10
State Management

BEFORE WE BEGIN discussing state management in ASP.NET, let's get one thing straight: Attempting to manage state in Web applications goes against the fundamental design principles of the Web. One of the primary goals of the Web and its underlying protocol, HTTP, is to provide a scalable medium for sharing information. Adding user state inherently reduces scalability because the pages shown to a particular user will be different from those shown to another user and thus cannot be reused or cached.

In spite of this fact, many applications deployed on the Web require user-specific state to function properly. Applications ranging from e-commerce shopping sites to local company intranet sites depend on the ability to track individual requests from distinct users and store state on behalf of each client, whether it's items in a shopping cart or which days were selected on a calendar as requested vacation days. Although maintaining client-specific state is not officially part of the HTTP protocol, there is a proposal in place for adding state management to HTTP. RFC 2109[14] defines a proposed standard for state management for HTTP also known as cookies. Although it is only a proposed standard and not yet an official part of the HTTP specification, cookies are in widespread use today in almost all browsers, and many Web sites rely on cookies for their functionality.

14. See http://www.w3.org/Protocols/rfc2109/rfc2109.

As a consequence, Web programmers must be very conscious about state management. Unlike traditional applications, Web applications must be very explicit about any state that is maintained on behalf of a client, and there is no one standard way to maintain that state.

10.1 Types of State

One of the most important decisions you face when designing a Web application is where to store your state. ASP.NET provides four types of state: application state, session state, cookie state, and view state. In this chapter, we explore each type of state, when it is most applicable, and any disadvantages you should be aware of if you decide to make use of it.

ASP.NET, like its predecessor, ASP, provides a pair of objects for managing application-level state and session-level state. Application state is where information that is global to the application may be stored. For efficiency, this state is typically stored once and then read from many times. Session state is maintained on a per-client basis. When a client first accesses any page in an application, an ASP.NET generated session ID is created. That session ID is then transmitted between the server and the client via HTTP either using client-side cookies or encoded in a mangled version of the URL (URL mangling is discussed in detail later in this chapter). On subsequent accesses by the same client, state associated with that session ID may be viewed and modified. Cookies provide the ability to store small amounts of data on a client's machine. Once a cookie is set, all subsequent pages accessed by the same client will transmit the cookie and its value.

Finally, view state is a yet another way of storing state on behalf of a client by saving and restoring values from a hidden field when a form is posted. Although this technique for retaining state has been used by Web developers in the past, ASP.NET provides a simplified mechanism for taking advantage of it. As we have seen in Chapter 8, it is possible to place items into the `ViewState` property bag available in every `Page`-derived class. When that page issues a `POST` request to itself, the values placed in the `ViewState` property bag can then be retrieved, the key restriction being that view state works only when a page posts to itself. Table 10-1 summarizes the

advantages and disadvantages of each of the four types of state available in ASP.NET.

TABLE 10-1: State Type Comparison in ASP.NET

Type of State	Scope of State	Advantages	Disadvantages
Application	Global to the application	• Shared across all clients	• Overuse limits scalability • Not shared across multiple machines in a Web farm or processors in a Web garden • Primary purpose subsumed by data cache in ASP.NET
Session	Per client	• Can configure to be shared across machines in a Web farm and processors in a Web garden	• Requires cookies or URL mangling to manage client association • Off-host storage can be inefficient
Cookie	Per client	• Works regardless of server configuration • State stored on client • State can live beyond current session	• Limited memory (~4KB) • Clients may not support cookies or may explicitly disable them • State is sent back and forth with each request
View	Across `POST` requests to the same page	• Works regardless of server configuration	• State is retained only with `POST` request made to the same page • State is sent back and forth with each request

10.2 Application State

Application state is something that should be used with care, and in most cases, avoided altogether. Although it is a convenient repository for global data in a Web application, its use can severely limit the scalability of an application, especially if it is used to store shared, updateable state. It is also an unreliable place to store data, because it is replicated with each application instance and is not saved if the application is recycled. With this warning in mind, let's explore how it works.

Application state is accessed through the `Application` property of the `HttpApplication` class, which returns an instance of class `HttpApplicationState`. This class is a named object collection, which means that it can hold data of any type as part of a key/value pair. Listing 10-1 shows a typical use of application state. As soon as the application is started, it loads the data from the database. Subsequent data accesses will not need to go to the database but will instead access the application state object's cached version. Data that is prefetched in this way must be static, because it will not be unloaded from the application until the application is recycled or otherwise stopped and restarted.

Listing 10-1: Sample Use of Application State for Data Prefetching

```
    // Inside of global.asax
void Application_Start(object src, EventArgs e)
{
  DataSet ds = new DataSet();
  // population of dataset from ADO.NET query not shown

  // Cache DataSet reference
  Application["FooDataSet"] = ds;
}

      // In some page within the application
private void Page_Load(object src, EventArgs e)
{
  DataSet ds = (DataSet)(Application["FooDataSet"]);
  // ...
  MyDataGrid.DataSource = ds;
  // ...
}
```

Because it is likely that multiple clients will be serviced by the same application, there is a potential for concurrent access to application state. The `HttpApplicationState` class protects access to its collection of objects with an instance of the `HttpApplicationStateLock` class, a derivative of the `ReadWriteObjectLock` class. This class provides two alternate mechanisms for locking, one for reading and one for writing. Multiple reader locks may be acquired simultaneously, but to acquire a writer lock, all other locks must be released first. This type of locking mechanism is particularly useful for protecting state in the application state bag because it allows multiple readers to pass through concurrently, and restricts access only when a request tries to write to the state bag. The general usage model of application-level state is to update it infrequently and read it frequently, so concurrent readers are a common occurrence.

In traditional ASP, it was always on the shoulders of the developer to call `Lock` and `Unlock` on the application object whenever it was modified or accessed. In ASP.NET, however, these calls are made implicitly for you whenever you insert items into or read items from the state bag in the form of either `AcquireWrite()` or `AcquireRead()`, depending on whether an item is being inserted or accessed. There is typically no need to explicitly call `Lock()` and `UnLock()` when working with the application state bag. These methods do exist, however, and internally calling the `Lock()` method acquires a writer lock on the internal `HttpApplicationStateLock` class. It is important to note that making explicit calls to `Lock()` and `UnLock()` defeats the multiple-reader efficiency of this new locking mechanism and should therefore be avoided in most cases.

The one case in which you still need to explicitly call the `Lock()` and `UnLock()` methods on the application state bag is when you are updating a shared piece of state. For example, Listing 10-2 shows a sample page that uses shared, updateable application state. In this example, each time the page is accessed, the string identifying the client browser type (`Request.Browser.Browser`) is used as an index into the `HttpApplicationState` collection, where a count is maintained to keep track of how many times this page was accessed with each client browser type. The page then renders a collection of paragraph elements displaying the browser names

along with how many times each browser was used to access this page. These statistics continue to accumulate for the lifetime of the application. Note that before the value in the application state bag is retrieved and updated, `Application.Lock()` is called, and once the update is complete, `Application.UnLock()` is called. This acquires a writer lock on the application state bag and guarantees that the value will not be read while the update is being performed. If we did not take care to call `Lock`, a potential race condition would exist, and the value keeping track of the number of browser hits for a particular browser type would not necessarily be correct.

Listing 10-2: Sample Use of Application State

```
<%@ Page Language='C#' %>
<script runat='server'>
private void Page_Load(object sender, System.EventArgs e)
{
  Application.Lock();

  // Modify a value in the HttpApplicationState collection
  if (Application[Request.Browser.Browser] != null)
    Application[Request.Browser.Browser] =
              (int)Application[Request.Browser.Browser] + 1;
  else
    Application[Request.Browser.Browser] = 1;

  Application.UnLock();

  // Print out the values in HttpApplicationState
  // to show client browser access statistics
  for (int i=0; i<Application.Count; i++)
    Response.Output.Write("<p>{0} : {1} hits</p>",
                Application.GetKey(i), Application[i]);
}
</script>
```

In almost every scenario that would have used application state in a traditional ASP application, it makes more sense to use the data cache in ASP.NET, discussed in Chapter 9. The most common need for application state is to provide a share point for accessing global, read-only data in an application. By placing global, read-only data in the data cache instead of in application state, you gain all the benefits of cache behavior, with the same ease of access provided by application state. Probably the most compelling

advantage of the data cache over application state is memory utilization. If the memory utilization of the ASP.NET worker process approaches the point at which the process will be bounced automatically (the recycle limit), the memory in the data cache will be scavenged, and items that have not been used for a while will be removed first, potentially preventing the process from recycling. If, on the other hand, data is stored exclusively in application state, ASP.NET can do nothing to prevent the process from recycling, at which point all of the application state will be lost and must be restored on application start-up.

The one feature of application state that cannot be replaced by the data cache is the ability to have shared updateable state, as shown earlier in Listing 10-2. Arguably, however, this type of state should not be used at all in a Web application, because it inherently limits scalability and is unreliable as a mechanism for storing meaningful data. In the previous example, we were using application state to save statistics on browser type access. This information is maintained only as long as the application is running, and it is stored separately in each instance of the application. This means that when the process recycles, the data is lost. It also means that if this application is deployed in a Web farm (or a Web garden), separate browser statistics will be kept for each running instance of the application across different machines (or CPUs). To more reliably collect this type of statistical information, it would make more sense to save the data to a central database and avoid application state altogether.

10.3 Session State

Maintaining state on behalf of each client is often necessary in Web applications, whether it is used to keep track of items in a shopping cart or to note viewing preferences for a particular user. ASP.NET provides three ways of maintaining client-specific state: session state, cookie state, and view state. Each technique has its advantages and disadvantages. Session state is the most flexible and, in general, the most efficient. ASP.NET has enhanced session state to address some of the problems associated with it in previous versions of ASP, including the abilities to host session state out of process (or in a database) and to track session state without using cookies.

Session state is maintained on behalf of each client within an ASP.NET application. When a new client begins to interact with the application, a new session ID (or session key) is generated and associated with all subsequent requests from that same client (either using a cookie or via URL mangling). By default, the session state is maintained in the same process and `AppDomain` as your application, so you can store any data type necessary in session state. If you elect to house session state in another process or in a database, however, there are restrictions on what can be stored, as we will discuss shortly. Session state is maintained in an instance of the `HttpSessionState` class and is accessible through the `Session` property of both the `Page` and `HttpContext` classes. When a request comes in to an application, the `Session` properties of the `Page` and `HttpContext` class used to service that request are initialized to the current instance of `HttpSessionState` that is associated with that particular client. Listing 10-3 shows the primary methods and properties of the `HttpSessionState` class, along with the property accessors in both the `Page` and `HttpContext` classes.

Listing 10-3: HttpSessionState Class

```
public sealed class HttpSessionState : ICollection,
                                        IEnumerable
{
    // properties
  public int CodePage {get; set;}
  public int Count {get;}
  public bool IsCookieless {get;}
  public bool IsNewSession {get;}
  public bool IsReadOnly {get;}
  public KeysCollection Keys {get;}
  public int LCID {get; set;}
  public SessionStateMode Mode {get;}
  public string SessionID {get;}
  public HttpStaticObjectsCollection StaticObjects {get;}
  public int Timeout {get; set;}
    // indexers
  public object this[string] {get; set;}
  public object this[int] {get; set;}
    // methods
  public void Abandon();
  public void Add(string name, object value);
```

```
  public void Clear();
  public void Remove(string name);
  public void RemoveAll();
  public void RemoveAt(int index);
  //...
}

public class Page : TemplateControl, IHttpHandler
{
  public virtual HttpSessionState Session {get;}
  //...
}

public sealed class HttpContext : IServiceProvider
{
  public HttpSessionState Session {get;}
  //...
}
```

Because the `HttpSessionState` class supports string and ordinal-based indexers, it can be populated and accessed using the standard array access notation that most developers are familiar with from traditional ASP. There are some new properties, however, including flags for whether the session key is being maintained with cookies or with mangled URLs (`IsCookieless`) and whether the session state is read-only (`IsReadOnly`). Also note that although the `CodePage` property is accessible through session state, this is for backward compatibility only. The proper way to access the response's encoding is through `Response.ContentEncoding.CodePage`.

For an example of using session state, let's consider an implementation of the classic shopping cart for a Web application. As a user navigates among the pages of an application, she selects items to be retained in a shopping cart for future purchase. When the user is done shopping, she can navigate to a checkout page, review the items she has collected in her cart, and purchase them. This requires the Web application to retain a collection of items the user has chosen across request boundaries, which is exactly what session state provides. Listing 10-4 shows the definition of a class called `Item`. Instances of this class are used to represent the selected items in our shopping cart.

Listing 10-4: Item Class

```
public class Item
{
  private string _description;
  private int    _cost;

  public Item(string description, int cost)
  {
    _description = description;
    _cost = cost;
  }

  public string Description
  {
    get { return _description; }
    set { _description = value; }
  }
  public int Cost
  {
    get { return _cost; }
    set { _cost = value; }
  }
}
```

To store `Item` instances on behalf of the client, we initialize a new `ArrayList` in session state and populate the `ArrayList` with items as the client selects them. If you need to perform one-time initialization of data in session state, the `Session_Start` event in the `Application` class is the place to do so. Listing 10-5 shows a sample handler for the `Session_Start` event in our application object, which in our case is creating a new `ArrayList` and adding it to the session state property bag indexed by the keyword `"Cart"`.

Listing 10-5: Initializing Session State Objects

```
// in global.asax
public class Global : System.Web.HttpApplication
{
  protected void Session_Start(Object sender, EventArgs e)
  {
    // Initialize shopping cart
    Session["Cart"] = new ArrayList();
  }
}
```

A sample page that uses the shopping cart is shown in Listings 10-6 and 10-7. In this page, two handlers are defined: one for purchasing a pencil and another for purchasing a pen. To keep things simple, the items and their costs have been hard-coded, but in a real application this information would normally come from a database lookup. When the user elects to add an item to her cart, the `AddItem` method is called. This allocates a new instance of the `Item` class and initializes it with the description and cost of the item to be purchased. That new item is then added to the `ArrayList` maintained by the `Session` object, indexed by the string `"Cart"`. Listings 10-8 and 10-9 show a sample page that displays all the items in the current client's cart along with a cumulative total cost.

LISTING 10-6: Session State Shopping Page Example

```
<!— File: Purchase.aspx —>
<%@ Page language="c#" Codebehind="Purchase.aspx.cs"
         Inherits="PurchasePage" %>

<HTML>
  <body>
    <form runat="server">
      <p>Items for purchase:</p>
      <asp:LinkButton id=_buyPencil runat="server"
                      onclick="BuyPencil_Click">
           Pencil ($1)</asp:LinkButton>
      <asp:LinkButton id=_buyPen runat="server"
                      onclick="BuyPen_Click">
           Pen ($2)</asp:LinkButton>
      <a href="purchase.aspx">Purchase</a>
    </form>
  </body>
</HTML>
```

LISTING 10-7: Session State Shopping Page Example—Code-Behind

```
// File: Purchase.aspx.cs
public class PurchasePage : Page
{
  private void AddItem(string desc, int cost)
  {
    ArrayList cart = (ArrayList)Session["Cart"];
```

continues

```
      cart.Add(new Item(desc, cost));
    }

    // handler for button to buy a pencil
    private void BuyPencil_Click(object sender, EventArgs e)
    {
      // add pencil ($1) to shopping cart
      AddItem("pencil", 1);
    }

    // handler for button to buy a pen
    private void BuyPen_Cick(object sender, EventArgs e)
    {
      // add pen ($2) to shopping cart
      AddItem("pen", 2);
    }
  }
```

LISTING 10-8: Session State Checkout Page Example

```
<!— File: Checkout.aspx —>
<%@ Page language="c#" Codebehind="Checkout.aspx.cs"
         Inherits="CheckoutPage" %>
<HTML>
 <body>
 <form runat="server">
   <asp:Button id=Buy runat="server" Text="Buy"/>
   <a href="purchase.aspx">Continue shopping</a>
 </form>
 </body>
</HTML>
```

LISTING 10-9: Session State Checkout Page Example—Code-Behind

```
// File: Checkout.aspx.cs
public class CheckOutPage : Page
{
  private void Page_Load(object sender, System.EventArgs e)
  {
      // Print out contents of cart with total cost
      // of all items tallied
    int totalCost = 0;

    ArrayList cart = (ArrayList)Session["Cart"];
    foreach (Item item in cart)
    {
      totalCost += item.Cost;
```

```
        Response.Output.Write("<p>Item: {0}, Cost: ${1}</p>",
                              item.Description, item.Cost);
      }

      Response.Write("<hr/>");
      Response.Output.Write("<p>Total cost: ${0}</p>",
                            totalCost);
    }
  }
```

The key features to note about session state are that it keeps state on behalf of a particular client across page boundaries in an application, and that the state is retained in memory on the server in the default session state configuration.

10.3.1 Session Key Management

To associate session state with a particular client, it is necessary to identify an incoming request as having been issued by a given client. A mechanism for identifying a client is not built into the essentially connectionless HTTP protocol, so client tracking must be managed explicitly. In traditional ASP, this was always done by setting a client-side cookie with a session key on the first client request. This technique is still supported in ASP.NET (in fact, it is the default technique) and is demonstrated in Figure 10-1.

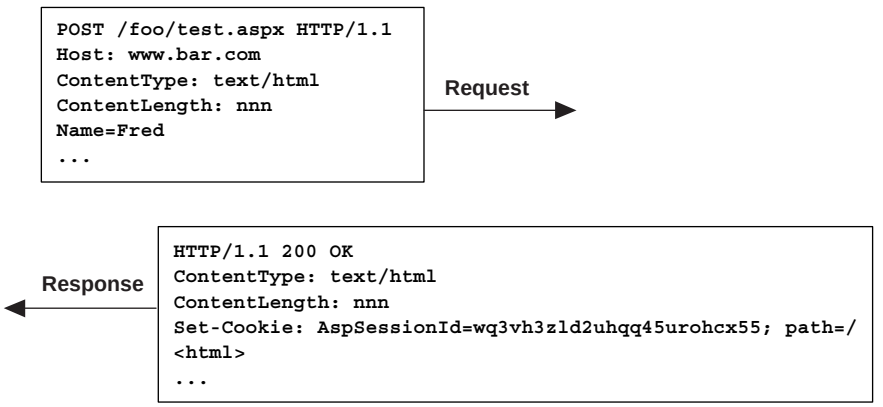

FIGURE 10-1: Session Key Maintained with Cookies

Because session keys are used to track clients and maintain potentially sensitive information on their behalf, they must not only be unique, they must also be next to impossible to guess. This has been a problem in the past when programmers used Globally Unique Identifiers (GUIDs) as session keys. Because the original algorithm for generating GUIDs was deterministic, if you knew one of the GUIDs generated by a server machine, you could guess subsequent GUIDs and thus access the session state associated with another client. Although GUIDs are no longer generated this way, ASP.NET takes the precaution of generating its own session keys by using the cryptographic service provider and its own encoding algorithm. Listing 10-10 shows some pseudocode demonstrating the technique used by ASP.NET for creating session keys.

Listing 10-10: Session Key Generation in ASP.NET

```
// Generate 15-byte random number using the crypto provider
RNGCryptoServiceProvider rng =
                          new RNGCryptoServiceProvider();
byte[] key = new byte[15];
rng.GetBytes(key);

// Encode the random number into a 24-character string
// (SessionId is a private class - not accessible)
string sessionKey = SessionId.Encode(key);
```

Using cookies to track session state can be problematic. Clients can disable cookie support in their browsers, and some browsers do not support cookies. As an alternative to using cookies, ASP.NET also supports a technique called URL mangling to track session keys without using client-side cookies. This technique works by intercepting the initial request made by a client, inserting the session key into the URL, and redirecting the client to the original page requested. When this page receives the request, it extracts the encoded session key from the request URL and initializes the current session state pointer to the correct block of memory. This technique is demonstrated in Figure 10-2. This technique works even with clients that have disabled cookie support in their browsers. On any subsequent navigation, either via anchor tags or explicit programmatic redirections,

ASP.NET will alter the target URL to embed the session key as well. This implicit URL mangling works only for relative URLs, however, so care must be taken with all links in an application using cookieless session key management to avoid absolute URLs.

Controlling whether cookies or URL mangling is used to manage your session keys (along with several other session state–related features) is performed through the `sessionState` element in your application's `web.config` file. Table 10-2 lists the various configuration settings available for the `sessionState` element of `web.config`. Listing 10-11 shows a sample `web.config` file that enables cookieless session key management for an application.

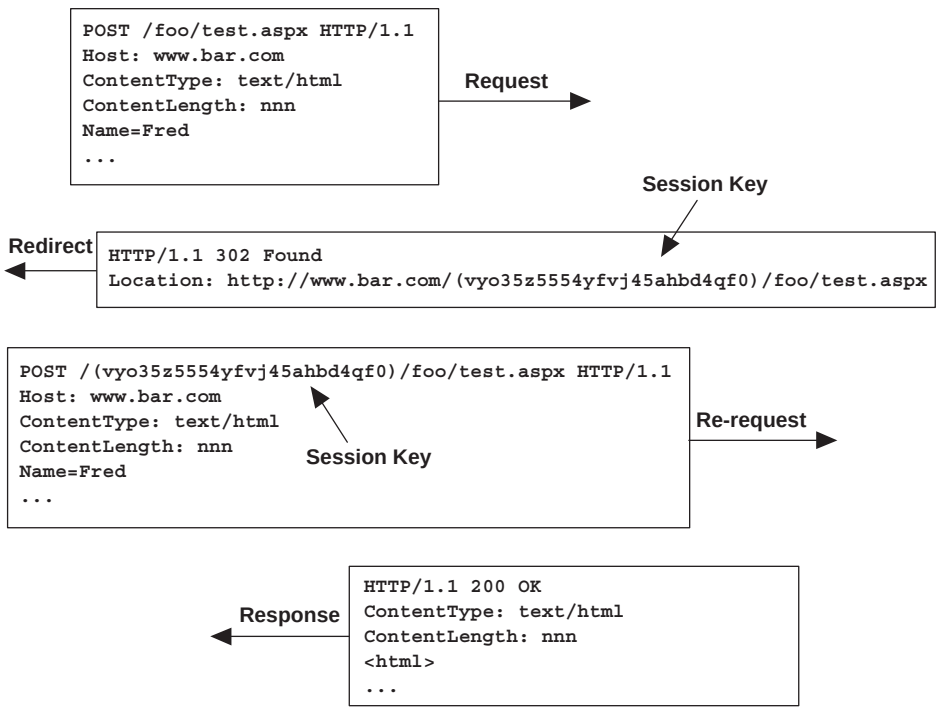

FIGURE 10-2: Session Key Maintained with URL Mangling

TABLE 10-2: sessionState Attributes

Attribute	Possible Values	Meaning
`cookieless`	`True, False`	Pass `SessionID` via cookies or URL mangling
`mode`	`Off, InProc, SQLServer, StateServer`	Where to store session state (or whether it is disabled)
`stateConnection-String`	Example: `'192.168.1.100:42424'`	Server name and port for `StateServer`
`sqlConnection-String`	Example: `'server=192.168.1.100; uid=sa;pwd='`	`SQLServer` connection string excluding database (tempdb is implied)
`timeout`	Example: `40`	Session state timeout value (in minutes)

LISTING 10-11: Sample web.config File Enabling Cookieless Session Key Management

```
<configuration>
  <system.web>
    <sessionState cookieless="true" />
  </system.web>
</configuration>
```

The choice of whether to use cookie-based or mangled URL–based session key management must be made at the application level. It is not possible to specify that the application should use cookie-based management if the client supports cookies, and otherwise default to mangled URL–based management. The trade-offs to consider when making this decision include efficiency, universal client support, and dealing with relative URLs. Cookies are more efficient because they avoid the redirection necessary to perform the URL mangling, although only one redirection per session will occur with URL mangling. Mangled URLs work with clients that don't have cookies enabled (or that don't support them). The mangled URL technique requires

that your application avoid absolute URLs so that the mangling can take place properly. Finally, URL mangling also prevents easy bookmarking and thus may be an inconvenience for your users.

10.3.2 Storing Session State out of Process

In addition to requiring cookies to track session state, traditional ASP only supported the notion of in-process session state. Confining session state to a single process means that any application that relies on session state must always be serviced by the same process on the same machine. This precludes the possibility of deploying the application in a Web farm environment, where multiple machines are used to service requests independently, potentially from the same client. It also prevents the application from working correctly on a single machine with multiple host processes, sometimes referred to as a Web garden. If session state is tied to the lifetime of the Web server process, it is also susceptible to disappearing if that process goes down for some reason. To build traditional ASP applications that scale to Web farms and/or maintain persistent client-specific state, developers must avoid session state altogether and rely on other techniques for tracking client-specific state. The most common approach is maintaining client-specific state in a database running on a network-accessible server. To distinguish one client's state from another, the table (or tables) used to store state is indexed by the session key, as shown in Figure 10-3.

ASP.NET introduces the ability to store session state out of process, without resorting to a custom database implementation. The `sessionState` element in an ASP.NET application's `web.config` file controls where session state is stored (see Table 10-2). The default location is in-process, as it was in traditional ASP. If the `mode` attribute is set to `StateServer` or `SqlServer`, however, ASP.NET manages the details of saving and restoring session state to another process (running as a service) or to an SQL Server database installation. This is appealing because it is possible to build ASP.NET applications that access session state in the normal way, and then by switching the `sessionState` mode in a configuration file, that same application can be deployed safely in a Web farm environment.

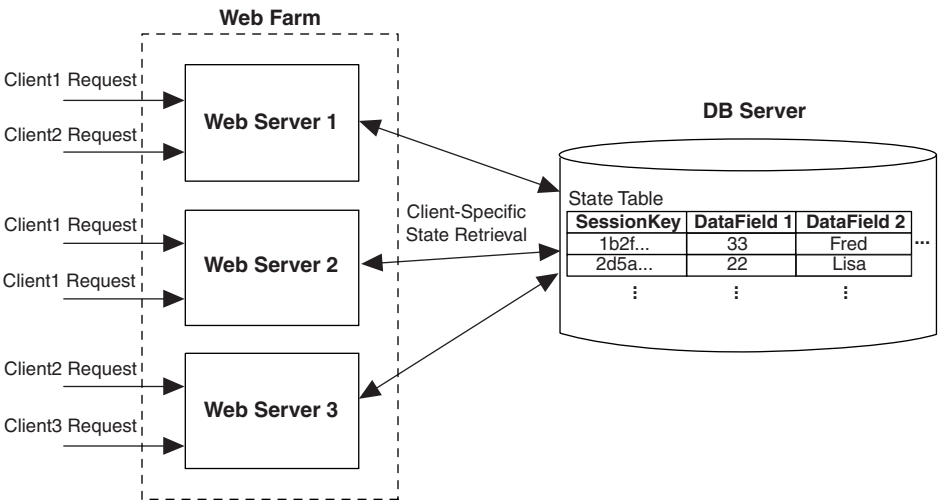

FIGURE 10-3: Maintaining Client-Specific State in a Web Farm Deployment

Whenever out-of-process session state is specified, it is also important to realize that anything placed into session state is serialized and passed out of the ASP.NET worker process. Thus, any type that is stored in session state must be serializable for this to work properly. In our earlier session state example, we stored instances of a locally defined `Item` class, which, if left in its existing form, would fail any attempts at serialization. The `ArrayList` class we used to store the instances of the Item class does support serialization, but since our class does not, the serialization will fail. To correct this, we would need to add serialization support to our class. Listing 10-12 shows the `Item` class correctly annotated to support serialization, which is now compatible with storage in out-of-process session state.

LISTING 10-12: Adding Serialization Support to a Class

```
[Serializable]
public class Item
{
  private string _description;
  private int    _cost;
  // ...
}
```

For session state to be transparently housed out of process, ASP.NET must assume that a page has all of its session state loaded before the page is loaded, and then flushed back to the out-of-process state container when the page completes its processing. This is inefficient when a page may not need this level of state access (although it is somewhat configurable, as we will see), so there is still a valid case to be made for implementing your own custom client-specific state management system, even with ASP.NET.

The first option for maintaining session state out of process is to use the `StateServer` mode for session state. Session state is then housed in a running process that is distinct from the ASP.NET worker process. The `StateServer` mode depends on the ASP.NET State Service to be up and running (this service is installed when you install the .NET runtime). By default the service listens over port 42424, although you can change that on a per-machine basis by changing the value of the `HKLM\System\CurrentControlSet\Services\aspnet_state\Parameters\Port` key in the registry. Figure 10-4 shows the ASP.NET State Service in the local machine services viewer.

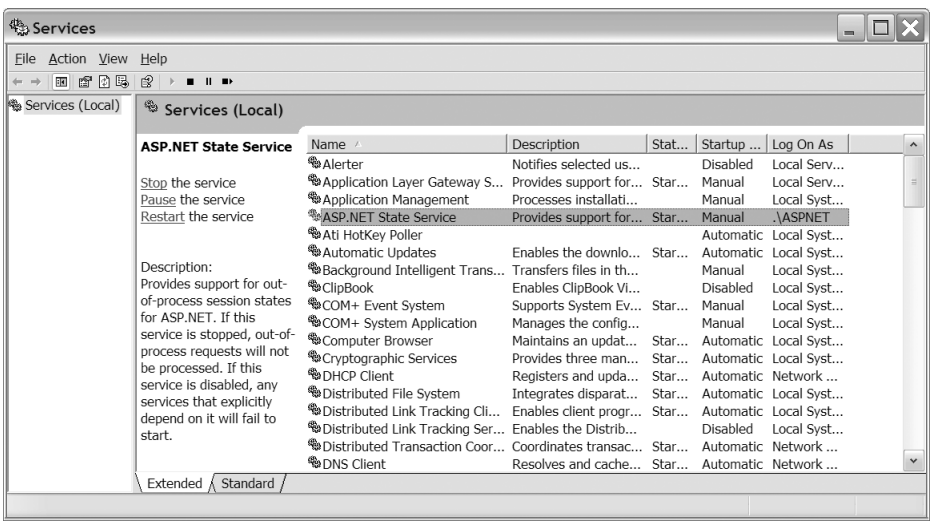

FIGURE 10-4: The ASP.NET State Service

The State Service can run either on the same machine as the Web application or on a dedicated server machine. Using the State Service option is useful when you want out-of-process session state management but do not want to have to install SQL Server on the machine hosting the state. Listing 10-13 shows an example `web.config` file that changes session state to live on server 192.168.1.103 over port 42424, and Figure 10-5 illustrates the role of the state server in a Web farm deployment scenario.

LISTING 10-13: web.config File Using State Server

```
<configuration>
  <system.web>
    <sessionState mode="StateServer"
       stateConnectionString="192.168.1.103:42424"
    />
  </system.web>
</configuration>
```

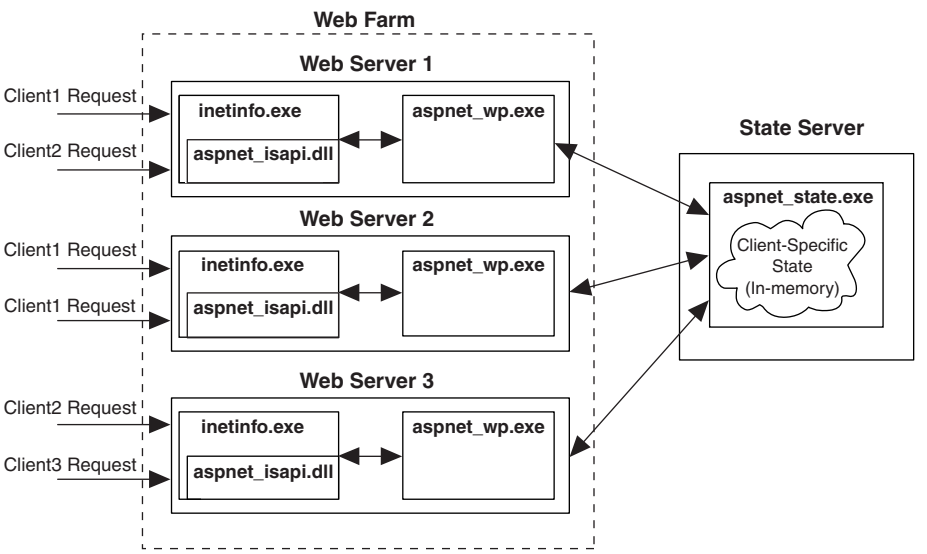

FIGURE 10.5: Using a State Server in a Web Farm Deployment

The last option for storing session state outside the server process is to keep it in an SQL Server database. ASP.NET supports this through the `SQLServer` mode in the `sessionState` configuration element. Before using this mode, you must run the `InstallSqlState.sql` script on the database server where session state will be stored. This script is found in the main Microsoft.NET directory.[15] The primary purpose of this script is to create a table that can store client-specific state indexed by session ID in the tempdb of that SQL Server installation. Listing 10-14 shows the `CREATE` statement used to create the table for storing this state. The ASP state table is created in the tempdb database, which is not a fully logged database, thus increasing the speed of access to the data. In addition to storing the state indexed by the session ID, this table keeps track of expiration times and provides a locking mechanism for exclusive acquisition of session state. The installation script also adds a job to clean out all expired session state daily.

LISTING 10-14: ASPStateTempSession Table

```
CREATE TABLE tempdb..ASPStateTempSessions (
  SessionId        CHAR(32) NOT NULL PRIMARY KEY,
  Created          DATETIME NOT NULL DEFAULT GETDATE(),
  Expires          DATETIME        NOT NULL,
  LockDate         DATETIME        NOT NULL,
  LockCookie       INT             NOT NULL,
  Timeout          INT             NOT NULL,
  Locked           BIT             NOT NULL,
  SessionItemShort VARBINARY(7000) NULL,
  SessionItemLong  IMAGE           NULL,
)
```

Listing 10-15 shows a sample `web.config` file that has configured session state to live in an SQL Server database on server 192.168.1.103. Notice that the `sqlConnectionString` attribute specifies a data source, a user ID, and a password but does not explicitly reference a database, because ASP.NET assumes that the database used will be tempdb.

15. On most 1.0 installations, this should be C:\WINNT\Microsoft.NET\Framework\v1.0.3705.

Listing 10-15: web.config File Using SQL Server

```
<configuration>
  <system.web>
    <sessionState mode="SQLServer"
     sqlConnectionString=
        "data source=192.168.1.103;user id=sa;password=" />
  </system.web>
</configuration>
```

Both the state server and the SQL Server session state options store the state as a byte stream—in internal data structures in memory for the state server, and in a `VARBINARY` field (or an `IMAGE` field if larger than 7KB) for SQL Server. While this is space-efficient, it also means that it cannot be modified except by bringing it into the request process. This is in contrast to a custom client-specific state implementation, where you could build stored procedures to update session key–indexed data in addition to other data when performing updates. For example, consider our shopping cart implementation shown earlier. If, when the user added an item to his cart, we wanted to update an inventory table for that item as well, we could write a single stored procedure that added the item to his cart in a table indexed by his session key, and then updated the inventory table for that item in one round-trip to the database. Using the ASP.NET SQL Server session state feature would require two additional round-trips to the database to accomplish the same task: one to retrieve the session state as the page was loaded and one to flush the session state when the page was finished rendering.

This leads us to another important consideration when using ASP.NET's out-of-process session state feature: how to describe precisely the way each of the pages in your application will use session state. By default, ASP.NET assumes that every page requires session state to be loaded during page initialization and to be flushed after the page has finished rendering. When you are using out-of-process session state, this means two round-trips to the state server (or database server) for each page rendering. You can potentially eliminate many of these round-trips by more carefully designing how

each page in your application uses session state. The session manager then determines when session state must be retrieved and stored by querying the current handler's session state requirements. There are three options for a page (or other handler) with respect to session state. It can express the need to view session state, to view and modify session state, or no session state dependency at all. When writing ASP.NET pages, you express this preference through the `EnableSessionState` attribute of the `Page` directive. This attribute defaults to `true`, which means that session state will be retrieved and saved with each request handled by that page. If you know that a page will only read from session state and not modify it, you can save a round-trip by setting `EnableSessionState` to `readonly`. Furthermore, if you know that a page will never use session state, you can set `EnableSessionState` to `false`. Internally, this flag determines which of the tagging interfaces your `Page` class will derive from (if any). These tagging interfaces are queried by the session manager to determine how to manage session state on behalf of a given page. Figure 10-6 shows the various values of `EnableSessionState` and their effect on your `Page`-derived class.

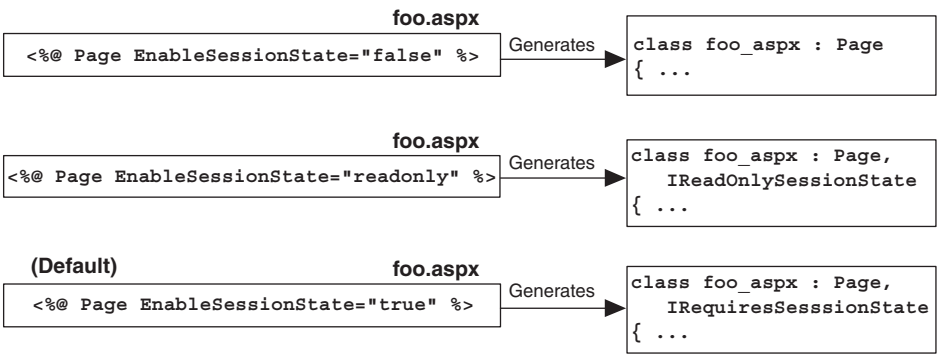

FIGURE 10-6: Indicating Session State Serialization Requirements in Pages

10.4 Cookie State

Although not part of the HTTP specification (yet), cookies are often used to store user preferences, session variables, or identity. The server issues a `Set-Cookie` header in its response to a client that contains the value it wants to store. The client is then expected to store the information associated with the URL or domain that issued the cookie. In subsequent requests to that URL or domain, the client should include the cookie information using the `Cookie` header. Some limitations of cookies include the fact that many browsers limit the amount of data sent through cookies (only 4,096 bytes are guaranteed) and that clients can potentially disable all cookie support in their browser.

ASP.NET provides an `HttpCookie` class for managing cookie data. Listing 10-16 shows the `HttpCookie` class definition and the cookie collection properties exposed by the request and response objects. Note that the request and response objects both expose the collection of cookies through the `HttpCookieCollection` type, which is just a type-safe derivative of the `NameObjectCollectionBase` class, designed for storing `HttpCookie` class instances. Each cookie can store multiple name/value pairs, as specified by RFC 2109, which are accessible through the `Values` collection of the `HttpCookie` class or indirectly through the default indexer provided by the class.

Listing 10-16: The HttpCookie Class

```
public sealed class HttpCookie
{
  public string              Domain           {get; set;}
  public DateTime            Expires          {get; set;}
  public bool                HasKeys          {get;     }
  public string              this[string key] {get; set;}
  public string              Name             {get; set;}
  public string              Path             {get; set;}
  public string              Secure           {get; set;}
  public string              Value            {get; set;}
  public NameValueCollection Values           {get;     }
  //...
}

public sealed class HttpRequest
{
  public HttpCookieCollection Cookies {get;}
```

```
  //...
}

public sealed class HttpResponse
{
  public HttpCookieCollection Cookies {get;}
  //...
}
```

To request that a client set a cookie, add a new `HttpCookie` instance to the response cookie collection before your page rendering. To access the cookies that the client is sending with any given request, access the `Cookies` collection property of the request object. Listing 10-17 shows an example of a page that sets and uses a cookie named `"Age"`. If the cookie has not been set, the page adds the cookie to the `Response.Cookies` collection with a value from a field on the form (`ageTextBox`). If the cookie has been set, the current value is pulled from the `Request.Cookies` collection and is used instead.

LISTING 10-17: Using Cookies in ASP.NET

```
protected void Page_Load(Object sender, EventArgs E)
{
  int age = 0;
  if (Request.Cookies["Age"] == null)
  {
    // "Age" cookie not set, set with this response
    HttpCookie ac = new HttpCookie("Age");
    ac.Value = ageTextBox.Text;
    Response.Cookies.Add(ac);
    age = Convert.ToInt32(ageTextBox.Text);
  }
  else
  {
    // use existing cookie value...
    age = Convert.ToInt32(Request.Cookies["Age"].Value);
  }
  // use age to customize page
}
```

Although cookies are typically used to store user-specific configuration information and preferences, they can be used to store any client-specific state needed by an application (as long as that state is converted to string

form). It is interesting to contrast our earlier shopping cart implementation using session state with an equivalent implementation using only cookies. The major change in our implementation is the population and retrieval of the shopping cart contents from cookies instead of directly from session state. This can be done by converting the contents of the shopping cart into string form so that it can be sent back as cookies to the client and later restored on subsequent requests. To facilitate this, we have added two new functions to our `Item` class: `HydrateArrayListFromCookies` and `SaveArrayListToCookies`. The first function is called from within the `Load` event handler of our shopping `Page` class, and the second function is called from within the `PreRender` event handler. The implementation of these two functions is shown in Listing 10-18. The rest of our code remains the same because we have changed only how the `ArrayList` is persisted. Listing 10-19 shows the cookie-based implementation of our shopping cart application.

Listing 10-18: Item Class with Cookie Serialization Support

```
public class Item
{
  public static ArrayList HydrateArrayListFromCookies()
  {
    int itemCount=0;
    HttpCookie itemCountCookie =
          HttpContext.Current.Request.Cookies["ItemCount"];
    if (itemCountCookie != null)
      itemCount = Convert.ToInt32(itemCountCookie.Value);
    else
    {
      itemCountCookie = new HttpCookie("ItemCount");
      itemCountCookie.Value = "0";
      HttpContext.Current.Response.Cookies.Add(
                                      itemCountCookie);
    }

    ArrayList cart = new ArrayList();
    for (int i=0; i<itemCount; i++)
    {
      HttpCookieCollection cookies =
         HttpContext.Current.Request.Cookies;

      int cost = Convert.ToInt32(
                  cookies[i.ToString()+"cost"].Value);
```

```
      string desc = cookies[i.ToString()+"desc"].Value;
      cart.Add(new Item(desc, cost));
    }

    return cart;
  }

  public static void SaveArrayListToCookies(ArrayList cart)
  {
    // Save array size first
    HttpCookie itemCountCookie =
                          new HttpCookie("ItemCount");
    itemCountCookie.Value = cart.Count.ToString();
    HttpCookieCollection cookies =
          HttpContext.Current.Response.Cookies;

    cookies.Add(itemCountCookie);
    int i=0;
    foreach (Item item in cart)
    {
      HttpCookie descCookie =
              new HttpCookie(i.ToString() + "desc");
      descCookie.Value = item.Description;
      cookies.Add(descCookie);

      HttpCookie costCookie =
              new HttpCookie(i.ToString() + "cost");
      costCookie.Value = item.Cost.ToString();
      cookies.Add(costCookie);
      i++;
    }
  }
  // remainder of class unchanged from Listing 10-4
}
```

LISTING 10-19: Cookie State Shopping Page Example

```
public class PurchasePage : Page
{
    // Maintain private cart array variable
  private ArrayList _cart;

  private void Page_Load(object sender, System.EventArgs e)
  {
    _cart = Item.HydrateArrayListFromCookies();
```

continues

```
    }

    private void Page_PreRender(object src, EventArgs e)
    {
      Item.SaveArrayListToCookies(_cart);
    }

    private void AddItem(string desc, int cost)
    {
      _cart.Add(new Item(desc, cost));
    }

    // remaining code identical to Listing 10-7
  }
```

Although it is technically possible to store any type of client-specific state using cookies, as shown in the previous shopping cart example, there are several drawbacks compared with other models. First, all of the state must be mapped into and out of strings, which in general requires more space to store the same amount of data. Second, as mentioned earlier, clients may disable cookies or may have a browser that does not support cookies, thus rendering the application inoperative. Finally, unlike session state, cookie state is passed between the client and the server with every request.

10.5 View State

In addition to session state and cookie state, ASP.NET introduces the ability to store client-specific state through a mechanism called view state. View state is stored in a hidden field on each ASP.NET page called `__VIEWSTATE`. Each time a page is posted to itself, the contents of the `__VIEWSTATE` field are sent as part of the post. The primary use of view state is for controls to retain their state across post-backs, as described in Chapter 2, but it can also be used as a mechanism for storing generic client-specific state between post-backs to the same page.

View state is accessible from any control and is exposed as a `StateBag` that supports storing any type that is serializable. Because the `Page` class is derived from the `Control` base class, you can access the view state

directly from within your pages and indirectly through server-side controls. Listing 10-20 shows the `ViewState` property of the `Control` class. The view state for a control is loaded just before the `Load` event firing, and it is flushed just before the `Render` method being invoked. This means that you can safely access the `ViewState` in your `Load` event handler and that you should make sure it has been populated with whatever state you need by the time your `Render` method is called.

LISTING 10-20: ViewState Property Accessor

```
public class Control : //...
{
  protected virtual StateBag ViewState {get;}
  //...
}
```

For an example of using view state, let's reimplement our shopping cart example one more time, this time using view state as the container for client-specific state. Because the `StateBag` class has a default indexer just as the `HttpSessionState` class does, the code needs to change very little from our original session state–based implementation. The `Item` class can be used in its original form with serialization support (not the altered form required for cookie state). The most significant change is that view state does not propagate between pages in an application, so to use it, we must aggregate all of the functionality that relies on client-specific state into a single page. In our example, this means that we must implement the `CheckOutPage` and the `ShoppingPage` together in one page. Listing 10-21 shows this implementation.

LISTING 10-21: ViewState Shopping Page Example

```
public class PurchasePage : Page
{
  private void Page_Load(object sender, EventArgs e)
  {
    ArrayList cart = (ArrayList)ViewState["Cart"];
    if (cart == null)
    {
      cart = new ArrayList();
```

continues

```
        ViewState["Cart"] = cart;
      }
        // Print out contents of cart with total cost
        // of all items tallied
      int totalCost = 0;

      foreach (Item item in cart)
      {
        totalCost += item.Cost;
        Response.Output.Write("<p>Item: {0}, Cost: ${1}</p>",
                              item.Description, item.Cost);
      }
      Response.Write("<hr/>");
      Response.Output.Write("<p>Total cost: ${0}</p>",
                            totalCost);
    }

    private void AddItem(string desc, int cost)
    {
      ArrayList cart = (ArrayList)ViewState["Cart"];
      cart.Add(new Item(desc, cost));
      _itemsInCart.Text = cart.Count.ToString();
    }

    // remaining code identical to Listing 10-7
  }
```

Notice that in contrast to the cookie state implementation, we were able to save the `ArrayList` full of `Item` instances directly to the `ViewState` state bag. When the page was rendered, it rendered the `ArrayList` into a compressed, text-encoded field added as the value of the `__VIEWSTATE` control on the form. On subsequent post-backs to this page, the view state was then reclaimed from the `__VIEWSTATE` field, and the `ArrayList` was once again available in the same form. Like cookie state, view state is sent between the client and the server with each request, so it should not be used for transmitting large amounts of data. For relatively small amounts of data posted back to the same page, however, it provides a convenient mechanism for developers to store client-specific state.

SUMMARY

State management influences almost every aspect of a Web application's design, and it is important to understand all the options available for state management as well as their implications for usability, performance, and scalability. ASP.NET provides four types of state, each of which may be the best choice in different parts of your application. State that is global to an application may be stored in the application state bag, although it is typically preferable to use the new data cache instead of application state in ASP.NET. Client-specific state can be stored either in the session state bag, as client-side cookies, or as view state. Session state is most commonly used for storing data that should not be sent back and forth with each request, either because it is too large or because the information should not be visible on the Internet. Cookie state is useful for small client-specific pieces of information such as preferences, authentication keys, and session keys. View state is a useful alternative to session state for information that needs to be retained across posts back to the same page. Finally, enhancements to the session state model in ASP.NET give developers the flexibility to rely on session state even for applications that are deployed on Web farms or Web gardens through remote session storage.