Performance Tuning and Optimizing ASP.NET Applications

JEFFREY HASAN WITH KENNETH TU

Apress™

Performance Tuning and Optimizing ASP.NET Applications

ISBN (pbk): 1-59059-072-4

Printed and bound in the United States of America 12345678910

Technical Reviewer: Michael Machowski
Editorial Directors: Dan Appleman, Gary Cornell, Simon Hayes, Karen Watterson, John Zukowski
Managing Editor: Grace Wong
Project Manager: Sofia Marchant
Copy Editor: Kim Wimpsett
Compositor: Impressions Book and Journal Services, Inc.
Indexer: Rebecca Plunkett
Cover Designer: Kurt Krames
Production Manager: Kari Brooks
Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com.

Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 9th Street, Suite 219, Berkeley, CA 94710.

Phone 510-549-5930, fax: 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

CHAPTER 4

Optimizing Application and Session State Management

ASP.NET ***STATE MANAGEMENT*** is the ability of a Web application to persist information, both at the application and session levels. There are two main types of state in a Web application:

- **Application state**: This is information that applies to all clients of the Web application. This information is shared by, and managed for, multiple clients.

- **Session state**: This is information that applies to a specific client session. You manage this information for individual clients.

Application and session state management are important for personalizing Web applications and for persisting information that is widely used across a Web application. ASP.NET expands the range of options available for managing application and session state. In particular, it overcomes previous limitations of classic ASP for managing state across Web farms.

You continue to manage application state using the Application object (technically, the `HttpApplicationState` class). In addition, ASP.NET provides the `Cache` class, which offers more granular control over managing application data.

Session state management has been greatly expanded compared to classic ASP; you are no longer confined to just using in-process Session objects. In classic ASP, developers liked Session objects for their ease of use but disliked them for negatively impacting application scalability and performance. ASP.NET faces similar challenges; however, the actual performance implications may surprise you. As we discuss in this chapter, Session objects are not necessarily performance killers. On the contrary, when used correctly, they can greatly improve the performance of your application and minimally impact scalability. Most books simply make a cursory reference to the "performance impact" of using Session objects. We, on the other hand, take the discussion a step further by running performance tests for each session mode and examining the numbers.

Overview of Session Management

ASP.NET provides a wide range of session state management capabilities, which allows for the dedicated storage and retrieval of user-specific information. Web applications are built on Hypertext Transfer Protocol (HTTP), which is inherently a stateless protocol. Web servers cannot typically recognize when a set of requests originates from a single user. (The exception would be if the user has a unique Internet Protocol that the Web application can reference from the HTTP Headers collection). This limitation makes it challenging to tailor a Web application experience to a single user. Personalized application sessions can usually only occur if the Web server retains session-specific information between requests. This process typically requires infrastructure support from the Web server and participation from the client. The server and the client establish a unique reference number for the session, or *session ID*, which is typically stored in a cookie on the client machine. Cookies alone may also enable session management because they allow session-specific information to be retained in a text file on the client machine. Cookies pass between the client and server during requests, which enables the server to customize a response based on client-specific information.

But cookies will only get you so far because they are limited both in size and in the complexity of information they can store. Cookies are limited to 4KB in size and are only capable of storing strings. You must store complex information, such as an array or an ADO.NET DataSet, in more sophisticated ways on the server side.

NOTE *Some developers prefer to create custom session management code rather than using the Session object. One approach is to persist session information in hidden fields or in the Uniform Resource Locator (URL) querystring. An alternate approach is to store session information in a back-end database and key the records using the session ID key that is automatically generated when you enable session state management. In these cases, neither the Web server nor the client requires direct session management support.*

There is actually a dual challenge to retaining and providing session-specific information. On the one hand, there is the challenge of how to retain and procure the information. And on the other hand, there is the challenge of how to do it *quickly*. Users will not appreciate their richly tailored individual experience if it requires them to wait for long periods of time between requests to the Web application.

Managing Session State in Classic ASP

Session state management was available in classic ASP, but it was much maligned for four important reasons:

Performance: Classic ASP provides in-process session management only. All session-specific information has to be stored in the Web server's memory heap, which becomes a drain on available resources as the number of sessions increases. This is especially true if the session-specific information is large or takes time to serialize. Session management in classic ASP is widely considered to have unacceptable impacts on application scalability.

Reliability: In-process session information will not persist if the Web server process ends unexpectedly or the connection between the client and the server is dropped.

Web farms: The in-process nature of classic ASP session management means that only one server at a time can retain session information. This limitation makes classic ASP session management incompatible with Web farms because this architecture routes a single user's requests to the most available server in the farm. Session information will get lost unless the user is consistently routed to the same machine. In recent years this has not been as much of an issue because modern load-balancing routers have the ability to consistently route a user to the same machine for every request. However, the user is still exposed to the risk of losing their session information if their specific server crashes between requests and they are forced to route to a different machine.

Cookie support: Classic ASP requires cookies for managing sessions, which is a problem for the minority of clients that do not enable cookies. Although this only affects a small number of clients, the greater problem is the lack of any alternative to using cookies.

Classic ASP developers use their skills to overcome these limitations as best they can. An especially popular approach is to retain all session information in a dedicated database, using the session ID as a primary key for referencing the information. This approach is not without its performance implications because database calls are slower than pulling data from memory. But the performance hit is worthwhile given that data is guaranteed to be available, especially from clustered SQL Servers, which are highly available. Of course, database server crashes will interrupt access to data. However, developers can greatly reduce the likelihood of crashes through a combination of reliable database software (SQL Server!) and fail-over measures, such as clustering database servers.

Managing ASP.NET Session State

ASP.NET addresses the limitations of classic ASP in the following ways:

Process independence: ASP.NET continues to support traditional in-process session state storage, which stores session values in the same process as the ASP.NET worker process. However, ASP.NET also provides two modes for storing session state out-of-process. The StateServer mode stores session state in a separate thread that is managed by a separate NT service. The SQLServer mode stores session state in a dedicated SQL Server database. Process independence improves the reliability and durability of session state information by decoupling it from the ASP.NET application's worker process. If this process crashes, then session state information does not need to be lost.

Cookieless support: ASP.NET does not require cookies for managing sessions. Cookie-based session state management continues to be the default, where the session ID is stored in a cookie on the client machine. In cookieless mode, ASP.NET automatically appends the session ID to all URLs. The drawback to this approach is that the Web application must contain relative links, with no absolute links. Otherwise, the session ID will fail to append to the URL, and the session association will be lost.

Web farms: ASP.NET provides the StateServer and SQLServer session state modes, which decouples session state management from an application's ASP.NET worker process. Multiple computers in a Web farm can manage session state using a centralized StateServer thread or a centralized SQL Server database. These session state modes are easy to configure and require no special coding.

In the "Understanding Session State Modes" section, we examine the various ASP.NET session state modes in detail. In addition, we discuss the performance implications of each mode. Clearly, there are performance implications when you require a server to manage session information. This task is an additional burden on the server and requires it to allocate valuable resources, both in terms of memory and processor utilization. The key is to pick a session state mode that provides the best session management for your application with the lowest overhead. That is, you must pick a mode that offers the optimal balance between performance and reliability for your particular state management requirements.

Configuring and Using ASP.NET Session State

Session state is enabled by default for a new ASP.NET project and is set to InProc (in-process) mode (described next). You configure session state in the `Machine.config` and `Web.config` files using the `<sessionState>` element:

```
<sessionState
         mode="Off|InProc|StateServer|SQLServer"
         stateConnectionString="tcpip=127.0.0.1:42424"
         sqlConnectionString="server= machineName\sqlServer;uid=sa;pwd=;"
         cookieless="true|false"
         timeout="20"
/>
```

In this example, the pipe symbol (|) indicates a mutually exclusive choice of options, and the connection string and timeout properties have default examples. Note that the `Web.config` file is case sensitive, so make sure you type all mode values using the correct case. "InProc" is a valid mode value, but "Inproc" is not. There is no special user interface (UI) for the `Web.config` file; otherwise this detail would be taken care of for you.

The minimum required `<sessionState>` attributes are `mode`, `cookieless`, and `timeout` (set in minutes). The `stateConnectionString` attribute is only required when the session mode is StateServer. Similarly, the `sqlConnectionString` attribute is only required when the session mode is SQLServer.

You can further configure session state at the individual page level using the `EnableSessionState` attribute of the `@ Page` directive:

```
<%@ Page EnableSessionState="True|False|ReadOnly" %>
```

If the attribute value is "True," then either a new session will be created or an existing session will be used. If the value is "False," then no new session will be created and no session values may be accessed on the page. If the value is "ReadOnly," then session values may be retrieved, but not modified.

Understanding Session State Modes

ASP.NET provides four modes for managing session state on the server:

- **Off**: Session state is disabled.
- **InProc**: Session state is stored and managed in-process, on the same thread as the ASP.NET application.

- **StateServer**: Session state is stored out-of-process and is managed by an NT Service called *ASP.NET State Service*.

- **SQLServer**: Session state is stored and managed by a SQL Server database called *ASPState*. A batch file that ships with .NET, called `InstallSqlState.sql`, creates this database.

Let's discuss each of the modes in turn, excluding the Off mode, which warrants no further explanation.

Using InProc Session State

The InProc mode is the default mode for session state and is equivalent to what classic ASP provides. This mode is the easiest to configure and only requires you to update the `Web.config` file:

```
<sessionState mode="InProc" cookieless="false" timeout="20"  />
```

The advantages of the InProc mode are as follows:

- It is easy to configure.

- It is the fastest mode available because session items are stored in the same thread as the ASP.NET application.

The disadvantages of the InProc mode are as follows:

- Session items are available on a single server only; you cannot share them across multiple Web servers.

- Session items are not durable. You will lose them if the server crashes or is restarted.

- Session items use up server memory and may negatively impact the scalability of the application.

The InProc mode is an excellent choice if the session items are modest in size and you are not concerned about potentially losing session items and having to re-create them. E-commerce applications, for example, cannot afford to lose session data. However, other applications can use Session objects to reduce redundant database calls that would return duplicate information. These applications can easily re-create session items if they are lost.

Using StateServer Session State

The StateServer mode provides out-of-process session storage and management. This mode stores session items in a dedicated process managed by an NT service called *ASP.NET State Service*. You configure the StateServer mode in a two-step process. First, you update the `Web.config` file:

```
<sessionState mode="StateServer" stateConnectionString="tcpip=127.0.0.1:42424"
        cookieless="false" timeout="20" />
```

Next, you have to start the ASP.NET State Service because its default startup type is manual. Open the MMC snap-in from the Windows Start menu button by selecting Start ➢ Programs ➢ Administrative Tools ➢ Services.

Highlight the ASP.NET State Service entry, as shown in Figure 4-1, and click the Start button. Alternatively, you can right-click the entry and select Start from the pop-up menu.

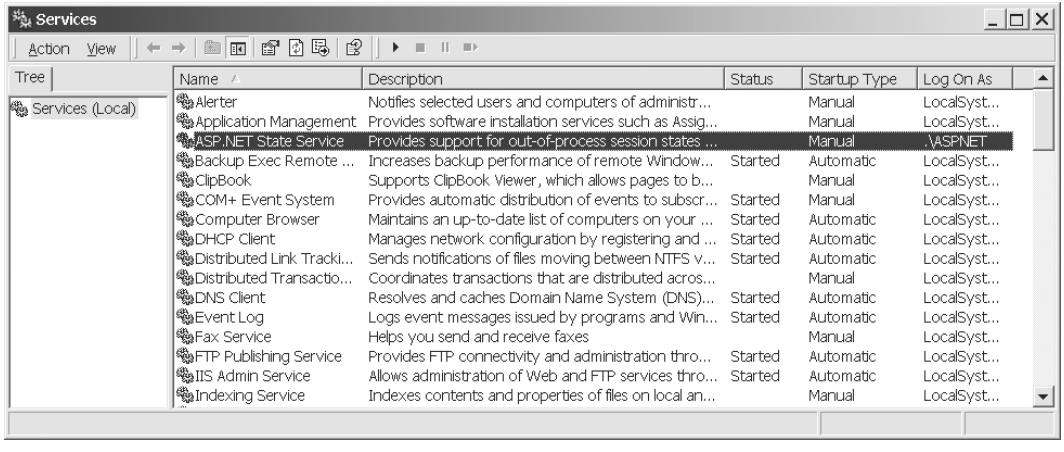

Figure 4-1. The ASP.NET State Service

If you forget to start the service but you update the `Web.config` file, then your application will throw the following error:

```
System.Web.HttpException: Unable to make the session state request to the
session state server. Make sure that the ASP.NET State service is started
and that the client and server ports are the same.
```

The advantages of the StateServer mode are as follows:

- Session storage is out-of-process, so it does not directly impact the scalability of the ASP.NET application.

- You can share session items across multiple Web servers.

The disadvantages of the StateServer mode are as follows:

- There is a high performance cost of marshaling session items across processes, even within the same server.

- There is a high performance cost of marshaling session items between servers if you have multiple servers accessing the same state service.

- Session items are not durable. You will lose them if the dedicated process crashes or is restarted.

- Session items must support binary serialization to work with the StateServer mode. Popular objects such as the DataSet object do support binary serialization. However, others such as the equally useful DataView object do not.

The StateServer mode is often the worst choice you can make for managing session state. The cost of marshaling data across process boundaries is high, even if the size of the data is small. If you must access Session data from multiple servers, then SQLServer mode is often a better choice.

In ASP.NET 1.1, by default, only the local machine can connect to its ASP.NET State Service. You can grant non-local machines access to the State Service via a registry setting. This is an improvement over ASP 1.0, which did not restrict access to the StateServer mode from any machine.

Using SQLServer Session State

The SQLServer mode provides out-of-process session storage and management using a SQL Server database. You configure the SQLServer mode in a two-step process. First, you update the `Web.config` file:

```
<sessionState mode="SQLServer"
    sqlConnectionString="server= machineName\sqlServer;uid=myid;pwd=123;"
    cookieless="false" timeout="20" />
```

You have some flexibility in the format of the SQL connection string. You could use the following alternate format:

```
<sessionState mode="SQLServer"
    sqlConnectionString="data source= machineName\sqlServer;
        user id=myid;password=123;" cookieless="false" timeout="20" />
```

Note that the connection string does not include a database name. In fact, the application will generate a runtime error if you include a specific database name in the connection string. For security purposes, you may prefer to use a trusted connection in place of specifying SQL credentials in the database connection string. (Chapter 3, "Writing Optimized Data Access Code," describes SQL Server trusted connections in detail.)

Next, you need to run the SQL batch script that creates the SQL Server session state database:

1. Open SQL Query Analyzer.

2. Open the `InstallSqlState.sql` script in a new window. The script is located at `%windir%\Microsoft.NET\Framework\%version%`, where `%version%` is a folder that is named equal to the current installed version of the .NET Framework.

3. Execute the SQL script in Query Analyzer.

The script creates a new database called `ASPState`, which contains a number of stored procedures for writing to, and reading from, the `tempdb` database. When a user assigns a session item, the information is inserted into a temporary table in the `tempdb` database. The new record includes an expiration timestamp that is equivalent to the `<sessionState>` element's timeout attribute value, in `Web.config`.

The advantages of the SQLServer mode are as follows:

- Session storage is out-of-process, so it does not directly impact the scalability of the ASP.NET application.

- You can share session items across multiple Web servers and potentially persist them until the service is stopped or the session item is explicitly removed.

- It is highly efficient storage and retrieval for simple data types and small DataSets.

The disadvantages of the SQLServer mode are as follows:

- It offers less efficient storage and retrieval for large DataSets.
- It potentially impacts application scalability when session items are large and/or the number of session reads and writes is high.
- It only works for objects that can be serialized (in other words, objects based on classes that implement the `ISerializable` interface).

The SQLServer mode is typically your only choice for session state if you need to guarantee that the session information will be durable. The exception would be if your ASP.NET application stores small strings, and you are willing to persist this information in cookies on the individual client machines. The SQLServer mode is an excellent combination of performance and durability, and it will typically have limited impact on the scalability of an ASP.NET application. This is provided that the session items are modest in size and the number of session reads and writes remains reasonable. The SQLServer mode may not be a good choice if you are persisting large amounts of data, especially in combination with complex object types, such as the DataSet object. The process of serializing information to and from the database is extremely fast for a smaller number of users. But you are likely to notice a measurable delay if your application makes a high number of concurrent requests to the database, especially for larger amounts of information.

Analyzing Session State Performance

We have all heard about the supposed performance implications of using Session objects, but rarely do we see actual performance numbers in print. There is probably a good reason for this—namely, that no published set of numbers really applies to your application. But there is value in looking at the relative performance numbers for a simple ASP.NET Web page that retrieves data from a SQL Server database. ASP.NET introduces a new and unfamiliar set of session management options, and it is interesting to look at how each mode performs relative to the others.

Visual Studio .NET Enterprise Edition provides a tool called Microsoft Application Center Test (ACT), which is a stress test tool for Web applications. The tool allows you to record a Web session and then execute it for multiple simulated users. ACT provides summary statistics and performance counter numbers for the test runs. These metrics enable you to analyze performance and scalability issues with your application. Chapter 7, "Stress Testing and Monitoring ASP.NET Applications," discusses how ACT works in great detail. For now, show simulations for an increasing number of concurrent browsers and measure three important performance and scalability counters:

- **Time to Last Byte (TTLB)**: This counter measures (in milliseconds) how long it takes for the Web application to service a request. TTLB is a key indicator of how scalable an application is.

- **Requests/Sec**: This counter measures how many pages the Web application can serve per second. (This counter is a good measure of scalability.)

- **% Committed Bytes in Use**: This counter measures the amount of memory being utilized on the Web server. This measure includes all processes running on the machine, so you need to adjust the final numbers for the amount of memory usage that is unrelated to the Web application.

Processor utilization is another important metric because it indicates whether your hardware is a limiting factor to your application's scalability. This metric factors into Transaction Cost Analysis (TCA), which provides a quantitative measure of the processing cost of your application for a specific user load. Note that TCA is not a part of this chapter's load testing because our purpose is to study the relative performance of each session state mode. However, Chapter 7, "Stress Testing and Monitoring ASP.NET Applications," discusses it in detail.

ACT also provides a summary of the HTTP Errors count, which is important because performance metrics are only relevant when a significant percentage of the requests have been successfully processed. As the number of concurrent browsers increases, the chance for errors increases as well. A successful request will return an HTTP response code of 200. ACT will commonly return two additional response codes:

- Response code 403 indicates that the server understood the request but is refusing to fulfill it.

- Response code 500 indicates that the server encountered errors in attempting to fulfill the request.

Response code 403 is frequently returned for higher numbers of concurrent browsers. We do not consider performance numbers meaningful unless greater than 97.5 percent of the requests are fulfilled successfully. For this reason, in the following performance test, we ignored all test runs with greater than 10 concurrent browsers.

Sample Web Page with Session State

The sample Web "application" is a single Web page called `ap_SalesQueryWithSession.aspx`, which executes a stored procedure in the

Northwind database and binds the resulting DataSet to a DataGrid on the page. Specifically, the page executes the `[Employee Sales By Country]` stored procedure, which accepts two input parameters: `@BeginningDate` and `@EndingDate`. Figure 4-2 shows the Web frontend screen for this stored procedure.

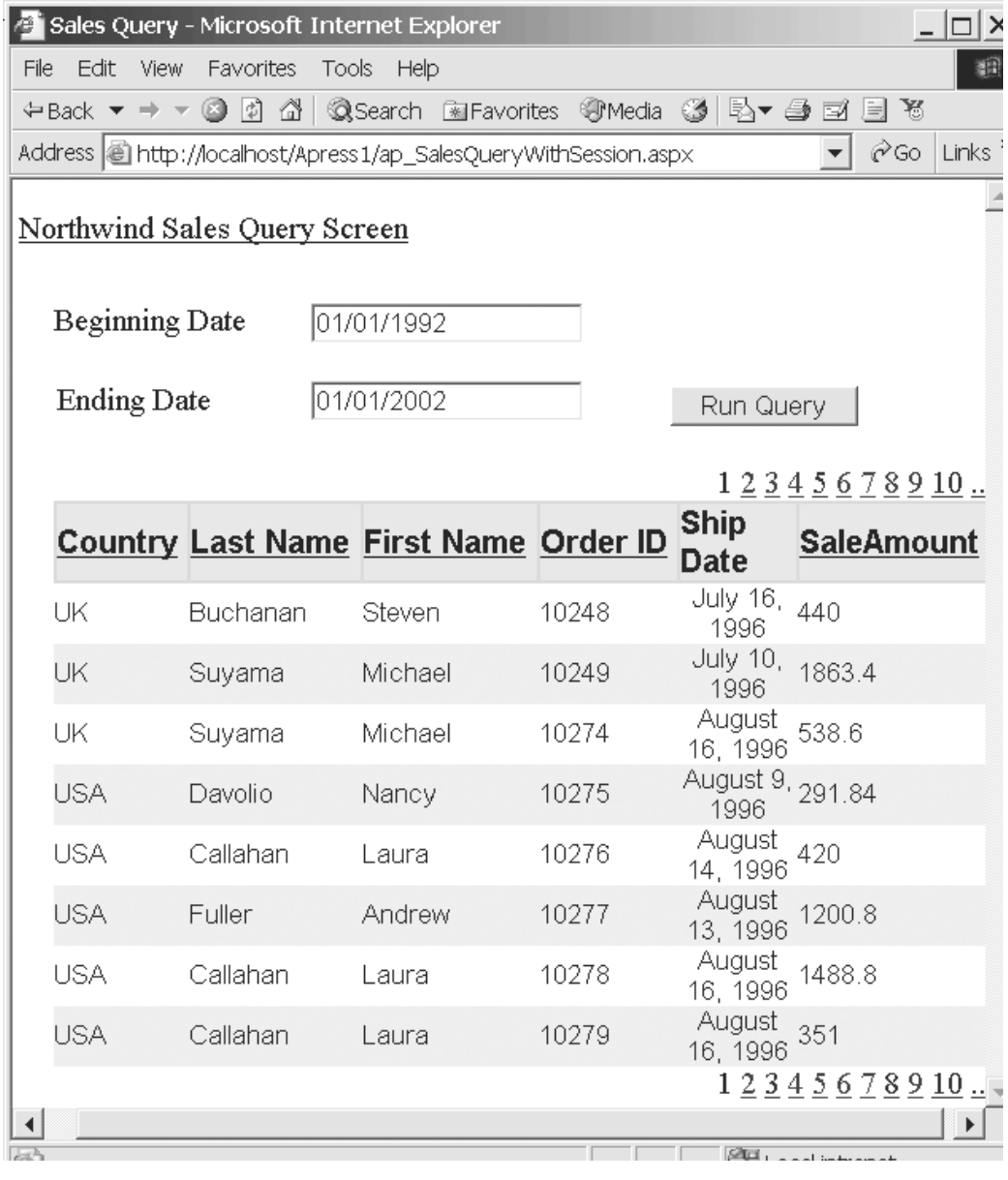

Figure 4-2. Using session state

The first time that the page executes, it retrieves a DataSet directly from the database. This DataSet gets bound to the DataGrid and then assigned to a Session object. In addition, the search parameters are persisted directly to view state so that they are available for comparison purposes. On subsequent requests, the code compares the current textbox values with the values in view state. If they are the same, then the code attempts to retrieve the DataSet from the Session object. If they are different, then the code executes a fresh database request.

This logic is handled inside of the `BindDataGrid()` function, as shown in Listing 4-1.

Listing 4-1. The `BindDataGrid()` *Function*

```
Private Sub BindDataGrid()
    Dim objDB As Apress.Database
    Dim arrParams() As String
    Dim sqlDS As DataSet
    Dim blnRefreshDS As Boolean = False
    Dim strJSScript As String = False

    ' Retrieve the connection string from Web.config
    Dim strConn As String
    strConn = ConfigurationSettings.AppSettings("ConnectionString")

    Try
        ' Did the search criteria change?
        If viewstate("BeginningDate") <> Me.ap_txt_beginning_date.Text Then _
            blnRefreshDS = True
        If viewstate("EndingDate") <> Me.ap_txt_ending_date.Text Then _
            blnRefreshDS = True
        ' Look for an existing DataSet object in a session variable
        sqlDS = CType(Session("sqlDataView"), DataSet)
        If sqlDS Is Nothing Then blnRefreshDS = True
        If blnRefreshDS Then
            ' Step 1: Instance a new Database object
            objDB = New Apress.Database(strConn)
            ' Step 2: Execute [Employee Sales By Country]
            arrParams = New String() { _
                "@Beginning_Date", Me.ap_txt_beginning_date.Text, _
                "@Ending_Date", Me.ap_txt_ending_date.Text}
            sqlDS = objDB.RunQueryReturnDS("[Employee Sales By Country]", _
                arrParams)
            Session("sqlDataView") = sqlDS ' Assign DataSet to Session object
```

```
            ' Persist the search parameters in ViewState, for future comparison
            viewstate("BeginningDate") = Me.ap_txt_beginning_date.Text
            viewstate("EndingDate") = Me.ap_txt_ending_date.Text
        End If

        ' Bind the DataView to the DataGrid
        DataGrid1.DataSource = sqlDS
        DataGrid1.DataBind()

    Catch err As Exception
        ' Report the error in a JavaScript alert
        strJSScript = "<SCRIPT LANGUAGE='JavaScript'>alert('" & _
            err.Message"');</SCRIPT>"
        RegisterStartupScript("JSScript1", strJSScript)
    Finally
        objDB = Nothing
    End Try

End Sub
```

Note that Listing 4-1 uses a wrapper function called `RunQueryReturnDS()`, which is a member of a custom data access component that encapsulates ADO.NET database calls. You can view the code listing for this component in the sample project that accompanies this chapter.

Stress Testing with Session State

We stress tested the sample page in four groups of tests: one group for each of the four session state modes. We performed the testing within each group as follows:

1. We configured the page for one of the session state modes: Off, InProc, StateServer, or SQLServer.

2. ACT recorded a Web browser session with three steps:

    a. Load `ap_SalesQueryWithSession.aspx` into the browser for InProc, StateServer, and SQLServer modes. For Off mode, load `ap_SalesQueryWithDataSet.aspx`.

    b. Enter a Beginning Date of 01/01/1992 and an Ending Date of 01/01/2002.

c. Click the Submit Query button twice: first, to retrieve a DataSet from the database and, second, to retrieve the DataSet from the Session object.

3. The recorded script ran three times, one time each for one, five, and 10 concurrent browsers. The script ran for a 35-second interval with a five-second warm-up period.

The database returned 809 records per query for the time period from 01/01/1992 to 01/01/2002. ACT generated from roughly 600 to 900 connections per test during the 35-second testing interval, depending on the session mode. This means that the tests created anywhere from 200 to 450 Session objects during the testing interval.

We executed the tests in two groups of runs with different architectures:

Group A: We executed these tests against a dedicated Web server using recorded scripts in ACT. The database resided on a separate server on the network. The ACT scripts were executed from the database server against the Web server to avoid generating simulated requests on the same server that processes them. This design spreads the processing burden between multiple servers so that IIS and SQL Server do not have to compete for processor time on the same server. This design should prevent the test results from being skewed by an overburdened processor.

Group B: We executed these tests on a single server that runs the Web server, the SQL Server, and the test scripts. This architecture imposes a high processor burden on the server, but it does not unusually skew the memory usage numbers. We chose this architecture because authentication issues prevented the Group A test results from generating memory usage numbers. For the client machine to bind to these remote counters, the Web server must authenticate requests using a domain account with administrative access (to the Web server). We chose not to set up these permissions levels for this round of testing.

The Group A tests represent better testing practices because the architecture spreads the processing burden between multiple servers. We ran the Group B tests because we could not otherwise generate memory usage numbers for different session state modes.

Before proceeding, we should point out that, in reality, you would likely not design a Web application to have tens to hundreds of session-stored data sets. The ACT tests represent unusually stressful conditions that would not likely be duplicated in the field because you would make a different design decision to avoid this situation. But this is, after all, what stress testing is all about.

Analyzing the Stress Testing Results

By session mode, Table 4-1 shows the change for Group A in the all-important Time To Last Byte (TTLB) parameter as the number of concurrent browsers increases. The numbers are normalized per 100 requests. You will recall that this parameter is a key indicator of application scalability.

Table 4-1. Normalized TTLB by Session State Mode (in Milliseconds per 100 Requests)

CONCURRENT BROWSERS	MODE = OFF	MODE = INPROC	MODE = STATESERVER	MODE = SQLSERVER
1	7.81	4.54	8.27	8.47
5	28.28	20.25	27.25	29.29
10	89.38	46.08	77.29	85.11

The TTLB numbers are similar for Off, StateServer, and SQLServer modes. However, the numbers are lower for InProc mode by up to a factor of two. This number becomes important when the Web server is under heavy load. A lower TTLB number translates into less latency—that is, more requests serviced per second. The testing results indicate this, as shown in Table 4-2, which presents Group A average request rates for each of the session state modes.

Table 4-2. Average Requests per Second by Session State Mode

CONCURRENT BROWSERS	MODE = OFF	MODE = INPROC	MODE = STATESERVER	MODE = SQLSERVER
1	18.86	24.17	18.31	18.11
5	21.66	25.74	21.54	21.34
10	17.23	23.8	18.11	17.6

These numbers may not look very different, but they can translate into a dramatically different number of total serviced requests. For example, over the course of the 35-second testing interval with 10 concurrent users, the Off mode serviced 603 total requests, and the InProc mode serviced 833 total requests.

Based on these numbers, the total number of serviced requests, from highest to lowest, is as follows: InProc, StateServer, SQLServer, Off.

This sequence should sound entirely logical: InProc mode is fastest because it operates in memory and on the same worker process as the application. StateServer mode is the next fastest because it also operates in memory, although you take a responsiveness hit for the time it takes to marshal session data across processes. SQLServer is the next fastest because it takes time to exchange session

information with the database. Finally, the Off mode is the least responsive because every response must be regenerated freshly.

One of the knocks against classic InProc session variables is that they are scalability killers. They exhaust server resources rapidly as the number of concurrent users increases. This is a double hit when you consider that the Web server could be using some of this memory for caching, which would help service requests even faster by avoiding a complete re-creation of the response. In fact, session variables continue to use server resources, even if the user is not actually storing any session-specific information. Even a lightly used session variable continues to consume server resources. The overall result is that the Web server services fewer requests as the number of concurrent users increases.

The numbers in Table 4-2 appear to verify this trend, although with an interesting twist. Each mode services the most requests for five concurrent users but a fewer number for one user and for 10 concurrent users. Figure 4-3 shows a graph of the Group A average requests per second by session state mode.

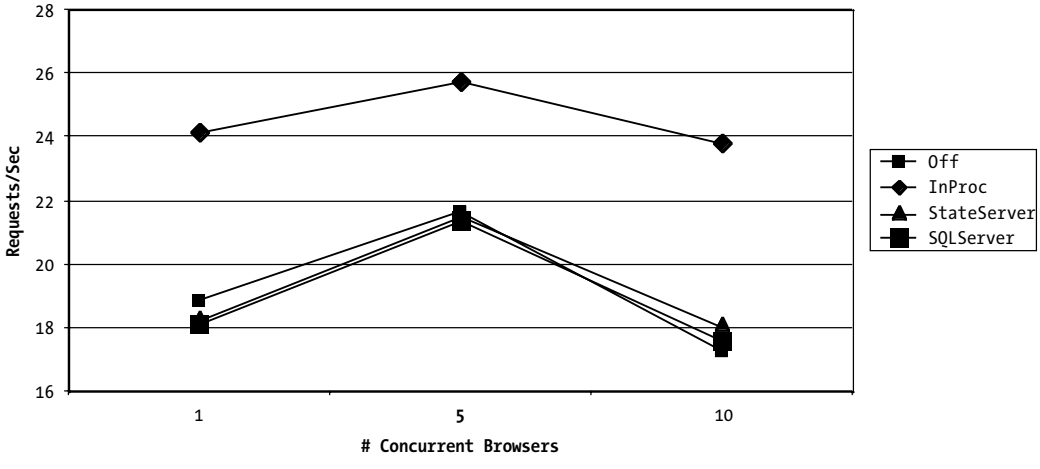

Figure 4-3. Group A: Average requests/sec by session state mode

This "triangular trend" indicates that five concurrent users receive better responsiveness than one concurrent user. This trend may reflect the influence of SQL Server, which caches data pages for successive requests, and SQL connection pooling, which makes a set of connections readily available for multiple users. The number drops again for 10 concurrent users because it exceeds the pool number and begins to be high enough to burden the server.

A better measure of scalability changes is to look at the change in TTLB as the number of concurrent users increases. Figure 4-4 graphs the change in TTLB for each session state mode as the number of concurrent users increases. The

numbers are normalized based on 100 requests to adjust for the fact that different session modes service different numbers of requests. For example, in the Group A tests, InProc mode serviced 846 total requests, and SQLServer mode serviced 634 total requests.

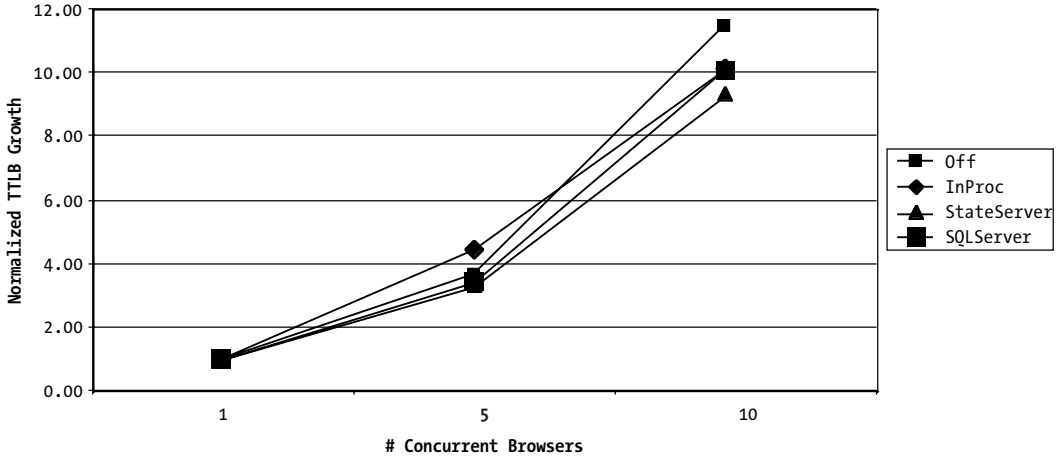

Figure 4-4. Group A: Normalized TTLB by session state mode

The TTLB numbers shown in Figure 4-4 exhibit subtle differences, except for InProc mode, which experienced the lowest TTLB numbers. This indicates that the InProc mode can service a superior number of requests and remain more responsive than other session modes. We attempted to test more than 10 concurrent browsers, but the number of request errors exceeded 20 percent, which would not produce meaningful numbers for comparison.

Based on our limited data set, it is useful to look at relative growth rates in TTLB, as shown in Figure 4-5. The TTLB is normalized for each session mode, based on one concurrent user. For example, TTLB grows a factor of 10.05 for SQLServer mode as the number of concurrent browsers increases from 1 to 10.

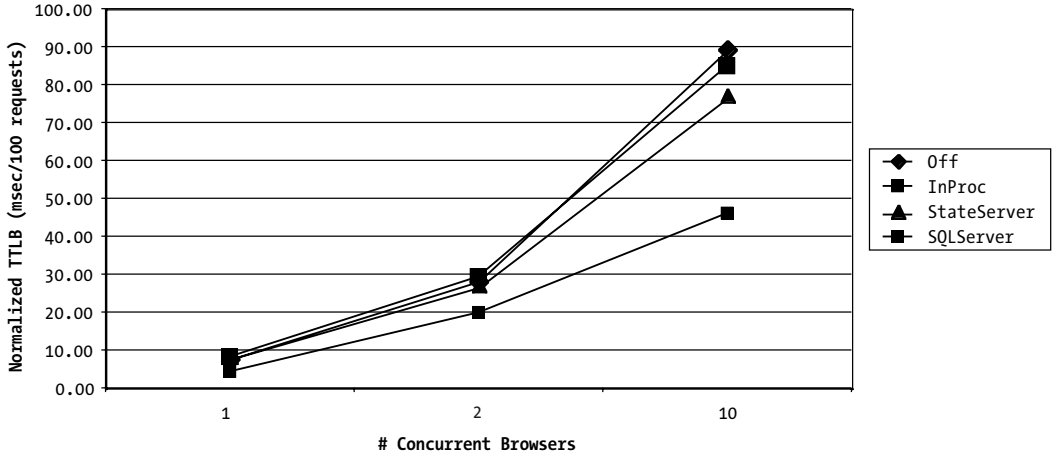

Figure 4-5. Group A: Normalized TTLB growth by session state mode

The differences in the TTLB growth rates are subtle, and it is perhaps a stretch to infer patterns from them. However, based on these numbers, the growth rate in TTLB for each session mode from highest to lowest is as follows: Off, InProc, SQLServer, StateServer.

This trend indicates that the Off mode experiences the greatest growth in TTLB as the number of concurrent users increases. The InProc mode and the SQLServer mode experience lesser growth in TTLB, and the StateServer mode experiences the lowest. The results simply indicate the trend in TTLB growth and are not a replacement for actual stress testing and observation at higher user loads. These limited results simply indicate that responsiveness goes down as the number of concurrent browsers increases and that the Off mode experiences the greatest decrease in responsiveness. As the stock market mantra goes, current results are not an indication of future performance. In a similar sense, TTLB growth changes at low user loads may not indicate their behavior at higher (and more critical) user loads.

A further note of wisdom is that every system will experience bottlenecks at some level, whether it is related to the processor speed, to available memory, to network latency, or to the number of active threads being processed. Your goal must be to stay ahead of the curve by designing your system to manage its expected loads as efficiently as possible. Ultimately, performance tuning is important because it allows your system to handle higher loads without a redesign or without having to purchase bigger, more expensive hardware.

The other piece of the scalability puzzle is memory usage. We were unable to generate memory usage numbers for Group A tests because ACT could not bind to the remote Memory counter on the Web server (recall that ACT is running on a separate server from the Web server). However, ACT has no problem binding to the Memory counter on the same server. As a workaround, we ran an alternative set of tests on a single server (Group B).

Figure 4-6 shows the Group B normalized TTLB values, based on 100 requests. The result pattern is different from the equivalent Group A test. The SQLServer and StateServer modes experience much higher TTLB values, compared to the InProc and Off modes, by up to two orders of magnitude. This difference may reflect the greater processor burden on the single server. Simply put, with more demands on the processor, the SQLServer and StateServer modes suffered because they are more dependent on processor availability. We are not attempting to explain the numbers away, but we are simply presenting the TTLB test results so that you can keep them in mind when evaluating the memory usage results.

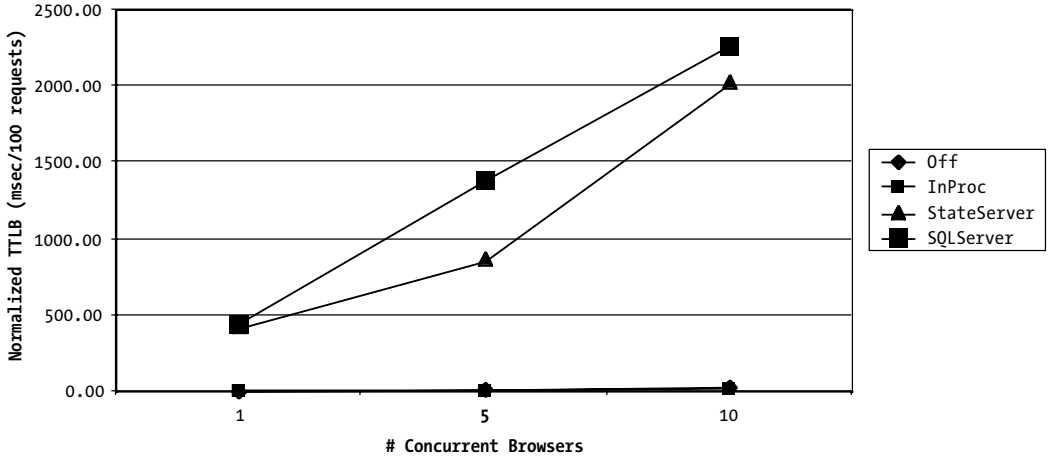

Figure 4-6. Group B: Normalized TTLB by session state mode

Figure 4-7 shows actual memory usage by session mode where memory usage is defined as the percentage of committed bytes in memory (as compared to the total amount of memory available). This is an actual measure of memory usage on the server, and it reflects the level of burden that each session mode places on available server memory.

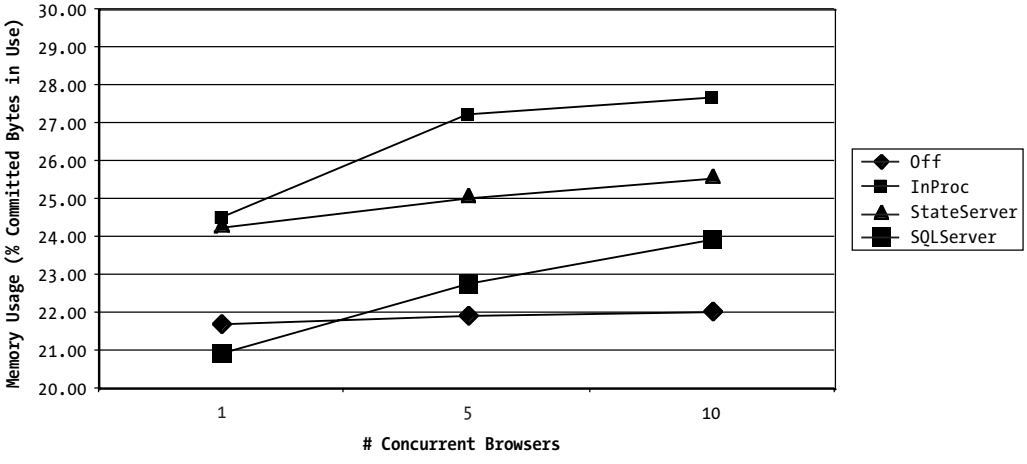

Figure 4-7. Group B: Actual memory usage by session state mode

The InProc mode clearly uses the highest amount of memory, followed by the StateServer mode. The Off mode uses the least amount of memory, which is to be expected. The SQLServer mode falls somewhere in between, although it is interesting to note that its growth curve in memory usage is steeper than for other modes. It is unfortunate that ACT could not generate meaningful numbers with more than 10 concurrent browsers because it would be interesting to see where the trends continued.

Memory usage numbers are an important indication of how a session mode impacts server resources. But as with every counter, it only tells a part of the story. For example, from Figure 4-7 alone, you might infer that the InProc mode is a potential scalability killer because it exerts the highest burden on server memory. But then, consider that it services a far greater number of requests than the other modes. Increased memory usage may be a small price to pay for the far greater number of requests that you can service, compared to other session modes. Add to this the fact that the InProc mode experiences lower TTLB growth rates than other session modes (based on both Group A and Group B test results). The InProc mode suddenly appears to be an attractive option for managing session state.

In closing out this section, we want to emphasize its overall message, which is that session state performance is not as clear-cut as many texts would lead you to believe. For example, many texts brand the InProc mode as a guaranteed scalability killer that should always be avoided on heavily trafficked Web sites. Our tests have demonstrated that the picture is more complex because the InProc mode offers far superior performance in exchange for higher memory usage.

Of course, there are other considerations that go into choosing a session state mode. For example, if you must persist session state to disk or manage it in a Web farm, then InProc mode will not meet your needs, no matter how good or

bad it may be. The previous section described the advantages and disadvantages of each session state mode and discussed the optimal usage scenarios for each mode.

The bottom line is that only you can decide which approach is best for your Web site. There is no set of test numbers that can ever tell the definitive story, and we ask you to keep this in mind and to possibly be inspired to extend our testing with your own.

Programming with Session State

Session objects provide a straightforward application programming interface (API) that is easy to code against. Table 4-3 summarizes useful Session object members.

Table 4-3. Session Object Members

MEMBER	DESCRIPTION
`SessionID`	This read-only property gets the unique session ID used to identify the session.
`Timeout`	This read-write property gets or sets the timeout period (in minutes) between requests before the session is terminated.
`Keys`	This read-only property gets a collection of the keys of all values stored in the session.
`IsReadOnly`	This read-only property gets a value indicating whether a session is read-only. You set this property at the individual page level using `<%@ Page EnableSessionState="ReadOnly" %>`.
`Add()`	This method adds a new item to session state. Its syntax is `Add(name As String, value As Object)`.
`Clear()`	This method clears all values and references from session state.

`Abandon()`	This method cancels the current session. If the session gets reinitialized, then the user gets a new session with a different session ID. Session objects store information by associating an object reference with a named index, or *key*, as follows:
	`Session("[Name]") = [Object]`
	Or, alternatively:
	`Session.Add("[Name]", [Object])`

NOTE *If you add a session variable that already exists, then the existing session variable will simply be updated with the new object or value. The previous object, or value, will be overwritten without a warning.*

Recall that in .NET, all data types inherit from the `System.Object` type. This enables Session objects to store virtually any .NET data type with two important exceptions:

- If the session mode is StateServer, then you can only assign objects that support binary serialization. For example, the DataSet object supports serialization, and the DataView object does not. *Serialization* is the process that allows an object to be represented in an XML document or as a binary stream. The object may then be stored, or transported in this form, and then faithfully re-created from the XML document or stream. However, if the session mode is InProc, then the object need not support binary serialization.

- You should not store objects, such as the DataReader, that maintain open connections to a database in Session objects. If you must assign a data object to a session, then use a disconnected object such as a DataSet or DataView.

You can iterate through a collection of session keys using the following:

```
Dim objKey As [Object]
For Each objKey in Session.Keys
    Console.WriteLn(objKey.Name) ' Write out the name of the key
Next
```

Retrieving session values is as simple as assigning the stored object back to a local variable:

```
sqlDV = Session("sqlDataView")
```

This method implicitly casts the session reference to the appropriate data type. You need to do this step because the Session object stores its references as Object data types for maximum flexibility. An alternative to implicit casting is to explicitly cast the data type when the reference is retrieved:

```
Dim sqlDV As DataView
sqlDV = CType(Session("sqlDataView"), DataView)
```

Explicit casting is always preferable to implicit casting. Once you have retrieved an object reference, you should always verify it was retrieved successfully before using it in code. The easiest way to do this is to execute the assignment and then check to see if the object exists:

```
sqlDV = CType(Session("sqlDataView"), DataView)
If sqlDV Is Nothing Then
    ' Recreate the object
End If
```

Finally, the Session object provides two event handlers for adding code when a session is first created and when it is abandoned. The "Understanding the Global.asax File" section discusses the `Session_Start()` and `Session_End()` event handlers in more detail.

Session State Management in Web Farms

ASP.NET makes it easy to manage session state in Web farms. The StateServer and SQLServer modes are equally good candidates for centralized session state management, so you need to decide which mode is right for your application. The StateServer mode may offer better performance than the SQLServer mode. However, the SQLServer mode guarantees that session state information will be durable. The StateServer mode cannot provide the same guarantee because it provides in-memory storage.

Keep in mind that you may not need a centralized State server in your Web farm. If you are using an IP redirector, such as Cisco's LocalDirector or F5 Network's BIGIP, then a client's requests get routed to the same server for the duration of their session. In this case, you can maintain session state on individual servers, using either the InProc or StateServer modes. You do run a risk that IP redirection may not always work. If a server crashes or becomes unavailable, then the client will be routed to another server, which will have no record of their session information. For this reason, you may want to consider using centralized session state management.

If you decide on using the StateServer mode, then you need to start the ASP.NET State Service on one of the servers in the Web farm. You must designate only one server in the Web farm for managing session state because you are proceeding on the assumption that there is no fixed redirection of requests in the Web farm. The advantage of this approach is its flexibility in being able to manage state for all servers in the Web farm. However, the disadvantage of this approach is that it creates a single potential point of failure. In exchange for flexibility, you run a higher risk that the State server may fail and be completely unavailable for all servers in the Web farm.

Next, you need to modify the `Web.config` file for each server in the Web farm to point to the centralized State server. For example:

```
<sessionState mode="StateServer" stateConnectionString="tcpip=127.0.0.1:42424"
        cookieless="false" timeout="20" />
```

Obviously, for this connection string to work, the State server must provide a fixed IP and must not use Dynamic Host Control Protocol (DHCP). If you decide on using the SQLServer mode, then you need to set up a central database server and run the SQL script that creates the ASPState database. Next, you need to modify the `Web.config` file for each server in the Web farm to point to the same SQL Server database. For example:

```
<sessionState mode="SQLServer"
     sqlConnectionString="server= machineName\sqlServer;uid=myId;pwd=myPwd;"
     cookieless="false" timeout="20" />
```

If the reliability of your session is of utmost importance, then you can implement state management on a cluster of multiple database servers so that no single point of failure exists.

This concludes the discussion of session state management. Next, turn your attention to the topic of application state management.

Overview of Application Management

Application state management enables information to be shared between multiple users of the same Web application. In classic ASP, you manage application state using an HttpApplicationState handler, which is encapsulated by the ASP Application object. This object is still available in ASP.NET, although it has additional members and gives you new ways to enumerate through a collection of HttpApplicationState variables. The Application object is easy to work with, in terms of configuration and coding, and this makes it a tempting option. Unfortunately, the Application object is also problematic because it does a poor job of

synchronizing changes from multiple users. Only one user at a time (technically, one thread at a time) should be allowed to modify an Application object variable. This is a big issue in the .NET environment, which supports free-threaded objects. To ensure single-thread updates, the Application object provides `Lock()` and `UnLock()` methods that prevent other users from simultaneously updating the object. The `Lock()` method actually locks the *entire* Application object, even though the user may be updating just one of several available object variables. This feature can cause concurrency problems if several users attempt to lock the object at the same time. Ultimately, concurrency problems lead to scalability problems as users are forced to wait to commit their changes. Worse yet, concurrency lockups could cause one or more users to deadlock and experience instability with their sessions. As a result, Application state is only appropriate for values that are read often but are updated infrequently.

ASP.NET provides a new and superior alternative to the Application object in the form of a Cache engine, which provides better control over data storage and retrieval. The Cache engine provides a more sophisticated API than the Application object as well as better concurrency handling. The following sections demonstrate the different options that ASP.NET provides for managing application state.

Permanent vs. Transient Application State

You need to store two kinds of information at the application level:

- **Permanent information**: This applies globally across an application and changes rarely. Examples include connection string information or configuration setting values referenced throughout the application.

- **Transient information**: This information is still global in scope, but it changes with some frequency; examples include counters, such as a Web site visitor counter. Users must all modify the same copy of this value to keep a running count.

Although you could store both kinds of information in an Application object, ASP.NET provides better alternatives.

Understanding Permanent Application State

You can store permanent information in the `Web.config` file and reference it programmatically at runtime. At the simplest level, you can assign custom information to a new key within the `<appSettings>` node:

```
<appSettings>
    <add key="ConnectionString" value="server=;uid=sa;pwd=ap1;database=dev;" />
    <add key="SysAdminEmailAddress" value="sysadmin@yourcompany.com" />
</appSettings>
```

You can then reference these keys from any code-behind file using the ConfigurationSettings object, which is a member of the `System.Configuration` namespace:

```
Dim strConn As String
strConn = ConfigurationSettings.AppSettings("ConnectionString")
```

The `<appSettings>` element is useful, but it is restricted to storing name-value pairs. The `Web.config` file also allows you to define a custom configuration section, which can have a more complex structure. For example, you could define a section that tracks parent (myMenuGroup) and child (myMenuItem) menu options:

```
<configSections>
        <!-- Declares a section group called myMenuGroup -->
        <sectionGroup name="myMenuGroup">
                <!-- Declares a section name called myMenuItem -->
                <section name="myMenuItem"
                    type="System.Configuration.DictionarySectionHandler, System"/>
        </sectionGroup>
</configSections>
```

You could then implement the sections as follows:

```
<myMenuGroup>
        <myMenuItem>
                <add key="Login" value="login.aspx"/>
                <add key="Logout" value="logout.aspx"/>
        </myMenuItem>
</myMenuGroup>
```

You must define and implement custom configuration settings inside `Web.config`. Once you do this, you can reference the settings from any code-behind file in the project using the `GetConfig()` method of the ConfigurationSettings object:

```
Dim dctMenuItems As IDictionary
Dim enmKeys As IDictionaryEnumerator
```

```
dctMenuItems = ConfigurationSettings.GetConfig("myMenuGroup/myMenuItem")
enmKeys = dctMenuItems.GetEnumerator()
While enmKeys.MoveNext
    If enmKeys.Value.GetType.ToString = "System.String" Then
        Response.Write(enmKeys.Key & " = " & enmKeys.Value & "<BR>")
    End If
End While
```

The `Web.config` file is an excellent choice for persisting permanent application information that must be referenced by, but never altered by, the application. Clearly, the `Web.config` file is only capable of storing a limited range of data types, so it is most suitable for storing configuration values. An added advantage is that the application automatically picks up changes to this file without requiring a restart. (ASP.NET automatically restarts when it detects a change event.) This makes it easy to change application settings on the fly or to deploy multiple versions of the same `Web.config` file to different environments, such as staging and production.

CAUTION *The* `Web.config` *file is a text file and should therefore not be used for storing sensitive information. IIS will not allow the* `Web.config` *file to be accessed by an outside browser, but you should still be cautious with the type of information you store in this file.*

Some developers refuse to store SQL connection string information in the `Web.config` file. We, on the other hand, store SQL login credentials as long as they reference an account that has highly restricted access to the database.

Chapter 2, "Introducing ASP.NET Applications," discusses the `Web.config` file in great detail.

Understanding Transient Application State

ASP.NET provides two main classes for managing transient application state:

- HttpApplicationState
- Cache

Let's discuss each of these in turn.

Configuring and Using the HttpApplicationState Class

The HttpApplicationState class is instanced once for every ASP.NET application, and it provides shared application state for all requests. This class is conveniently exposed by the Page object's `Application` property. From here on, we refer to the HttpApplicationState class as the *Application object*, which represents a single instance of the class. Think of the Application object as a collection container that enables you to manage a collection of globally scoped variables. Like the Session object, the Application object provides a straightforward API that is easy to code against. Table 4-4 summarizes useful Application object members.

Table 4-4. HttpApplicationState Class Members

MEMBER	DESCRIPTION
`Add()`	This method adds a new item to application state. Its syntax is `Add(name As String, value As Object)`.
`Lock()`	This method locks access to a specific application state variable.
`Unlock()`	This method unlocks access to a specific application state variable.
`Set()`	This method sets the value of a specific application variable. Its syntax is `Set(name As String, value As Object)`.
`Contents`	This read-only property gets a reference to the collection of Application variables that were added through code using the Application object's API.
`StaticObjects`	This read-only property gets a reference to the collection of Application objects that were added in `Global.asax` using the `<object>` tag.
`RemoveAll()`	This method removes the entire collection of Application variables.
`Remove()`	This method removes a specific Application variable from the collection. Its syntax is `Remove(name As String)`.
`RemoveAll()`	This method removes the entire collection of Application variables.

The Application object is programmatically accessible via the `Application` property of the Page object. Unlike the Session object, the Application object does not require any settings in the `Web.config` file. You can add application-level variables to the collection in two ways:

- Programmatically using the Application object's API
- Using the `<object>` tag in `Global.asax`

For example, you can assign a String object programmatically:

```
Dim objStr As System.String
Page.Application.Add("myStr", objStr)
```

Alternatively, you can instance the String object at the application level in `Global.asax`:

```
<%@ Application Codebehind="Global.asax.vb" Inherits="Apress1.Global" %>
<object runat="server" id="myStr" class="System.String" scope="application" />
```

Now that you have instanced the String object, you can set and retrieve its value from any page within the application. For example, on `Page1.aspx`, you can set the value:

```
' Set the string
Dim MyString1 As String = "My global string value."
Page.Application.Set("myStr", MyString1)
```

Then on `Page2.aspx`, you can retrieve the value:

```
' Retrieve the string
Dim MyString2 As String = Page.Application.Item("myStr")
Response.Write("MyString2 = " & MyString2.ToString())
```

Observant readers will notice that we assigned MyString1 to the Application object without locking the object first. Had we been more careful, we would have used the available locking methods:

```
' Alternative set
Dim MyString2 As String = "My global string value2."
Page.Application.Lock()
Page.Application.Set("myStr", MyString2)
Page.Application.UnLock()
```

The moral of the story is that the Application object allows you to set values without requiring the safety check of the `Lock()` and `UnLock()` methods. Without this check, you risk a collision with another user who is updating the Application

object at the same time. On the other hand, if you keep the Application locked for too long, you risk a deadlock with another user.

In summary, the advantages of the Application object are that Application objects are easy to code with and easy to configure.

The disadvantages of the Application object are as follows:

- You cannot share Application objects across multiple Web servers. Stored values are only available to the application thread that instanced them.
- Application objects are not durable. They reside in memory and will be lost if the dedicated process crashes or is restarted.
- Application objects greatly impact scalability because they are multi-threaded and run a high risk of causing deadlocks and concurrency issues when multiple users attempt updates at the same time.
- Application objects use memory resources, which can potentially have a significant impact on the Web application's performance and scalability—particularly if the Application object stores significantly sized objects, such as a populated DataSet.
- Application objects do not optimize resource usage, for example, by expiring underused items. Application items remain in memory all the time, whether they are heavily used or not.

Let's now take a look at another alternative for managing transient application state: the Cache class.

Configuring and Using the Cache Class

ASP.NET supports application data caching, which allows expensive resources to be stored in memory for fast retrieval. Chapter 5, "Caching ASP.NET Applications," discusses caching in full detail, so this section serves as a quick introduction to the feature. We present just enough detail to demonstrate how caching is a good alternative to the Application object for managing transient application state.

The Cache class provides optimized storage for persisting objects in memory. Unlike the Application object, cached items remain available only for as long as they are needed. You can assign cached items with expiration policies. The Cache class provides much more control over cached items compared to the Application object. These advantages include the following:

- **Customized expiration**: You can assign cache items individual expiration policies that indicate when they should expire and be removed from the cache. The Cache class supports three expiration modes, including absolute, sliding, and dependency expiration:
  - Absolute mode specifies an exact date and time for expiring the item.
  - Sliding mode specifies a time interval for expiring the item, based on the last time that the item was accessed.
  - Dependency mode links an item's expiration to a fixed resource, such as a file. The item automatically expires and refreshes whenever the dependency changes.
- **Memory management**: The Cache class automatically removes underused items from the cache. In addition, items will be systematically evicted from the cache when server resources become low.
- **Concurrency management**: The Cache class automatically manages concurrent updates to the same item, without requiring that the user place a lock on the item.

Durability is the key difference between items stored in the cache vs. those stored in an Application object. Cache items are not guaranteed to persist in memory, although you can ensure they will by setting specific expiration policies. As server resources become low, the Cache class will evict items based on their relative priority. Heavily used items have high priority and will typically be evicted last. You can set items with specific priorities to influence their eviction order. But, ultimately, all items are subject to eviction if server resources become tight enough. The Application object, on the other hand, will continue to hold its references, regardless of the impact on server resources.

Like the Application object, the Cache class allows you to add items implicitly, using basic key-value pairs:

```
Dim sqlDS As DataSet
Page.Cache("MyDS") = sqlDS
```

The Cache class also provides explicit `Add()` and `Insert()` methods for adding cache items with advanced settings, such as expiration policies and priorities. The `Insert()` method is overloaded, so it provides the most flexibility for adding items. For example, this is how you add an item using a 30-minute sliding expiration:

```
Dim sqlDV As DataView
Page.Cache.Insert("MyDV", sqlDV, Nothing, Cache.NoAbsoluteExpiration, _
    New TimeSpan(0, 0, 30))
```

You can retrieve items from the cache implicitly:

```
Dim sqlDV As DataView
sqlDV = Page.Cache("MyDV")  ' Returns Nothing reference if item has been evicted
```

Or, explicitly using the `Get()` method:

```
Dim sqlDV As DataView
sqlDV = Page.Cache.Get("MyDV") ' Returns Nothing reference if item has been evicted
```

Finally, you can explicitly remove items from the cache using the `Remove()` method:

```
Dim MyDS As DataSet
MyDS  = Page.Cache.Remove("MyDS") ' Evaluates to True
```

Because cache items may expire, or be evicted, any code that uses them must have the ability to re-create the object in the event that it cannot be pulled from the cache. Consider the following code, which uses the `GenerateDataSet()` method to create a populated DataSet object:

```
If Not IsNothing(Page.Cache.Item("MyDS")) Then
     sqlDS = Page.Cache.Get("MyDS")
Else
     sqlDS = GenerateDataSet() ' Regenerate the DataSet
     Page.Cache.Insert("MyDS", sqlDS, Nothing, "12/31/2020", _
         Cache.NoSlidingExpiration)
End If
```

In this example, the code attempts to retrieve the DataSet from the cache. If it cannot be found, then it must be regenerated and added to the cache again. This example illustrates an important point: The Cache class is best suited for storing objects that have page-level scope and that can be re-created if needed. The Cache is global to an application, so it may technically be used for storing "application-level" objects. But in practice, you would not want every page to have to check for, and re-create, the Application object item. However, this is not an issue for page-level objects.

The Cache class is a superior alternative to the Application object for all purposes except when you need to store a truly global object reference—that is, a reference that may be accessed from any page within an application and that must be counted on to be there. The Application object is not as efficient as the Cache class, but it does offer more convenience when you want to guarantee that an item will always be available. The Application object does not persist items in the event that the application crashes, but then, neither does the Cache class.

In summary, the advantages of the Cache class are as follows:

- The Cache class optimizes memory management by using expiration policies and by automatically removing underused items.
- The Cache provides automatic concurrency management.
- The Cache class is easy to code with and easy to configure.

The disadvantages of the Cache class are as follows:

- Cache items are not guaranteed to be persistent in the cache. This requires contingency coding in code blocks that use cached object references.
- You cannot share cached objects across multiple Web servers. Object references are only available to the application thread that instanced them.

This concludes our discussion on application state management. Next, we discuss the `Global.asax` file and show how it helps you design optimal ASP.NET applications.

Understanding the Global.asax File

The `Global.asax` file provides access to events handlers for the HttpApplicationState class, for the HttpSessionState class, and for any HTTP module registered for the application. The file is optional, and you are not required to implement any of the event handlers. The `Global.asax` file essentially provides a gateway to all HTTP requests received by the application. It provides a centralized location where you can intercept client requests and use that information to modify custom application state information. The `Global.asax` file generally serves two purposes:

- Handling events for the Application and Session objects
- Centralizing application-wide tasks

This section focuses on the role of `Global.asax` both for state management and for centralizing application-wide tasks.

Table 4-5 summarizes the important Application and Session object event handlers that you can access in the `Global.asax` file.

Table 4-5. `Global.asax` *Event Handlers*

EVENT HANDLER	DESCRIPTION
`Application_Start()`	Called the first time an HttpApplication class is instanced. The `Global.asax` file has access to a pool of HttpApplication instances, but this event handler is called only once.
`Application_BeginRequest()`	Handles the HttpApplication `BeginRequest()` event. This is called when a new HTTP request is received by the application.
`Application_EndRequest()`	Handles the HttpApplication `EndRequest()` event. This is called when an HTTP request has finished processing but before the response has been delivered to the client.
`Application_End()`	Called when all HttpApplication instances unload. This occurs when the application is restarted, which may occur manually or when the `Web.config` file changes.
`Application_Error()`	Called when an unhandled exception is raised anywhere in the application. You can add generic code for managing unhandled exceptions, such as logging the issue and emailing a system administrator.
`Session_Start()`	Called when a new session is started.
`Session_End()`	Called when a session is abandoned. This event handler will not be called if the client simply closes their browser. It will be called when the current session is explicitly abandoned.

For example, consider a simple set of counters that track the following information:

- AllRequests: This tracks the total number of requests received by the application.

- AllUniqueSessions: This tracks the number of unique sessions created in the application.

- SalesQueryCounter: This tracks the number of requests for a specific page in the application, namely, `ap_SalesQuery.aspx`.

Listing 4-2 shows one example of how the `Global.asax` file manages these counters.

Listing 4-2. Seeing `Global.asax` *in Action*

```
Public Class Global
    Inherits System.Web.HttpApplication
    Sub Application_Start(ByVal sender As Object, ByVal e As EventArgs)
        ' Fires when the application is started
        Application("AllRequests") = 0
        Application("AllUniqueSessions") = 0
        Application("SalesQueryCounter") = 0
    End Sub

    Sub Session_Start(ByVal sender As Object, ByVal e As EventArgs)
        ' Fires when the session is started
        Application("AllUniqueSessions") += 1
    End Sub

    Sub Application_BeginRequest(ByVal sender As Object, ByVal e As EventArgs)
        ' Fires at the beginning of each request
        Application("AllRequests") += 1
        If InStr(Me.Request.Url.ToString, "ap_SalesQuery.aspx") > 0 Then
            Application("SalesQueryCounter") += 1
        End If
    End Sub
End Class
```

These counters are all initialized in the `Application_Start()` event, which fires the first time the application is instanced. The AllUniqueSessions counter gets incremented in the `Session_Start` event (assuming that session state is enabled for the application). Finally, the SalesQueryCounter counter gets incremented in the `Application_BeginRequest` event, which fires every time the application receives a new request. The code uses the Request object's `Url` property to determine which page the user has requested.

Managing Unhandled Exceptions with the Application_Error() Event Handler

The Application_Error() event handler is another useful method that is called whenever an unhandled exception occurs anywhere within the application. You can design an application for all foreseeable exceptions, but it is likely that unhandled exceptions will occur, particularly when the application is moved from a development to a production environment. Listing 4-3 shows how you can have unhandled exceptions logged to the application event log, then emailed to the system administrator.

Listing 4-3. Managing Unhandled Exceptions with the Application_Error() *Event Handler*

```
Imports System.Diagnostics
Imports System.Web.Mail

Sub Application_Error(ByVal sender As Object, ByVal e As EventArgs)

    ' Step 1: Write an error to the event log
    Dim strMessage As String
    strMessage = "Url " & Me.Request.UserHostAddress & Me.Request.Path & _
        " Error: " & Server.GetLastError.ToString()

    Dim Log As New EventLog()
    Log.Source = "ASP.NET 1.0.3705.0"
    Log.WriteEntry(strMessage, EventLogEntryType.Error)

    ' Step 2: Send a mail message to the System Administrator
    Dim objMail As Mail.MailMessage = New Mail.MailMessage()
    With objMail
        .BodyFormat = Mail.MailFormat.Html
        .To = "sysadmin@yourcompany.com"
        .From = "sysadmin@yourcompany.com"
        .Subject = "Exception Report for " & Me.Request.UserHostAddress
        .Body = "<html><body><h2>" & Me.Request.UserHostAddress & _
            Me.Request.Path & "</h2>" & Me.Server.GetLastError.ToString() & _
            "</body></html>"
    End With

    ' Step 4: Send the Mail message (SMTP must be configured on the Web server)
    Dim objSmtpMail As Mail.SmtpMail
    objSmtpMail.SmtpServer = "MySMTPServer"
```

```
    objSmtpMail.Send(objMail)
    objSmtpMail = Nothing
    objMail = Nothing
End Sub
```

As an added convenience, you can set the `<customErrors>` element in the `Web.config` file to automatically redirect remote users to a friendly custom error page. This redirection will occur after the `Application_Error()` event handler has been called. Local users (in other words, developers who are working on localhost) will continue to see a standard error screen that displays full exception details, including the call stack:

```
<customErrors mode="RemoteOnly" defaultRedirect="ap_CustomErrorPage.aspx"/>
```

In summary, the `Global.asax` file serves as a central location for efficiently managing application and session state and as central location for managing application-wide tasks. The `Global.asax` file plays a key role in developing optimal ASP.NET applications.

Using a Custom Base Class for Global.asax

The Application object is not the only way to store application-wide values. In fact, it may be inefficient to store certain kinds of information this way. For example, consider the counter example from Listing 4-2. The three counters are initialized and incremented within the `Global.asax` file only, and they are never modified outside of this file. There is no need to use an Application object for storing this information, particularly if you want to keep the counter values private and inaccessible from the rest of the application.

An alternative approach to using the Application object is to create a custom base class for the `Global.asax` file. This base class inherits from the HttpApplication class, just like the default Global class that sits behind the `Global.asax` file. The custom base class provides the same members as the default `Global.asax` file, but even better, you can extend the class with additional members, such as custom properties for tracking counters.

Listing 4-4 illustrates one possible custom base class.

Listing 4-4. Creating a Custom Base Class for the `Global.asax` *File*

```
Imports System.Diagnostics
Public Class apCustomModule
    Inherits System.Web.HttpApplication
```

```
    Private m_Counter As Integer

    Public Property MyCounter() As Integer
        Get
            MyCounter = m_Counter
        End Get
        Set(ByVal Value As Integer)
            m_Counter = Value
        End Set
    End Property

    Sub Application_Start(ByVal sender As Object, ByVal e As EventArgs)
        ' Fires when the application is started
        MyCounter = 0
    End Sub

    Sub Application_BeginRequest(ByVal sender As Object, ByVal e As EventArgs)
        ' Fires at the beginning of each request
        MyCounter = MyCounter + 1
    End Sub

    Sub Application_End(ByVal sender As Object, ByVal e As EventArgs)
        ' Fires when the application ends
        Dim Log As New EventLog()
        Log.Source = "ASP.NET 1.0.3705.0"
        Log.WriteEntry("Number of Application Requests: " & MyCounter, _
            EventLogEntryType.Information)
    End Sub

End Class
```

You can find this code implemented in the sample application, AspNetChap4A, which accompanies this chapter. Notice that the class inherits from the HttpApplication class and that it implements selected event handlers. The class provides a property called `MyCounter`, which is equivalent to the AllRequests counter from Listing 4-2. This property value gets incremented in the `Application_BeginRequest()` event handler—that is, once for every client request.

The next and final step is to update the `@ Application` directive in the `Global.asax` file to inherit from the custom base class instead of from the default Global class:

```
<%@ Application Codebehind="Global.asax.vb"
      Inherits="MyApp.apCustomModule" %>
```

The custom base class resides in memory continuously for as long as the application remains loaded. As a result, the `MyCounter` property acts like a static variable, such that all application users will share one instance. When the application does unload, the current counter value gets written to the application event log.

One caveat with this approach is that you run the risk of thread blocking issues if ASP.NET fails to manage the user load correctly. ASP.NET does a good job of managing its thread pool and is efficient at managing its pool of HttpApplication instances. You should not encounter problems updating custom properties if they encapsulate simple data types. To be on the safe side, make sure you stress test your Web application and monitor the number of errors the application encounters under heavy load.

In summary, the `Global.asax` file serves as a central location for efficiently managing application and session state and as a centralized location for managing application-wide tasks. The `Global.asax` file plays a key role in developing optimal ASP.NET applications.

Choosing the Right ASP.NET State Management Option

State management is a vastly more complicated topic in ASP.NET than it is in classic ASP. The choices you need to make are not as clear-cut as before because you now have different options for accomplishing the same task. ASP.NET *does* allow you to manage state in the most optimal way for your Web application. The burden is on you, the developer, to make the right choices on which approach you need to take.

When considering using session state, ask the following questions:

Does the application require centralized session state management, or can it be managed on individual Web servers? ASP.NET provides StateServer and SQLServer modes for centralized session state. ASP.NET provides InProc, StateServer, and SQLServer modes for server-specific session state.

Does the application require cookie-based or cookieless session state? Most Web clients support cookies, so cookie-based session state is a good approach for the vast majority of Web clients. Cookieless session state requires the application to contain relative links only. Also, the application is more vulnerable to losing a session reference because the ID is stored in plain text in the URL, which can be easily tampered with.

What kind of information needs to be stored? The InProc session state mode stores any data type, although you should be careful not to store objects that could present threading issues. The StateServer and SQLServer session state modes can only store objects that support binary serialization. This includes most of the simple data types (string, integer, Boolean) as well as some specialized objects, including the DataSet object.

Does the application really need a Session object for all information? Session state management is typically more expensive than application state management because the server provides every client with its own copy of the same information. You should only store information in session state that is truly specific to an individual client. Technically, the `ap_SalesQueryWithSession.aspx` page presented earlier is *not* a good use of session state and would be better suited for caching. This is because the DataSet contents vary by request parameters, not by individual client.

When considering using application state, ask the following questions:

Does the application require permanent application state? Permanent state values are guaranteed to be available as long as the ASP.NET application remains loaded. You can store permanent state values in the `Web.config` file. This file is suitable for storing configuration values, but it cannot be used to store objects. Permanent state values may also be stored in the HttpApplicationState class, but then they must be compiled with the application, and there is nothing to prevent them from being modified at runtime. Alternatively, you can set up a public shared variable in the `Global.asax` file and initialize it with a reference value or object. This variable is accessible throughout the application; however, it does not provide concurrency management. You should not set shared variables more than once, and they should be primarily read-only for the application to prevent concurrency problems. Often these variables are set once (initialized) in the `Global.asax` file and then are treated as read-only throughout the rest of the application.

Does the application require transient application state? The HttpApplicationState class (the Application object) stores a wide range of objects and data types in memory, and it will persist them until a user alters them or until the application unloads. The Cache class provides more granular control over application data, but it does not guarantee that items will remain persistent in memory. Application code that references cached items must have a contingency for re-creating an item that cannot be retrieved from the cache. The Application object avoids this inconvenience, but it provides none of the storage efficiencies of the Cache class.

How frequently will stored items be updated? You should store reference values used throughout an application in the `Web.config` file because ASP.NET will automatically reload the application when this file changes. You must store reference objects used throughout an application in the Application object. If one or more of these references changes, then you must recompile the application. Alternatively, you can store object references in the Cache class using dependency expiration. For example, a DataSet may be added to the cache, and the cached reference will be used throughout the application as long as its dependency resource remains unchanged.

Does the client need direct access to the item? If the client does not require direct access to an application item, then consider creating a custom base class for the `Global.asax` file and storing the item using a class property. For example, you can store a request counter in a class property and automatically increment in it the `Global.asax` `Application_BeginRequest()` method.

Ultimately, your choice for managing state comes down to the type of item, how it gets accessed, and whether the item must remain persistent or can be recreated. If used correctly, state management is an important factor in developing optimal ASP.NET applications.

Summary

ASP.NET provides new ways to manage session state and application state. In this chapter we discussed session state management using classic ASP and contrasted it with the new capabilities offered by ASP.NET. We reviewed the three modes of managing session state in ASP.NET, which are InProc, StateServer, and SQLServer. Many texts refer to the performance degradation you can expect to see in your application when you manage session state. However, you rarely see performance numbers that back up these statements. To address this issue, we conducted performance stress tests of a Web page using different session state management modes. The results showed that the effect on performance is not clear-cut and that you may actually recognize performance benefits. Next we discussed application state management in ASP.NET. There are effectively two kinds of information that need to be stored at the application level: transient information and permanent information. You can store transient information using the HttpApplicationState class, and you can store permanent information in the `Web.config` file. After that, we discussed the important role that `Global.asax` plays

in ASP.NET applications. This class provides numerous event handlers and allows you to execute code at various points in the request process. We showed how to extend the functionality of the `Global.asax` file by writing a custom base class. Finally, we ended the chapter with design considerations for implementing session and application state management.