Performance Tuning and Optimizing ASP.NET Applications

JEFFREY HASAN WITH KENNETH TU
# Contents at a Glance

About the Authors ................................................................. xi
About the Technical Reviewer ............................................... xi
Acknowledgments ................................................................. xii
Introduction ........................................................................... xiii
Chapter 1 Introducing Performance Tuning and Optimization ..... 1
Chapter 2 Introducing ASP.NET Applications ....................... 21
Chapter 3 Writing Optimized Data Access Code ................. 63
Chapter 4 Optimizing Application and Session State Management ......................................................... 123
Chapter 5 Caching ASP.NET Applications .......................... 167
Chapter 6 Writing Optimized Web Services ....................... 207
Chapter 7 Stress Testing and Monitoring ASP.NET Applications ......................................................... 253
Chapter 8 Debugging and Tracing ASP.NET Applications .... 297
Index .................................................................................. 353
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:

```xml
<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](image-url)

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:

```xml
<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.

![Northwind Sales Query Screen](image)

**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**

```vbscript
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
    End Try
End Sub
```
' 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()

Try
Catch err As Exception
' Report the error in a JavaScript alert
strJSScript = "\n<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>
<thead>
<tr>
<th>CONCURRENT BROWSERS</th>
<th>MODE = OFF</th>
<th>MODE = INPROC</th>
<th>MODE = STATESERVER</th>
<th>MODE = SQLSERVER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.81</td>
<td>4.54</td>
<td>8.27</td>
<td>8.47</td>
</tr>
<tr>
<td>5</td>
<td>28.28</td>
<td>20.25</td>
<td>27.25</td>
<td>29.29</td>
</tr>
<tr>
<td>10</td>
<td>89.38</td>
<td>46.08</td>
<td>77.29</td>
<td>85.11</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>CONCURRENT BROWSERS</th>
<th>MODE = OFF</th>
<th>MODE = INPROC</th>
<th>MODE = STATESERVER</th>
<th>MODE = SQLSERVER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>18.86</td>
<td>24.17</td>
<td>18.31</td>
<td>18.11</td>
</tr>
<tr>
<td>5</td>
<td>21.66</td>
<td>25.74</td>
<td>21.54</td>
<td>21.34</td>
</tr>
<tr>
<td>10</td>
<td>17.23</td>
<td>23.8</td>
<td>18.11</td>
<td>17.6</td>
</tr>
</tbody>
</table>

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](image)

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.

![Graph showing TTLB growth by session state mode](image)

**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.
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](image)

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.
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

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

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

Or, alternatively:

```csharp
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:

```vbnet
Dim objKey As [Object]
For Each objKey In Session.Keys
    Console.WriteLine(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:

```vbnet
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:

\[
\text{Dim sqlDV As DataView} \\
\text{sqlDV = CType(Session("sqlDataView"), DataView)}
\]

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:

\[
\text{sqlDV = CType(Session("sqlDataView"), DataView)} \\
\text{If sqlDV Is Nothing Then} \\
\quad \text{' Recreate the object} \\
\text{End If}
\]

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:

```xml
<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:

```xml
<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:

```vbnet
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:

```xml
<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:

```xml
<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:

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

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:
Chapter 4

- Programatically using the Application object's API

- Using the <object> tag in Global.asax

For example, you can assign a String object programatically:

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

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

```vbnet
<%@ 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:

```vbnet
' 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:

```vbnet
' 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:

```vbnet
' 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:
Chapter 4

- **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:

```vbscript
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
    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("MyOV") ' 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
        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**

<table>
<thead>
<tr>
<th>EVENT HANDLER</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application_Start()</td>
<td>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.</td>
</tr>
<tr>
<td>Application_BeginRequest()</td>
<td>Handles the HttpApplication BeginRequest() event. This is called when a new HTTP request is received by the application.</td>
</tr>
<tr>
<td>Application_EndRequest()</td>
<td>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.</td>
</tr>
<tr>
<td>Application_End()</td>
<td>Called when all HttpApplication instances unload. This occurs when the application is restarted, which may occur manually or when the Web.config file changes.</td>
</tr>
<tr>
<td>Application_Error()</td>
<td>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.</td>
</tr>
<tr>
<td>Session_Start()</td>
<td>Called when a new session is started.</td>
</tr>
<tr>
<td>Session_End()</td>
<td>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.</td>
</tr>
</tbody>
</table>

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**

```csharp
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
    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
        .To = "sysadmin@yourcompany.com"
        .From = "sysadmin@yourcompany.com"
        .Subject = "Exception Report for " & Me.Request.UserHostAddress
    End With

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

```xml
<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

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

```vbnet
objSmtpMail.Send(objMail)
objSmtpMail = Nothing
objMail = Nothing
End Sub
```
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:
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 it in 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 re-created. 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.
Index

Symbols and Numbers
@ Application directive, 161–162
@ OutputCache directive, 169
fragment caching, 176–183
page-level output caching, 171–176
@ Page directive, 30–31
disabling view state, 39
Trace attribute, 42, 332
% Committed Bytes in Use counter, 133
| (pipe symbol), 127
* (wildcard character), 117
128-character cryptographic key, 45

A
Abandon() method, 144
absolute expiration, 188, 190
absoluteExpiration parameter, 188
AcceptChanges() method, 93–94
Accept-Language header, 175
AccessCodeValue property, 243
ACT client processor usage counter, 278
ACT tool, 17–18, 263–295
analyzing test results, 280–282
Application object model, 282–284
customizing tests, 268–273
test duration, 270–271
test load level, 269–270
user groups and users, 271–273
features, 17–18, 263–264
interfaces, 18, 265
Memory counter and, 142
Microsoft newsgroups for, 295
performance counters, 273–274, 278
reports, 18
saving, 278–280
setting the reporting level, 275
using the report interface, 277
running tests, 275–278
checking performance counters, 278
Test Status window, 275, 276
runtime issues, 293–295
NTLM authentication error, 293
Provider load failure error, 294
response code problems, 294–295
test script length limitation, 295
saving tests, 268
scripting in, 17, 289–292
session state performance and,
132–133
setting up tests, 265–268
defining a dynamic test, 267–268
recording browser sessions, 265–267
test architecture, 264
Test object model, 285–292
overview of objects in, 283–289
scripting dynamic tests with, 289–292
See also performance testing; stress testing
Actor property, 223
ACTTrace.log file, 291
Add() method
Cache class, 154, 188, 191
HttpApplicationState class, 151
Session object, 144
StateBag class, 36
AddNew() method, 108
ADO.NET, 63–121
architecture, 65
benefits of using, 67–68
Command object, 66, 117
data access best practices, 116–121
data access design decisions, 110–115
decision flow diagram, 114–115
n-tier applications, 110–112
Web services, 112–114
DataSet class, 87–110
features, 89–91
updating data, 91–96
XML integration, 96–106
DataAdapter class, 85–87
DataView class, 107–108
important objects, 66
.NET managed data providers, 68–87
Command class, 77–80
Connection class, 69–77
DataAdapter class, 85–87
DataReader class, 81–85
overview of, 64–66
ADO.NET (continued)
SqlException class, 108–110
summary of, 121
XML integration, 63, 64, 67–68, 88, 96–106
AllRequests counter, 157
AllUniqueSessions counter, 158
analysis, performance, 15, 280–282
ap_crypt.aspx page, 45
ap_ExpensiveProducts.aspx page, 217
API (application programming interface), 4, 23, 144, 245
application builds. See build configurations
Application Center Test (ACT) tool. See ACT tool
application data caching, 168, 186–196
adding cache items, 190–193
Cache class members, 188–189
Cache object features, 186–187
callback notification, 194–196
expiration policies, 189–190
removing cache items, 194
retrieving cache items, 193–194
@ Application directive, 161–162
application latencies, 255, 256–257
Application object, 123, 147–148
Cache class vs., 153–156
class members, 151
configuring and using, 151–153
Application object model (ACT tool), 282–284
application programming interface (API), 4, 23, 144, 245
Application() property, 32
application settings, 54, 55–56
application state, 123, 147–164
overview of, 147–148
permanent, 148–150
temporary, 148, 150–156
Cache class, 153–156
HttpApplicationState class, 151–153
application tracing, 334–335
Application_BeginRequest() event handler, 157
Application_End() event handler, 157
Application_EndRequest() event handler, 157
Application_Error() event handler, 157, 159–160, 344, 347
Application_Start() event handler, 157
AppRequestQueueLimit attribute, 61
Apps Performance counters, 16
AppSettings element, 55, 148–149
Apress Web site, 29
APSoapHeader class, 242, 244
ap_WSAsynchCustomer1.aspx client page, 217
ap_WSConsumer1.aspx client page, 217
ap_WSGetMostExpensiveProduct.aspx client page, 243
architecture
ACT tool, 264
ADO.NET, 65
ASP.NET, 2–4, 24–48
“Argument not specified” error message, 311
ASP.NET applications, 21–62
architecture, 2–4, 24–48
benefits of, 23–24
caching, 167–205
classic ASP vs., 3–4
configuration system, 48–61
debugging, 297–298, 311–330
exception management, 341–351
helper technologies, 120–121
HTTP handlers, 26–29
n-tier, 110–112
overview of, 21–23
Page class, 30–34
performance testing, 17–18, 253–296
state management, 123–165
tracing, 297–298, 330–341
view state, 34–48
Web services, 112–114, 207–251
ASP.NET runtime engine, 24, 25–26, 49
ASP.NET State Service, 129
ASP.NET worker process, 25, 57–60
ASPState database, 131
Assert() method
Debug class, 322, 323, 325
Trace class, 337
assertions, 322, 323–325
asynchronous Web services
client-side consumers, 230–240
server-side consumers, 225–230
Attach to Process dialog box, 328
authentication, 17
mixed-mode, 76–77
Web services, 240–244
Authentication element, 53
Authorization element, 53
AutoEventWireUp attribute, 31
Autos window, 313, 315
availability
assessing, 8–10
defined, 8
B
bandwidth, 9
base64-encoded string, 43
baseline, 13–14
batch log, 294
BeginGetEmployeeOrders() method, 226
benchmarks, 7–8, 13
best practices
  data access, 116–121
  performance testing, 262–263
  throwing exceptions, 343–344
BindDataGrid() function, 135
binding data, 84–85
Body property, 286, 288
BooleanSwitch class, 324, 339
boot.ini file, 261
Breakpoint Properties dialog box, 318–319
breakpoints, 312, 314, 318, 329
Breakpoints window, 313, 318–320
Browse Performance Counters dialog box, 274
browsers
  recording a browser session, 265–267
  simulating browser compatibility, 289–291
BrowseTest.vbs script, 289
build configurations, 299–311
  characteristics of, 300
  conditional compilation, 310–311
  custom builds, 301–302
  debug builds, 299–302
  environments for, 300
  levels of, 299
  project configurations, 299, 306–309
  release builds, 299, 300, 302
  resolving errors in, 311
  settings for, 302–304
  solution configurations, 299, 304–306
Build tab, 309
BytesReceived property, 288

C
Cache API, 48, 168, 186, 187–189
  Cache class members, 188–189
  Cache object features, 186–187
Cache class, 153–156, 187–189
  explained, 187
  members of, 188–189
Cache object, 186–187
Cache property, 168
CacheDependency object, 188
CacheDuration property, 196, 210, 249
CacheItemPriority enumeration value, 188
caching, 120, 167–205
  application data, 168, 186–196
  adding cache items, 190–193
Cache class members, 188–189
Cache object features, 186–187
callback notification, 194–196
expiration policies, 189–190
removing cache items, 194
retrieving cache items, 193–194
ISA Server, 203–204
output, 168, 169–186
  enabling, 169–171
  explained, 169
  fragment, 176–183
HttpCachePolicy object, 183–186
  page-level, 171–176
overview of, 168–169
summary of, 205
transient application state and, 153–156
view state vs., 48
Web farm, 201–203
Web services, 196–201
  proxy browser interface for, 196–198
  Web application client for, 198–200
Call Stack window, 313
callback function, 235
callbacks
  notification, 194–196
  validation, 187
callService() method, 235, 237
camel-case convention, 52
capacity, 257–258
Catch statement, 120
centralized data access functions, 118–119
classic ASP
  ASPNET vs., 3–4
  session state management in, 125
Clear() method, 144
Click() event handler, 229–230
Client-Activated mode, 246
ClientConnectedCheck attribute, 59
ClientFaultCode, 223
clients
  cached pages on, 170
  Web application, 198–200
client-side consumers, 230–240
  consuming the Web service, 232–234
  exception handling for, 238–240
  implementing the WebService behavior, 234–238
  overview of the WebService behavior, 231–232
  using the WebService behavior, 232, 233, 234–238
client-side invocation, 250
Index

client-side scripts, 22
debugging, 325–330
Web services and, 250
Close() method, 73, 283, 287
CloseConnection property, 78
code efficiency, 257
code modules, 22
Code property, 223, 239
code-behind files, 21, 30
coding conventions, 250
Command class, 77–80
CommandBehavior enumeration members, 78–79
ExecuteXmlReader() method, 80
command mode, 321
Command object, 66, 117
Command window, 313, 320–321
CommandText property, 117
CommandTimeout property, 79, 112
CommandType property, 79, 117
Committed Bytes in Use counter, 133
Common Language Runtime (CLR), 58
Common properties, 307
compilation
conditional, 310–311
exceptions, 297
just-in-time, 23, 250
Compilation element, 53
compiled code, 4, 23, 250
complex data types, 249
complex stored procedures, 117–118
component calls, 4
Component Designer tool, 104–105
concurrency management, 154, 187
conditional compilation, 310–311
<configSections> tags, 51
<configuration> node, 51
configuration implementation section, 50
Configuration Manager, 303–304, 308–309
Configuration properties, 307
configuration section handler (CSH) section, 50
configuration setting, 50
configuration system (ASP.NET), 48–61
benefits of, 49–50
custom elements, 54–57
application settings, 54, 55–56
custom configuration settings, 54, 56–57
Machine.config file, 50–52
optimal configuration, 57–61
ASP.NET worker process, 57–60
HTTP runtime engine, 60–61
session state configuration, 127
Web.config file, 52–53
ConfigurationSettings class, 54, 71
Connect Timeout/Connection Timeout parameter, 71
connection bandwidth, 9
Connection class, 69–77
closing connections, 73
connection pooling, 73–75
ConnectionString property, 70–73
trusted connections, 75–77
Connection Lifetime parameter, 74, 75
Connection object
ACT Application object model, 286–287
ADO.NET, 66, 85
connection pooling, 73–75, 118
Connection Reset parameter, 74
ConnectionString property, 70–73
important parameters, 71–72
storing connection strings, 70
consuming Web services, 216–240
client-side consumers, 230–240
consumption types, 217
examples of, 216–217
server-side consumers with asynchronous calls, 225–230
synchronous server-side consumers, 217–225
Contains() method, 337
Contents property, 151
Control class, 33
Control Tree trace output section, 333
Controller object, 283, 284
Controls property, 32
cookies, 48, 124, 125, 126, 162
Cookies Collection trace output section, 334
Cookies property, 288
counters, 5–6, 16–17, 262
checking, 278
setting up, 273–274
See also specific counters
CreateConnection() method, 285
CreateRequest() method, 285
cross-process calls, 249
cryptographic key, 45–46
custom base class, 160–162, 164
custom build configuration, 301–302
custom configuration elements, 54–57
application settings, 54, 55–56
custom configuration settings, 54, 56–57
System.Configuration namespace, 54–55
custom configuration settings, 54, 56–57
custom error pages, 349–351
Custom Errors element, 53, 160
custom log file, 346
customized expiration, 154, 187
data access
  best practices, 116–121
  centralized functions, 118–119
  design decisions, 110–115
    decision flow diagram, 114–115
  n-tier applications, 110–112
  Web services, 112–114
  .NET Framework namespaces involved in, 64–65
Data Access Application Block, 119
data binding, 84–85
Data Cache engine, 4
Data Manipulation Language (DML), 117
Data Source/Server parameter, 71
data types
  complex, 249
  SQL, 83–84, 118
  Web service-supported, 215
DataAdapter class, 85–87
  methods, 86
  properties, 85, 86
DataAdapter object, 66, 89
database efficiency, 257
DataColumn objects, 89
DataGrid controls
  data binding and, 85
  DataSets updates and, 94–96
  view state and, 41
DataReader class, 81–85
  accessing data, 82–84
  data binding, 84–85
DataReader object, 66, 81–82
DataRow objects, 89
DataSet class, 87–110
  data access, 112
  DiffGram format, 106
  features, 89–91
  overview, 87–89
  typed DataSets, 104–106
  updating data, 91–96
  validation using XSD schemas, 101–104
  XML integration with, 96–106
DataSet object, 64, 66, 90–106, 116, 249
DataTable object, 89
DataView class, 107–108
DataView object, 66, 67, 107, 116
Date object, 188
DCOM technology, 244, 294
dcomcnfg.exe utility, 294
Debug.Assert() method, 323, 325
debug builds, 299–302
Debug class, 298, 322–323
Debug.Listeners collection, 325, 337
debug mode, 312
debugger windows, 312–321
  Autos window, 315
  Breakpoints window, 318–320
  Command window, 320–321
  functionality overview of, 313–314
  Locals window, 315–316
  QuickWatch dialog box, 317–318
  Watch window, 316–317
debugging, 297–298, 311–330
  assertions for, 323–325
  client-side script, 325–330
  setting the debug mode, 312
  summary of, 352
  VS.NET tools for, 311–325
    debugger windows, 312–321
    programmatic tools, 321–325
    Task List window, 311
Debugging tab, 308
DebugWindows2.aspx file, 314
decision flow diagram, 114–115
declarative code, 169, 171, 183
default error page, 350
Default property, 79
default Service Description file, 212–213
defaultRedirect parameter, 350, 351
DefaultTraceListener, 337
delegate function, 188
Delete() method, 108
DeleteCommand property, 86
dependencies parameter, 188
dependency expiration, 190, 191–193
Deployment tab, 309
Description property, 210
Detail property, 223
detecting exceptions, 342
DictionarySectionHandler class, 54
DiffGram format, 67, 106
Disassembly window, 313
distributed caching, 204
Distributed Component Object Model (DCOM), 244, 294
<div> tag, 234
downtime, 10
DropDownList controls, 237
Duration attribute
  fragment caching, 176
  page-level output caching, 172
Duwamish sample site, 266, 282
Dynamic Host Configuration Protocol (DHCP), 147
dynamic tests, 267–268

E
Edit and Continue options window, 320
embedded timestamps, 230
Enable attribute, 58
Index

Enabled property, 335, 339
EnableSession property, 210
EnableViewState attribute, 31, 38
EnableViewStateMac attribute, 31, 44
encoding, 43
encryption, 44
EndGetEmployeeOrders() method, 226, 230
Enlist parameter, 72
error pages, 349–351
errorDetail object, 238–239
errors
  build, 311
  HTTP, 133
  NTLM authentication, 293
  provider load failure, 294
  SQL, 108–110
See also exception handling

ErrorSample.aspx page, 347, 349
event logging, 298
Event Properties dialog box, 347, 348
Event Viewer, 29, 347
exception handling, 341–351
  appropriate use of, 119–120
  custom error pages, 349–351
  detecting exceptions, 342
  filtering exceptions, 343
  logging exception information, 345–348
  logical/semantic exceptions, 297
  managing unhandled exceptions, 159–160, 344–345
  notification process, 348–349
  SoapException class, 222–225
  SqlException class, 108–110
  syntax/compilation exceptions, 297
  throwing exceptions, 343–344
  Web services, 222–225, 238–240
See also errors

ExecuteNonQuery() method, 77, 79
ExecuteReader() method, 77, 78, 79
ExecuteScalar() method, 77, 79
ExecuteXmlReader() method, 77, 80

ExecutionTimeout attribute, 61
expiration
  absolute, 188, 190
  cache policies, 189–190
  customized, 154, 187
  dependency, 190, 191–193
  sliding, 170, 184, 188, 190, 191
explicit casting, 82, 146
extensible configuration, 49
Extensible Markup Language. See XML
Extensible Schema Definition. See XSD

F
Fail() method
  Debug class, 322
  Trace class, 337
federated security, 240
Fill() method, 86, 105, 106
filtering exceptions, 343
FindControl() method, 32
FOR XML clause, 80
forward caching, 203
fragment caching, 48, 176–183
  @ OutputCache attributes for, 176–177
  VaryByControl attribute for, 181–183
  VaryByParam attribute for, 178–180

G
Generate Users dialog box, 272, 273
GenerateDataSet() method, 155, 200
Get() method, 155, 189
GetBoolean() method, 83
GetChanges() method, 93, 96
GetConfig() method, 55, 149
GetCurrentUser() method, 285
GetCustomerList() method, 216, 227
GetCustomerOrders() method, 217, 227
GetDateTimeStamp() method, 196, 197
GetEmployeeOrders() method, 226
GetEmployeeSales() method, 216, 218, 221–222, 224
GetEnumenator() method, 189
GetErrors() method, 93
GetMostExpensiveProducts() method, 242
GetNextUser() method, 285
GetSqlDouble() method, 83
GetSqlMoney() method, 83
GetString() method, 83
GetXml() method, 97
GetXmlSchema() method, 97, 100, 103
Global.asax file, 22, 156–162, 163
  Application_Error event handler, 159–160, 346
  custom base class for, 160–162, 164
  event handlers accessible in, 157
  purposes served by, 156
Globalization element, 53

H
HandleOrderDetails() method, 230
Handler factory classes, 26
Handler processing classes, 26
handlers, HTTP, 26–29
hardware issues
  availability and, 9
  latency and, 257
  scalability and, 12
HasControls() method, 32
headers
  Accept-Language, 175
  SOAP, 241–244, 249
Headers Collection trace output section, 334
helper technologies, 120–121
hierarchical caching, 261
HTML (Hypertext Markup Language), 21, 167
HTML Control (HTC) file, 231
HTTP (Hypertext Transfer Protocol), 22, 207
  errors count, 133, 278
  handlers, 26–29, 335
  response codes, 133, 281
  runtime engine, 60–61
  security, 43–44
HttpApplicationState class, 151–153
  class members, 151
  configuring and using, 151–153
HttpCachePolicy object, 169, 183–186
  class members, 184–185
  example of using, 185
<httpRuntime> configuration element, 60–61
HTTPS (HTTP Secure), 44
HTTPVersion property, 286

I
ICorrelationSectionHandler class, 54
IDictionary interface, 36
IdleTimeout attribute, 58
IEnumerable interface, 36
IEXPLORE.EXE process, 327
IHttpHandler interface, 26
  immediate mode, 321
  implicit casting, 146
  implied exceptions, 222
InferXmlSchema() method, 98
information
  permanent, 148–150
  transient, 148, 150–156
Initial Catalog/Database parameter, 71
Initialize() function, 235
Initialize lifecycle stage, 33
InnerException property, 223
InnerXML() method, 101
InProc mode, 127, 138, 139, 146, 163
Insert() method, 154, 189, 191
InsertCommand property, 86
instances, Web service, 235
instrumentation, 298
integrated development environment (IDE), 265, 298
integrated security, 76
Integrated Security/Trusted_Connection parameter, 71
Internet Explorer
  client-side script debugging with, 325–330
  simulating compatibility with, 289
  WebService behavior used with, 232
Internet Information Server (IIS), 2, 24, 76, 169
Internet Protocol (IP) redirectors, 201, 203
Internet Security and Acceleration (ISA) server, 170, 203–204
interpreted code, 4
IP redirectors, 201, 203
ISA Server, 170
  caching with, 203–204
  information resources, 204
ISerializable interface, 37
IsOpen() method, 287
IsPostBack() property, 32
IsReadOnly() property, 144
IsReusable() property, 26
Item property, 36
ItemRemovedCallback() function, 195

J
JavaScript, 237
just-in-time (JIT) compilation, 23, 250

K
keep-alive connections, 293
Kelvin, Lord, 254
key name parameter, 188
Keys property, 36, 144
key-value assignment, 191

L
latency, 254–257
  application, 255, 256–257
  network, 255–256
  user load vs., 255, 281
  utilization vs., 259–260
  See also response time
Listeners collection
  Debug class, 325, 337
  Trace class, 337–338
Index

ListMostExpensiveProducts() method, 216, 233–234
load, 9, 11
testing, 260–261, 269
See also user load
loading, 15
LoadPostData() method, 33
LoadViewState() method, 33
local area network (LAN), 9
localOnly attribute, 335
Locals window, 313, 315–316
Location attribute, 172
Lock() method, 148, 151, 152
logging exception information, 345–348
logical/semantic exceptions, 297

M
Machine Authentication Check (Mac), 31
Machine.config file, 48–49, 50–52
code example, 51
sections in, 50
machine key-based encryption, 44
<machineKey> element, 44
MakeIE60GETRequest() function, 289–290
managed code, 4
managed provider objects, 64
management
  concurrency, 154, 187
  exception, 341–351
  memory, 154, 187
  state, 123
Max Pool Size parameter, 74
MaxIOThreads attribute, 60
maxRequestLength attribute, 61
MaxWorkerThreads attribute, 60
measuring performance, 8–12
  availability, 8–10
  scalability, 10–12
  view state, 41–43
memory
  cache management and, 154, 187
  performance counter for, 262
  scalability and, 142–143
Memory window, 313
MemoryLimit attribute, 59
Message property, 223
MessageName property, 210
metrics. See performance metrics
Microsoft newsgroups, 295
Min Pool Size parameter, 74
MinFreeThreads attribute, 61
MinLocalRequestFreeThreads attribute, 61
mixed-mode authentication, 76–77
modes
  .NET remoting, 246
  session state, 127–132, 138–144
Modules window, 313
monitoring performance, 15–17
  Performance Monitor for, 16–17
  steps involved in, 15
  unattended monitoring, 341–342, 351
See also performance testing
MustUnderstandFaultCode, 223
MyCounter property, 161, 162

N
Name property, 210, 288
Namespace property, 210
namespaces
  .NET Framework, 64–65
  See also specific namespaces
NameValueSectionHandler class, 54
.NET Framework
  asynchronous method invocation, 229–230
  data typing system, 83–84
  helper technologies, 121
  namespaces involved in data access, 64–65
  remote object invocation, 244–248
  scalability features, 11–12
  Software Development Kit, 250, 298
  thread management capabilities, 11
  Web service supported data types, 215
.NET managed data providers, 68–87
Command class, 77–80
  CommandBehavior enumeration members, 78–79
  ExecuteXmlReader() method, 80
Connection class, 69–77
  closing connections, 73
  connection pooling, 73–75
  ConnectionString property, 70–73
  trusted connections, 75–77
DataAdapter class, 85–87
  methods, 86
  properties, 86
DataReader class, 81–85
  accessing data, 82–84
  data binding, 84–85
.NET remoting, 244–248
  ASP.NET Web services vs., 246–248
  highlights of, 245–246
  information resources, 248
  network latencies, 9, 255–256
New() constructor, 77
New() method, 87
newsgroups, 295
NextResult() method, 117
NoAbsoluteExpiration field, 189
NoSlidingExpiration field, 189
notification
callback, 194–196
exception, 348–349
n-tier Web applications, 110–112
NTLM authentication error, 293
NUMPROC parameter, 261

O
ODBC .NET data provider, 69
Off mode, 127, 128, 138, 351
OLE DB.NET data provider, 68
On mode, 351
onProductListResult() callback function, 236, 237
onRemoveCallback parameter, 188
Open() method, 283
optimization, 1
ASP.NET configuration, 57–61
HTTP runtime engine, 60–61
worker process, 57–60
Web service design, 248–250
Oracle .NET data provider, 68
output caching, 168, 169–186
enabling, 169–171
explained, 169
fragment, 176–183
HttpCachePolicy object, 183–186
page-level, 171–176
Web service, 249
Output window, 313
@ OutputCache directive, 169
fragment caching, 176–183
page-level output caching, 171–176
OutputCacheModule, 170

P
Packet Size parameter, 72
Page.Cache property, 183
Page class, 30–34
@ Page directive, 30–31
lifecycle stages, 33–34
members, 31–32
@ Page directive, 30–31
disabling view state, 39
Trace attribute, 42, 332
Page object, 30, 168
Page_Disposed() event, 34
Page_Error event, 344
Page_Init event, 33
page-level output caching, 48, 171–176
@ OutputCache attributes, 172–173
VaryByHeader attribute, 174–175
VaryByParam attribute, 173–174
page-level tracing, 332–333
enabling with @ Page directive, 332
interpreting output of, 333–334
Page_Load() event, 33
pageOutput attribute, 335
Page_PreRender() event, 34
PageTest.vbs script, 291
Page_Unload() event, 34
partial page-level caching. See fragment caching
Pascal-case convention, 52
Password property, 288
Password/Pwd parameter, 71
Path property, 286
percentiles, 282
performance, 1
benchmarks, 7–8
counters, 5–6, 16–17, 262
measuring, 8–12
metrics, 4–5, 254–260
monitoring, 15–17
profiling, 13–15
testing, 17–18, 253–296
view state considerations, 39–41
performance metrics, 4–5, 254–260
latency, 254–257
relationships between, 259–260
throughput, 257–258
utilization, 258–259
Performance Monitor (PerfMon), 5–7, 16–17
ACT tool integration, 17
graphical view, 7
selection screen, 6
performance testing, 17–18, 253–296
ACT tool for, 263–295
analyzing results of, 280–282
approaches to, 260–261
best practices for, 262–263
counters for, 262, 273–274, 278
customizing tests, 268–273
dynamic tests and, 267–268
metrics for, 4–5, 254–260
overview of, 253–254
reporting level for, 275
running performance tests, 275–278
saving tests and reports, 268, 278–280
setting up tests, 265–268
summary of, 296
See also ACT tool; stress testing
performance tuning, 254, 263, 282
permanent application state, 148–150, 163
pipe symbol (|), 127
Platform setting, 304
pooling, connection, 73–75, 118
Pooling parameter, 74
POST operations, 230–231, 257
priority parameter, 188
process independence, 126
Processes dialog box, 327
<processModel> configuration element, 58–60
Processor object, 17
processor utilization, 133, 262
ProcessRequest() method, 26
profiling performance, 13–15
programmatic code, 169, 171, 183
programmatic debug tools, 321–325
Programmer Database files, 300
programming
  debug tools for, 321–325
  session state, 144–146
  using good sense in, 119
project build configurations, 299, 306–309
Project Contexts setting, 304
project dependencies, 306
Project object, 283
propagating exceptions, 343–344
Properties dialog box, 269, 270
property pages
  project configuration, 306–308
  solution configuration, 304–306
protected configuration settings, 50
“Provider load failure” error message, 294
proxy browser interface, 196–198
proxy classes, 211–215
proxy servers, 170

Q
QueryProducts() method, 98–99
QuickWatch dialog box, 313, 317–318

R
RaisePostBackEvent() method, 34
Raw property, 239
Read() method, 82
ReadXml() method, 97
ReadXmlSchema() method, 97
Registers window, 313
relational database, 346
release builds, 299, 300, 302
remote object invocation, 244–248
ASP.NET Web services, 245, 246–248
.NET remoting technology, 244–248
remote scripting, 231
RemoteOnly mode, 351
Remove() method
  Cache class, 155, 189
  HttpApplicationState class, 151
  StateBag class, 36
RemoveAll() method, 151
Render() method, 34
reports, 18
  saving, 278–280
  setting the reporting level, 275
  using the report interface, 277
request bytes out total, 5
Request Details trace output section, 334
Request object, 286
Request() property, 32
RequestLimit attribute, 59, 335
RequestQueueLimit attribute, 59
Requests/Sec counter, 133
resource utilization, 258–259
response codes, 133
  runtime problems, 294–295
  test results, 281
Response object, 288–289
Response() property, 32
response time, 5, 254
  baseline performance, 14
  measuring, 254–257
See also latency
Response.Write() method, 194
ResponseDeadlockInterval attribute, 60
responseRestartDeadlockInterval attribute, 60
RestartQueueLimit attribute, 59
ResultCode property, 288
resultsets, 117–118
return exceptions, 222
reverse caching, 204
RNGCryptoServiceProvider class, 45
Running Documents window, 313
RunQueryReturnDS() wrapper function, 136
runtime engine, 24, 60–61
runtime issues, 293–295
  NTLM authentication error, 293
  Provider load failure error, 294
  response code problems, 294–295
  test script length limitation, 295

S
SalesQueryCounter, 158
SaveViewState() method, 34
scalability
assessing, 10–12
defined, 8, 10–11
dimensions, 12
horizontal, 261
memory usage and, 142–143
session state and, 138–144
testing, 261
vertical, 261
scaling up/out, 12
scavenging, 168, 187
ScheduleTaskRunScript1.vbs file, 276, 284
SchemaOnly property, 78
scripts
ACT tool, 17, 289–292, 295
client-side, 22, 250, 325–330
remote, 231
section handlers, 51–52
Secure Sockets Layer (SSL) protocol, 17
SecureNorthwind.asmx Web service, 241–242
security
federated, 240
integrated, 76
view state, 43–44
Web services, 240–244
Select() method, 315
SelectCommand property, 86
Send() method, 287
SendRequest() method, 285, 290
SequentialAccess property, 78
serialization, 37, 145, 247
server farms
implementing view state in, 44–46
See also Web farms
Server Variables trace output section, 334
ServerFaultCode, 223
servers
ISA, 170, 203–204
proxy, 170
See also SQL Server
server-side consumers, 217–230
asynchronous Web method calls, 225–230
consuming the Web service, 220–222, 226–229
exception handling for, 222–225
synchronous Web method calls, 217–225
Service Description file, 212–213
session ID, 124
Session objects, 123, 144, 163
Session() property, 32
session state, 123, 124–147
ASP.NET management of, 126
classic ASP management of, 125
configuring and using, 127
managing in Web farms, 125, 126, 146–147
modes, 127–132, 138–144
InProc mode, 127
Off mode, 127, 128
SQLServer mode, 128–130
StateServer mode, 127–128
overview of, 124
performance analysis, 132–133
programming with, 144–146
sample Web page with, 133–136
scalability and, 138–144
stress testing with, 136–144
analyzing results of, 138–144
steps in process of, 136–137
view state vs., 46–48
Session variables, 46–48
Session_End() event handler, 157
SessionID property, 144
Session_Start() event handler, 157
SessionState element, 53, 127
Set() method, 151
SetCacheability() method, 183, 184
SetExpires() method, 183, 184
SetLastModified method, 184
SetNoServerCaching method, 185
SetSlidingExpiration method, 184
SetValidUntilExpires method, 184
Shared attribute, 177, 178
shared caching, 177
ShutdownTimeout attribute, 59
Simple Object Access Protocol. See SOAP
simulation, 17
SingleCall mode, 246
SingleResult property, 79
SingleRow property, 79
Singleton mode, 246
sliding expiration, 170, 184, 188, 190, 191
slidingExpiration parameter, 188
SOAP (Simple Object Access Protocol)
exception handling, 222–225, 238–240
headers, 241–244, 249
security extensions, 240–244
Web service communications, 22, 112, 196, 207, 208
SoapException class
exception handling using, 222–224
properties used in constructing, 223
raising a SOAP exception server fault
code, 224–225
SoapHeader base class, 241
SoapHeaderAttribute(), 241
Index

Software Development Kit (SDK), 250, 298
software issues, 9
solution build configurations, 299, 304–306
SQL data types, 118
SQL Server
  caching mechanism in, 202
data typing system, 83–84
SQL data types used with, 118
trusted connections, 75–77
XML functionality, 80
SQL Server.NET data provider, 68
SqlException class, 108–110
SQLServer mode, 128–130, 138, 146, 163
SSL protocol, 17
StartTest() method, 284
startup projects, 306
state management, 4, 123–165
  application state, 123, 147–162
  considerations for choosing, 162–164
  Global.asax file, 156–162
  session state, 123, 124–147
  summary of, 164–165
StateBag class, 36
StaticObjects property, 151
steady-state, 255
sticky sessions, 201
StopTest() method, 284
stored procedures, 117–118
stress testing, 15, 17, 260–261
  ACT tool for, 132, 263–295
  analyzing results of, 138–144, 280–282
customizing tests, 268–273
  dynamic tests and, 267–268
  performance counters for, 273–274, 278
  reporting level for, 275
  running performance tests, 275–278
  saving tests and reports, 268, 278–280
  session state, 136–144
  setting up tests, 265–268
  See also performance testing
String object, 152
synchronous Web services, 217–225
syntax/compilation exceptions, 297
System.Configuration namespace, 54–55
System.Data.Common namespace, 64
System.Data namespace, 2, 64
System.Data.OleDb namespace, 64
System.Data.SqlClient namespace, 64
System.Data.SqlTypes namespace, 64, 83, 118

System.Diagnostics namespace, 324, 346
System.Security.Cryptography namespace, 45
System.Web.Caching namespace, 187
System.Web namespace, 2, 183
System.Web.Services namespace, 209–211
System.Web.UI namespace, 33
System.Xml namespace, 64
System.Xml.Schema namespace, 65
System.Xml/Xsl namespace, 65

T
Task List window, 311
TCP/IP (Transmission Control Protocol/Internet Protocol), 207
tempdb database, 131
TemplateControl class, 33
Test object
  ACT Application object model, 283, 284
  ACT Test object model, 285
Test object model (ACT tool), 285–292
  Connection object, 286–287
  Request object, 286
  Response object, 288–289
  scripting dynamic tests with, 289–292
  configuring tests for multiple pages, 291–292
  simulating browser compatibility, 289–291
Test object, 285
User object, 287–288
test scripts, 289–292
  configuring for multiple pages, 291–292
  length limitation for, 295
  simulating browser compatibility, 289–291
Test Status window, 275, 276
Test.Trace() method, 291
testing. See performance testing; stress testing
TestIsRunning property, 284
This (Me) window, 314
thread blocking, 162
Threads window, 314
thread-safe code, 9
throughput, 5, 8, 257
  baseline performance, 14
  measuring, 257–258
  user load vs., 280
throwing exceptions, 119–120, 343–344
Time to First Byte (TTFB) counter, 255
Time to Last Byte (TTLB) counter, 133, 138, 139–142, 255
Timeout attribute, 58
Timeout property, 144, 248
TimeSpan object, 188
timestamps, embedded, 230
Trace attribute, 42, 332
Trace.axd handler, 335–336
Trace class, 298, 336–341
    enabling, 336–337
    listeners, 337–338
    methods, 337
    trace switching, 339–341
Trace element, 53
Trace Information trace output section, 334
Trace.Listeners collection, 337–338
Trace() method, 285
TraceContext class, 298, 331–336
    application tracing, 334–335
    enabling, 332
    interpreting trace output, 333–334
    methods and properties, 331–332
    page-level tracing, 332–334
    Trace.axd handler, 335–336
TraceLevelSwitch, 340–341
traceMode attribute, 335
TraceSwitch class, 339–340
tracing, 297–298, 330–341
    application, 334–335
    attributes, 335
    enabling, 42–43
    page-level, 332–334
    interpreting output, 333–334
    summary of, 352
    tools for, 330–341
    Trace class, 336–341
    TraceContext class, 331–336
Transaction Cost Analysis (TCA), 133, 261
TransactionOption property, 210
Transact-SQL statements, 117
transient application state, 148, 150–156, 163
    Cache class, 153–156
    HttpApplicationState class, 151–153
Triple DES encryption algorithm, 44
trusted connections, 75–77
TTLB property, 288
tuning performance, 254, 263, 282
typed accessor methods, 82–84
typed DataSets, 66, 104–106

U
unattended monitoring, 341–342, 351
unhandled exceptions, 159–160, 344–345
Uniform Resource Identifiers (URIs), 27, 199
Uniform Resource Locators (URLs), 124, 212
Universal Discovery, Description, and Integration (UDDI), 209
Unlock() method, 148, 151, 152
Update() method, 86, 87, 96
UpdateCommand property, 86
UpdateDSWithDataGrid.aspx sample projects, 94
updating data, 91–96
uptime, 9–10
user controls, 22
user groups, 271–273
User ID parameter, 71
user interface (UI), 265
user load, 9, 11
    latency vs., 255, 281
    testing, 260–261, 269
    throughput vs., 258, 280
    utilization vs., 259
User object, 287–288
UserHostAddress property, 28
users and user groups, 271–273
useService() method, 235
utilization, 258–259
    latency vs., 259–260
    scaling options, 259

V
validation
    callbacks, 187
    DataSet, 101–104
validation key, 44–45
VaryByControl attribute, 177, 181–183
VaryByCustom attribute
    fragment caching, 177
    page-level output caching, 173
VaryByHeader attribute, 173, 174–175
VaryByHeaders method, 184
VaryByParam attribute
    fragment caching, 176, 178–180
    page-level output caching, 172, 173–174
VaryByParams method, 185
VBScript, 17, 265, 284
Verb property, 286
VersionMismatchFaultCode, 223
vertical scalability, 261
view state, 34–48
alternatives to, 48
disabling, 38–39
how it works, 35–37
implementing in server farms, 44–46
measuring performance cost, 41–43
performance considerations, 39–41
persisting across multiple pages, 37–38
security considerations, 43–44
session state vs., 46–48
virtual private network (VPN), 9
Visual Studio .NET (VS.NET), 23
ACT tool, 132, 263
client-side script debugging, 325–330
code editor, 106
Component Designer tool, 104–105
debugging tools, 311–325
shell integration, 265
trace log, 42
Web service features, 207, 209

W
Warn() method, 332, 333
Watch window, 314, 316–317
Web application client, 198–200
Web Application Stress (WAS) tool, 263
Web applications
data access design for, 110–114
n-tier applications, 110–112
performance counters for, 262
Web browsers
recording a browser session, 265–267
simulating browser compatibility, 289–291
Web.config file, 22, 48–49, 52–53
configuration elements, 53
enabling application tracing in, 334–335
permanent information in, 148–150
sensitive information and, 150
session state modes and, 128, 129, 130
storing connection strings in, 70
Web farms
 caching in, 201–203
session state management in, 125, 126, 146–147
See also server farms
Web forms, 21
Web reference, 220–222
Web server processor usage counter, 278
Web service method name, 235
Web service method parameters, 236
Web services, 22, 207–251
building using ASP.NET, 209–216
genrating a proxy class with
WSDL, 211–215
System.Web.Services namespace
members, 210–211
caching, 196–201
proxy browser interface for, 196–198
Web application client for, 198–200
consumers of, 216–240
client-side, 230–240
eamples of, 216–217
server-side, asynchronous, 225–230
server-side, synchronous, 217–225
data access design, 112–114
design optimization, 248–250
exception handling, 222
SoapException class, 222–225
WebService behavior, 238–240
.NET remoting vs. ASP.NET, 244–248
choosing between, 246–248
overview of features, 245–246
security and authentication, 240–244
setting Web references to, 220–222
summary of, 251
supported data types, 215–216
technology overview, 208–209
Web Services Description Language
(WSDL), 207, 208
genrating a proxy class using, 211–215
WebMethodAttribute class, 210
WebService behavior, 231–232
exception handling, 238–240
implementing, 234–238
overview of, 231
using, 232, 233, 234–238
WebService class, 209, 211
WebServiceAttribute class, 210
WebServiceBindingAttribute class, 211
wildcard character (*), 117
Windows event log, 345–346
wrapper functions, 118–119
Write() method
Debug class, 322
Trace class, 337
WriteEntry() method, 28
WriteIf() method
Debug class, 322
Trace class, 337
WriteLine() method
Debug class, 322
Trace class, 337
WriteLineIf() method
  Debug class, 322
  Trace class, 337, 340
WriteTrace() subroutine, 339
WriteXml() method, 98
WriteXmlSchema() method, 98
WS-Security specification, 240

X
XML (Extensible Markup Language), 22
  ADO.NET integration with, 63, 64, 67–68, 88, 96–106
  ASP.NET configuration files and, 49
  DataSet integration with, 88, 96–106
    generating from a DataSet, 98–106
    serialization, 88
  Web Services Description Language, 207
  XML data document, 64
  XML-MSDATA namespace, 208
  XmlReader object, 66, 80
  XmlValidatingReader class, 103–104
  XPATH queries, 98, 101
  XSD (Extensible Schema Definition), 68, 88
    DataSet validation using, 101–104
    Web service supported data types, 215