decryptionKey* attribute values, which are used in many scenarios—for example, *ViewState*, Forms authentication, and session—to provide encryption and validation of values sent back and forth from client to server.

By default, the values for the *validationKey* and *decryptionKey* attributes are set to *AutoGenerate*, which enables the server to randomly create the values. Randomly selected values work well in a single server environment; however, in a Web server farm in which there is no guarantee which server will satisfy the client's request, the values for *validationKey* and *decryptionKey* must be precise and predictable.

Optimizing Out-of-Process Session Use

When using out-of-process session, ASP.NET makes two requests to the out-of-process session store for every one page requested (if the request comes in with a session ID). When the request first starts, the session state module connects to the state store, reads the session state data, and marks the session as locked. At this point the page is executed, and the *Session* object is accessible. When the page completes execution, the session state module connects to the state store, writes the session state data if it changed, and unlocks the session data.

The locking mechanism implemented is a reader-writer lock using the ReaderWriterLock class. Multiple requests can read data simultaneously when the session isn't locked. If the session is locked, read requests are blocked until the write lock is released. This locking strategy guarantees that the *Session* object with which the user is interacting is always an accurate reflection of the data. If you attempt to build a site that uses frames, and each page within a frame requires session state, the pages will execute serially. You can configure session state to be read-only on a page-by-page or even application basis. Configuring a page to use session data in a read-only manner allows the page to be requested without locking the session data and prevents serialized access.

If session is configured as read-only, in out-of-process mode, the session module does not need to go back to the session store to release the lock. Multiple requests can read session data simultaneously without serialized access, which yields better throughput. To configure the read-only option on a page-by-page basis, simply set the page-level directive *EnableSessionState*:

```
<%@ Page EnableSessionState="ReadOnly" %>
```

Another option is to configure session to *enableSessionState="false"* as the default setting (you can change this in web.config or machine.config) and use *EnableSessionState="ReadOnly"* or *EnableSessionState="true"* at the page level. Code Listing 5-4 shows the code for disabling session state.

Code Listing 5-4 Disabling Session State

```
<configuration>
    <system.web>
        <pages enableSessionState="false" />
    </system.web>
</configuration>
```

Important For out-of-process session, set session state to *enableSessionState="false"* within the configuration file and set the *EnableSessionState* page directives to either *true* or *ReadOnly* based on what behavior is needed. Note that the length of the session will still be reset (even when set to *false*).

When you apply this strategy for optimizing out-of-process session, you get fewer requests to the out-of-process session store, which increases the scalability and throughput of the site.

C0561900x.fm Page 123 Monday, September 15, 2003 11:01 AM

Cookieless Session

Session state, to HTTP purists, is a frowned upon but necessary feature for building real-world Web applications. Session state was designed to work around the limitations of the stateless nature of HTTP. To do so, the browser and the server must share a common piece of data: the *SessionID*. This shared value must be stored somewhere—we're certainly not going to ask the user to re-enter an ID value upon each request to the server! To solve this problem, we take advantage of another HTTP feature known as a cookie, which you learned about briefly earlier in the chapter.

A cookie is a highly contentious, much debated feature supported by all browsers that allows the server to store a small amount of data, private to the server, on the client. Upon each client request to the server, the browser sends along any cookie data belonging to that server.

For both ASP and ASP.NET, the *SessionID* is stored within a cookie. When the client makes requests to the server, the client presents the cookie, giving ASP.NET the opportunity to fetch any associated session data belonging to the presented *SessionID*.

Important Using the *SessionID* as a key for user data is not recommended. The *SessionID* is randomly generated, and session data—as well as session IDs—do expire. Additionally, although a *SessionID* might be generated on each request, a *SessionID* is set only when a *Session* value is set server side. This means that if no session values are set server side, new *SessionID*s are issued on each request.

Storing the *SessionID* in a cookie works very well except when the client chooses not to accept cookies. (By default, cookies are accepted, and the user has to explicitly disable cookie support to avoid using them.) When cookies are not supported, ASP.NET provides a cookieless option in which the *SessionID* is stored within the URL instead of an HTTP cookie.

An ASP.NET Web application cannot be configured to support both cookie and cookieless *SessionID* storage; that is, the application cannot dynamically choose whether to use cookies. This can be seen as advantageous because designing an application to accommodate various cookie scenarios can be very difficult.

When building applications to take advantage of a cookieless session, you must carefully design navigation in the user interface. Any links within the site that are not relative (those starting with *http://*) will cause the user to lose her session when clicked. For relative URLs (for example, /MyStore/default.aspx), the embedded *SessionID* is automatically added by ASP.NET when generating the page output.

> **Tip** If you have to develop an application that supports both cookie and cookieless sessions, your best strategy is to write an HTTP module to redirect the browser to the appropriate application or server for the supported browser feature, for example, configure a dedicated application that is used for cookieless sessions.

Using *ViewState* to Store State in the Page Output

As stated earlier in the section on session state, Web application communication takes place over HTTP, a stateless protocol. The ASP.NET session state feature circumvents the stateless nature of HTTP by storing its *SessionID* in an HTTP cookie or embedding it within the URL of the page. This shared session key is then used to associate data stored on the server with the browser making the request.

In some cases, it isn't necessary or desirable to require session state, and a common technique that many developers have used in the past is to store data in hidden form fields like this: *<input type="hidden" value="some value here"/>*. When the client submits the page and causes either an HTTP POST or GET request to the server, this data, along with other *input* form data, is sent to the server in either the POST body or the query string.

ASP.NET has taken this concept of hidden input form fields and utilized them for maintaining state for pages that participate in postback. (All ASP.NET pages that use *<form runat="server"/>* send the contents of the *form* back to the same page. Additionally, the *action* attribute of *form* is ignored when the *form* is marked with *runat="server"*.) This feature is known as view state; data is stored in a special hidden *<input type="hidden" name="__VIEWSTATE"/>* form element. The data stored in the *value* attribute of the *__VIEWSTATE* form element consists of a base-64 encoded string that contains all the serialized *ViewState* data for the current page plus a MAC (Message Authentication Code). When the page is posted back to the server, the ASP.NET page framework deserializes the data in *__VIEWSTATE* and automatically repopulates the *ViewState* state bag. Thus data added to the view state is available when the page is posted back again. View state is very useful for building complex server controls since data not usually sent as a *form* element can be stored in the view state and retrieved when the page is posted back. (For more details on the inner workings of view state, I highly recommend you take a look at Chapter 7 of *Developing Microsoft ASP.NET Server Controls and Components,* written by Nikhil Kothari and Vandana Datye and published by Microsoft Press.)

Note A MAC is a key-dependent, one-way hash. A MAC is used to verify *ViewState* data by recomputing the MAC on post back and comparing it to the MAC stored in *__VIEWSTATE*. If the MACs match, the data in *__VIEWSTATE* is valid. If they do not match, the *ViewState* data is invalid and an exception is thrown.

Programming *ViewState*

ViewState data is accessible in much the same way that *Session* data is; both use a key to set or retrieve data. However, unlike *Session*, *ViewState* data is available only in pages that utilize *<form runat="server"/>*, which causes the page to perform a postback. The data types that can be stored in *ViewState* are limited to *Int32*, *Boolean*, *String*, *Unit*, and *Color*. Data types other than these incur significant overhead and must first either be converted to a string or be serialized using the same binary serializer used by session state.

The view state can be extremely useful in cases in which there is a costly piece of data to fetch that is necessary for the duration of the page (where duration of the page is equal to the first request and all postbacks). A great example of this is in the source code for the ASP.NET Forums (*www.asp.net/forums*). One of the controls used frequently within the Forums is a server control used for paging data. This server control (Paging.cs) allows the user to page through multiple records of data in SQL Server (as opposed to paging through the data in ASP.NET). One of the paging control's tasks is to keep track of the total number of available records. Using this number and the requested page size, the control can calculate the total number of pages available. The total records available are not computed on each request—instead, this data is fetched once and stored in view state, alleviating the stress on the server from making multiple requests to the database for the same information.

Below is a code snippet from the Paging.cs file that demonstrates this technique—the full source is available as part of the ASP.NET Forums downloadable from *www.asp.net*.

```
/// <summary>
/// TotalRecords available
/// </summary>
public int TotalRecords {
    get {
        // The total records available is stuffed into
        // ViewState so that we don't pay the lookup cost
        // across postbacks.
        if (ViewState["totalRecords"] == null)
            return defaultTotalRecords; // 0

        return Convert.ToInt32(ViewState["totalRecords"].ToString());
    }
    set {
    // Recalculate the total number of pages in case page size changed
    TotalPages = CalculateTotalPages(value, PageSize);

    // set the ViewState
    ViewState["totalRecords"] = value;
  }
}
```

ViewState's Liability

Using *ViewState* does have a liability: it increases the total size of the page that must be created. Although it doesn't affect the UI generated by the page, the HTML payload can dramatically increase depending upon how much view state is used by the page. We recommend that you disable *ViewState* for page or controls that don't require it.

> **Tip** The view state can be disabled in a page by using *<%@ Page EnableViewState="false" %>*, or in a control by specifying *Page.EnableViewState="false"* on the server control.

Without disabling view state, the following code sample demonstrates using the *DataGrid* server control bound to a *DataSet* to serialize the XML document BookData.xml to base64 in an XML attribute value:

Code Listing 5-5 Serializetobase64.aspx

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Data" %>

<script runat="server">

    public void Page_Load (Object sender, EventArgs e) {

      // Load some data
      DataSet ds = new DataSet();
      ds.ReadXml(Server.MapPath("BookData.xml"));

      // Now databind to the datagrid
      DataGrid1.DataSource = ds;
      DataGrid1.DataBind();

    }
</script>
<form runat="server">
<asp:Button runat="server" Text="PostBack" />
</form>
<asp:DataGrid id="DataGrid1" runat="server" />
```

When this page is requested and the HTML source is viewed, the value for __*VIEWSTATE* contains this:

```
<input type="hidden" name="__VIEWSTATE"
value="dDwxMzg3MzYyMzg7dDw7bDxpPDI+Oz47bDx0PEAwPHA8cDxsPERhdGFLZXlzO18hSXRlbUNv
dW5[…30 lines removed…]z47dDxwPHA8bDxUZXh0Oz47bDxQYXJpczs+Pjs+Ozs+O3Q8cDxwPGw8V
GV4dDs+O2w8Jm5ic3BcOzs+Pjs+Ozs+O3Q8cDxwPGw8VGV4dDs+O2w8RnJhbmNlOz4+Oz47Oz47Pj47
Pj47Pj47Pj47Phd0PzYb9Lz7N2ZqMReiGAMMnwyz" />
```

In this case, we're not using a view state, so we should disable it by adding an *EnableViewState* attribute to *DataGrid*:

```
<asp:DataGrid id="DataGrid1" runat="server" EnableViewState="false" />
```

Now when this page is requested, the value for __*VIEWSTATE* is more reasonable:

```
<input type="hidden" name="__VIEWSTATE"
value="dDwxMzg3MzYyMzg7Oz6TQ21xg8KTWseIQ341mOOdKXguIw==" />
```

View state is a powerful technique for managing state for pages that participate in post back. However, you need to be aware that view state has an associated cost that can easily increase the size of your page output, as demonstrated in this code sample.

Using Cookies for Client State Management

The last technique that we'll examine for managing client state is the cookie, which you learned about briefly earlier in the chapter. Unbeknownst to many Web developers, cookies are not an approved standard, although all major browsers support them and all Web application development technologies use them. To review, cookies are small state bags that belong to a particular domain and are stored on the client's machine rather than on the server. ASP.NET utilizes cookies for two tasks:

- **Session state** The associated cookie is .ASPXSession. The cookie stores the *SessionID* used to associate the request with its session data.
- **Forms authentication** The associated cookie is .ASPXAUTH. The cookie stores encrypted credentials. Credentials can be decrypted and the user re-authenticated.

You can view all the cookies on your system by opening Microsoft Internet Explorer and selecting Tools\Internet Options to open the Internet Options dialog box. Click the Settings button to open the Settings dialog box. Click the View Files button to open Explorer and access your temporary Internet files directory. You can then sort by type *Text Document* or by items named Cookie. As you can see, you've got lots of cookies!

Cookies are actually a great way to manage state if you can guarantee that your clients use them. They can store multiple name/value combinations as long as the value is of type *string* (or can be converted to string). The only limitation with cookies is the amount of data that can be stored; most browsers support a maximum cookie size of 4 KB (4096 bytes, to be more precise).

Working with cookies in ASP.NET is simple. We use them in many of our sample applications, including the *www.asp.net* Web site, in which we store the roles that a user belongs to. Rather than fetching the user roles on each request from the database, we fetch the user roles only if a specific *UserRoles* cookie doesn't exist. We then create the *UserRoles* cookie and add the roles the user belongs to. On subsequent requests, we can simply open the *UserRoles* cookie, extract the roles, and add them to the roles the current user belongs to. The following code fragment illustrates this.

```
//*********************************************************************
//
// Application_AuthenticateRequest Event
//
// If the client is authenticated with the application, then determine
// which security roles he/she belongs to and replace the "User" intrinsic
// with a custom IPrincipal security object that permits "User.IsInRole"
// role checks within the application
//
// Roles are cached in the browser in an in-memory encrypted cookie.
// If the cookie doesn't exist yet for this session, create it.
//
//*******************************************************************
void Application_AuthenticateRequest(Object sender, EventArgs e) {
    String[] roles = null;

    if (Request.IsAuthenticated == true) {
        // Create roles cookie if it doesn't exist yet for this session.
        if ((Request.Cookies["userroles"] == null) ||
            (Request.Cookies["userroles"].Value == "")) {

            // Get roles from UserRoles table, and add to cookie
            roles = UserRoles.GetUserRoles(User.Identity.Name);

            CreateRolesCookie(roles);

        } else {

            // Get roles from roles cookie
            FormsAuthenticationTicket ticket =
                FormsAuthentication.Decrypt(
                    Context.Request.Cookies["userroles"].Value);

            // Ensure the user logged in and the user
            // the cookie was issued to are the same
            if (ticket.Name != Context.User.Identity.Name) {

            // Get roles from UserRoles table, and add to cookie
            roles = UserRoles.GetUserRoles(User.Identity.Name);

            CreateRolesCookie(roles);
        } else {
            // convert the string representation of the role
            // data into a string array
            ArrayList userRoles = new ArrayList();

            foreach (String role in
                    ticket.UserData.Split( new char[] {';'} )) {
                        userRoles.Add(role);
            }

            roles = (String[]) userRoles.ToArray(typeof(String));
        }
    }
    // Add our own custom principal to the request
    // containing the roles in the auth ticket
    Context.User = new GenericPrincipal(Context.User.Identity, roles);
    }
}


//*********************************************************************
//
// CreateRolesCookie
//
// Used to create the cookie that store the roles for the current
// user.
//
//*********************************************************************
private void CreateRolesCookie(string[] roles) {

    // Create a string to persist the roles
    String roleStr = "";
    foreach (String role in roles) {
        roleStr += role;
        roleStr += ";";
    }

    // Create a cookie authentication ticket.
    FormsAuthenticationTicket ticket = new FormsAuthenticationTicket(
                1,                              // version
                Context.User.Identity.Name,     // user name
                DateTime.Now,                   // issue time
                DateTime.Now.AddHours(1),       // expires every hour
                false,                          // don't persist cookie
                roleStr                         // roles
    );

    // Encrypt the ticket
    String cookieStr = FormsAuthentication.Encrypt(ticket);

    // Send the cookie to the client
    Response.Cookies["userroles"].Value = cookieStr;
    Response.Cookies["userroles"].Path = "/";
    Response.Cookies["userroles"].Expires = DateTime.Now.AddMinutes(5);

}
```

The first method, *Application_AuthenticateRequest*, is an event delegate that gets called when ASP.NET is ready to authenticate the request. Within this method, we check to see whether we have a cookie named *UserRoles* and whether it has a value.

C0561900x.fm Page 131 Monday, September 15, 2003 11:01 AM

If the cookie isn't found, we load the roles for the user and then call the *CreateRolesCookie* method, passing in a *string[]* of role names. Within *CreateRolesCookie*, we simply format the *string[]* into a semicolon-delimited string, encrypt it using APIs from Forms Authentication, and then store the encrypted data in the *UserRoles* cookie.

If the *UserRoles* cookie is found, we first decrypt the value of the cookie, ensure that the user the cookie belongs to is the same user that is currently logged in, split the roles using a semicolon as the delimiter, and finally create a new *GenericPrinicpal* (authenticated identity), passing in the roles as one of the arguments.

Obviously this code works on each request, but it doesn't go to the database on each request to refetch the roles. The *www.asp.net* site averages about 85,000 unique users per day. If each user made an average of 30 requests, by using cookies for storing the user roles, we would eliminate at least 2,465,000 requests to the database!

Summary

In this chapter, we examined three techniques for managing client state: session state, view state, and cookies. Session state is a powerful tool that you can use to store data associated with individual users. Session state requires a storage location for this user data. By default, user data is stored in the current process's memory space, as was the case with ASP. However, ASP.NET introduces a powerful new concept known as out-of-process session state that allows for all servers in a farm to use a common store such as SQL Server. Additionally, session state requires the use of a session ID. The session ID is a token shared between the client and the server that is used to identify the client's session to the server on subsequent requests.

View state solves a problem that many developers have solved in the past through custom code. Unlike session state, view state never times out, but it is limited to the postback life cycle of a page—once you navigate away from the page, you lose your view state. View state allows for simple types to be stored in the hidden input in the HTML of the page, but caution should be used when using view state because the size of *ViewState* affects the size of the page the client must download.

Cookies can be used to store client state independent of the server. However, cookies are not an approved standard and have data storage limitations.

Choosing the appropriate client state management technique depends on what you need to accomplish within your application. ASP.NET provides you with easy-to-use APIs for working with the three client state management techniques covered in this chapter. In Chapter 6, we'll examine another type of state: application state.