CHAPTER 22

# Session Management

The HTTP session API is an essential component in constructing interactive web sites. The session API of the Java Servlet specification provides a mechanism for associating a series of requests with a specific browser or user. This is required because the Hypertext Transfer Protocol (HTTP) employed for web browser to web server requests is a stateless protocol. As a result, a web server has no means of associating a series of requests with a specific browser or user. This chapter will expand on the coverage of HTTP session from Chapter 4, "Build and Deploy Procedures," by providing a brief overview of HTTP session and will then discuss the WebSphere Application Server (WAS) session management implementation, as well as the specifics of configuring the various session management options that exist in WAS.

## Introduction to HTTP Session

It's almost impossible to visit any interactive web site today that does not make use of the HTTP session API. By providing multiple options for tracking a series of requests and associating those requests with a specific user, HTTP session allows applications to appear dynamic to application users. The most often cited example of HTTP session is the creation of a "shopping cart" for shoppers on a web site. In this example, information associating the user and their prior navigation through the web site and their selections are stored as objects in HTTP session. Once the users are ready to check out from the web site and purchase their selections, the application typically constructs a page composed of all the selected items stored in the "shopping cart."[1] By maintaining application state between browser requests, HTTP session overcomes the default stateless behavior for HTTP requests.

1 It's worth noting that this is actually a poor use of a session. Sessions are not designed for robust "permanent" storage. By storing the shopping cart in a session, if there were a client failure (perhaps the browser crashed), the user would lose his or her entire shopping cart. That's a great way to lose a sale. Important information is better stored in a database directly.

The HTTP session API component of the Java Servlet specification provides a mechanism for web applications to maintain a user's state information, and this mechanism addresses some of the problems with other options for maintaining state, such as those based solely on cookies. This mechanism, known as a session, allows a web-application developer to maintain user state information on the server, while passing minimal information back to the user to track the session via one of three options: cookies, URL encoding, and SSL sessions.

## Session Tracking

In order to associate the user's session with a particular browser, WAS needs to maintain some form of association with the client browser. There are several techniques available, and we'll detail them here. It is important to note that while WAS allows for great flexibility in choosing the tracking mechanism, cookies are by far the best approach.

### Cookies

The use of a cookie for tracking session state is the default in WAS. This option differs from a pure cookie-based solution in that the HTTP session uses a single cookie named JSESSIONID that contains the session ID, which is used to associate the request with information stored on the server for that session ID, while an entirely cookie-based solution would employ multiple cookies, each containing possibly sensitive user state information (account number, user ID, etc.). With HTTP session, all attributes associated with the user's request are stored on the server. Since the only information transmitted between the server and the browser is the session ID cookie, which has a limited lifetime, HTTP session can provide a much more secure mechanism than cookies for tracking application state when configured in conjunction with SSL.

The mechanics of using a cookie for tracking session are depicted in Figure 22-1. A request arrives at the server requiring that a session be created as the result of a getSession() method call. The server creates a session object, associating a session ID with it. The session ID is transmitted back to the browser as part of the response header and stored with the rest of the cookies in the browser. On subsequent requests from the browser, the session ID is transmitted as part of the request header, allowing the application to associate each request for a given session ID with prior requests from that user.

The interaction between browser, application server, and application are all handled transparently to the end user and the application program, aside from the getSession() method call inside the application. The application and the user need not be, nor are they likely aware of the session ID provided by the server.

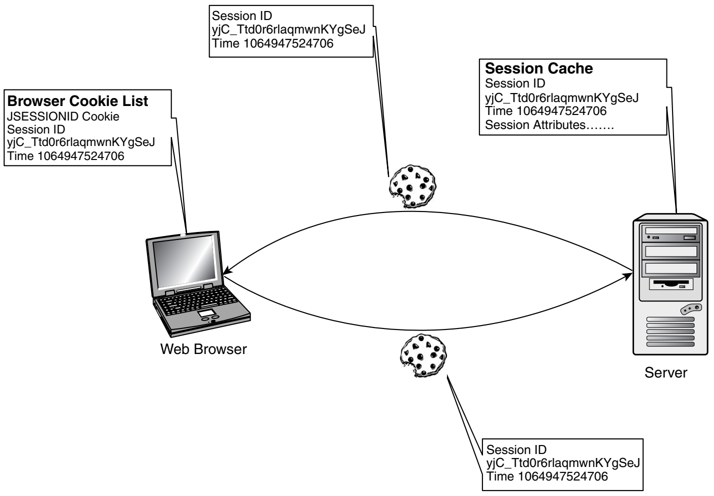


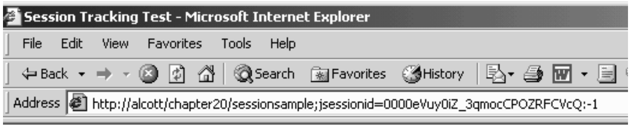
**Figure 22-1** Browser-session manager cookie interaction.

Unfortunately, not all users configure their browsers to accept cookies. Often, this is related to security concerns over accepting a cookie into the browser. Most often the restrictions on accepting a cookie into the browser are the result of concerns about persistent cookies. Persistent cookies remain in the browser after the browser is closed and allow a web site to "remember" you on ensuing visits. Most important in terms of presenting a risk, persistent cookies may contain personal information about you, which is then accessible to any application that has access to the cookie folder in your browser. However, WAS generates a session cookie that exists only until the browser has been closed and is used only to ensure that you are "recognized" as you move from page to page within the web site. This technique is not generally considered to be a security or privacy concern. In fact, most modern browsers can be configured to accept all session cookies while still blocking persistent cookies.

Finally, in a WAS environment with security enabled, disabling all cookies is actually counterproductive since the most often used authentication mechanism is LTPA (Lightweight Third Party Authentication), and LTPA relies on creating a token (cookie) that is used to represent the identity of the authenticated user to the browser. In any event, other options exist for maintaining an application state such as URL rewriting, although they are not recommended.

## URL Rewriting

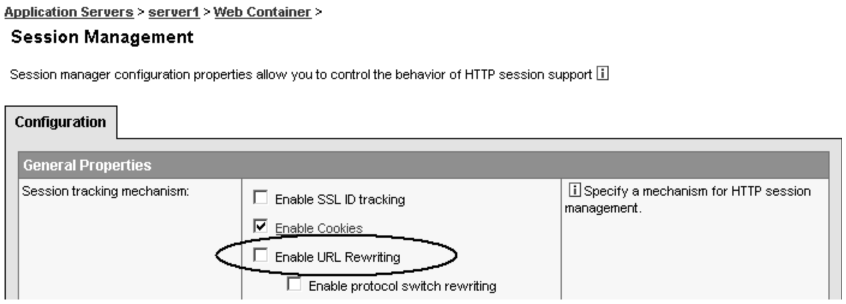
Most often, URL rewriting is employed when a browser is configured not to accept cookies. URL rewriting stores the session identifier in the page returned to the user. WAS encodes the session identifier as a parameter on any link or form the user submits from a web page. This option requires that the Servlet code be modified to include either an encodeURL() or encodeRedirectURL() method. When a browser user clicks on a link that utilizes one of these methods, the session identifier passes into the request as a parameter, with the result shown in Figure 22-2, where the session ID is appended to the URL for the web page.


Session Tracking Test - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address

http://alcott/chapter20/sessionsample;jsessionid=0000eVuy0iZ_3qmocCPOZRFCVcQ:-1

**Figure 22-2** URL rewriting example in browser.

The requirement that the application developer write additional code for a Servlet or JSP presents a major disadvantage for URL rewriting when compared to other available session tracking mechanisms. Additionally, URL rewriting limits the flow of site pages exclusively to dynamically generated pages (those generated by Servlets or JSPs). While WAS can insert the session ID into dynamic pages, it cannot insert the user's session ID into static pages (.htm or .html pages). As a result, after the application creates the user's session data, the user must visit dynamically generated pages exclusively until he or she finishes with the portion of the site requiring sessions. URL rewriting forces the site designer to plan the user's flow in the site to avoid losing his or her session ID. Lastly, the system administrator must configure WAS to enable URL rewriting by checking the box shown in Figure 22-3.


Application Servers > server1 > Web Container >

Session Management

Session manager configuration properties allow you to control the behavior of HTTP session support

Session tracking mechanism:

- [ ] Enable SSL ID tracking
- [✓] Enable Cookies
- [ ] Enable URL Rewriting
  - [ ] Enable protocol switch rewriting

Specify a mechanism for HTTP session management.

**Figure 22-3** Session tracking mechanism dialog.

Fortunately, the WAS session management implementation can recognize when a browser is configured to accept cookies and will use this option instead of URL rewriting in cases when both cookies and URL rewriting are enabled.

## SSL ID Tracking

Another alternative tracking mechanism is the Secure Socket Layer (SSL) ID that is negotiated between the web browser and the HTTP server. While secure, this alternative presents a number of disadvantages. Since this ID is negotiated between the browser and the HTTP server, the failure of the HTTP server results in the loss of this ID. In a clustered environment, the IP sprayer used to direct requests to the HTTP servers must provide some sort of affinity mechanism so that browser requests return to the HTTP server that negotiated the ID with the browser. Additionally, in a clustered environment with multiple application servers, either cookies or URL rewriting must be enabled for affinity between the HTTP server and application server. Lastly, in a fashion similar to URL rewriting, the administrator must also explicitly configure WAS to employ this option by selecting this option on the configuration dialog (refer to Figure 22-3).

## The Session API

The interface declaration of HttpSession contains over a dozen methods in all. Minimally, an application will likely call three of these methods. The first of these is the getSession() method, which is used to either create a session object if one does not already exist or to associate a request with an existing session. In order to store information in the session object, the setAttribute() method is called, and the application retrieves information that is stored via a call to the getAttribute() method. Often overlooked in an application is the invalidate() method call. There should also be some provision in the application to invalidate the session object once the application no longer requires the session:

```
session.invalidate() ;
```

Unless configured to never do so, the web container will eventually invalidate the session object once the inactive interval for the session is reached. When the application explicitly invalidates the session, it reduces the overhead on the runtime of tracking sessions that are no longer required. You can also minimize the size of session objects prior to invalidation by removing attributes no longer required by explicitly calling the removeAttribute() method in the application.

Another constraint exists for applications that implement distributable sessions, which are sessions that can be handled by more than one web container, typically in order to provide for failover. In this case, all objects placed into HttpSession via the `setAttribute()` method must be serializable. WebSphere Application Server does provide for exceptions to this requirement for some J2EE objects that are not serializable:

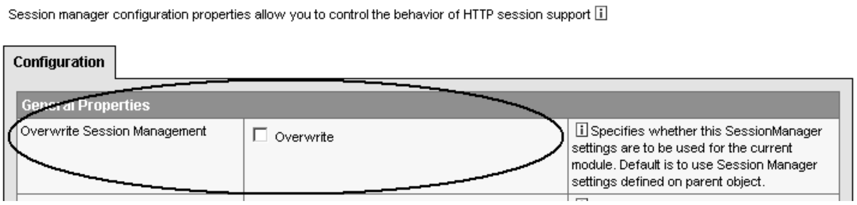
- javax.ejb.EJBObject
- javax.ejb.EJBHome
- javax.naming.Context
- javax.transaction.UserTransaction

The specifics on how WAS overcomes this restriction vary for each of these objects. Except for UserTransaction, which requires a WebSphere specific public wrapper object, the mechanism employed is transparent to the application.

It is best if all objects placed into HTTP session are serializable or you'll experience a java.io.NotSerializable Exception, as shown in Code Snippet 22-1. This provision ensures transparent deployment in both non-clustered and clustered environments, without requiring application changes for the latter. It's also important to note that it's not sufficient that the application simply implement the java.io.Serializable interface; the objects must actually be serializable, which is an important but subtle distinction that we'll return to later.

## WAS Session Management Configuration

WebSphere Application Server V5 provides a large number of options for configuring the behavior of the session manager portion of the runtime. The default is for these options to apply to all applications running inside a given application server, but they can also be configured on a case-by-case basis for each enterprise application running on an application server or web applications within an enterprise application. By tuning at the application level, you ensure that the application behaves in the same fashion, regardless of the configuration of the application server. Customization of the session manager for a web application or enterprise application is performed by selecting the **Overwrite Session Management** check box for the enterprise application or web application, as shown in Figure 22-4.



Session manager configuration properties allow you to control the behavior of HTTP session support

| | | |
|---|---|---|
| Overwrite Session Management | ☐ Overwrite | Specifies whether this SessionManager settings are to be used for the current module. Default is to use Session Manager settings defined on parent object. |

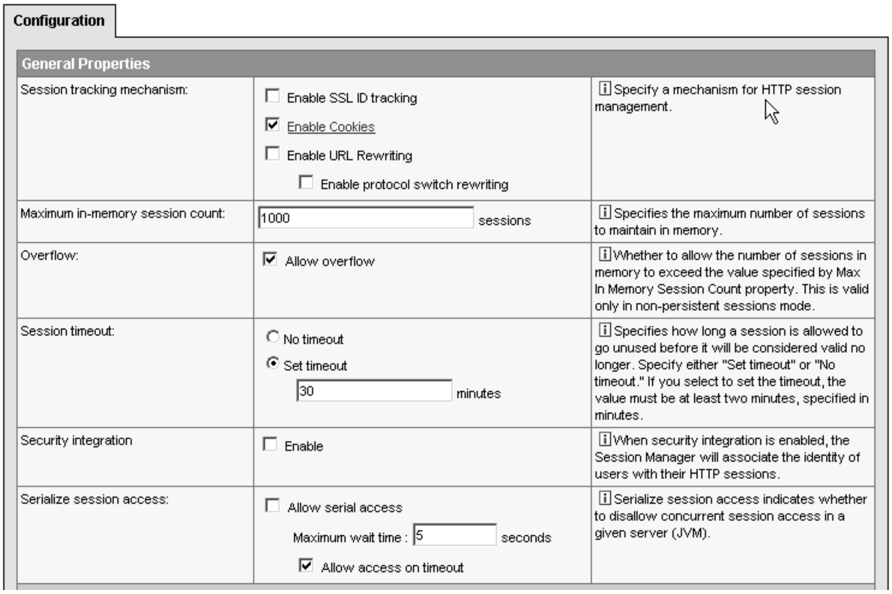
**Figure 22-4** Overwriting session management for an EAR or web application.

The Configuration tab depicting the Overwrite property is located on the Session Manager dialog for both the enterprise application and web applications, which is reached from the administration console by navigating to **Applications > Enterprise Applications > Additional Properties > Session Management**. In cases where you want to overwrite the properties for a web application, the dialog is reached by selecting the **Web Modules > Session Management** for a specific enterprise application. All of the options described here can be configured specifically to a web application or enterprise application, so as an example, one application in an application server could persist session to a database, while other applications could use memory-to-memory replication.

### Local and Distributed Session Options

The default with WAS V5 is to store the HTTP session object locally as part of the application server JVM. WAS also provides for distributed sessions that can be replicated in memory to other application servers or persisted to a database. Distributed sessions provide for failover of the session object to a surviving application server in the case of an application server outage. Through the use of cookies or URL rewriting, WAS provides affinity so that requests from a specific browser return to the application server where the session object was initially created. The configuration options common to both local and distributed sessions are depicted in Figure 22-5, though the meaning for some of these varies on whether local or distributed sessions are in use.

This dialog is reached by navigating to **Application Servers** > *application server name* > **Web Container** > **Session Management** in the administration browser client.


Configuration

General Properties

| | | |
|---|---|---|
| Session tracking mechanism: | Enable SSL ID tracking [✓ Enable Cookies] Enable URL Rewriting Enable protocol switch rewriting | Specify a mechanism for HTTP session management. |
| Maximum in-memory session count: | 1000 sessions | Specifies the maximum number of sessions to maintain in memory. |
| Overflow: | [✓ Allow overflow] | Whether to allow the number of sessions in memory to exceed the value specified by Max In Memory Session Count property. This is valid only in non-persistent sessions mode. |
| Session timeout: | No timeout [✓ Set timeout] 30 minutes | Specifies how long a session is allowed to go unused before it will be considered valid no longer. Specify either "Set timeout" or "No timeout." If you select to set the timeout, the value must be at least two minutes, specified in minutes. |
| Security integration | Enable | When security integration is enabled, the Session Manager will associate the identity of users with their HTTP sessions. |
| Serialize session access: | Allow serial access Maximum wait time : 5 seconds [✓ Allow access on timeout] | Serialize session access indicates whether to disallow concurrent session access in a given server (JVM). |

**Figure 22-5** Session management configuration.

## Maximum Session Count and Allow Overflow

The next two parameters on this dialog are for the maximum number of session objects stored in memory and whether the maximum can be exceeded. This allows you to limit the memory footprint of sessions stored locally or in local cache when distributed sessions are in use. When local sessions are in use, the session count specifies the number of sessions that are stored in memory, assuming that you do not specify “Allow Overflow.” When Allow Overflow is specified and local sessions are in use, a second session memory table is constructed, and all sessions greater than the specified maximum are stored there, up to the available memory for the JVM. For distributed sessions, when Allow Overflow is specified, the session manager employs a Least Recently Used (LRU) algorithm so that only the most recently used sessions, up to the maximum count, are kept in the local cache, with the remainder of the session objects either replicated in memory to a remote application server or persisted to a database depending on your configuration. Specifying Allow Overflow should probably never be configured. Doing so removes the limiting mechanism that it provides. As a result, the number of session objects could grow to consume the entire application server JVM, either as the result of a spike in load or as the result of a denial of service attack.

## Session Timeout

As its name implies, the Session Timeout parameter specifies how long an unused session exists before it times out and is removed from the memory table for local sessions or the cache for distributed sessions. This setting is an important one from a performance perspective because of the memory impact that unused sessions can have on the application server JVM until they are removed. In the same vein, specifying "No Timeout" can result in a memory leak since the session objects are never eligible for garbage collection by the JVM, unless the application explicitly calls session.invalidate(). As a result, No Timeout is generally not recommended, though it might be appropriate for a small percentage of applications where the user population is very small and stable. In order to minimize the memory impact of session objects, this setting should be set as low as practical in order to satisfy application requirements and use patterns. For applications with a large number of short-lived visits of perhaps a few minutes, a timeout of 5–10 minutes would likely be appropriate. Some web sites even provide a timer to show when the current session will expire and a logout function that invalidates the session.

## Session Security Integration

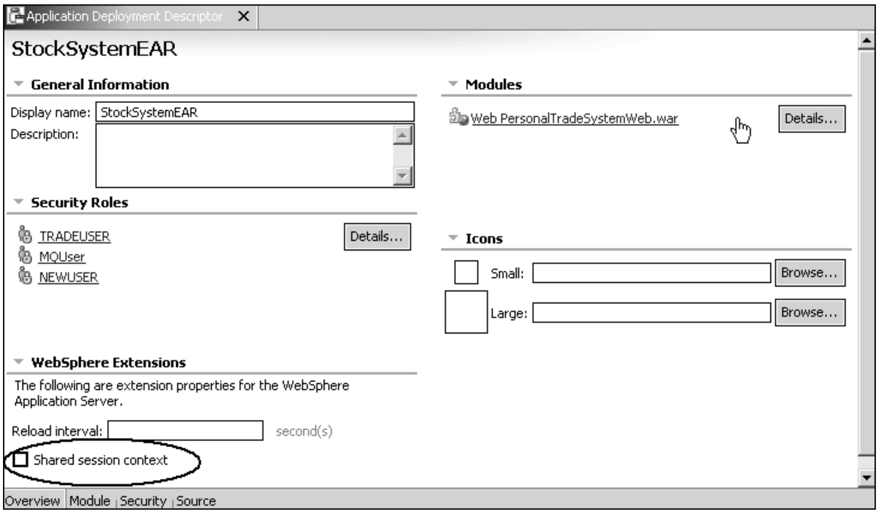
When "Security Integration" is enabled, WAS checks the user ID of the HTTP request against the user ID for the session object as part of the processing of the request.getSession() method. If the check fails, an UnauthorizedSessionRequestException is thrown. When used in conjunction with SSL, Security Integration can be used to prevent "man in the middle" intrusions. Thus, when architecting a secure infrastructure, it's recommended that this option be employed.

## Serialize Session Access

This option is somewhat misnamed since, when specified, the session manager actually synchronizes access to a session inside a JVM in order to provide thread-safe access to the session object. Prior to WAS V3.5.3, this was the default for session access, but in order to improve performance, the responsibility for thread safety was shifted from the runtime to the application developer. For best performance, it's recommended that applications continue to take appropriate measures inside the application to ensure thread safety rather than relying on this option. This means that application code must be careful when modifying a session to ensure that multiple threads don't simultaneously modify the same session object in conflicting ways. This is most likely to occur in web applications that use frames where multiple Servlets are executing on behalf of the same client simultaneously. This is fairly easy to prevent by ensuring that only one of the Servlets in the frame modifies the session and the others only read from it.

## Shared Session Context

Before leaving the session management options, there is one additional option that can be configured in the Application Server Toolkit (ASTK), WebSphere Studio, or the Application Assembly Tool (AAT). This is a J2EE extension that allows for the session object to be shared across the multiple Servlet contexts inside a single enterprise application. While this is a useful option when migrating pre-J2EE applications (Servlet 2.1) to J2EE compliance, we recommend that this be avoided if at all possible. Sharing session in this fashion tends to lead to large session objects that in turn are usually detrimental to application performance. This option is enabled in the ASTK or Studio by selecting the check box indicated in Figure 22-6.


Application Deployment Descriptor

StockSystemEAR

General Information

Display name: StockSystemEAR

Description:

Security Roles

TRADEUSER

MQUser

NEWUSER

WebSphere Extensions

The following are extension properties for the WebSphere Application Server.

Reload interval: second(s)

☐ Shared session context

Modules

Web PersonalTradeSystemWeb.war

Icons

Small:

Large:

**Figure 22-6** Session sharing option in ASTK.

## Distributed Sessions

As previously noted, distributed sessions provide for failover in case of application server outage, allowing an end user to continue using a web site without any loss of an application state that could require a re-login or navigation through previously viewed pages. While local sessions only provide for a single copy of the session object, distributed sessions provide at least one copy of the session object in addition to the local copy that is cached in the application server JVM. It's worth mentioning that aside from ensuring that the information placed into the HTTP session is serializable, there's no impact to application development when choosing to use this option.

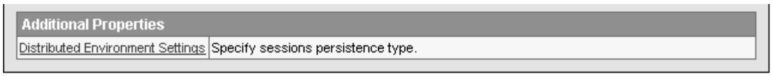
### Session Affinity

Strictly speaking, session affinity isn't an option; it's the default behavior for WAS, but before discussing the other session-management settings in WAS, it's important to have some understanding of how WebSphere Application Server implements this feature. Session affinity allows WAS to return requests associated with a given browser and session object back to the same application server instance in a clustered environment. This in turn aids performance by allowing sessions to be accessed from the cache in the application server when distributed sessions are in use and allows local sessions to be utilized in cases where session failover is not a requirement. Without affinity, browser requests would be distributed across all application servers in a cluster. The impact of requests being distributed in this fashion would depend on whether distributed or local sessions were in use. In an environment using distributed sessions, the application server would make an out-of-process request to retrieve the session object from the copy of the session object stored either in memory in another application server or in a database. In an environment employing local sessions, the getSession() method call in the application code would result in the creation of a new session object, with the loss of any data previously stored in session for that client.

In a clustered environment, WAS provides affinity to a server by appending the server ID to the session ID that is contained in the JSESSIONID cookie or URL rewrite. This information is used by the HTTP server plug-in to dispatch the request to the correct server. This can be seen by looking at the request header for the HTTP request via a programmatic call to request. getHeader("Cookie"), which will return the JSESSIONID cookie with a result similar to **0001kJLEJhMoitCnI0QTgAo5z8z:v1efc643**. The first four characters are the cache ID, followed by the session ID, then a ":" for a separator, followed by the server clone ID. In this case, **v1efc643** is the server clone ID, which corresponds to the application server defined in the plugin-cfg.xml file with the Server CloneID="v1efc643" property.

## Distribution Mechanisms

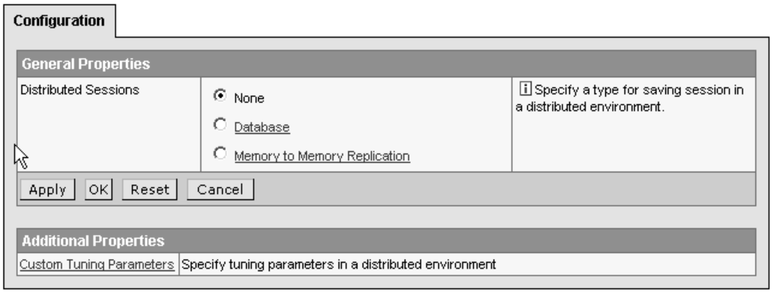
In order to maintain session state across multiple application servers, some mechanism for distribution of the session object is required. That way, if one application server should fail, when the request is routed to another application server, that server can obtain the state from a remote persistent store. In looking at the link to the Distributed Environment Settings dialog (see Figure 22-7), you'll note the word "persistence," which is probably not the best term for this, but it has been used historically, so we continue to use it here, even though only one of the options provides persistence. Keep in mind that sessions are not intended for long-term stable storage. The lifetime of their "persistence" is essentially the lifetime of the client browser. In any case, WAS provides two persistence mechanisms for maintaining session state: database and memory to memory. The dialog for configuration of Distributed Sessions is accessed from Distributed Environment Settings under the Additional Properties heading at the bottom of the session configuration dialog, as shown in Figure 22-7.



| Additional Properties | |
|---|---|
| Distributed Environment Settings | Specify sessions persistence type. |

**Figure 22-7** Link to Distributed Environment Settings dialog.

This in turn leads to the dialog depicted in Figure 22-8, where the options for Distributed Sessions are specified.



Configuration

General Properties

| | | |
|---|---|---|
| Distributed Sessions | [✓ None] Database Memory to Memory Replication | Specify a type for saving session in a distributed environment. |

Additional Properties

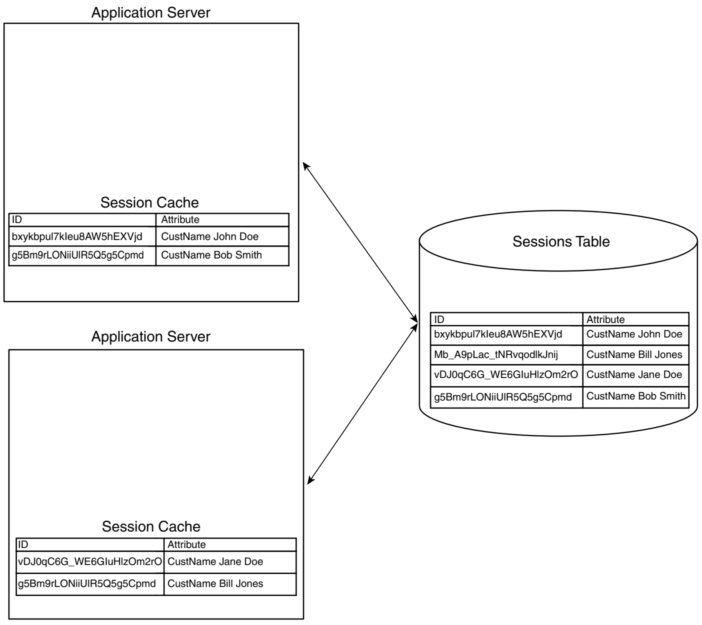
| | |
|---|---|
| Custom Tuning Parameters | Specify tuning parameters in a distributed environment |

**Figure 22-8** Session distribution options.

The default of "None" is depicted, with options for "Database" and "Memory to Memory Replication." If your web site requires that application state be maintained in the event of a server outage, you will want to provide for failover by selecting one of the two options. The decision on which option is appropriate for your environment will depend on several factors, which we'll explore, but first let's discuss how each of these options is implemented and configured.

## Database Session Persistence

When a database is used for session failover, the WAS session manager uses a database table to store a copy of the session object. This is depicted in Figure 22-9. Each application server maintains a local cache of the session object with a copy maintained in the database.



| ID | Attribute |
|---|---|
| bxykbpul7kleu8AW5hEXVjd | CustName John Doe |
| g5Bm9rLONiiUlR5Q5g5Cpmd | CustName Bob Smith |

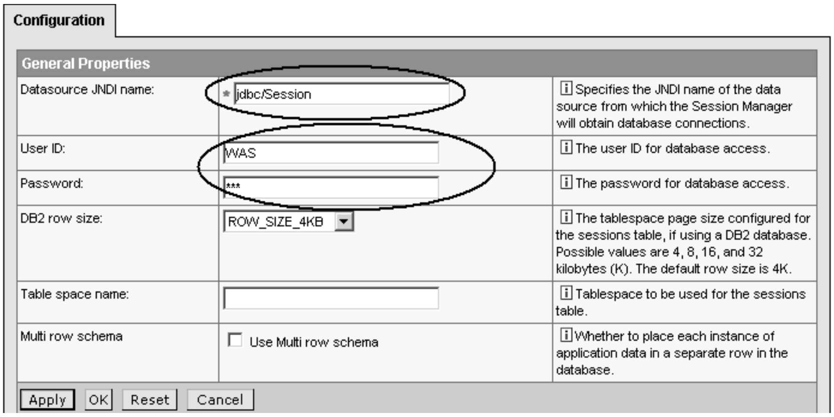
| ID | Attribute |
|---|---|
| bxykbpul7kleu8AW5hEXVjd | CustName John Doe |
| Mb_A9pLac_tNRvqodlkJnij | CustName Bill Jones |
| vDJ0qC6G_WE6GIuHlzOm2rO | CustName Jane Doe |
| g5Bm9rLONiiUlR5Q5g5Cpmd | CustName Bob Smith |

| ID | Attribute |
|---|---|
| vDJ0qC6G_WE6GIuHlzOm2rO | CustName Jane Doe |
| g5Bm9rLONiiUlR5Q5g5Cpmd | CustName Bill Jones |

**Figure 22-9** Database session persistence.

In the event that one of the application servers were to go offline, requests would be directed to the surviving application servers, and when the application calls the getSession() method, the session object is retrieved from the database and placed in the local cache.

By selecting the **Database** option shown in Figure 22-8, the session database dialog is displayed, as shown in Figure 22-10.



General Properties

| | | |
|---|---|---|
| Datasource JNDI name: | * jdbc/Session | Specifies the JNDI name of the data source from which the Session Manager will obtain database connections. |
| User ID: | WAS | The user ID for database access. |
| Password: | *** | The password for database access. |
| DB2 row size: | ROW_SIZE_4KB | The tablespace page size configured for the sessions table, if using a DB2 database. Possible values are 4, 8, 16, and 32 kilobytes (K). The default row size is 4K. |
| Table space name: | | Tablespace to be used for the sessions table. |
| Multi row schema | ☐ Use Multi row schema | Whether to place each instance of application data in a separate row in the database. |

**Figure 20-10** Configuring a database for session persistence.

At a minimum, you need to supply a JNDI name for a data source that corresponds to a database to be used, as well as the database user ID and password to be used for accessing the database. Note that when specifying the data source, it should be a non-XA data source. This is because the session manager runs with the JDBC "autocommit on" parameter, and some XA data sources don't support the use of "autocommit on." Additionally, the update of the session database is not transactional, nor will the session database participate in transactions with other resources, so there is no need for using an XA data source with the session manager.

The default maximum pool size of 10 is a good starting point when sizing the connection pool for database persistence. Unless the session object is quite large or your site is handling thousands of concurrent users, it's unlikely that the pool will need to be much larger. In most cases, the default maximum will suffice, or perhaps a moderate increase (to 22–30) may be required. Recall that a connection pool is created for each application server instance, so three or four application servers result in three to four times that many total connection objects that are accessing the database.

Depending on the size of the session object, you may find it beneficial to use one of the other options available on this dialog. We'll discuss determining the size of the session object later in this chapter, but for large session objects, meaning those greater than 4KB in size, the options available can be used to improve performance.

When DB2 is used as the database, the page size can be adjusted from the default of 4KB to 8KB, 16KB, and 32KB. This allows the **varchar** database column used for the session object to fit onto a single database page. The result is faster performance for session objects that are approximately 7KB, 15KB, and 31KB in size.

The multi-row schema option can be employed to improve performance in some cases for large session objects, regardless of the database in use. With multi-row sessions, each attribute in the session object is stored in its own row in the database, instead of the default of storing all attributes in one column in a single row. This can be used to improve performance in two ways. The first is for non-DB2 databases, where WAS makes no provision to adjust the table page size. If each attribute stored in session is smaller than the page or block size for the database, specifying multi-row allows each row to be contained in a single database block. This can be combined with the optional write setting "only updated attributes" so that only the rows corresponding to each updated attribute are written to, instead of a write to all rows.

The other advantage to using multi-row sessions is that they allow you to have a session object larger than 2MB in size. Such extremely large session objects are most definitely not recommended from a performance and failover perspective, but WAS does provide a mechanism for dealing with objects of this size.

### Memory-to-Memory Replication

As implied by the name, memory-to-memory replication stores a copy of the session object in the memory of one or more application server processes. In this mode, the WAS Distributed Replication Service (DRS) is used to replicate session information among application servers. Since DRS is used by multiple WAS runtime components, there's a discussion of the DRS architecture and configuration options in Chapter 20, "WAS Network Deployment Architecture," including instructions that walk through configuring DRS as a failover mechanism for HTTP session. As a result, if you plan to use DRS/Internal Messaging for an HTTP session failover, you should refer to that chapter. In order not to duplicate content, all we'll provide here is a brief review of the components in DRS and their role in HTTP session management:

- **Replicator**—The data transfer component of DRS running in an application server.
- **Replication domain**—The set of replicators that are connected together to share data.
- **Session manager**—The web container component that manages HTTP session objects.
- **Replication mode**—A server is a "client" if it only forwards changes to other session managers, while it is considered a "server" if it only receives changes from other session managers, and "both" (or peer to peer) if it does both.
- **Group**—A session object is assigned to a group; by default, all session managers listen to all groups, but you can partition where a session object is replicated to.
- **Single replica**—An alternative to groups in which, as the name implies, the session object is only replicated once.

There are four configuration options for memory to memory replication:

- Peer-to-peer, where each application server has a “replicator” process and stores a copy of the session object for all other application servers in the domain.
- Peer-to-peer with standalone replicators, where each application server stores a copy of the session object for all other application servers in the domain, but additional dedicated application servers run the replicator process.
- Client-server, where the application servers running the application have just a local copy of the session object for requests to that application server, and dedicated application servers store session copies and run the replicator process.
- Client-server with dedicated replicators and dedicated stores, where each application server maintains just the session object for requests to that application server, with application servers running the replicator process and additional application servers storing copies of the session objects for the replication domain.

Memory-to-memory replication provides failover in much the same manner as with database persistence. When a request arrives at an application server and the session object is not in the local cache, the session manager component of the runtime will attempt to retrieve it from the local or remote replica.

In order to configure the session manager to use memory-to-memory replication, a replication domain must be defined. This is somewhat analogous to creating a database when database persistence is to be employed and must be performed prior to specifying “Memory to Memory Replication” in the Distributed Session dialog shown in Figure 22-8.

Once you specify memory to memory as the mechanism for distributing sessions, the steps for configuring and tuning DRS are the same as those described in Chapter 20.

Before leaving the topic of distribution mechanisms, let’s give some consideration to what might lead you to choose one option over the other. Performance will not be a factor since 95% of the cost of replicating session is serialization/deserialization of the session object, which must occur regardless of how the session is distributed (memory to memory or database). Additionally, as the size of the session object increases, performance degrades, again about equally for both session distribution options. Instead, the decision will be based partially on how the two technologies differ:

- With a database, you actually persist the data (to disk), so a highly available database server can survive a cascading failure, while using application servers as session stores and replicators for this purpose may not.

- In the case of a "gold standard" (two identical cells/domains), a highly available database can pretty much assure session failover between domains, while with memory to memory, there can only be a single replicator common to the two cells; hence, it becomes a single point of failure (SPOF).[2]

Thus, for configurations where cross-cell session failover is a requirement, a highly available database is the only option for eliminating a SPOF. Note that while sharing sessions across cells is supported, this is not generally recommended. By sharing state between cells, it makes it significantly more difficult to independently upgrade components (application and WAS) in the two cells.

In the end, the decision then becomes based on what technology you are most comfortable with and which delivers the required quality of service for your availability requirements.

## Custom Tuning Parameters for HTTP Session

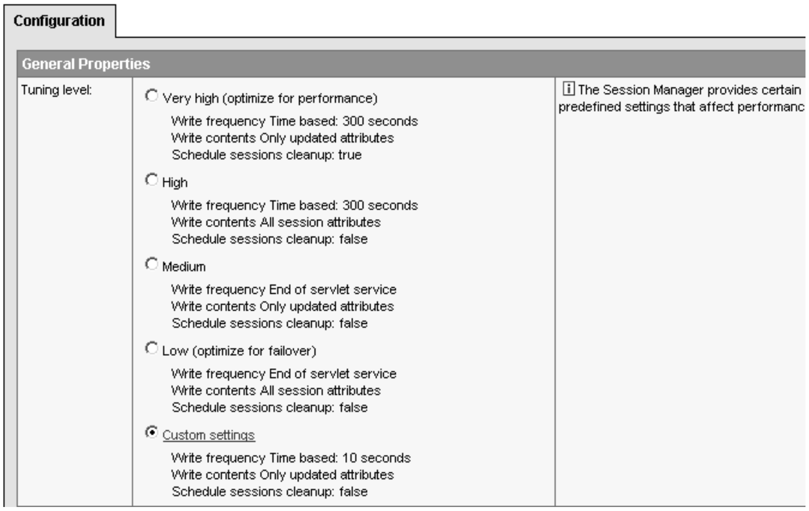
The Custom Tuning Options dialog controls the frequency and type of updates that the session manager makes to either the database or to memory-to-memory replicas. This dialog is reached by selecting **Custom Tuning Parameters** on the bottom of the **Distributed Environment Settings** dialog (Figure 22-8) and is depicted in Figure 22-11. In prior versions of WebSphere Application Server, the default was to update the session object at the end of the Servlet service method, and while this remains an option in V5 ("medium tuning" in the Tuning dialog), the default in WAS V5 is for a time-based write of the session updates. As of V5.02 (and above), the default time interval is every 10 seconds; earlier WAS V5 releases defaulted to 120 seconds.[3]

While time-based writes offer better performance than writing at the end of the Servlet service method, they also introduce the possibility of inconsistent application state. By deferring updates to the end of the specified time interval, an application could appear to move "backward" in state in the event of an application server failure during the specified update interval. Session changes made between updates will be lost while the end user will continue to use the web site. It is possible that the application's session state could move backward in time while the web page being viewed does not reflect this, potentially resulting in problems. Tests by the WebSphere performance lab have shown that a 10-second interval provides essentially all of the performance benefit of longer intervals, while limiting the vulnerability of inconsistent session state.

While requiring more resources (CPU, I/O), the update of the session object at the end of each Servlet service method ensures data consistency in the event of the failover of requests from one application server to another.

2 This is because all the application servers (in both cells) must be defined as "clients" to that server, and the admin console only gives you the ability to provide one replicator IP address and port at a time on a server, so if that replicator were to go down, then it would amount to a SPOF (single point of failure). This differs from the default behavior in a single cell, where the multiple application servers in a replication domain can be configured in a "client server" configuration.

3 The longer default interval was simply the result of limited testing and development resources that were available prior to the V5.0 release. The 10-second interval is a recommended starting point in V5.0 and V5.01, though you may want to tune it for your application and environment.



Configuration

General Properties

Tuning level:

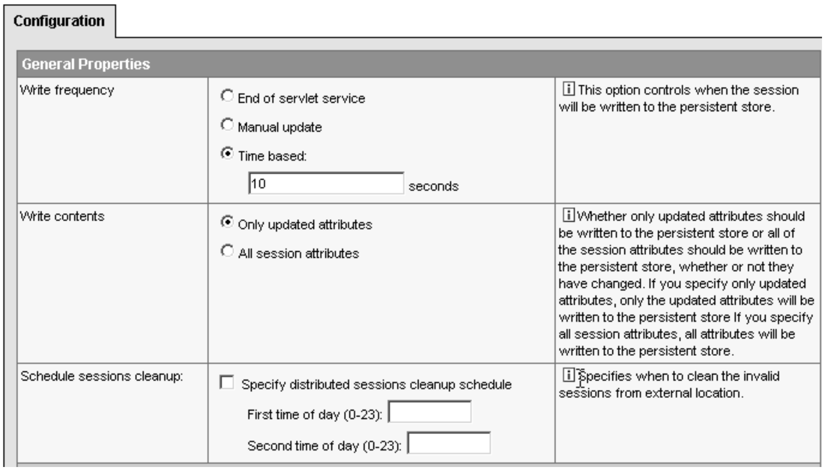
- ( ) Very high (optimize for performance)
  Write frequency Time based: 300 seconds
  Write contents Only updated attributes
  Schedule sessions cleanup: true
- ( ) High
  Write frequency Time based: 300 seconds
  Write contents All session attributes
  Schedule sessions cleanup: false
- ( ) Medium
  Write frequency End of servlet service
  Write contents Only updated attributes
  Schedule sessions cleanup: false
- ( ) Low (optimize for failover)
  Write frequency End of servlet service
  Write contents All session attributes
  Schedule sessions cleanup: false
- [✓ Custom settings]
  Write frequency Time based: 10 seconds
  Write contents Only updated attributes
  Schedule sessions cleanup: false

The Session Manager provides certain predefined settings that affect performanc

**Figure 22-11** Session Write Frequency Tuning dialog.

In order to avoid possible application state inconsistencies, the authors favor the **Medium** tuning setting shown in Figure 22-11 since it minimizes the cost of writing updates by only writing the session attributes that have been updated.

You can further refine the tuning levels depicted by selecting **Custom Settings**, which invokes the dialog shown in Figure 22-12.



Configuration

General Properties

| | | |
|---|---|---|
| Write frequency | ( ) End of servlet service<br>( ) Manual update<br>[✓ Time based:] 10 seconds | This option controls when the session will be written to the persistent store. |
| Write contents | [✓ Only updated attributes]<br>( ) All session attributes | Whether only updated attributes should be written to the persistent store or all of the session attributes should be written to the persistent store, whether or not they have changed. If you specify only updated attributes, only the updated attributes will be written to the persistent store If you specify all session attributes, all attributes will be written to the persistent store. |
| Schedule sessions cleanup: | ☐ Specify distributed sessions cleanup schedule<br>First time of day (0-23):<br>Second time of day (0-23): | Specifies when to clean the invalid sessions from external location. |

**Figure 22-12** Session Tuning Custom Settings.

The primary options of interest here are the ability to specify a manual update, though this requires that the application code use the IBMSession class, which is an extension to the HTTP session API, for managing sessions. With a manual update, the session manager only writes the session data and last access time to the session replica when the application invokes the sync() method in the IBMSession class. The session data that is written out to the replica is controlled by the write contents option selected. If the Servlet or JSP terminates without invoking the sync() method, the session manager saves the contents of the session object into the session cache (if caching is enabled) but does not update the modified session data in the session replica. The session manager will only update the last access time in the replica asynchronously at a later time. While this option requires using a non-J2EE extension and, as a result, is not portable to other application servers, it does provide the most control over writing updates of the session object. Use of this option is most beneficial for applications that only read or update the session object rarely. If changes in the session object don't occur on every browser interaction, the manual update will likely outperform the use of end-of-service method updates.

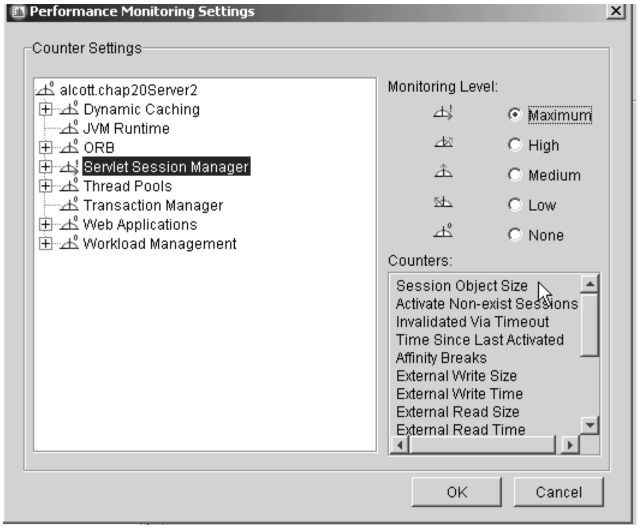
The other setting of interest is the ability to schedule session cleanup at certain times of day instead of when the session times out. While in some cases, such as where sessions are extremely long-lived, this option might be of value, however it's best to remove unneeded sessions as soon as they are no longer needed in order to minimize the impact of managing and tracking the sessions that are no longer needed.

## Session Tuning and Troubleshooting

### Session Object Size

As noted previously, the information contained in HTTP session is stored in the application server JVM, which is a limited resource that is shared by all applications and users. As a result, the more information that is stored in session, the greater the memory footprint for HTTP session, with a proportional decrease in JVM memory available for creation and execution of application objects. In turn, performance can degrade as the decreased heap memory leads to frequent garbage collection (GC). Another factor is the amount of time it takes to serialize and deserialize HTTP session as it is being written to a remote copy—the authors know of cases where the write of the updated session objects from the application server to the database server could not complete due to the size of the session object as the application server was failing, thus negating any failover. Keeping these factors in mind, as well as the primary purpose of HTTP session, which is simply to maintain state between browser invocations, you should strive to keep session objects small so that HTTP session serves as a bookmark and not as a library. How small, you ask? Ideally, the session object should be less than 4KB in size, which coincidentally is also the size limit for a cookie. Of course, it's not always possible to architect an "ideal application," so with this in mind, you should strive to maintain an upper size limit in the 30–60KB in range. In this range, there will be performance degradation, but it will not be as severe as with much larger sessions.

With the size of the session object serving as a primary contributor to application performance, it's fortunate that WAS V5 provides a mechanism for determining session size via the WAS Performance Monitoring Infrastructure (PMI). By configuring the maximum monitoring level in the Tivoli Performance Viewer, as shown in Figure 22-13, you can monitor the size of the session object, as shown in Figure 22-14. The size is shown in bytes. Those of you not familiar with PMI and the Tivoli Performance Viewer will want to see Chapter 25, "Performance Tuning Tools," for a more comprehensive discussion of these two subjects.

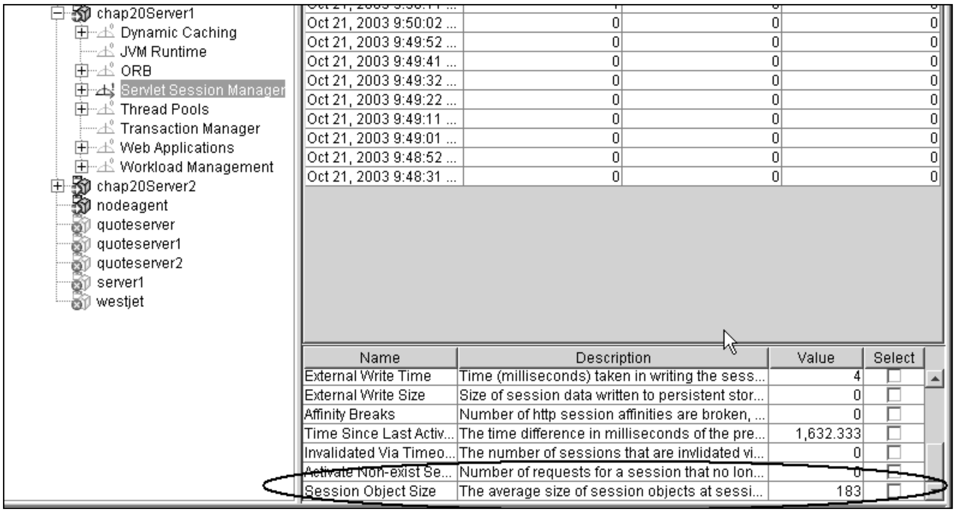


- alcott.chap20Server2
  - Dynamic Caching
  - JVM Runtime
  - ORB
  - Servlet Session Manager
  - Thread Pools
  - Transaction Manager
  - Web Applications
  - Workload Management

Monitoring Level: [✓ Maximum] High Medium Low None

Counters:
Session Object Size
Activate Non-exist Sessions
Invalidated Via Timeout
Time Since Last Activated
Affinity Breaks
External Write Size
External Write Time
External Read Size
External Read Time

**Figure 22-13** Session Manager PMI Monitoring Level Settings.



- chap20Server1
  - Dynamic Caching
  - JVM Runtime
  - ORB
  - Servlet Session Manager
  - Thread Pools
  - Transaction Manager
  - Web Applications
  - Workload Management
- chap20Server2
- nodeagent
- quoteserver
- quoteserver1
- quoteserver2
- server1
- westjet

| | | | |
|---|---|---|---|
| Oct 21, 2003 9:50:02 ... | 0 | 0 | 0 |
| Oct 21, 2003 9:49:52 ... | 0 | 0 | 0 |
| Oct 21, 2003 9:49:41 ... | 0 | 0 | 0 |
| Oct 21, 2003 9:49:32 ... | 0 | 0 | 0 |
| Oct 21, 2003 9:49:22 ... | 0 | 0 | 0 |
| Oct 21, 2003 9:49:11 ... | 0 | 0 | 0 |
| Oct 21, 2003 9:49:01 ... | 0 | 0 | 0 |
| Oct 21, 2003 9:48:52 ... | 0 | 0 | 0 |
| Oct 21, 2003 9:48:31 ... | 0 | 0 | 0 |

| Name | Description | Value | Select |
|---|---|---|---|
| External Write Time | Time (milliseconds) taken in writing the sess... | 4 | ☐ |
| External Write Size | Size of session data written to persistent stor... | 0 | ☐ |
| Affinity Breaks | Number of http session affinities are broken, ... | 0 | ☐ |
| Time Since Last Activ... | The time difference in milliseconds of the pre... | 1,632.333 | ☐ |
| Invalidated Via Timeo... | The number of sessions that are invlidated vi... | 0 | ☐ |
| Activate Non-exist Se... | Number of requests for a session that no lon... | 0 | ☐ |
| Session Object Size | The average size of session objects at sessi... | 183 | ☐ |

**Figure 22-14** Session object size in the Tivoli Performance Viewer.

### Non-Serializable Objects in Session

Another common issue deals with the application placing an object in HTTP session that's not serializable. This typically shows up in an application server SystemErr file, as shown in Code Snippet 22-1.

**Code Snippet 22-1** Session exception example from logs

```
[3/17/02 10:16:44:253 PST] 61602dc6 SessionContex X SESN0058E:
➥BackedHashtableMR: a problem occurred inserting a new session into
➥the database. If a SQLException has occurred then refer to the
➥appropriate database documentation for your environment. Also, assure
➥that you have properly configured a datasource for Session Manager.
java.io.NotSerializableException:
➥com.ibm.servlet.engine.webapp.WebApp...
```

In most cases, you can turn on session trace to determine the non-serializable session object, but in some cases such as Code Snippet 22-2, this still is not definitive (you can see that the "non-serializable stanza" is empty).

**Code Snippet 22-2** Non-serializable stanza from logs

```
[3/17/03 10:16:55:940 PST] 61602dc6 SessionContex >
SessionContext:getIHttpSession - leaving and returning session of

Session Object Internals: id : LE0W4B1E0SPCR2TT4AI1BZA

<omitted for brevity>

non-serializable app specific session data : {}

serializable app specific session data :

{webControler=com.sun.j2ee.blueprints.petstore.control.web.
➥ShoppingClientControllerWebImpl@69246dcc, currentScreen=MAIN,

profilemgr=com.sun.j2ee.blueprints.petstore.control.web.
➥ProfileMgrWebImpl@1e272dcc,

customer=com.sun.j2ee.blueprints.petstore.control.web.CustomerWebImpl@
➥2d572dcc,

cart=com.sun.j2ee.blueprints.petstore.control.web.ShoppingCartWebImpl@
➥33292dcc,

mm=com.sun.j2ee.blueprints.petstore.control.web.ModelManager@20bbedcc,
➥language=en_US}
<omitted for brevity
```

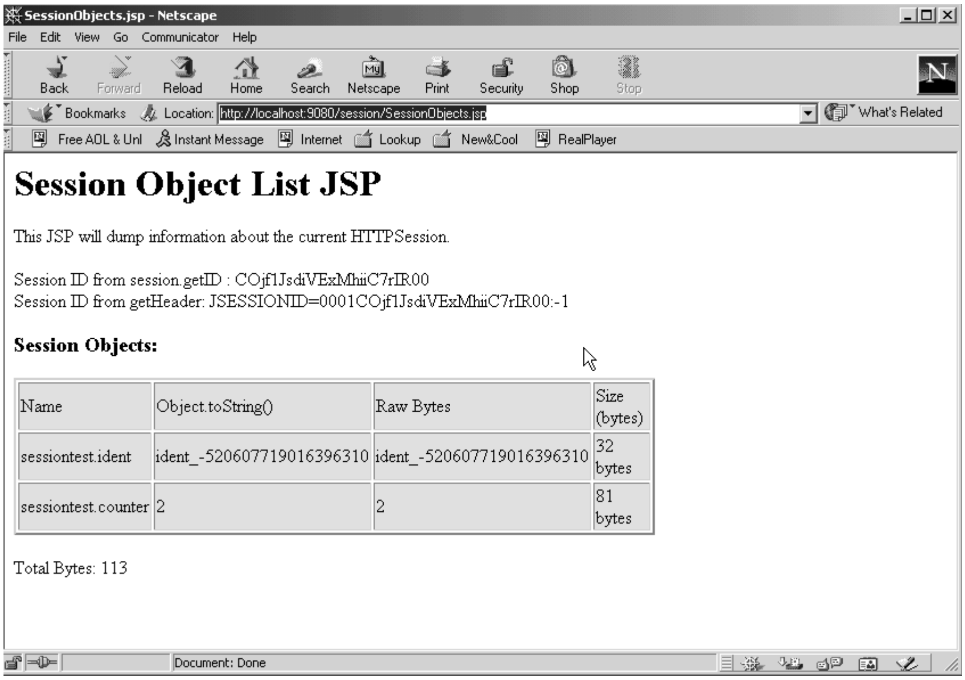
This is because the object implements serializable but isn't actually serializable.

Code Snippet 22-3 contains the source code for a JSP, SessionObjects.jsp, which iterates through all the attributes in the session object, actually tries to write/read all the objects out as a "true" test of serialization, and then displays the results, as shown in Figure 22-15. The JSP displays, left to right:

- The attribute name
- The attribute value
- The results of reading the object after serialization/deserialization

Any attribute that displays "read object null" is in fact not serializable and should not be placed into session if you intend to use distributed sessions.

You simply can drop the JSP into the installedApps directory for the application, <*wasinstallroot/installedApps\installedapp.ear\installed app.war*, along with the rest of the JSPs. Next, create a session object in the application though normal application flow and then invoke the JSP by invoking the appropriate URL: <webapp URL>/SessionObjects.jsp. This will show the offending object(s). You'll also note that this shows the size of each session attribute, so this can be helpful in determining if multi-row sessions would be of value. Please note that the JSP uses a large amount of Java scriptlet and, as a result, is not necessarily representative of "application best practices." Instead, it was developed rather quickly as a debugging aid.


SessionObjects.jsp - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: http://localhost:9080/session/SessionObjects.jsp
What's Related
Free AOL & Unl Instant Message Internet Lookup New&Cool RealPlayer
Document: Done

Session Object List JSP

This JSP will dump information about the current HTTPSession.

Session ID from session.getID : COjf1JsdiVExMhiiC7rIR00
Session ID from getHeader: JSESSIONID=0001COjf1JsdiVExMhiiC7rIR00:-1

**Session Objects:**

| Name | Object.toString() | Raw Bytes | Size (bytes) |
|---|---|---|---|
| sessiontest.ident | ident_-520607719016396310 | ident_-520607719016396310 | 32 bytes |
| sessiontest.counter | 2 | 2 | 81 bytes |

Total Bytes: 113

**Figure 22-15** Session Object JSP.

**Code Snippet 22-3** Session Object List JSP

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
<HEAD>
<%@ page
language="java"
contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"
%>
<META http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<META name="GENERATOR" content="IBM WebSphere Studio">
<TITLE>SessionObjects.jsp</TITLE>
</HEAD>
<BODY>
<H1>Session Object List JSP</H1>
This JSP will dump information about the current HTTPSession.<br><br>
<%@ page import="java.io.*,java.util.*,javax.servlet.*" session="false" %>
<%! public void dumpSession(HttpServletRequest request, JspWriter out)
➥throws IOException {
    HttpSession session = request.getSession(false);
        Object ro = null ;



    out.println("Session ID from session.getID : "
          + session.getId() + "<br>");


    out.println("Session ID from getHeader: "
          + request.getHeader("Cookie") + "<br>");



     Enumeration enum = session.getAttributeNames();
        if ( enum.hasMoreElements() )
        {
            int totalSize = 0;


            out.println("<h3>Session Objects:</h3>");
            out.println("<TABLE Border=\"2\" WIDTH=\"65%\" BGCOLOR=\
            ➥"#DDDDFF\">");
out.println("<tr><td>Name</td><td>Object.toString()</td>");
out.println("<td>Raw Bytes</td><td>Size (bytes)</td></tr>");
            while ( enum.hasMoreElements() )
            {
       String name = (String)enum.nextElement();


       Object sesobj = session.getAttribute(name) ;




       ObjectOutputStream oos = null;
               ByteArrayOutputStream bstream = new
               ➥ByteArrayOutputStream();
       try {


           oos = new ObjectOutputStream(bstream);
           oos.writeObject(sesobj);
           }
       catch (Exception e) {
           e.printStackTrace();
       }
           finally {
               if (oos != null) {
               try {oos.flush();}
               catch (IOException ioe) {}
               try {oos.close();}
               catch (IOException ioe) {}
           }
       }
```

```
        ObjectInputStream ois = null;
        ro = null ;

        try {
            ois = new ObjectInputStream(new
            ➥ByteArrayInputStream(bstream.toByteArray()));
        ro = ois.readObject();
                }
        catch (Exception e) {
        e.printStackTrace();
        }
            finally {
                if (ois != null) {
                try {ois.close();}
                catch (IOException ioe) {}
            }
        }

          totalSize += bstream.size();
                out.println("<tr><td>" + name + "</td><td>" +
                ➥session.getAttribute(name) +
    "</td><td>" + ro + "</td>");
          out.println("<td>" + bstream.size() + " bytes </td></tr>");
            }
            out.println("</table><BR>");
    out.println("Total Bytes: " + totalSize + "<br><br>");
        } else {
        out.println("No objects in session");
}
  }
  %>

<%
```

```
    response.setHeader("Pragma", "No-cache");
    response.setHeader("Cache-Control", "no-cache");
    response.setDateHeader("Expires",0);

    HttpSession session = request.getSession(false);
    if (session == null) {
          out.println("No session");
    }  else {
          dumpSession(request, out);
    }

%>
</BODY>
</HTML>
```

## Conclusion

In this chapter, we have examined the basics of HTTP session management and outlined some options for maintaining application state with HTTP session. We've seen how WAS implements HTTP session and covered the two approaches for persisting session data to a database and memory to memory. We also covered some of the performance implications of the session management configuration options and discussed some troubleshooting approaches.