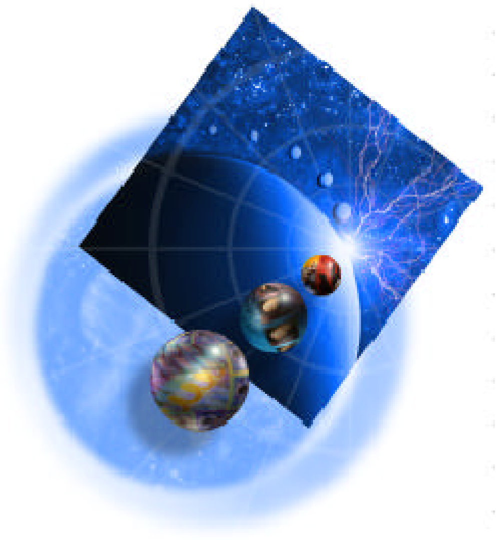
IBM WebSphere

Standard and Advanced Editions

WebSphere Application Server Best Practices using HTTP Sessions



Written by David Draeger and Jay Toogood

Document Revision Control: 1.2
Date: December 14, 2000

Intended Audience

This paper is intended for application architects, developers, management and anyone else responsible for the development and deployment of well performing and scalable WebSphere applications that use session persistence.

Acknowledgments

A special thanks to Gabe Montero for his insight into the topics discussed in this document.

WebSphere Application Server Best Practices using HTTP Sessions − Overview

This white paper contains tips and techniques to help developers program more efficiently and administrators to tune WebSphere appropriately. This document first takes a brief look at the options available in WebSphere for configuring how sessions are handled. In the second section, a discussion of situations on when to use each option is presented to help the system administrator or developer identify when and where an option is valuable to use or not. Finally application development best practices for servlets and JSPs specifically focusing on HTTP Sessions and Session Persistence are presented.

Note: This document only applies towards WebSphere Advanced/Standard Edition 3.02.2 and greater. While some of the items mentioned may work in WebSphere versions prior to 3.02.2, they were not tested while developing this document.

What are sessions?

A session is a series of requests to a servlet, originating from the same user at the same browser. Sessions allow servlets running on a servlet engine to keep track of individual users, a concept known as personalization.

For example, a servlet might use sessions to provide "shopping carts" to on−line shoppers. Suppose the servlet is designed to record the items each shopper indicates he or she will purchase from the Web site. It is important that the servlet be able to associate incoming requests with particular shoppers. Otherwise, the servlet might mistakenly add Shopper_1’s choices to the cart of Shopper_2.

A servlet distinguishes users by their unique session IDs. The session ID arrives with each request. If the user’s browser is cookie−enabled,the session ID can be stored as a cookie. Otherwise, it can conveyed to the servlet by URL rewriting, in which the session ID is appended to the URL of the servlet or JavaServer Pages (JSP) file from which the user is making requests.

WebSphere provides facilities, grouped under the heading Session Manager, that support the javax.servlet.http.HttpSession interface described in the Servlet API specification. Using the

WebSphere Application Server 3.x implementation of the Java Servlet API (Version 2.1) session framework, your server can maintain session state information.

What are persistent sessions (session cluster)?

The Session Manager session support allows multiple application server instances to share a common pool of sessions, known as a session cluster. A cluster is the binding of two or more virtual hosts that reside on separate nodes, such that each virtual host runs servlets across all of the nodes and processes sessions on any one of those nodes.

The implementation of clustering in WebSphere Application Server Version 3.x allows failover. This preserves session data integrity and the common pool of sessions in the event of a system failure in one or more of the clustered Java virtual machines (JVMs) running servlets within a virtual host group.

The session clustering implementation also allows load balancing (Work Load Management), whereby the session workload is distributed among the virtual hosts comprising the cluster.

IBM WebSphere Application Server Version 3.x uses a database to maintain session clusters. In a clustered environment, the session may be accessed on any virtual host in a cluster. Which virtual host is actually accessed will be transparent to the end user. During a session transaction, if the virtual host fails during the WebSphere HttpSession transaction, then the update to the database does not occur. Still, the common pool of sessions continues to function, including the session being processed during the failure, minus any updates made during the uncompleted transaction.

For non−catastrophic failures (such as when the virtual host remains functional), any session changes that cannot be completed are rolled back. The session reverts to its state prior to the start of the transaction. If instead the transaction is completed successfully and the changes are committed, the session is still accessible, regardless of the failure of an individual node.

At what point do you need to use persistent sessions?

Whenever data is shared across multiple application servers (JVMs), persistent sessions are used as a method to share that data. This occurs when cloning application servers (WLM), having two applications, in separate application servers, talk to each other using sessions, or when configuring WebSphere clusters with fail−over support.

What is session caching?

The IBM WebSphere Application Server session support maintains a list of the most recently used sessions in memory. It avoids using the database to read in or access the session when it is determined that the cache entry is still the most recently updated. To determine if the cached session is still valid, WebSphere does a small read of the session’s last access time field from the database and compares it to the last access time of the cached session.

Understanding the Session Manager in WebSphere

Listed below are brief descriptions of the components of the WebSphere Session Manager. Understanding how each option affects WebSphere sessions is valuable both with tuning WebSphere and with developing applications that run on WebSphere. Along with the brief description listed here, more in−depth descriptions can be found online in section **6.6.11.1.4** of the WebSphere InfoCenter. *(See the Online Resources section at the end of this document.)*

Session Manager − Enable Tab			
Label	***Options***	***Default***	***Description***
Enable Sessions	Yes/No	Yes	Specifies whether session tracking is enabled, meaning the session−related methods for the request and response objects will be functional.
Enable Cookies	Yes/No	Yes	Specifies whether session tracking uses cookies to carry sessions IDs.
Enable URL Rewriting	Yes/No	No	Specifies whether the Session Manager uses rewritten URLs to carry the session IDs. If it is enabled, the Session Manager recognizes session IDs that arrive in the URL and, if necessary, rewrites the URL to send the session IDs.
Enable Protocol Switch Rewriting	Yes/No	No	Specifies whether the session ID is added to URLs when the URL requires a switch from HTTP to HTTPS or HTTPS to HTTP.
Enable Persistent Sessions	Yes/No	No	Specifies whether to save session data in a database, or lose the session data when the server shuts down.

Session Manager − Cookies Tab			
Label	***Options***	***Default***	***Description***
Cookie Name	Valid String*	sesessionid	Specifies a unique name for the cookie.
Cookie Comment	Valid String*	Servlet session support	Specifies information about the cookie.
Cookie Domain	Valid String*		Specifies the value of the domain field of a session cookie. This value will restrict where the cookie is sent. For example, if you specify a particular domain, session cookies will be sent only to hosts in that domain.
Cookie Maximum Age	Positive Integer or −1	−1	Specifies the maximum number of milliseconds the cookie will live on the client browser. Corresponds to the Time to Live (TTL) value described in the Cookie specification. When value is −1, cookie timeouts when browser session is closed.
Cookie Path	Any valid path on the server	/	Specifies the value of the path field that will be sent for session cookies. Specify a value to restrict which paths on the server the cookie will be sent to. By restricting paths, you can keep the cookie from being sent to certain servlets, JHTML, and HTML files.
Cookie Secure	Yes/No	No	Specifies whether session cookies include the secure field. Specify Yes to restrict the exchange of cookies to only HTTPS sessions.
* See the cookie specification at http://www.netscape.com/newsref/std/cookie_spec.html			

Session Manager − Persistence Tab			
Label	***Options***	***Default***	***Description***
Persistence Type	Directodb / EJB	Directodb	Specifies the mechanism to use for recording persistent session data. **Note:** EJB is not fully implemented in WebSphere 3.0 − 3.5
Datasource	Defined Datasource		Specifies the data source object from which the Session Manager will obtain database connections.
Userid	Valid Userid for given datasource		Specifies the user ID for accessing the session database and tables.
Password	Valid Password for given Userid		Specifies the password for accessing the session database and tables.
Note: Changes on the Persistence tabbed page take effect the next time the application server containing the Session Manager is started.			

Session Manager − Intervals Tab			
Label	***Options***	***Default***	***Description***
Invalidate Time (seconds)	Positive Integer or −1	1800	Specifies the number of seconds a session is allowed to go unused before it will no longer be considered valid. When value is −1, no time−out.
Note: Changes on the Interval tabbed page take effect immediately.			

Session Manager − Tuning Tab			
Label	***Options***	***Default***	***Description***
Using Multirow Sessions	Yes/No	No	Specifies whether to place each instance of application data in a separate row in the database, allowing larger amounts of data to be stored per session. This can yield better performance in certain usage scenarios.
Using Cache	Yes/No	No	Specifies whether to maintain an in−memory list of the most recently used sessions. In some cases, the list (cache) can help avoid database accesses.
Using Manual Update	Yes/No	No	Specifies whether to automatically send session updates to the database. By default, Application Server updates the database with the last access time and any changes effected by the servlet, such as updating or removing application data.
Using Native Access	Yes/No	No	N/A for V3.02.2 and V3.5
Allow Overflow	Yes/No	Yes	Specifies whether to allow the number of sessions in memory to exceed the value specified by Base In−memory Size property.
Base Memory Size	0..Memory Max	1000	Specifies the number of sessions to maintain in memory. The meaning differs somewhat, depending on whether you are using in−memory or persistent sessions.
Note: Changes on the Tuning tabbed page take effect the next time the application server containing the Session Manager is started.			

Understanding When to Use the Session Options in WebSphere

When to use single row sessions or multi row sessions

The decision on using single row session persistence or multirow session persistence is largely dependent on how much data a developer wants to store for each session. The following graphics represent how the data is stored in the database for persistent sessions.

Column Ids of a WebSphere Session Record:

ID	PROPID	APPNAME	LISTENERCNT	LAST ACCESS	CREATION TIME	MAX INACTIVETIME	USERNAME	SMALL	MEDIUM	LARGE

Example of a Single Row Session Record:

WW3QP2 YAAAAA GCIFO5A CTBA	WW3QP2 YAAAAA GCIFO5A CTBA	MyApp	0	969416563671	969416561625	1800	anonymous			java.lang.Integer,"My String","My Large String"

Example of a Multi Row Session Record:

WWTVE 1YAAAA ACCIFO5 AH53Q	WWTVE1YA AAAACCIFO 5AH53Q	MyApp	0	969415276250	969415273031	1800	anonymous			
WWTV...	IntVariable							java.lang.Integer		
WWTV...	StrVariable							"My String"		
WWTV...	LargeStrVar									"My Large String"

The following items help an administrator or developer decide whether to use single row or multi row sessions in WebSphere.

Reasons to use single row:

1. Can read or write all values with just one record read/write.
2. Takes up less space in a database since you are guaranteed that each session is only one record long.

Reasons not to use single row:

1. 2 Megabyte limit of stored data per session.

Reasons to use multi row:

1. The application can store an unlimited amount of data. (Limited only by size of database and 2 MB per record limit)
2. The application can read in individual fields instead of the whole record. When larger amounts of data are stored in the session but only small amounts are specifically accessed during a given servlet's processing of a http request, multi row sessions can improve performance by avoiding unneeded Java object serialization.

Reasons not to use multi row:

1. If data is small in size, you don't want the extra overhead of multiple row reads when everything can be stored in one row.

MultiRow Gotcha: In the case of multirow usage, design your application data objects not to have references to each other, and to prevent circular references. For example, suppose you are storing two objects A and B in the session using HttpSession.put(..) , and A contains a reference to B. In the multirow case, because objects are stored in different rows of the database, when objects A and B are retrieved later, the object graph between A and B is different than stored. A and B behave as independent objects.

When to use the cache option

The decision of using the cache option is dependent on the administrator and the WebSphere configuration.

Note: It is highly recommended to use this feature to improve WebSphere performance.

Reasons to use the cache:

1. Session affinity is being used.

Reasons not to use the cache:

1. The server has limited memory available for a cache.

When to use the Manual Update option

The decision on the Manual Update option is entirely dependent on the developer as they need to use the com.ibm.websphere.servlet.session.IBMSession object instead of the HttpSession object. This is so that the sync() function can be used to commit the information to the database.

Reasons to use Manual Update:

1. The servlets of an application typically read in the session data but don’t write it back as much.
2. The developer wants direct control over when the session information is persisted to the database.
3. The servlets of an application take a long time to finish processing, thereby holding locks on the database records for a long time.

(**Note:** The latest versions of WebSphere no longer perform select for updates when request.getSession() is called. The session manager will now release any database lock before returning back to the servlet engine and the servlet. Not using the database for locking now places a hard requirement on affinity and not allowing multiple web applications to access a session concurrently.)

Reasons not to use Manual Update:

1. The developer doesn’t want to explicitly control when to persist data to the database using the com.ibm.websphere.servlet.session.IBMSession object and wants to let WebSphere control persisting to the database.
2. The servlets of an application are updating (writing) session information frequently.
3. The developer needs to be completely compliant with the servlet 2.1 specifications. When the application uses the sync() function that is part of the IBMSession object, this code may not be portable to other systems since session persistence is not part of the servlet 2.1 specifications, but is an IBM extension to the specifications.

When to Allow Overflow

The decision on when to Allow Overflow is dependent on the administrator and the resources of the WebSphere server.

Reasons to Allow Overflow:

1. You do not want to be limited by the amount of memory specified in the Base−memory field.

Reasons not to Allow Overflow:

1. Malicious users can turn off cookies and use a script to make new session requests causing the server to use up all its resources to fill those requests.
2. You want to limit the amount of physical memory used as a cache.

What value to use for Base Memory Size

The decision on what value to make the Base Memory size depends on the resources of the WebSphere server and whether the cache is on or not. Also, the value means different things depending on how other session options are configured.

For in−memory sessions, this value specifies the number of sessions in the base session table. Use the Allow Overflow property to specify whether to limit sessions to this number for the entire Session Manager, or allow additional sessions to be stored in secondary tables.

For persistent sessions, this value specifies the size of the general cache. If the Cache property is enabled, the Base In−memory Size specifies how many session updates will be cached before the Session Manager reverts to reading session updates from the database automatically.

This value holds when you are using in−memory sessions, persistent sessions with caching, or persistent sessions with manual updates. (The manual update cache keeps the last n time stamps representing "last access" times, with n being the Base Memory Size value).

The default is 1000. How you customize this setting will depend on your hardware system, the usage characteristics of your site, and your willingness to increase the stack sizes of the Java processes for your Application Servers to accommodate a larger value.

Additional Notes about WebSphere

The latest versions of WebSphere (WS 3.5 PTF 3 and greater, WS 3.02.2 plus e−fix PQ42166) no longer perform select for updates when request.getSession() is called. The session manager will now release any database lock before returning back to the servlet engine and the servlet. Not using the database for locking now places a hard requirement on affinity and not allowing multiple web applications to access a session concurrently. The responsibility of not accessing sessions concurrently is placed on the developer of the applications. This complies with the servlet 2.2 API.

The session invalidation time−out is not accurate to the second.
The session invalidation based on the time−out is not accurate to the second. There is a low priority thread that runs to clear out the sessions that have timed out. Based on the time−out value, the invalidation thread is triggered during a block of time which may not be exactly at the second that is in the time−out field. For a configured interval of 30 minutes (default value), the time−out is accurate to within 5 minutes. This means the invalidation process will run every 5

minutes and if it finds a session that has not been accessed for 30 minutes or more, it will invalidate it. So if a session was created right after an invalidation check was made and was inactive since it was created, it would be 35 minutes before it is invalidated.
For a time−out of 0−15 minutes, there is a 1 minute fluctuation. For a time−out of 15−60 minutes, there is a fluctuation of 5 minutes. For a time−out of 60 minutes − 6 hours, there is a fluctuation of 30 minutes. For a time−out of greater than 6 hours, there is a fluctuation of 2 hours.

Session In−memory Cache: When this cache becomes full, the least−recently used session will be discarded to make room for the new session. This occurs when persistent sessions are disabled. When persistent sessions are enabled, sessions persisted to a database will always be served from the database (not from cache) once the Base Memory Size is exceeded.

Best Practices for Using HTTP Sessions

1. **With the latest e−fixes that remove the notion of locking the session for the scope of the servlet service() method, you now have to be aware of your servlets accessing a session concurrently.** For example, if two servlets are changing the same session property at the same time, or if one servlet invalidates the session while another one is still trying to use it, data inconsistencies may occur. If these situations are not handled in your application, you may want to consider synchronizing on the session object following the request.getSession() call, along with checking the validity of the session as the first step following that java synchronization. This, along with the points on http session affinity mentioned below, should ensure single access if that is desirable for you.

2. **When developing new objects which will be stored in the HTTP session, make sure to implement the Serializable class.** This allows the object to properly serialize to the database when using persistent sessions. Without this extension, the object will not persist correctly and will throw an error. An example of this is:
   public class MyObject **implements** java.io.Serializable

3. **When adding Java objects to a session, make sure they are in the correct classpath.** If Java objects will be added to a session, be sure to place the class files for those objects in the application server classpath or in the directory containing other servlets used in WebSphere Application Server. In the case of session clustering, this applies to every node in the cluster.

   Because the HttpSession object is shared among servlets that the user might access, consider adopting a site−wide naming convention to avoid conflicts.

   Additionally if objects are only in the web application classpath and there is more than one web application sharing sessions, the following restrictions apply:

   1. You cannot use single row session persistence due to the fact that the applications which do not have the objects in the classpath will not be able to read in the session data.
   2. You cannot have two web applications reading in the same session concurrently (i.e. via a multi−framed JSP)

4. **Don't store large Object graphs in HttpSession.** In most applications each servlet only requires a fraction of the total session data. However, by storing the data in the HttpSession as one large object, an application forces WebSphere to process all of it each time.
   (See IBM WebSphere Whitepaper − WebSphere Application Development Best Practices for Performance and Scalability written by Harvey W. Gunther for more details.)

5. **Release HttpSessions When Finished**

HttpSession objects live inside the WebSphere servlet engine until:

1. The application explicitly and programmatically releases it using the API,

javax.servlet.http.HttpSession.invalidate(); quite often, programmatic invalidation is part of an application logout function.

2. WebSphere destroys the allocated HttpSession when it expires (default = 1800 seconds or 30 minutes). WebSphere can only maintain a certain number of HttpSessions in memory. When this limit is reached, WebSphere simply does not cache any new sessions and session updates are automatically sent back to the database, without checking for their presence in the cache.
   (See IBM WebSphere Whitepaper − WebSphere Application Development Best Practices for Performance and Scalability written by Harvey W. Gunther for more details.)

6. **Do not try to save and reuse the HttpSession object outside of each servlet/JSP.** The HttpSession object is a function of the HttpRequest (you can only get if via req.getSession) and a copy of it is only valid for the life of the service() method of the servlet or JSP. You **cannot** cache the HttpSession object and reference it outside the scope of a servlet or JSP.

7. **Utilize Session Affinity to help achieve higher cache hits in WebSphere.** In addition to information listed in the InfoCenter, WebSphere 3.02.2 and WebSphere 3.5 have functionality in the HTTP Server Plugin to help with session affinity. The plugin will read the cookie data (or encoded URL) from the browser and help direct the request to the appropriate application or clone based on the assigned session key. This helps to achieve a greater use of the in−memory cache and reduces hits to the session database.

8. Maximize use of session affinity and prevent breaking affinity. Using session affinity properly can enhance the performance of WebSphere. Session affinity in WebSphere is a way to maximize the use of the in−memory cache of session objects and reduce the amount of reads to the backend database. Session affinity works by caching the session objects in the JVM of the application a user is interacting with. If there are multiple clones of this application, the user can be directed to any one of the clones which are also in their own JVM. If the users starts on clone1 and then a comes in on clone2 a little later, all the session information must be persisted to the backend database and then read in by the JVM that clone2 is running in. This database read can be avoided by using session affinity. With session affinity, the user would start on clone1 for the first request and then for every successive request, the user would be directed back to clone1. By doing this, clone1 only has to look at the cache to get the session information and never has to make a call to the session database to get the information.

   You can improve performance by not breaking session affinity. Some suggests to help prevent breaking session affinity are:

   1. Do not use multi−framed JSPs where the frames point to different web applications. This will break affinity and will cause separate JVMs to process a session concurrently. When

this happens, consistent state cannot be guaranteed.

2. If possible, combine all web applications into a single JVM (Application Server) and use modeling / cloning to provide fail−over support.
3. See #9 − "When using multi−frame Java Server Pages (JSP) create the session using the frame page (JSP) but don’t create it in the pages (JSPs) within the frame" below.

9. **When applying security to servlets/JSPs that use sessions, make sure you secure all the pages not just some of the pages.** When it comes to security and sessions, it’s all or nothing. It does not make sense to protect access to session state only part of the time. When WebSphere security is enabled, every resource from which sessions are created or accessed must be secured or or it must be unsecured. You cannot mix secured and unsecured resources. The problem with securing only a couple of pages is that sessions created in secured pages are created under the identity of the authenticated user. They can only be accessed in other secured pages by the same user. To protect these sessions from use by unauthorized users, they cannot be accessed from an insecure page. When a request from an unsecure page occurs access is denied and an UnauthorizedSessionRequest Exception is throw. (UnauthorizedSessionRequest Exception is a runtime exception and does not need to be explicitly caught)
10. **Use Manual Update and sync() in applications that mostly read session data but update infrequently.** When an application is using sessions, anytime data is read from or written to that session, the LastAccess time field is updated. If persistent sessions are being used, this produces a new write to the database. This is a performance hit that can be avoided by using Manual Update and having the record written back to the database only when data values are updated, not on every read/write of the record.

    To use manual update, you first need to turn it on in the session manager. (See the tables above to show where it is located.) Additionally, the application code must use the com.ibm.websphere.servlet.session.IBMSession instead of the generic HttpSession. Within the IBMSession there is a method called sync(). This method tells WebSphere that the data in the session object should be written out to the database. This allows the developer to improve overall performance by having the session information persist only when necessary.

    The following is an actual example of another use of Manual Update:
    In a customer’s application, JSPs only read the page bean from the session. They have many JSPs and for every page displayed to the user, they have four JSPs that might execute (multi−frame − not recommended but possible if taking the correct precautions mentioned in this document). Each one of those JSPs, when manual update mode is turned off, will cause a write to the session database to update the LastAccess timestamp field. So, one of the big reasons they switched Manual Update mode on was to avoid these very frequent (albeit short duration) hits to the Session database. Additionally their Servlets write out frequently to set

everything up for the JSPs before they execute. Every screen presented to the user will write once regardless of how many JSPs or servlets are executed to display the page. Turning manual update mode on had effectively removed about 4 accesses to the session database.

That was a good example of how you can have applications that update the session database frequently yet still want to have manual update mode on to avoid the multi−JSP initiated updates to the LastAccess timestamp.

11.**When using multi−frame Java Server Pages (JSP) create the session using the frame page (JSP) but don't create it in the pages (JSPs) within the frame**. By default JSPs create HTTPSessions using the request.getSession(true) method. By doing this, each page in the browser is requesting a new session, but only one session is used per browser instance. A developer can use <%@ page session="false"%> to turn off the automatic session creation. Then if the page needs to access session information use <% HttpSession session = javax.servlet.http.HttpServletRequest.getSession(false); %> to get the already existing session which was created by the frame JSP. This allows you to not break session affinity on the initial loading of the frame pages.

12.Utilize the following suggestions to achieve high performance.

1. Turn on Caching.

2. Use Network Dispatcher taking advantage of the session affinity capabilities. Also use WebSphere 3.02.2 or higher to utilize the web server plugin session affinity capabilities.

3. If your applications do not change the session data frequently, use Manual Update and the sync() function to efficiently persist session information.

4. Keep the amount of data stored in the session as small as possible. With the ease of using sessions to hold data, sometimes too much data is stored in the session objects. A proper balance of data storage and performance must be determined to effectively use sessions.

5. Use a dedicated database for the session database. Don't use the WebSphere repository database or another application's database. This helps to avoid contention for JDBC connections and allows for better database performance.

6. Verify that you have the latest e−fixes for WebSphere.
   http://www−4.ibm.com/software/webservers/appserv/efix.html

13.Utilize the following tools to help monitor session performance.

1. Run the com.ibm.servlet.personalization.sessiontracking.IBMTrackerDebug servlet.
   − To be able to run this servlet, you must have the servlet invoker running in the web application you want to run this from.
   − OR−
   −You can explicitly configure this servlet in the application you want to run it from.
2. Use the WebSphere Resource Analyzer.

http://www-4.ibm.com/software/webservers/appserv/doc/v35/ae/infocenter/was/atsepm02.html

3. Use Database tracking tools such as "Monitoring" in DB2.
   (See the respective documentation for the database system used.)
4. Turn on the object level tracing in WebSphere for the session manager. This can be done by right clicking on the Application Server and choosing Trace. Then select **Components−>com−>ibm−>servlet−>personalization−>sessiontracking**. Right click on the box in front of sessiontracking and choose the level of tracing you wish to export. Click the "Set" button. Run the application you are monitoring the session on. Return to the Trace Administration screen and enter a filename of where to export the trace dump. Click on the "Dump" button.
   (For information on how to read the dump file, see section **8.4** of the InfoCenter documentation.)

Online Resources:

InfoCenter for WebSphere V 3.5
http://www−4.ibm.com/software/webservers/appserv/doc/v35/ae/infocenter/index.html

The following are chapters of the InfoCenter the pertain specifically to HTTP Sessions or this documentation.
0.11.2 − What is session clustering (i.e. persistent sessions)
1.2.3.8 − Miscellaneous database tips
4.4.1.1 − Session programming model and environment
4.4.1.1.1 − Deciding between session tracking approaches
4.4.1.1.1.1 − Using cookies to track sessions
4.4.1.1.1.2 − Using URL rewriting to track sessions
4.4.1.1.2 − Locking and unlocking session transactions
4.4.1.1.3 − Securing Sessions
4.4.1.1.4 − Switching from single to multirow schema
4.4.1.1.5 − Using sessions in a clustered environment
4.4.1.1.6 − Session Limitations
4.4.1.1.7 − Tuning Session Support
4.4.1.1.7.2 − Tuning Session Support − Session Affinity
4.4.1.1.7.3 − Tuning Session Support − Multirow Schema
4.4.1.1.7.4 − Tuning Session Support − Manual Update
4.4.1.1.7.5 − Tuning Session Support − Base in−memory session pool size
6.6.11.1 − Administering Session Support
6.6.11.1.4 − Session Manager properties
6.6.0.1.6− Java command line arguments reference
8.4 − Viewing Traces

Session Management Tips
http://www−4.ibm.com/software/webservers/appserv/doc/v35/ae/infocenter/was/08010301_v35.html#5.4