

Give Your Session State Keys and Let It Go Out (of Process)

by Dino Esposito

Although HTTP is a stateless protocol, Web applications can't just do without certain forms of state. One of the most used is session state; that is, state specific to a user that is valid as long as that user works with the application. When ASP was introduced, session management across Web farms and other clustered Web server architectures was a huge problem. Web farms arose from the need for highly scalable systems that were also robust enough to survive failovers and any sort of runtime anomalies, but they created their own problems for developers.

Classic ASP doesn't provide the developer much in the way of facilities designed to specifically address the issue of clustered servers. In essence, the introduction and success of ASP as a Web development technology rushed the demand for scalable multiserver hardware solutions, causing a chicken-and-egg problem.

In classic ASP, the Session object soon became the best friend of many programmers mostly because of its immediacy and ease of use in making the state of a page persistent, despite the stateless nature of the underlying protocol. Here, I'll ignore any scalability issues raised by a massive use of session state and go straight to the next step: how to bring the session state beyond the physical boundaries of a single Web server machine.

The problem of distributed session storage is probably as old as ASP itself. In ASP.NET, session management supports distributed scenarios natively and supplies a common API irrespective of the storage medium and the location of the data. This article will discuss the various implementations of session state and explores the impact each may have on the rest of the system.

Don't Worry, Be Global

In ASP, the native Session object has one important limitation: it can store read and write state only from and to the local memory of the host machine-the Web server. In addition, there's no way for you to configure the object to behave differently. To achieve a global session state, many programmers replace the Session object with another COM object that uses a cross-machine storage medium.

There are various ways to "replace" the native Session object; the simplest and least intrusive is just to avoid the use of Session. You create a new COM object, make sure it exposes a dictionary-like interface, instantiate it in the global.asa file, and go. If you name the instance of this object MySession, then you just replace any occurrence of Session with MySession throughout the pages and you're finished.

Using a dictionary-like programming interface for your MySession object means you can maintain the same coding style you would use with the original Session object.

The key difference is in the storage medium used to persist data-memory for the native ASP object; typically a database table for the custom object.

Obviously, I've left a lot of things out in this quick outline of a typical ASP solution. Some developers created a table row per each session entry using the session ID as the key; others packed all the session entries into a custom array-like structure, then packed into a single table row. All of us had to decide how to retrieve and set data. Lots of developers chose to read/write directly from/to the database with no memory caching in the middle.

Finally, ASP.NET came to the rescue with its variegated implementation of the session state.

The State of the Session in ASP.NET

In ASP.NET, session state has one common API, but three different implementations for the storage media. The building blocks of session state in ASP.NET are detailed in the following table:

Block	Details	Description
Session ID generator	HTTP module	Generates the ID using a random algorithm
Session storage medium	User's choice	State can be stored in the local memory, the memory of a remote server, or even a SQL Server table
Session state format	SessionStateItem	Class used to serialize and deserialize all items in a single session state
Session	Dictionary	Class used to back the familiar Session syntax

The session ID generator is an HTTP module. The component is called to ensure that valid IDs can't be guessed from existing ones. The algorithm generates 15 random numbers and maps them to the alphabet of valid symbols—letters and digits. The

resulting combination of characters forms the session ID. The session ID is the primary key to identify session state objects in the storage medium.

The storage medium is where the session state is actually persisted for the lifetime of the session. In ASP.NET 1.x, there are three possible options, as below.

Storage	Description
Memory	Default option, nearly identical to classic ASP. The session state is stored in the Cache object in the memory of the ASP.NET worker process. The process is aspnet_wp.exe or w3wp.exe depending on the operating system.
State Server	Session state is held in the memory of an external process, specifically a Windows NT service named aspnet_state.exe. The server state requires manual start and uses .NET Remoting to move data back and forth.
SQL Server	Session state is persisted in a SQL Server table. The instance of SQL Server can be local or remote and it uses the credentials you set.

The working mode is configurable in the web.config file and can be changed at any time. The impact on the existing pages is null in terms of required changes but significant in terms of performance. There's no need to touch pages if you simply want to move session state persistence to another medium. However, any change to the web.config file automatically restarts the application, thus delaying any first new request to pages.

Of the building blocks listed in the previous table, only the storage medium is modifiable in ASP.NET 1.x. Other aspects of the session state implementation are hard to customize in version 1.x. That will change in the forthcoming ASP.NET 2.0.

Customizing modules part of the session management infrastructure isn't impossible; it's just objectively hard and challenging. Wintellect's Jeff Prosis showed the way in his **August 2004 Wicked Code column** in MSDN Magazine.

Despite the three different storage media supported, the programmer is given just one API—our old friend the Session object. In classic ASP, it is a COM object instantiated in the asp.dll ISAPI extension and injected in the memory space of the ActiveX Scripting engine that is called to parse and process the .asp script.

In ASP.NET 1.x, there's a collection object behind the Session property of the Page class. The exact type is HttpSessionState; it's a not-further-inheritable class that implements ICollection and IEnumerable. An instance of this class is created during

the startup of each request that requires session support. The collection is filled with name/value pairs read from the specified medium and attached to the context of the request—the HttpContext class. The Page's Session property just mirrors the Session property on the HttpContext class.

In classic ASP, the Session COM object provides both the access and storage APIs at the same time. The object hides the details of the storage medium (local memory) and just reads and writes from and to it. An extended and extensible model is available in ASP.NET. See the diagram below.

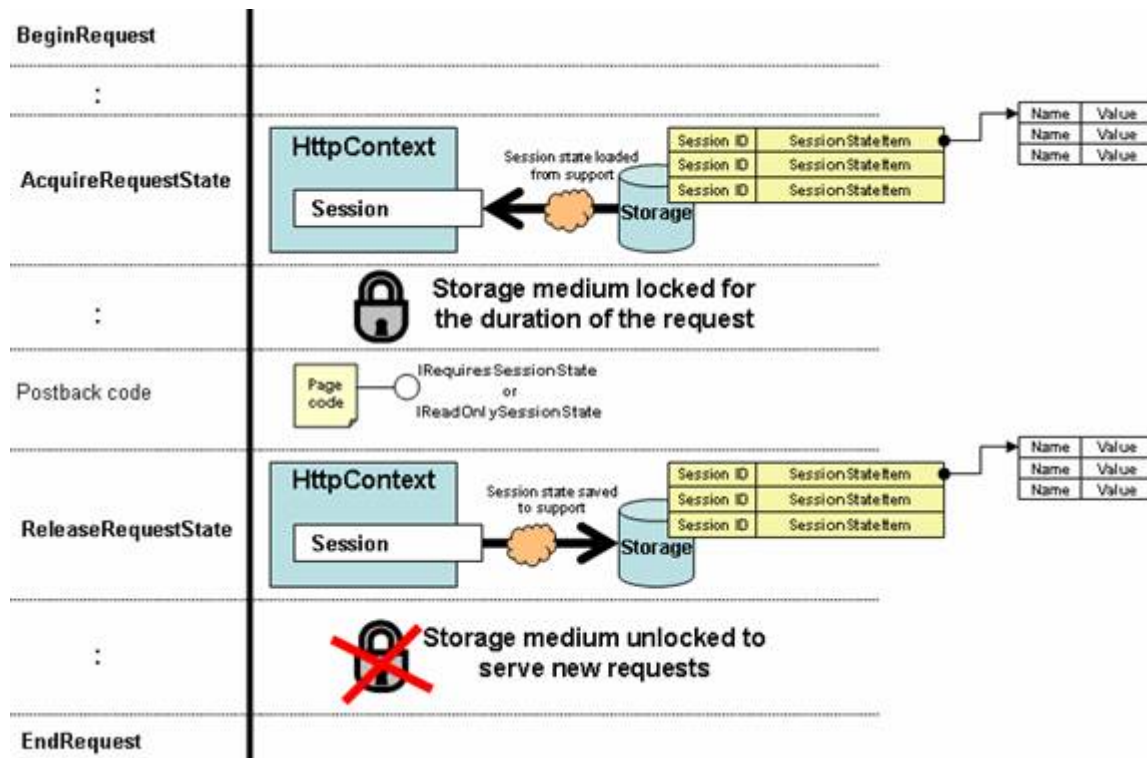


FIGURE 1-Session state lifecycle in the context of a request

The session state is managed by an HTTP module (named `SessionStateModule`) that wires up to the `AcquireRequestState` application event, reads any settings in the `web.config` file, and access the session state provider. The state provider is an object that wraps a storage medium—either local memory, another process memory, or SQL Server. The state provider implements the `IStateManager` interface so the module can call it.

Session state is stored in rows identified by the session ID. The value of the row is an all-encompassing object that contains a serialized version of the classic name/value dictionary. The contents of the state is then loaded into an instance of the `HttpSessionState` object and attached to the request.

What happens next? The current request walks its way to the end and executes any postback code. The `Session` keyword refers to the previously extracted session state. Each request is processed independently and concurrently on separate threads in the same `AppDomain`. What if a second request in the same session comes in while the system is processing the first? And what if the incoming request needs to access session state earlier than the first completes?

To preserve state consistency, a locking mechanism is essential. The row in the storage medium that represents a session with pending requests is locked and access to it is serialized. The second request is queued and served only once the first has completed. You can't assume anything about the time each request takes to terminate; there's no guarantee either that requests won't affect one another's state. Requests are processed concurrently but access to session state (and the application's state stored in Application and Cache) is automatically synchronized on a rigorous first-in-first-out basis.

This architecture has one key benefit. The model is highly extensible and allows you to store the session state in various locations, even user-defined locations. (This will happen in ASP.NET 2.0 but, architecturally speaking, there's nothing to prevent it already in ASP.NET 1.x.)

The drawback is that data must be copied into an intermediate and temporary data container—the HttpSessionState collection. For obvious reasons, this must take place once per request. The performance hit is not relevant as long as the data store is local worker process memory. It gets more significant if you keep session state out of the current process. In this case, in fact, you ought to account for an extra serialization/deserialization step. Depending on the nature of the data store, this additional step can be quite costly. The diagram below outlines the connection between the session module and the data stores.

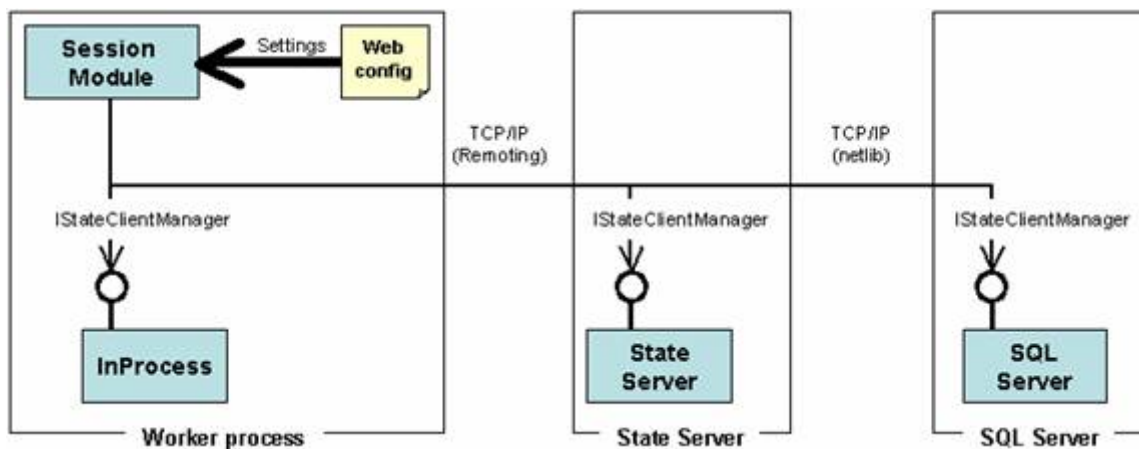


FIGURE 2—Provider model for session management

Let's examine the impact that out-of-process schemas have on Web applications, and why you might want to use them.

Serialization & Sons

The HTTP session state module uses the `IStateClientManager` interface to communicate with state providers. In ASP.NET 1.x the number of possible state providers is hardcoded; this aspect of the model is open and extensible in ASP.NET 2.0.

The main advantage of out-of-proc session is application robustness. The application can happily survive any Web server stops, failures, and anomalies. When the Web server restarts, the session state is still there if it has not timed out. The application can restart for runtime anomalies but also because of mundane things like changes to the Bin folder or the `web.config` or `global.asax` files.

In an out-of-proc scenario, though, data must be moved back and forth across two distinct `AppDomains`. The physical link with providers is the network and TCP/IP is the underlying protocol. The high-level API used to exchange data is `.NET Remoting` if you opt for a state server; it is the network library chosen in the connection string in case of SQL Server-based storage.

To move data over the wire, a serialization format is used. As you know, the `.NET Framework` comes with a couple of formatter classes—that is, classes that take an object (or a graph of objects) and write them down as an array of characters. The `BinaryFormatter` class, for instance, returns an array of bytes; the `SoapFormatter` returns a SOAP representation of the object. Choosing between the two in this case is a no-brainer—you have no real choice other than the binary formatter.

The problem is that even the faster formatter—the binary formatter—is not fast enough. In ASP.NET 1.x, an optimized, private formatter class is used for the most common types. The standard binary formatter implements sophisticated features such as circular references and nested type management that slow performance, yet are necessary for a general-purpose formatter. A straight-to-the-point formatter (an internal class named `AltSerialization`) is used to serialize numbers, strings, dates, and arrays in the fastest possible way. Other classes, as well as user-defined classes, are serialized using the `BinaryFormatter` class.

The bottom line is that to maintain session state performance at an acceptable level, “simple” types and a comma-separated sequence of strings are better than a neater and more elegant custom class. Non-serializable objects and COM objects cannot be stored in the session state when an out-of-process storage mechanism is used.

The State Server Option

Storing session state into an external process memory requires this process to be up and running all the time on the Web server machine or any other production box you want to use. By default, ASP.NET expects you use the state server process on the local machine, but this is not mandatory. Any settings about the state server are read from the `<sessionState>` section in `web.config`.

The two attributes in the section that are specific to the state server are `stateConnectionString` and `stateNetworkTimeout`. The `stateConnectionString` attribute indicates the server name (or TCP/IP address) and port to use for the communication. The default address is 127.0.0.1 (local machine), whereas the default port is 42424. The `stateNetworkTimeout` attribute indicates the allowable timeout for network operations. The default is 10 seconds.

The ASP.NET state provider is a Windows NT service, `aspnet_state.exe`, which is installed along with the framework. The service requires manual startup and runs under the ASP.NET account. The account, though, can be configured and changed at will using the Service Control Manager interface. The state service is slim and simple and does not implement any special features. It simply holds data and listens to the specified port for requests to serve. The service isn't cluster-aware (i.e., it doesn't provide a failover monitor to be error tolerant) and can't be used in a cluster when another server takes on the one that fails.

What's the impact of out-of-proc serialization in this case? For the state server, it adds up at least 15% of extra cost per request. This number represents a lower bound and can even be higher if custom classes are widely used.

The SQL Server State Option

Using a Windows NT service to manage session state storage gives applications more stability in return for a diminished per-request speed. If you are more sensitive to robustness, instead, you might want to consider SQL Server storage. In the `<sessionState>` section you define the connection string for the instance of Microsoft SQL Server to use. You're allowed to specify credentials and network path, but not the database name and structure.

The database must be created with all of its bells and whistles (read, tables and stored procedures) using a T-SQL script that ships with the ASP.NET framework. Two pairs of install/uninstall scripts are available. One configures the new database to store session information in the temporary `tempDB` memory table; one creates persistent tables.

The database contains as many rows as there are active sessions; each row is identified by the session ID. A background job will clear sweep away unused sessions periodically.

Using SQL Server, each request will take about 25% longer, in order to download and upload state to the table.

In classic ASP, storing an ADO Recordset object in the session state was a potentially dangerous action due to threading issues. Fortunately, in ASP.NET no thread-related issues need make you sweat. However, you can't just store any object to Session and be happy. If you use an out-of-process scheme, then you ought to pay a lot of attention to storing DataSets. Because the DataSet is a complex type, it gets serialized through the binary formatter. The serialization engine of the DataSet, though, generates a lot of XML data, which turns out to be a serious flaw especially for large applications that store a large quantity of data. In fact, you can easily find yourself moving megabytes of data per each request. Just avoid DataSets in ASP.NET 1.x out-of-process sessions. In ASP.NET 2.0, set the new RemotingFormat property before you store it.

Summary

In ASP.NET, the architecture of the session state has been generalized to support a variety of data stores, both in-process and out-of-process. In this article, I reviewed some aspects of the out-of-process session state implementation, paying attention to the underlying plumbing. The high-level AP doesn't change as you switch the back-end data store.