TChapter 6—Page and Session ManagemenT

Page and Session Management

The Hypertext Transfer Protocol (HTTP) is the reason for the success of the Internet. It is also the source of the drawbacks in the usability of applications. To overcome these limitations, ASP.NET provides several ways of maintaining the state of pages—and the controls between postbacks—throughout the whole session. Whatever the type of data existing within an application, there are two things to consider: the storage of application data on the server, and how to integrate the user’s session with the application data.

In this chapter, I’ll explain how to customize in both of these areas for your real-life applications. In particular, you’ll find information about:

\* The page persister technique

\* Writing a custom page persister

\* The internals of the session state provider

\* Writing a custom session state provider

The approach presented here is based on the provider model as explained in Chapter 4. Please refer back for useful information about providers, their purpose, and how they fit into the ASP.NET framework.

# The Page State Persister

Handling active content and creating rich internet applications are difficult to integrate. Page persistence is the key to overcoming this challenge, and a good understanding of this feature is the basis of creating complex applications.

## A Look Back

The protocol spoken between server and browser is HTTP, and there are many books, articles, blogs, and newsgroup posts complaining about its limitations. However, HTTP has succeeded mainly because of its simplicity. There are three key properties which make HTTP unique:

\* It uses text to transport both messages and content

\* It runs on top of TCP

\* It is a stateless protocol

“Stateless” can be thought of as “fire and forget”. The client initiates a request to the server; the server processes the request and responds with the answer. After that, the server forgets all about the procedure. Once the conversation has ended, nothing is left for further processing—at least, this is the case for the protocol.

## The Default Page State Persister

The statelessness of HTTP means that there is no formal way of remembering, either on the client or the server, the state of any past actions involving the same user. However, every web development environment provides some means of recognizing session states and storing user data. ASP.NET is no exception, and as for many other parts of the .NET framework, there are several ways of achieving this.

### State Storage in ASP.NET

Because all state management approaches require custom coding by application developers, it’s worth looking into them in more detail. Understanding the pros and cons of each method makes it easier to decide which one to use, where to customize it, and how to get the most out of it.

ASP.NET can store information about the page’s state in several collections:

\* Cookies: You can create private cookies and store them on the client’s machine in order to obtain information from them. This is not the best method, but it’s the default method.

\* View State: ASP.NET controls, as well as several other parts, use view state to store control-specific information about the previous state. We’ll discuss all facets of view state later in this chapter.

\* Session: This is a collection of key-value pairs related to the current user’s session. It’s a way of adding more information, on the server side, to a session state stored elsewhere (in a cookie, for instance).

The Application collection is another set of key-value pairs which retain application-level data shared among all users. In contrast, the Context collection is limited to the current request-response flow—data in the Context collection does not survive the page cycle.

Cookies, View State and the Session collection give the developer the illusion of stateful page handling. Most commonly, they will use a single cookie, called the session cookie, to store an ID with the shortest lifetime possible in the browser’s cookie memory, just until the session ends. When the client makes a new request, the cookie is returned in the request. The server decodes the ID to determine which browser instance (and hence, which user) is sending the request.

As an experienced developer, you might already know how to use the Cookie, Session, and Application collections. Each of them has a different storage mechanism and must be used wisely. You may remember that there are many code snippets scattered throughout the page’s code which obtain access to the information whenever it is required. However, this is not ideal, as it detracts from the declarative nature of ASP.NET pages. Not only does it make pages harder to read, but it could lead to issues if overused.

## Persisting Page State Information

The first step in managing state data is to master the techniques for retaining the page state through each postback. The information on the “current” page is generally stored within several controls. Therefore, we’ll need a way of storing the controls’ state information.

The state storage relies on two similar techniques: the view state and the control state. The internal nature and pitfalls of the view state were described in Chapter 1. Here, I’ll show how to change the storage and retrieval mechanism.

### View State Explained

Recall the purpose of view state: to store data about the properties of controls that are set programmatically. View state is not responsible for the sticky form behavior that retains the controls’ current state according to the user’s selection. The control state holding this information is retrieved from the form’s data, which is posted back to the server during a normal postback. Additional information is stored in a special hidden field. Internally, view state uses an object of type StateBag. It retains data and includes code which serializes the data into text format and stores it in the hidden field.

Consider using a Label control, where you set the color to turn red if an error occurs. You’ll want to maintain this state until the error has been corrected, without checking and setting the property repeatedly. Once set programmatically, view state stores this value in the StateBag and persists it in the page via a hidden field. When the page is initialized after a postback, the value is retrieved and assigned to the property—no custom code required.

Clearly, view state plays a crucial role in page development, and can be used internally to store a control’s state.

### Control State Explained

View state and control state are related and therefore need to be discussed together. Whereas view state contains programmatically-set property values, control state contains the data required for one control to make it appear after a postback in the same state that it was in beforehand. Frustratingly, the control state is stored as part of the view state data. This means that there are two different ways of saving control data during postbacks, but both use the same storage—the hidden view state field.

In fact, there is no way to treat both states separately from each other. Disabling the view state suppresses the serialization and storage mechanism entirely, so the control state gets lost. Complex controls such as GridView store information about paging and sorting, because there are no corresponding HTML fields able to store and send such data as part of a regular form. The GridView has several indices that retain the internal state. Changes to underlying data are tracked and used to fire appropriate events. These are properties such as EditIndex, PageIndex, and SortDirection. Several internal actions might lead to changes that need to be stored during postbacks, but this does not use the same path through the code as the view state would use for public properties.

Control developers should be careful about using the control state because of its dependency on view state. A disabled view state could cause the control state to not function properly. The view state data stored in the hidden field would expand and developers using your control might by concerned by the growing page and form data.

However, as a control developer, you have several options for managing the way your control stores internal state values. Firstly, the Page.RegisterRequiresControlState method must be called to ensure that the control starts storing the control state. Apart from within view state, there is no StateBag that stores values directly. It’s up to you to implement the whole storage level. This is both good and bad, as although you’ll have more control of what happens internally, it requires more effort to store a few primitive values.

The following code listing shows a simple control that stores several values in the control state.

Listing 6-1: A custom control using the Control State

public class MyControl : Control

{

string stateData;

protected override OnInit(EventArgs e)

{

Page.RegisterRequiresControlState(this);

base.OnInit(e);

}

protected override void LoadControlState(object savedState)

{

object[] ctrlState = (object[]) savedState;

base.LoadControlState(ctrlState[0]);

stateData = [string] ctrlState[1];

}

protected override object SaveControlState()

{

object[] ctrlState = new object[2];

object[0] = base.SaveControlState();

object[1] = stateData;

return ctrlState;

}

}

If you plan to write a custom page state persister, you’ll have to handle the control state’s data as well as the view state’s data. The persister’s provider must store both, otherwise almost all custom controls will stop functioning properly.

An alternative is to store the values your own way—using private hidden fields, perhaps—in order to remain independent of the provider. However, this is beyond the scope of this chapter. Besides, adding infrastructure logic to custom controls run counter to the ASP.NET paradigm, in which the base framework contains and exposes a rich infrastructure.

In conclusion, view state and control state are stored in the same place in order to completely and automatically restore a control’s properties. Since the storage and management of view state and control state are related, I’ll treat them together in this chapter. When I use the term ‘view state’, consider it as including control state as well.

### The Default Providers

ASP.NET uses a provider to manage storage of the view state. The description above assumes the usage of one of the two providers available out-of-the-box. Both providers derive from an abstract base class, PageStatePersister:

\* HiddenFieldPageStatePersister

\* SessionPageStatePersister

By default, the HiddenFieldPageStatePersister is used to store the view state in a hidden field. When you look into a page’s source code in the browser, you’ll find a field with the name \_\_VIEWSTATE. This is the serialized, encoded, and optionally encrypted collection of properties stored in the StateBag.

This figure is not up to date

Figure 6-1: Typical view of the view state‘s hidden field

This is what you’ll see if the HiddenFieldPageStatePersister is used. There are several pros and cons to using this method. Many pages store information and many users have individual data. The state is associated with the current page, and apart from session data that is independent of the current page, this data is specific to the page and the user. Storing the data on the client computer moves the memory required from the server to the client. This costs bandwidth, but saves server resources.

If you have 1 KB of session data and 1,000 users accessing the server simultaneously, you’ll need to store 1 MB of data. If this happens for 100 pages, you’ll need to store 100 MB. This isn’t usually significant, but growing traffic will increase the resource consumption accordingly. For this reason, using the browser’s memory is a better option for scalability, and this is why the default provider uses the HiddenFieldPageStatePersister option.

However, sometimes it’s not the server’s resources that cause the bottleneck but the bandwidth, particularly on mobile devices. View state is transmitted twice for each request. Accessing a dozen pages with 1 KB of session data each would force the device to transfer and process an additional 24 KB of data, and this could be even higher for badly-developed pages.

To investigate the contents of the view state, a tool such as Fritz Onion’s *ViewStateDecoder* is invaluable. Knowing what is stored in view state makes it’s easier to design a custom page state persister. You can find the most recent version of *ViewStateDecoder* at: *http://www.pluralsight.com/community/media/p/51688.aspx*.

Insert ASPEXTf0602.tif

Figure 6-2: Examine the contents of the view state

At this point, the particular contents of the view state aren’t important. Decoding the data is merely to understand the internal behavior.

### Changing the Default Provider

The alternative provider, SessionPageStatePersister, moves the storage strategy from the client to the server. The data is stored in the session, along with other session variables. However, view state data is not session data—merely the state of controls. View state for each page is stored separately in this provider.

To change from the default provider, you must override the PageStatePersister property of the Page class:

protected override PageStatePersister PageStatePersister

{

get

{

return new SessionPageStatePersister(this);

}

}

It’s also possible to set this globally by assigning the following attribute to the pages tag in the *web.config* file.

<System.web>

<pages pageBaseType="PersisterBasePage">

…

</System.web>

This assigns a base page to all pages—*PersisterBasePage*, for example, which derives from the Page class and overrides the PageStatePersister property. However, this prevents you from using code behind.

You could set the base class individually for each page. This is more flexible, but more demanding. The page’s definition could look like:

public partial class \_Default : PersisterBasePage

As easy as this seems, it always involves a compromise. The workload is either on the client or the server. Fortunately, the extensibility model of ASP.NET allows you to create custom providers to transparently replace the existing ones.

# Developing a Custom Page State Provider

As already mentioned, the entry point into a custom provider is the implementation of an abstract base class. Before you start coding, you’ll need to answer two questions:

1. Where shall I store data during postbacks?

2. How do I identify when data is no longer current?

The first answer is easy. Data is typically stored in a file system, a database, or in memory.

The second answer is more complicated. Data is related to a particular page, and when a user leaves the page, the data becomes redundant. This event is difficult to recognize. The user can leave the page by navigating to another page within the same application, by jumping to a different website on another server, or even by simply closing the browser. Keeping the data in storage forever is not an option, as this would increase the storage space used and the effort of retrieving the right data. While the default provider is not the ideal solution either, it does answer the second question: if the user moves on to another page, the browser removes the data from the client’s memory. Thus, you don’t need to worry about obsolete data.

There is no satisfactory answer to this question. It’s usually sufficient simply to remove outdated data which has not been refreshed for a specific period of time. This assumes that a user is no longer connected to the server. Selecting a value corresponding to the session timeout will be adequate for most applications.

## Choosing the Data Storage

In this example, I’ll make the storage as easy as possible to implement. The data is stored on the server in the file system. This could even be a shared drive, if you need to support a web farm. File systems are highly efficient for storing and retrieving data and avoid dependency on a database. However, this is simply an example of how to extend the page state persister. The storage solution you use depends entirely on the characteristics of your application.

You’ll notice that the hidden field is still in charge, because it contains some information for identifying unique pages. This is why the hidden field is not empty when using the default provider, even if the view state is disabled. However, the identifier only needs a few bytes, and shouldn’t affect bandwidth requirements.

### Analyzing a Provider

Before implementing the provider from scratch, let’s examine the SessionPageStatePersister class for firsthand information about the internal structure of such a provider. This class stores data on the server, just as the custom provider is intended to do. If you investigate the view state value captured from the hidden field, you’ll see a simple hexadecimal number, such as 8cb5cd5d086eee0. This is the hexadecimal representation of the procedure. Using the following code sequence, you can decode the value:

new DateTime(0x8cb5cd5d086eee0, DateTimeKind.Local).ToString()

This value decodes to 14 February 2009, 4:09pm and 57 seconds. The SessionPageStatePersister stores not one, but nine of these values internally, to keep the history of page accesses alive. When the user presses the browser’s back button, a previous page is pulled from the server. Using the timestamp as a key, the correct data is retrieved from the server’s storage. The name of the internal queue is \_\_VIEWSTATEQUEUE. Constraining the number of pages in the history solves the second question raised earlier, and limits the storage space used. Because the user’s behavior depends on the application’s navigation structure, you can alter the queue size using the following entry in the *web.config* file:

<sessionPageState historySize="12" />

This value allows twelve pages in the history list. The value you choose is always a compromise between convenience and resource usage. If the user exceeds the queue size limit, the view state becomes invalid and an exception is thrown. You can catch the exception, deliver an error page, and end the session, or simply redirect to a default page. However, as this is not an ideal solution, the following example shows an alternative way of dealing with “dead” data.

### Implementing the Provider

As we learned from the implementation, the whole procedure consists of two methods, Load and Save. Both use the fully implemented serialization and deserialization methods provided by the base class to convert between the StateBag object and a text string. The example stores the data in single files in the file system. Each page’s data becomes one file. To retrieve the values, two pieces of information must be held in the view state field: the name of the file and the timestamp. The timestamp is needed for private garbage collection to remove expired data. The file name associates the page’s view state field with the file itself. After rendering to the browser, the hidden view state looks like:

<input type="hidden" name="\_\_VIEWSTATEID" id="\_\_VIEWSTATEID"   
 value="8cb5cd5d086eee0-2qi0qprzodivhh55uwog3k45.vs" />

The first part of the value, before the hyphen, is the timestamp. I have retained the coding schema. The second part is a randomly generated value, plus the extension, *.vs*.

Listing 6-2 shows the code for the custom provider, as described thus far.

Listing 6-2: Persisting the view state using local file system

public class FilePagePersister : PageStatePersister

{

private const string ViewStateFormFieldID = "\_\_VIEWSTATEID";

private const string StateFileFolderPath = "~/StateData/";

public FilePagePersister(Page p)

: base(p)

{

}

public override void Load()

{

string stateIdentifierValue = ⮰

HttpContext.Current.Request.Form[ViewStateFormFieldID];

if (stateIdentifierValue.Length > 0)

{

string fileName = stateIdentifierValue;

string filePath = HttpContext.Current.Server.MapPath( ⮰

StateFileFolderPath + fileName);

string contents = File.ReadAllText(filePath);

Pair state = base.StateFormatter.Deserialize(contents) as Pair;

if (state != null)

{

base.ViewState = state.First;

base.ControlState = state.Second;

}

}

}

public override void Save()

{

if (base.Page.Form != null ⮰

&& (base.ControlState != null || base.ViewState != null))

{

// Create filename for save

string fileName = String.Format("{0:x8}-{1}.vs", DateTime.Now.Ticks, ⮰

HttpContext.Current.Session.SessionID);

string filePath = HttpContext.Current.Server.MapPath( ⮰

StateFileFolderPath + fileName);

Pair p = new Pair(base.ViewState, base.ControlState);

File.WriteAllText(filePath, base.StateFormatter.Serialize(p));

HiddenField hf = new HiddenField();

hf.Value = fileName;

hf.ID = ViewStateFormFieldID;

base.Page.Form.Controls.AddAt(0, hf);

}

}

}

The base class provides methods of serializing and deserializing the data. You can access these methods by calling base.StateFormatter. The provider saves both the control state and the view state, and creates a Pair from them, to group the values together. The serialized Pair is written directly into the file using File.WriteAllText and retrieved using File.ReadAllText. We are not concerned about threading here, because each storage location is dedicated to one user session and one page. There’s a chance that a user could send the same page twice to the server at the same moment, but that’s unlikely—and out of our scope. The Load method is the reverse of the Save method, using the same approach. The Save method creates a hidden field using the HiddenField control and adds it to the beginning of the current control collection.

### Extending the Provider

Although, from this example, custom implementation seems very simple, this isn’t entirely true. Over time, the folder for state files accumulates more and more files. You might consider deleting session state files triggered by the session end event, but that’s not reliable. The event is fired only if the session ends by command. If the user closes the browser the session dies “silently”, and only the session timeout can be used to remove remaining objects.

One option is to remove old files regularly, but using such a cleanup process for each page request would increase the number of file accesses and thus decrease performance. A better approach is to include a regular folder check independent of any requests. Using this test, you can delete all files over a certain age.

Implementing the tidy-up code in the provider is undesirable. Recall the goals and intended behavior of a provider: it’s not a good idea to have them running basic tasks. Rather, the health monitoring system provided by ASP.NET is ideal for managing the “health” of the provider’s storage.

### Maintaining the Storage by Using the Health Monitor

Because the cleaning up procedure must be run regularly, the WebHeartbeatEvent is an ideal trigger. This event fires at regular intervals, invoking another provider. The health monitoring system itself is based on a provider model. Therefore, the second part of a custom view state storage solution consists of a “custom clean up provider”.

Before you implement this provider, take a look at the configuration step needed in the *web.config* file.

Listing6- 3: Activating the Health Monitor

<healthMonitoring enabled="true" heartbeatInterval="5">

<providers>

<add name="FileCleanupProvider"

type="Basta.EventProvider.FileCleanupProvider, Basta.EventProvider"/>

</providers>

<eventMappings>

<add name="FileCleanupEvent" type="System.Web.Management.WebHeartbeatEvent"

startEventCode="0" endEventCode="2147483647"></add>

</eventMappings>

<rules>

<add name="CleanupEvent" eventName="FileCleanupEvent"

provider="FileCleanupProvider" />

</rules>

</healthMonitoring>

The healthMonitoring element requires three child elements:

*\** provider: The provider that issues the action

*\** eventMapping: The event that invokes the action

*\** rules: A relation between a specific event and a provider

The event is based on the WebHeartbeatEvent class. Enable the monitor using the enabled="true" attribute. The heartbeatInterval="5" attribute sets the heart beat to five seconds; this is just for testing and debugging purposes. In real life scenarios, set a value that provides a compromise between memory consumption and clean-up effort in order to monitor the number of files stored in the data folder. A heartbeat interval of one hour is a good starting value.

The example provider configured here is called the FileCleanupProvider. The next code listing demonstrates its implementation. Before beginning, you’ll need to know a few things about health monitoring providers. The monitoring system is implemented to observe an ASP.NET application. It must be active before the application starts and after it ends, and it must exist independently of current requests, and IIS. Therefore, the class must be implemented in an assembly that can be loaded separately. Adding the class to *App\_Code* or any place in your current project will fail. Instead, create a simple class library project, reference it from the Web application you wish to monitor, and add the following class to that project.

Listing 6-4: Simple health monitor provider that tidies up a folder

public class FileCleanupProvider : WebEventProvider

{

private const string StateFileFolderPath = "StateData/";

public FileCleanupProvider()

: base()

{

}

public override void Flush()

{

// not required

}

public override void ProcessEvent(WebBaseEvent raisedEvent)

{

DateTime dtRaised = raisedEvent.EventTime;

// Remove files

string filePath = Path.Combine(HttpRuntime.AppDomainAppPath, ⮰

StateFileFolderPath);

foreach (string file in Directory.GetFiles(filePath))

{

if (dtRaised - File.GetCreationTime(file) > TimeSpan.FromHours(6))

{

File.Delete(file);

}

}

}

public override void Shutdown()

{

// Clean up on shut down

string filePath = Path.Combine(HttpRuntime.AppDomainAppPath, ⮰

StateFileFolderPath);

foreach (string file in Directory.GetFiles(filePath))

{

File.Delete(file);

}

}

}

Each heartbeat of the monitor invokes the ProcessEvent method. A simple loop locates all the files exceeding the specified age—six hours in the example—and deletes them. The file expiry test can be extended to suit your needs. You could check if the corresponding session is still alive, or if the data in the file matches certain conditions.

The current timestamp is passed to the method with the raisedEvent.EventTime value. The path to the application can be retrieved using the HttpRuntime object. Remember that HttpContext might not be available, because the first heartbeat arrives when the application has not yet begun. Other monitoring events could provide the HttpContext object, but basic events such as the WebHeartbeatEvent don’t support this.

### Conclusion

In this section, we looked at replacing the default page state persister provider that is responsible for storing view state and embedded control state information. Using the health monitoring system with another custom provider allows us to maintain clean up routines.

Page states are not the only state persisted during a user’s session. Data is also stored in session variables, which are, by default, stored in memory. However, as in several other parts of ASP.NET, the session state is built on the provider model.

## Session State Providers

As shown in chapter 4, providers are responsible for allowing storage access to services. The session state module is one such service relying on several out-of-the-box storage options. For all available options, a dedicated provider exists. You can also add your own provider to implement a fully customized storage strategy.

### The Session State Service

The session state service is defined in the System.Web.SessionState.SessionStateModule. Each instance creates a module that in turn creates a space to store data. The data is—as the name implies— associated with the user’s session. System.Web.SessionState.SessionStateStoreData is the class that defines the storage space. One instance is created for each user session. The SessionStateStoreData class is responsible for serializing and deserializing the session data. The serialization uses a highly efficient binary format partly customized with hardcoded transformations for scalar values and the BinaryFormatter type for all other values. This type is defined in the System.Runtime.Serialization.Formatters.Binary namespace. Because the session serializes all objects, the process includes two types of data: static and non-static. Non-static data are held in an Items collection. This collection is defined as an interface of type [ISessionStateItemCollection](http://www.aisto.com/roeder/dotnet/Default.aspx?Target=code://System.Web:2.0.0.0:b03f5f7f11d50a3a/System.Web.SessionState.ISessionStateItemCollection), which is a simple collection based on ICollection and IEnumerable. It’s exposed by the Session object through an indexer and the Items property. The StaticObjects property relies on the HttpStaticObjectsCollection type, which in turn is a collection that implements the same interfaces. However, the serialization is implemented slightly differently here. For both Items and StaticObjects, you can call the Serialize and Deserialize methods to translate the data to and from a storable format.

Apart from the two types of data mentioned above, the data store has its own Timeout property. This is necessary as the service is implemented as an HTTP module and relies on request events. If the final event is not raised, the store would cease to function if the timeout did not terminate it. Internally, the session state service class is defined as:

public sealed class SessionStateModule : IHttpModule

The session state module is executed before the handlers in the AcquireRequestState pipeline state. This ensures that session data is already present when the page’s handler is loaded. The module attempts to retrieve the session ID from the request and uses this to obtain data from the session store provider. If both function correctly, the session data is rebuilt as a dictionary and exposed to the HttpSession object. The session data is stored at the end of the request processing, within the ReleaseRequestSession stage of the pipeline. If no session ID is present, a new one is created. If the ID is there and some session data is changed, the provider is used to store these data.

The service’s providers use the SessionStateStoreProviderBase provider base class, which defines a subset of methods all providers share.

Listing 6-1: The SessionStateStoreProviderBase is the base for all session state providers

[AspNetHostingPermission(SecurityAction.LinkDemand, ⮰

Level=AspNetHostingPermissionLevel.Minimal), AspNetHostingPermission(SecurityAction.InheritanceDemand, ⮰

Level=AspNetHostingPermissionLevel.Minimal)]

public abstract class SessionStateStoreProviderBase : ProviderBase

{

protected SessionStateStoreProviderBase()

{

}

public abstract SessionStateStoreData CreateNewStoreData(HttpContext context, ⮰

int timeout);

public abstract void CreateUninitializedItem(HttpContext context, ⮰

string id, ⮰

int timeout);

public abstract void Dispose();

public abstract void EndRequest(HttpContext context);

public abstract SessionStateStoreData GetItem(HttpContext context, ⮰

string id, ⮰

out bool locked, ⮰

out TimeSpan lockAge, ⮰

out object lockId, ⮰

out SessionStateActions actions);

public abstract SessionStateStoreData GetItemExclusive(HttpContext context, ⮰

string id, ⮰

out bool locked, ⮰

out TimeSpan lockAge, ⮰

out object lockId, ⮰

out SessionStateActions actions);

internal virtual void Initialize(string name, ⮰

NameValueCollection config, ⮰

IPartitionResolver partitionResolver)

{

}

public abstract void InitializeRequest(HttpContext context);

public abstract void ReleaseItemExclusive(HttpContext context, ⮰

string id, ⮰

object lockId);

public abstract void RemoveItem(HttpContext context, ⮰

string id, ⮰

object lockId, ⮰

SessionStateStoreData item);

public abstract void ResetItemTimeout(HttpContext context, string id);

public abstract void SetAndReleaseItemExclusive(HttpContext context, ⮰

string id, ⮰

SessionStateStoreData item, ⮰

object lockId, ⮰

bool newItem);

public abstract bool SetItemExpireCallback( ⮰

SessionStateItemExpireCallback expireCallback);

}

Inspecting the GetItem and RemoveItem methods, you’ll notice that the provider is responsible for creating (via the out parameter) the session specific SessionStateDataStore. The vital parameter is HttpContext, as it passes all the information required for identifying the session.

However, the base class is abstract. In order to understand the code and learn how to implement a custom provider, we’ll need to examine certain implementations.

### Identifying the Session

Despite storage management, the session state service needs to retrieve the session ID. The first attempt to overcome this stateless behavior led to the invention of cookies.

#### Cookies make HTTP taste good…

A cookie is a small piece of text-based information stored in the browser’s memory or disk space. It’s under the browser’s control and it exists to send information about the state back to the server. When the browser requests a new resource, it searches for cookies stored by this specific server and returns this information. The server recognizes the information, reading it back from the headers of the HTTP request, and uses it to obtain the client’s state. However, cookies have some serious drawbacks. If a server only sends portions of a page, this very portion would send the information back to another server than the one serving the main page. An advertising banner is an example of such a section.

#### …and bad

At the end of the last century, the company Doubleclick (eventually bought out by Google) invented a business concept based on cookies. Once a banner ad appears, the cookies related to it are placed in the browser’s memory. When the user moves to a page drawn from another server, but containing a banner pulled from the original ad server, the ad server’s code would receive the same cookie back. The original server knows that the user has been on different pages, and it also recognises where these pages originated. The cookie makes it possible for the ad server to trace the user and track his or her behavior. By the third page, the banner ad has not been selected at random. Based on the user’s behavior, the ad server chooses the banner that best fits the user’s interests. This leads to a much higher click rate, which in turn results in more money made. This is the concept invented by Doubleclick—with which Google now generates a lot of revenue. It is as simple as it sounds. However, it has faced criticism in that it compromises the user’s privacy. The ad server knows much more of a user’s behavior than people would tell strangers voluntarily. Since cookies work silently, the user is never asked to accept or deny them. Even if modern browsers managed cookies well, it’s still a complicated task to deal with them, erasing them where appropriate and retaining them if harmless.

#### Beyond Cookies

Avoiding the usage of cookies or even limiting them to just one—to store the session state—would give your users the opportunity to operate within a restrictive cookie usage policy. This is an advantage of a site created this way. However, it makes your life as a developer much more difficult. Fortunately, ASP.NET provides a number of sophisticated techniques which help you handle the drawbacks of HTTP’s statelessness. In terms of the session state, you can decide whether to work with session cookies or without cookies at all.

ASP.NET maintains the cookieless session state by automatically inserting a unique session ID into the URL. This would look like:

http://www.apress.com/(S(lit3jk26t05z64v14vlm04s09))/default.aspx

The following configuration in the *web.config* file is required:

<configuration>

<system.web>

<sessionState cookieless="true" regenerateExpiredSessionId="true" />

</system.web>

</configuration>

A session ID will eventually expire when the session ends, or has been timed out. In this case, the ID is recycled; a later session could have the same ID. If a URL is stored somewhere on the client—such as within the favorites folder, sent by email, or gathered by a search engine— this could cause a security hole, as the older session ID located in the URL could be the same as that of another current session. The regenerateExpiredSessionId="true" attribute reduces that risk by forcing it to create a new ID every time.

If this behavior is not sufficient for your application, you can write your own session ID module. For complete control, consider implementing the ISessionIDManager interface. To just control the creation of the ID, inherit from the SessionIDManager base class. Both types are defined in the System.Web.SessionState namespace.

The next example demonstrates how to replace the session ID with a custom value.

Listing 6-2: Simple session ID manager with custom ID creation

namespace Apress.SessionState.Code

{

public class SimpleSessionIDManager : SessionIDManager

{

public override string CreateSessionID(HttpContext context)

{

return Guid.NewGuid().ToString();

}

public override bool Validate(string id)

{

try

{

Guid testGuid = new Guid(id);

if (id.Equals(testGuid.ToString()))

{

return true;

}

}

catch

{

}

return false;

}

}

}

The only methods to override are CreateSessionID and Validate. However, the basic form of the ID, such as (S(…)), is still used. To obtain completely different behavior, you must implement the interface, which requires several additional methods to be implemented. This is achieved behind the scenes by adding the ID to the URL first and then redirecting to the new URL. The changed URL is then passed to the browser.

However you implement this, you must configure the new session ID manager in the *web.config* file:

<sessionState mode="InProc"

cookieless="true"

regenerateExpiredSessionId="false"

timeout="20"

sessionIDManagerType="Apress.SessionState.Code.MySessionIDManager, ⮰

Apress.SessionState"/>

The sessionIDManagerType attribute contains the class’s name and the namespace of the containing assembly. All other options depend on the entire configuration and are not directly associated with the session ID manager.

### The Internal State Providers

ASP.NET includes three internal state providers. You should investigate these before planning the development of a custom provider.

\* InProcSessionStateStore

\* OutOfProcSessionStateStore

\* SqlSessionStateStore

This section gives a brief overview of the internal state providers, which have several features in common. The user can control their behavior with the EnableSessionState attribute located in the @Page directive.

Figure 6-3: Principles of the session state providers

#### InProcSessionStateStore

The InProcSessionStateStore class defines the default session state provider if no other provider is specified. It stores session data within the current process, using the HttpCache module to store data in memory.

This provider is the most efficient. The overhead is limited to the extracting of the session ID and the serializing and deserializing of the data. Everything happens in memory using a cache lookup, which merely retrieves the data and refreshes the cache expiration timestamp.

However, the InProcSessionStateStore has some drawbacks. This provider does not survive an application restart, since the cache objects are rebuilt on restart—meaning all previous sessions would be lost. It also does not function reliably in web farm scenarios, where the next request might reach a different server that doesn’t see the session data in the first server’s memory.

The following *web.config* file snippet shows the configuration:

<sessionState

mode="InProc"

cookieless="false"

/>

#### OutOfProcSessionStateStore

This class enables out-of-process storage based on an unmanaged stream that is created when an incoming request is handled by the session state service. The stream is read from and written back to a state server service. This Windows service runs independently of IIS and allows the sharing of memory data through several servers that make up a web farm.

The provider is not as efficient as the in-process version, as it takes time to pass data beyond process boundaries. Internally, the process communicates with a Windows service that passes the data from one server to another using HTTP. This is a very slim and fast implementation of a Web server handling only a subset of Web server functions. While the HTTP protocol adds overhead to requests, the OutOfProcSessionStateStore will survive an application restart, as the storage is not part of the ASP.NET process.

The following snippet from the *web.config* file shows the configuration:

<sessionState

mode="StateServer"

stateConnectionString="tcpip=127.0.0.1:42424"

cookieless="false"

/>

The service runs on local machines and on port 42424 by default.

The state service can be found here:

%systemroot%\Microsoft.NET\Framework\ *<version>*\aspnet\_state.exe

It must be installed and started as a service. By default, it’s installed, but set to start manually. To launch the service after the Web server starts, set the ASP.NET State service’s start option to *Automatic*.

#### SqlSessionStateStore

ASP.NET provides the SqlSessionStateStore in order to share state across multiple front end servers. The provider consists of several pieces you must assemble:

\* A data table for the session data

\* Stored procedures to write and read the session data from SQL Server

\* A connection string defined in *web.config* which declares the database connection

\* The SqlSessionStateStore set in the *web.config* as the default provider

To create the data table and add the stored procedures, use the *Aspnet\_regsql.exe* tool. You can find it at:

C:\Windows\Microsoft.NET\Framework\v2.0.50727

Details about all the command line options are at:

http://msdn.microsoft.com/en-us/library/ms229862.aspx

The tool also supports settings required by other SQL Server based modules. For the session state, the following options are important:

\* -S servername: The name of the database server, such as *locahost\sqlexpress*.

\* -d databasename: The name of the database; requires -sstype c.

\* -ssadd: Adds support.

\* -ssremove: Removes support.

\* -sstype t: Installs the data table into the *tempdb* database.

\* -sstype p: Installs the data table into a persistent database, like *aspnetdb*.

\* -sstype c: Installs the data table into a custom database. Use -d databasename to choose one.

Tip: If you start the Aspnet\_reqsql tool without any parameters, a wizard is lauched which leads you through the required steps.

This *web.config* file snippet shows an example of the configuration:

<sessionState

mode="SQLServer"

cookieless="true"

sqlConnectionString=" Integrated Security=SSPI;data source=MySqlServer;"

sqlCommandTimeout="10" />

### Improving the Session State

The session state service adds a considerable overhead to the performance of your application. Because it’s a valuable tool, most developers choose to use session data. However, the serializing and deserializing procedures required for each request take time. If the application comes under pressure, and if you decide to work with a web farm in order to balance the load, things might become worse. The session related traffic would increase on the local network between the SQL Server or State server and the web servers. There are several ways of improving performance, and it’s important to think about these before you start creating your own session state providers.

Some optimizations are implemented by default in order to avoid having a bottleneck for standard features. Firstly, the ability to handle the session state can be disabled. You can set this using the @Page directive:

\* EnableSessionState="true"

\* EnableSessionState="false"

\* EnableSessionState="ReadOnly"

Enabling and disabling the session state is trivial; when disabled, no session data is stored. The ReadOnly option is more subtle. Imagine that a user has several browser windows or tabs open and all of them are in the same session using the same session ID. If the user accesses the web server multiple times at once, the first request gains exclusive access, locking access for other requests. The second request must wait in order to obtain its exclusive access when the first request ends. This could lead to an unsatisfactory user experience, as the user will probably open several windows to launch simultaneous tasks, such as downloading files. Instead of speeding things up, the windows would be blocked. To avoid this, the ReadOnly option is used to allow concurrent read access to the session data. The lock is still there for write access and it blocks other requests with same session ID. You will only benefit from this setting if you use the session variables carefully.

## Implementing the Session State Store Provider

You might wish to create a custom session-state store provider to:

\* Store session-state information in a data source other than SQL Server, such as a MySQL database or an Oracle database.

\* Manage session-state information using a database schema that is different from the default database schema.

\* Use a totally different storage strategy, such as the file system.

### The Session State Module

Session state is managed by the SessionStateModule class, which calls the session-state stored provider to read and write data to the data store during a request. At the start of a request, the SessionStateModule instance retrieves data from the data source by calling the GetItemExclusive or GetItem method. The exclusive access is the default; the GetItem method also supports the EnableSessionState="ReadOnly" attribute, allowing shared read access as explained in the section above. At the end of a request, if the session-state values have been changed, the SessionStateModule instance calls the SetAndReleaseItemExclusive method to write the updated values to the data store.

The SessionStateModule class itself determines the session ID value, rather than relying on the session-state store provider. The session ID manager is responsible for the ID format, which is based on the implementation of the SessionIDManager class. The SessionStateModule class reverts to the ASP.NET process identity in order to access any secured resource, such as a database server. You can specify the SessionStateModule instance to impersonate the process identity by setting the useHostingIdentity attribute of the <sessionState> configuration element to false.

<sessionState useHostingIdentity="false" />

If the useHostingIdentity attribute is true, ASP.NET will impersonate either the process identity or the user credentials when connecting to the data source. Usually this is a cumbersome process, as you’ll have to manage all your users.

### Preface

Before developing a new session state persister, let’s briefly investigate what it can be used for. There are three major tasks:

\* Handling session locking for concurrent requests to the same data

\* Identifying and occasionally removing expired session data

\* Allowing more than one application to use the provider

#### Locking Session-Store Data

ASP.NET applications respond to multiple concurrent requests. Those requests might attempt to access the same session information, as explained above. When implementing your own provider, you’ll need to manage the locking behavior. A lock is set on session-store data at the beginning of the request in the call to the GetItemExclusive method. When the request is completed, the lock is released during the call to the SetAndReleaseItemExclusive method. If the SessionStateModule instance encounters locked session data during the call to either the GetItemExclusive or GetItem method, it will re-request the session data. This occurs at half-second intervals until either the lock is released or the amount of time specified in the ExecutionTimeout property has elapsed. After a timeout occurs, the ReleaseItemExclusive method is called in order to free the session-store data. Additionally, a lock identifier is used to distinguish between concurrent requests accessing the same session data. This is necessary as these requests run on separate threads. A call to the ReleaseItemExclusive appears before the SetAndReleaseItemExclusive method is called for the current response.

#### Handling Expired Data

Expiration occurs when a session times out or explicitly ends. A session ends with a user action, when the Abandon method is called. The data for that session is deleted from the data store using the RemoveItem method. The mechanism for deleting expired session data depends on the capabilities of your data source. If the session expires according to the session Timeout property, you can use the SetItemExpireCallback method to handle the Session\_OnEnd event.

#### Set Application Name

To maintain session scope, session-state providers store session information uniquely for each application. This allows multiple ASP.NET applications to use the same data source without conflicting. How you ensure uniqueness depends entirely on the storage mechanism. For a file based provider, as in the following example, a subfolder named after the application will suffice. In a database scenario, an *ApplicationName* column in the data table would be necessary.

### Implementation Details

There are many steps to implementation. None are difficult, but careful preparation is required.

#### Required Classes

To implement a session store provider, you must create a class that inherits the SessionStateStoreProviderBase abstract class. This class in turn inherits the ProviderBase abstract class, so you must also implement the required members of the ProviderBase class. (Refer to chapter 4 to read more about the structure of a provider.) The following tables list and describe the properties and methods you must implement from the abstract classes.

Table 6-1: Required ProviderBase Members

|  |  |
| --- | --- |
| Member | Description |
| Initialize | Takes as input the name of the provider and a NameValueCollection instance of configuration settings. This method is used to set the property values for the provider instance, including implementation-specific values and options specified in the configuration file (Machine.config or Web.config). |

Table 6-1: Required SessionStateStoreProvider Members

|  |  |
| --- | --- |
| Member | Description |
| InitializeRequest | Uses the HttpContext of the current request and performs any initialization required by the provider. |
| EndRequest | Uses the HttpContext of the current request and performs any cleanup. |
| Dispose | Frees any resources no longer in use by the provider. |
| GetItemExclusive | Retrieves session values and information from the session data store for a given Session ID and HttpContext. Locks the session-item data in the store for the duration of the request. Sets several output-parameter values that inform SessionStateModule about the state of the current item. |
| GetItem | As for the GetItemExclusive method, except that it does not attempt to lock the session item in the data store. This method is used when you set the EnableSessionState attribute of @Page directive to ReadOnly. |
| SetAndReleaseItemExclusive | Updates or releases the lock for the current session-state data. The newItem property forces the creation of a new item. Calls the ResetItemTimeout method internally to extend the expiration period. |
| ReleaseItemExclusive | This method is called when the GetItem or GetItemExclusive method is called and the item is locked, but the lock duration has exceeded the ExecutionTimeout value. Frees the item and releases the lock. |
| RemoveItem | This method is called when the Abandon method of the Session object is called. Deletes the session information from the data store. |
| CreateUninitializedItem Adds an uninitialized item to the session data store with an actionFlags value set to InitializeItem. |  |
| CreateNewStoreData | Returns a new SessionStateStoreData object with empty data collections. |
| SetItemExpireCallback | Takes a delegate that references the Session\_OnEnd event. If this event is supported the method returns true; otherwise it returns false. |

The GetItemExclusive method is the most significant part of the provider. If no session item data is found in the data store, the method sets the locked output parameter to false and returns null. This causes the SessionStateModule to call the CreateNewStoreData method in order to create a new SessionStateStoreData object for the request. If the item is located but the data is locked, the GetItemExclusive method sets the locked output parameter to true, sets the lockAge output parameter to the current age of the item (that is, current timestamp minus creation timestamp), sets the lockId output parameter to the lock identifier retrieved from the data store, and returns null. This causes the SessionStateModule to call the GetItemExclusive method again after a half-second interval in an attempt to retrieve the session-item information and obtain a lock on the data. If the lockAge exceeds the ExecutionTimeout value, SessionStateModule calls the ReleaseItemExclusive method in order to clear the lock on the session-item data and calls the GetItemExclusive method again.

The actionFlags parameter is used with sessions whose Cookieless property is true and whose regenerateExpiredSessionId attribute is set to true. The actionFlags value can be set to either InitializeItem or None. InitializeItem indicates that the entry in the session data store is a new session requiring initialization. Uninitialized entries are created by a call to the CreateUninitializedItem method. If the item from the session data store is already initialized, the actionFlags parameter is set to None.

If your provider supports cookieless sessions, set the actionFlags output parameter to the value returned from the session data store for the current item. If the actionFlags parameter value for the requested session-store item equals InitializeItem, the GetItemExclusive method should set the value in the data store to zero after setting the actionFlags out parameter.

The CreateUninitializedItem method is used with cookieless sessions when the regenerateExpiredSessionId attribute is set to true. This causes the SessionStateModule to generate a new session ID value when an expired session is encountered. The process of generating a new session ID value requires the browser to be redirected to a URL that contains the newly generated ID. Refer to Listing 6-2 for an example of how to control the session ID generation process. The CreateUninitializedItem method is called during an initial request containing an expired session. After the SessionStateModule acquires a new ID to replace the expired one, it calls the CreateUninitializedItem method. The browser is then redirected to the URL containing the newly generated ID. The existence of the uninitialized entry in the session data store ensures that the redirected request with the newly generated ID is not mistaken for a request for an expired session, and is treated as a new session.

### Implementing a File-Based Session State Persister

The following example shows the basic implementation steps for a file-based store. This is not intended for real-life scenarios, but simply for learning and testing. Files are easy to monitor and you can see where the data goes.

Listing 6-3: A simple file-based session state persister

using System;

using System.Collections.Specialized;

using System.IO;

using System.Web;

using System.Web.Configuration;

using System.Web.SessionState;

namespace Apress.SessionState.Code

{

public sealed class FileSessionStateStore : SessionStateStoreProviderBase

{

private SessionStateSection pConfig = null;

private string basePath;

private string pApplicationName;

public string ApplicationName

{

get

{

return pApplicationName;

}

}

public override void Initialize(string name, NameValueCollection config)

{

if (config == null)

{

throw new ArgumentNullException("config");

}

if (String.IsNullOrEmpty(name))

{

name = "FileSessionStateStore";

}

if (String.IsNullOrEmpty(config["description"]))

{

config.Remove("description");

config.Add("description", "File Session State Store ⮰

provider example");

}

base.Initialize(name, config);

pApplicationName = System.Web.Hosting.HostingEnvironment⮰

.ApplicationVirtualPath;

basePath = config["basePath"];

System.Configuration.Configuration cfg = WebConfigurationManager⮰

.OpenWebConfiguration(ApplicationName);

pConfig = (SessionStateSection)cfg ⮰

.GetSection("system.web/sessionState");

}

public override void Dispose()

{

}

public override bool SetItemExpireCallback(⮰

SessionStateItemExpireCallback expireCallback)

{

return false;

}

public override void SetAndReleaseItemExclusive(HttpContext context, ⮰

string id, ⮰

SessionStateStoreData item, ⮰

Object lockId, ⮰

bool newItem)

{

// Serialize the SessionStateItemCollection as a string.

string sessItems = Serialize((SessionStateItemCollection)item.Items);

string path = Path.Combine(basePath, String.Format("{0}.ssd", id));

FileStream fs = null;

try

{

if (newItem)

{

if (File.Exists(path))

{

File.Delete(path);

}

fs = new FileStream(path, FileMode.CreateNew, ⮰

FileAccess.Write, FileShare.Read);

}

else

{

// update item

fs = new FileStream(path, FileMode.Open, ⮰

FileAccess.Write, FileShare.Read);

}

StreamWriter sw = new StreamWriter(fs);

sw.Write(sessItems);

sw.Close();

}

catch (IOException exception)

{

// add error handling

}

finally

{

if (fs != null)

{

fs.Close();

fs.Dispose();

}

}

}

public override SessionStateStoreData GetItem(HttpContext context, ⮰

string id, ⮰

out bool locked, ⮰

out TimeSpan lockAge,

Object lockId, ⮰

out SessionStateActions actionFlags)

{

return GetSessionStoreItem(false, context, id, out locked, ⮰

out lockAge, out lockId, out actionFlags);

}

public override SessionStateStoreData GetItemExclusive(HttpContext context,⮰

string id, ⮰

out bool locked, ⮰

out TimeSpan lockAge,

Object lockId, ⮰

out SessionStateActions actionFlags) {

return GetSessionStoreItem(true, context, id, out locked, ⮰

out lockAge, out lockId, out actionFlags);

}

private SessionStateStoreData GetSessionStoreItem(bool lockRecord,

HttpContext context,⮰

string id, ⮰

out bool locked, ⮰

out TimeSpan lockAge,

Object lockId, ⮰

out SessionStateActions actionFlags)

{

// Initial values for Return value and out parameters.

SessionStateStoreData item = null;

lockAge = TimeSpan.Zero;

lockId = null;

locked = false;

actionFlags = SessionStateActions.None;

string serializedItems = String.Empty;

string path = Path.Combine(basePath, String.Format("{0}.ssd", id));

FileStream fs = null;

try

{

fs = new FileStream(path, FileMode.Open, FileAccess.Read, ⮰

FileShare.Read);

StreamReader sr = new StreamReader(fs);

serializedItems = sr.ReadToEnd();

sr.Close();

}

catch (IOException)

{

// add error handling here

}

finally

{

if (fs != null)

{

fs.Close();

fs.Dispose();

}

}

item = Deserialize(context, serializedItems, 1024);

return item;

}

private string Serialize(SessionStateItemCollection items)

{

MemoryStream ms = new MemoryStream();

BinaryWriter writer = new BinaryWriter(ms);

if (items != null)

{

items.Serialize(writer);

}

writer.Close();

return Convert.ToBase64String(ms.ToArray());

}

private SessionStateStoreData Deserialize(HttpContext context, ⮰

string serializedItems, ⮰

int timeout)

{

MemoryStream ms = new MemoryStream( ⮰

Convert.FromBase64String(serializedItems));

SessionStateItemCollection sessionItems = ⮰

new SessionStateItemCollection();

if (ms.Length > 0)

{

BinaryReader reader = new BinaryReader(ms);

sessionItems = SessionStateItemCollection.Deserialize(reader);

}

return new SessionStateStoreData(sessionItems, ⮰

SessionStateUtility.GetSessionStaticObjects(context), timeout);

}

public override void ReleaseItemExclusive(HttpContext context, ⮰

string id, object lockId)

{

// release lock

}

public override void RemoveItem(HttpContext context, string id,

object lockId, SessionStateStoreData item)

{

string path = Path.Combine(basePath, String.Format("{0}.ssd", id));

File.Delete(path);

}

public override void CreateUninitializedItem(HttpContext context, string id,

int timeout)

{

string path = Path.Combine(basePath, String.Format("{0}.ssd", id));

FileStream fs = File.Create(path);

fs.Close();

}

public override SessionStateStoreData CreateNewStoreData(

HttpContext context, int timeout)

{

return new SessionStateStoreData(new SessionStateItemCollection(), ⮰

SessionStateUtility.GetSessionStaticObjects(context), timeout);

}

public override void ResetItemTimeout(HttpContext context, string id)

{

// refresh item

}

public override void InitializeRequest(HttpContext context)

{

}

public override void EndRequest(HttpContext context)

{

}

}

}

The implementation lacks error handling and several advanced features. However, the basic parts of the session provider are implemented to demonstrate how you can implement actions appropriate to the provider.

Essentially, the provider obtains session data, serializes it, and stores it in a file. The expiration handling is based on the file’s last access time stamp. Concurrent access is also handled on the file level.

#### Configuring the Provider

Before you start working with the provider, you must configure the sessionState element in the *web.config* file.

Listing 6-4: Configuring the Provider

<sessionState

cookieless="true"

regenerateExpiredSessionId="false"

timeout="20"

mode="Custom"

customProvider="FileSessionProvider"

sessionIDManagerType="Apress.SessionState.Code.SimpleSessionIDManager,

Apress.SessionState">

<providers>

<add name="FileSessionProvider"

type="Apress.SessionState.Code.FileSessionStateStore,

Apress.SessionState"

basePath="c:\temp"/>

</providers>

</sessionState>

There are at least three required settings. Firstly, the mode must be set to “Custom”. Secondly, the customProvider attribute must be set to the name of the provider. Finally, the provider itself must be configured as a sub-element of the sessionState element. The additional basePath attribute is a private configuration setting used in the code above.

With these settings, all session data is written to and retrieved from the file system, using the path *C:\temp* for the default (root) application and *c:\ temp\<appname>* for any other application.

# Summary

In this chapter, we examined several persistence techniques, and discovered that both page persisters and session state persisters exist in order to overcome the limitations of HTTP’s statelessness.

Page persisters store page or form related data using control state and view state. The default storage location is a hidden field within the form. An alternative method is to store the data in local memory. The provider model is used to customize this storage, and the sample implementation demonstrates how to store data in the file system.

Session state persisters, which are also based on the provider model, store session related data. The example above also shows how to replace the default storage with custom storage, and how to store the data in the file system.

Extending persisters is a great way of improving performance, maintaining data within a web garden or farm, adding special features in order to handle specific environmental conditions, or better serving particular clients such as mobile devices.