Chapter 12 of *WebSphere Application Server V7 Administration and Configuration Guide*, SG24-7615

WebSphere Application Server V7: Session Management

Session support allows a Web application developer to maintain state information across multiple user visits to the application. In this chapter, we discuss HTTP session support in WebSphere Application Server V7 and how to configure it. We also discuss the support for stateful session bean failover.

We cover the following topics:

HTTP session management

In many Web applications, users collect data dynamically as they move through the site based on a series of selections on pages they visit. Where the user goes next, and what the application displays as the user's next page, or next choice, depends on what the user has chosen previously from the site. For example, if the user clicks the checkout button on a site, the next page must contain the user's shopping selections.

In order for this to happen, a Web application needs a mechanism to hold the user's state information over a period of time. However, HTTP does not recognize or maintain a user's state. HTTP treats each user request as a discrete, independent interaction.

The Java™ servlet specification provides a mechanism for servlet applications to maintain a user's state information. This mechanism, known as a *session*, addresses some of the problems of more traditional strategies, such as a pure cookie solution. It allows a Web application developer to maintain all user state information at the host, while passing minimal information back to the user through cookies, or another technique known as *URL rewriting*.

(New in V7) In WebSphere Application Server V7 session tracking using SSL ID is deprecated, you should use cookies or URL rewriting.

Session manager configuration

Session management in WebSphere® Application Server can be defined at the following levels:

- Application server:

  This is the default level. Configuration at this level is applied to all Web modules within the server.

- Application:

  Configuration at this level is applied to all Web modules within the application.

- Web module:

  Configuration at this level is applied only to that Web module.

Session management properties

With one exception, the session management properties that you can set are the same at each configuration level:

- *Overwrite session management,* for enterprise application and Web module level only, determines whether these session management settings are used for the current module, or if the settings are used from the parent object.
- *Session tracking mechanism* lets you select from cookies, URL rewriting, and SSL ID tracking. Selecting cookies will lead you to a second configuration page containing further configuration options.
- *Maximum in-memory session count* lets you specify the maximum number of sessions to keep in memory and whether to allow this number to be exceeded, or overflow.
- *Session timeout* specifies the amount of time to allow a session to remain idle before invalidation.
- *Security integration* specifies that the user ID be associated with the HTTP session.
- *Serialize session access* determines if concurrent session access in a given server is allowed.
- *Distributed environment settings* determines how to persist sessions (memory-to-memory replication or a database) and set tuning properties. Memory-to-memory persistence is only available in a Network Deployment distributed server environment.

Accessing session management properties

You can access all session management configuration settings using the administrative console.

Application server session management properties

To access session management properties at the application server level, from the administrative console, do the following steps:

1. Select **Servers** → **Server Types** → **WebSphere application servers**.
2. Click the application server.
3. In the Container Settings section of the Configuration tab, click **Session management**.

Application session management properties

To access session management properties at the application level, from the administrative console, do the following steps:

1. Click **Applications** → **Application Types** → **WebSphere enterprise applications.**
2. Click the application name to open its configuration page.
3. In the Web Module Properties section of the Configuration tab, click **Session management**.

Web module session management properties

To access session management properties at the Web module level, from administrative console, do the following steps:

1. Click **Applications** → **Application Types** → **WebSphere enterprise applications.**
2. Click the application.
3. In the Modules section of the Configuration tab, click **Manage Modules**.
4. Click the Web module.
5. In the Additional Properties section, click **Session Management**.

Session identifiers

WebSphere session support keeps the user's session information about the server. WebSphere passes the user an identifier known as a session ID, which correlates an incoming user request with a session object maintained on the server.

Note: The example session IDs provided in this chapter are for illustrative purposes only and are *not* guaranteed to be absolutely consistent in value, format, and length.

Choosing a session tracking mechanism

WebSphere supports three approaches to tracking sessions:

- SSL session identifiers (deprecated)
- Cookies
- URL rewriting

It is possible to select all three options for a Web application. If you do this:

- SSL session identifiers are used in preference to cookie and URL rewriting.
- Cookies are used in preference to URL rewriting.

Note: If SSL session ID tracking is selected, we recommend that you also select cookies or URL rewriting so that session affinity can be maintained. The cookie or rewritten URL contains session affinity information enabling the Web server to properly route a session back to the same server for each request.

To set or change the session mechanism type, do the following steps:

1. Open the session management properties for the application server, enterprise application, or Web module.
2. Select the session tracking mechanism (Figure 1).

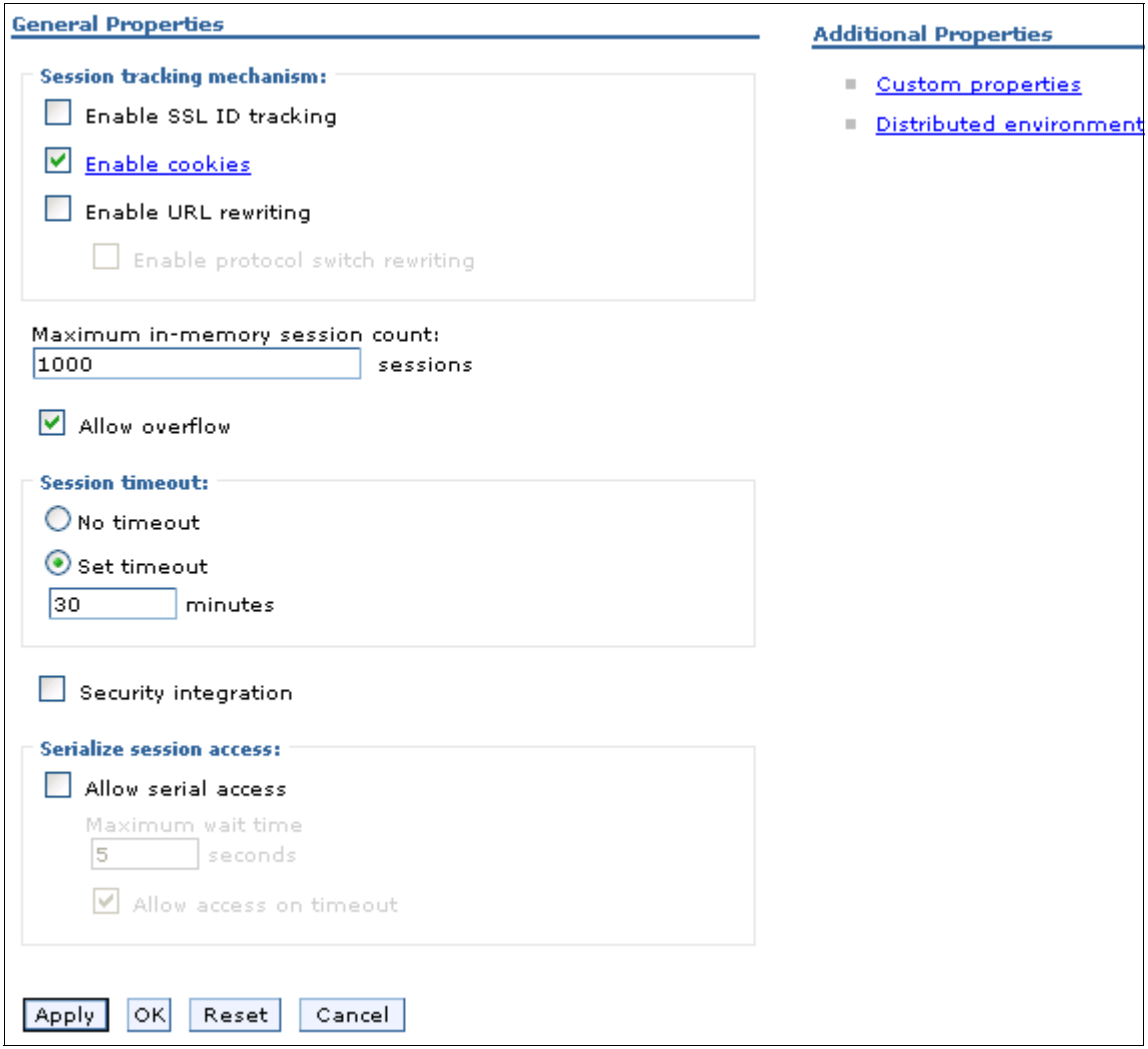

Figure 1 Selecting a session tracking mechanism window

3. Click **OK**.
4. Save and synchronize the configuration changes.
5. Restart the application server or the cluster.

Cookies

Many sites choose cookie support to pass the user's identifier between WebSphere and the user. WebSphere Application Server session support generates a unique session ID for each user, and returns this ID to the user's browser with a cookie. The default name for the session management cookie is JSESSIONID. See Figure 2.

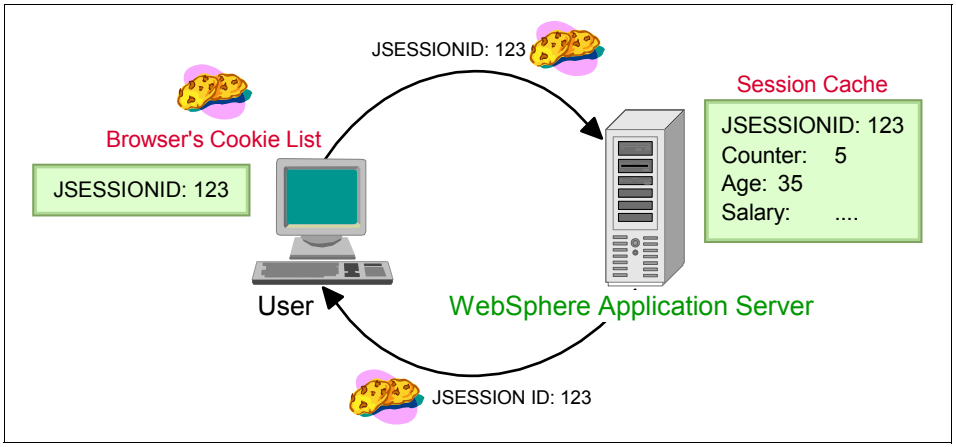

Figure 2 Cookie overview

A cookie consists of information embedded as part of the headers in the HTML stream passed between the server and the browser. The browser holds the cookie and returns it to the server whenever the user makes a subsequent request. By default, WebSphere defines its cookies so they are destroyed if the browser is closed. This cookie holds a session identifier. The remainder of the user's session information resides at the server.

The Web application developer uses the HTTP request object's standard interface to obtain the session:

```
HttpSession session = request.getSession(true);
```

WebSphere places the user's session identifier in the outbound cookie whenever the servlet completes its execution, and the HTML response stream returns to the end user. Again, neither the cookie or the session ID within it require any direct manipulation by the Web application. The Web application only sees the contents of the session.

Be aware that some users, either by choice or mandate, disable cookies from within their browser.

Cookie settings

To configure session management using cookies, do the following steps from the administrative console:

1. Open the Session Manager window at your preferred level.
2. Click the box for **Enable Cookies** as the session tracking mechanism. See Figure 1 on page 6.
3. If you would like to view or change the cookies settings, select the **Enable Cookies** hot link. The following cookie settings are available:
   - Cookie name:

     The cookie name for session management should be unique. The default cookie name is JSESSIONID. However, this value can be configured for flexibility.
   - Restrict cookies to HTTPS sessions:

     Enabling this feature restricts the exchange of cookies only to HTTPS sessions. If it is enabled, the session cookie's body includes the secure indicator field.
   - Cookie domain:

     This value dictates to the browser whether or not to send a cookie to particular servers. For example, if you specify a particular domain, the browser will only send back session cookies to hosts in that domain. The default value in the session manager restricts cookies to the host that sent them.

     Note: The LTPA token/cookie that is sent back to the browser is scoped by a single DNS domain specified when security is configured. This means that *all* application servers in an *entire* WebSphere Application Server domain must share the same DNS domain for security purposes.

   - Cookie path:

     The paths on the server to which the browser will send the session tracking cookie. Specify any string representing a path on the server. Use the slash (/) to indicate the root directory.

     Specifying a value restricts the paths to which the cookie will be sent. By restricting paths, you can keep the cookie from being sent to certain URLs on the server. If you specify the root directory, the cookie will be sent no matter which path on the given server is accessed.

– Cookie maximum age:

  The amount of time that the cookie will live in the client browser. There are two choices:

  - Expire at the end of the current browser session
  - Expire at a configurable maximum age

  If you choose the maximum age option, specify the age in seconds.

4. Click **OK** to exit the page and change your settings.
5. Click **OK** to exit the session management settings.
6. Save and synchronize your configuration changes.
7. Restart the application server or the cluster.

URL rewriting

WebSphere also supports URL rewriting for session ID tracking. While session management using SSL IDs or cookies is transparent to the Web application, URL rewriting requires the developer to use special encoding APIs, and to set up the site page flow to avoid losing the encoded information.

URL rewriting works by storing the session identifier in the page returned to the user. WebSphere encodes the session identifier as a parameter on URLs that have been encoded programmatically by the Web application developer. This is an example of a Web page link with URL encoding:

```
<a href="/store/catalog;$jsessionid=DA32242SSGE2">
```

When the user clicks this link to move to the /store/catalog page, the session identifier is passed in the request as a parameter.

URL rewriting requires explicit action by the Web application developer. If the servlet returns HTML directly to the requester, without using a JavaServer™ Page, the servlet calls the API, as shown in Example 1, to encode the returning content.

Example 1 URL encoding from a servlet

```
out.println("<a href=\");
out.println(response.encodeURL ("/store/catalog"));
out.println("\>catalog</a>");
```

Even pages using redirection, a common practice, particularly with servlet or JSP™ combinations, must encode the session ID as part of the redirect, as shown in Example 2.

Example 2 URL encoding with redirection

```
response.sendRedirect(response.encodeRedirectURL("http://myhost/store/c
atalog"));
```

When JavaServer Pages (JSPs) use URL rewriting, the JSP calls a similar interface to encode the session ID:

```
<% response.encodeURL ("/store/catalog"); %>
```

URL rewriting configuration

When you select URL rewriting, an additional configuration option, Enable protocol switch rewriting, is available. This option defines whether the session ID, added to a URL as part of URL encoding, should be included in the new URL if a switch from HTTP to HTTPS or from HTTPS to HTTP is required. For example, if a servlet is accessed over HTTP and that servlet is doing encoding of HTTPS URLs, URL encoding will be performed only when protocol switch rewriting is enabled, and vice versa.

Considerations for using URL rewriting

The fact that the servlet or JSP developer has to write extra code is a major drawback over the other available session tracking mechanisms.

URL rewriting limits the flow of site pages exclusively to dynamically generated pages, such as pages generated by servlets or JSPs. WebSphere inserts the session ID into dynamic pages, but cannot insert the user's session ID into static pages, .htm, or .html.

Therefore, after the application creates the user's session data, the user must visit dynamically generated pages exclusively until they finish 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 their session ID.

Local sessions

Many Web applications use the simplest form of session management: the in-memory, local session cache. The local session cache keeps session information in memory and local to the machine and WebSphere Application Server where the session information was first created.

Local session management does not share user session information with other clustered machines. Users only obtain their session information if they return to the application server. Most importantly, local session management lacks a persistent store for the sessions it manages. A server failure takes down not only the WebSphere instances running on the server, but also destroys any sessions managed by those instances.

WebSphere allows the administrator to define a limit on the number of sessions held in the in-memory cache from the administrative console settings on the session manager. This prevents the sessions from acquiring too much memory in the Java VM associated with the application server.

The session manager also allows the administrator to permit an unlimited number of sessions in memory. If the administrator enables the **Allow overflow** setting on the session manager, the session manager permits two in-memory caches for session objects. The first cache contains only enough entries to accommodate the session limit defined to the session manager, 1000 by default. The second cache, known as the overflow cache, holds any sessions the first cache cannot accommodate, and is limited in size only by available memory. The session manager builds the first cache for optimized retrieval, while a regular, un-optimized hash table contains the overflow cache.

For best performance, define a primary cache of sufficient size to hold the normal working set of sessions for a given Web application server.

Important: If you enable overflow, the session manager permits an unlimited number of sessions in memory. Without limits, the session caches might consume all available memory in the WebSphere instance's heap, leaving no room to execute Web applications. For example, here are two scenarios under which this could occur:

- The site receives greater traffic than anticipated, generating a large number of sessions held in memory.
- A malicious attack occurs against the site where a user deliberately manipulates their browser so the application creates a new session repeatedly for the same user.

If you choose to enable session overflow, the state of the session cache should be monitored closely.

Note: Each Web application will have its own base, or primary, in-memory session cache, and with overflow allowed, its own overflow, or secondary, in-memory session cache.

General properties for session management

The session management settings allow the administrator to tune a number of parameters that are important for both local or persistent sessions (see Figure 1 on page 6). Next we describe the settings:

- Maximum in-memory session count:

  This field specifies the maximum number of sessions to maintain in memory. The meaning differs depending on whether you are using local or persistent sessions. For local sessions, this value specifies the number of sessions in the base session table. Select **Allow overflow** to specify whether to limit sessions to this number for the entire session manager, or to allow additional sessions to be stored in secondary tables. Before setting this value, see “Local sessions” on page 10.

  For persistent sessions, this value specifies the size of the general cache. This value determines how many sessions will be cached before the session manager reverts to reading a session from the database automatically. Session manager uses a least recently used (LRU) algorithm to maintain the sessions in the cache.

  This value holds when you use local sessions, persistent sessions with caching, or persistent sessions with manual updates. The manual update cache keeps the last n time stamps representing the last access times, where n is the maximum in-memory session count value.

- Allow overflow:

  Choosing this option specifies whether to allow the number of sessions in memory to exceed the value specified in the maximum in-memory session count field. If **Allow overflow** is not checked, then WebSphere limits the number of sessions held in memory to this value.

  For local sessions, if this maximum is exceeded and Allow overflow is not checked, then sessions created thereafter will be dummy sessions and will not be stored in the session manager. Before setting this value, see “Local sessions” on page 10.

  As shown in Example 3, the IBM® HttpSession extension can be used to react if sessions exceed the maximum number of sessions specified when overflow is disabled.

Example 3 Using IBMSession to react to session overflow

```
com.ibm.websphere.servlet.session.IBMSession sess =
   (com.ibm.websphere.servlet.session.IBMSession) req.getSession(true);
if(sess.isOverFlow()) {
   //Direct to a error page…
```

}

Note: Allowing an unlimited amount of sessions can potentially exhaust system memory and even allow for system sabotage. Someone could write a malicious program that continually hits your site, creating sessions, but ignoring any cookies or encoded URLs and never utilizing the same session from one HTTP request to the next.

- Session timeout:

  If you select **Set timeout**, when a session is not accessed for this many minutes it can be removed from the in-memory cache and, if persistent sessions are used, from the persistent store. This is important for performance tuning. It directly influences the amount of memory consumed by the JVM™ in order to cache the session information.

  Note: For performance reasons, the session manager invalidation process runs at regular intervals to invalidate any invalid sessions. This interval is determined internally based on the Session timeout interval specified in the Session manager properties. For the default timeout value of 30 minutes, the invalidation process interval is around 300 seconds. In this case, it could take up to 5 minutes (300 seconds) beyond the timeout threshold of 30 minutes for a particular session to become invalidated.

  The value of this setting is used as a default when the session timeout is not specified in a Web module's deployment descriptor.

  If you select **No timeout**, a session will be never removed from the memory unless explicit invalidation has been performed by the servlet. This can cause a memory leak when the user closes the window without logging out from the system. This option might be useful when sessions should be kept for a while until explicit invalidation has been done, when an employee leaves the company, for example. To use this option, make sure that enough memory or space in a persistent store is kept to accommodate all sessions.

- Security integration:

  When security integration is enabled, the session manager associates the identity of users with their HTTP sessions. See "Session security" on page 51 for more information.

Note: Do not enable this property if the application server contains a Web application that has form-based login configured as the authentication method and the local operating system is the authentication mechanism. It will cause authorization failures when users try to use the Web application.

- Serialize session access:

  This option is available to provide serialized access to the session in a given JVM. This ensures thread-safe access when the session is accessed by multiple threads. No special code is necessary for using this option. This option is not recommended when framesets are used heavily because it can affect performance.

  An optional property, Maximum wait time, can be set to specify the maximum amount of time a servlet request waits on an HTTP session before continuing execution. The default is 5 seconds.

  If you set the Allow access on timeout option, multiple servlet requests that have timed out concurrently will execute normally. If it is false, servlet execution aborts.

Session affinity

In a clustered environment, any HTTP requests associated with an HTTP session must be routed to the same Web application in the same JVM. This ensures that all of the HTTP requests are processed with a consistent view of the user's HTTP session. The exception to this rule is when the cluster member fails or has to be shut down.

WebSphere assures that session affinity is maintained in the following way: Each server ID is appended to the session ID. When an HTTP session is created, its ID is passed back to the browser as part of a cookie or URL encoding. When the browser makes further requests, the cookie or URL encoding will be sent back to the Web server. The Web server plug-in examines the HTTP session ID in the cookie or URL encoding, extracts the unique ID of the cluster member handling the session, and forwards the request.

This situation can be seen in Figure 3, where the session ID from the HTTP header, request.getHeader("Cookie"), is displayed along with the session ID from session.getId(). The application server ID is appended to the session ID from the HTTP header. The first four characters of HTTP header session ID are the cache identifier that determines the validity of cache entries.

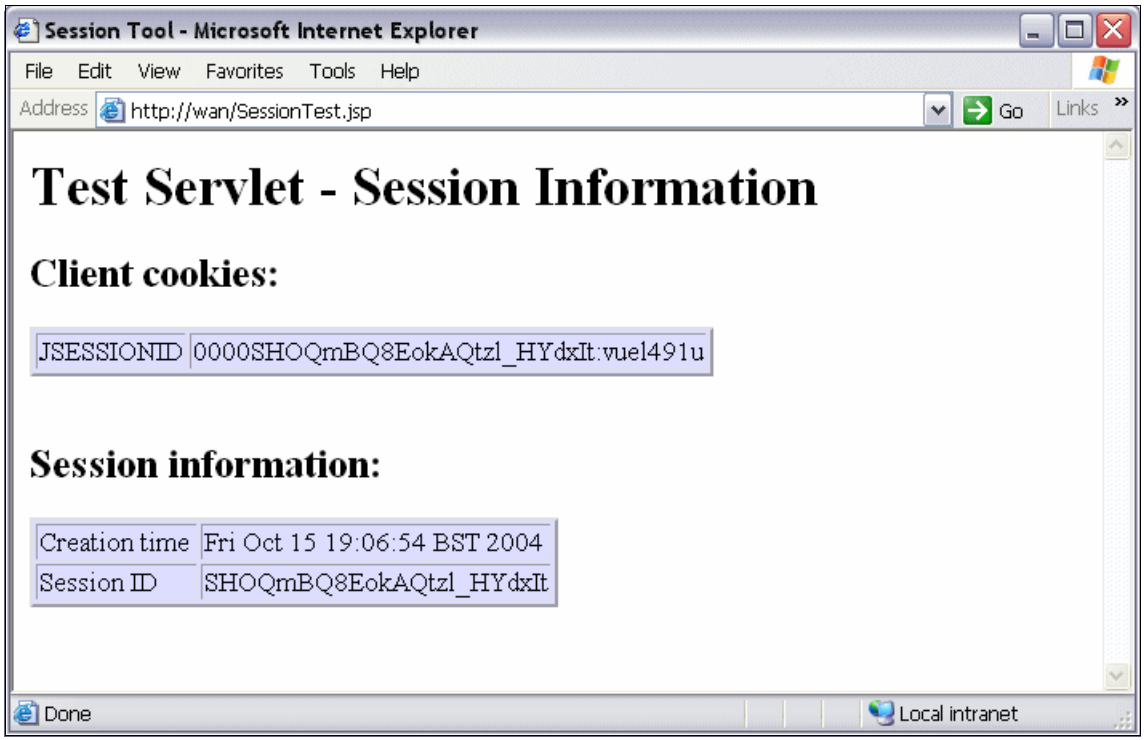

Figure 3 Session ID containing the server ID and cache ID

The JSESSIONID cookie can be divided into these parts: cache ID, session ID, separator, clone ID, and partition ID. JSESSION ID will include a partition ID instead of a clone ID when memory-to-memory replication in peer-to-peer mode is selected. Typically, the partition ID is a long numeric number.

Table 1 shows their mappings based on the example in Figure 3. A clone ID is an ID of a cluster member.

Table 1 Cookie mapping

content	value in the example
Cache ID	0000
Session ID	SHOQmBQ8EokAQtzl_HYdxIt
separator	:
Clone ID	vuel491u

The application server ID can be seen in the Web server plug-in configuration file, plug-in-cfg.xml file, as shown in Example 4.

Example 4 Server ID from plugin-cfg.xml file

```
<?xml version="1.0" encoding="ISO-8859-1"?><!--HTTP server plugin
config file for the cell ITSOCell generated on 2004.10.15 at 07:21:03
PM BST-->
<Config>
......
   <ServerCluster Name="MyCluster">
      <Server CloneID="vuel491u" LoadBalanceWeight="2"
Name="NodeA_server1">
        <Transport Hostname="wan" Port="9080" Protocol="http"/>
        <Transport Hostname="wan" Port="9443" Protocol="https">
    ......
</Config>
```

Note: Session affinity can still be broken if the cluster member handling the request fails. To avoid losing session data, use persistent session management. In persistent sessions mode, cache ID and server ID will change in the cookie when there is a failover or when the session is read from the persistent store, so do not rely on the value of the session cookie remaining the same for a given session.

Session affinity and failover

Server clusters provide a solution for failure of an application server. Sessions created by cluster members in the server cluster share a common persistent session store. Therefore, any cluster member in the server cluster has the ability to see any user's session saved to persistent storage.

If one of the cluster members fails, the user can continue to use session information from another cluster member in the server cluster. This is known as failover. Failover works regardless of whether the nodes reside on the same machine or several machines. Only a single cluster member can control and access a given session at a time. See Figure 4.

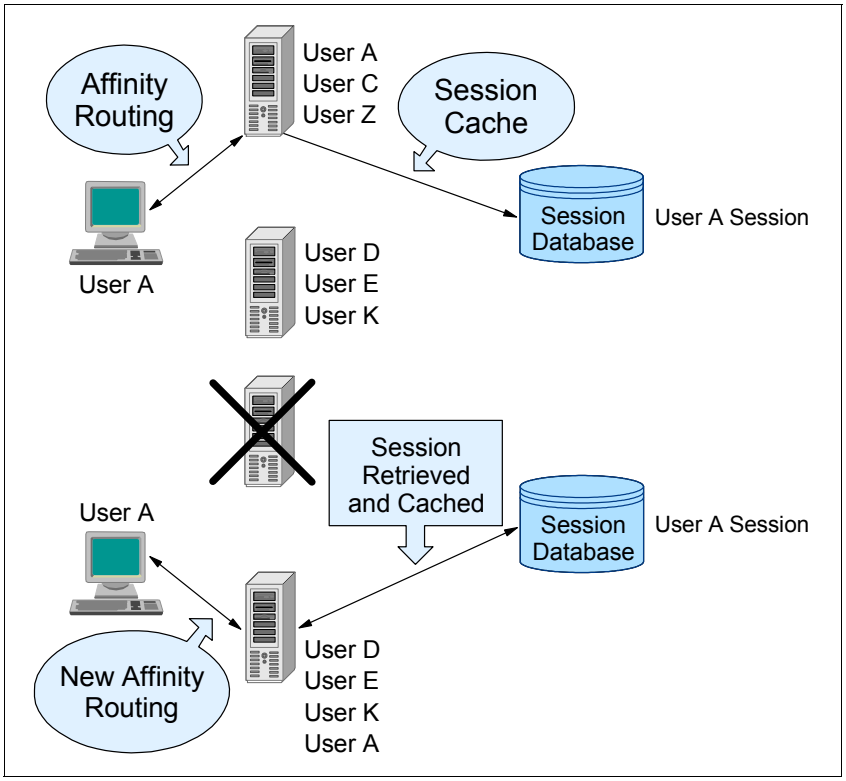

Figure 4 Session affinity and failover

After a failure, WebSphere redirects the user to another cluster member, and the user's session affinity switches to this replacement cluster member. After the initial read from the persistent store, the replacement cluster member places the user's session object in the in-memory cache, assuming that the cache has space available for additional entries.

The Web server plug-in maintains a cluster member list and picks the cluster member next in the list to avoid the breaking of session affinity. From then on, requests for that session go to the selected cluster member. The requests for the session go back to the failed cluster member when the failed cluster member restarts.

WebSphere provides session affinity on a best-effort basis. There are narrow windows where session affinity fails. These windows are as follows:

- When a cluster member is recovering from a crash, a window exists where concurrent requests for the same session could end up in different cluster members. The reason for this is that the Web server is multi-processed and each process separately maintains its own retry timer value and list of available cluster members. The end result is that requests being processed by different processes might end up being sent to more than one cluster member after at least one process has determined that the failed cluster member is running again.

  To avoid or limit exposure in this scenario, if your cluster members are expected to crash very seldom and are expected to recover fairly quickly, consider setting the retry timeout to a small value. This narrows the window during which multiple requests being handled by different processes get routed to multiple cluster members.

- A server overload can cause requests belonging to the same session to go to different cluster members. This can occur even if all the cluster members are running. For each cluster member, there is a backlog queue where an entry is made for each request sent by the Web server plug-in waiting to be picked up by a worker thread in the servlet engine. If the depth of this queue is exceeded, the Web server plug-in starts receiving responses that the cluster member is not available. This failure is handled in the same way by the Web server plug-in as an actual JVM crash. Here are some examples of when this can happen:
  - The servlet engine does not have an appropriate number of threads to handle the user load.
  - The servlet engine threads take a long time to process the requests. Reasons for this include: applications taking a long time to execute, resources being used by applications taking a long time, and so on.

Persistent session management

By default, WebSphere places session objects in memory. However, the administrator has the option of enabling persistent session management, which instructs WebSphere to place session objects in a persistent store. Administrators should enable persistent session management when:

- The user's session data must be recovered by another cluster member after a cluster member in a cluster fails or is shut down.
- The user's session data is too valuable to lose through unexpected failure at the WebSphere node.

- The administrator desires better control of the session cache memory footprint. By sending cache overflow to a persistent session store, the administrator controls the number of sessions allowed in memory at any given time.

There are two ways to configure session persistence as shown in Figure 5:

- Database persistence, supported for the Web container only
- Memory-to-memory session state replication using the data replication service available in distributed server environments

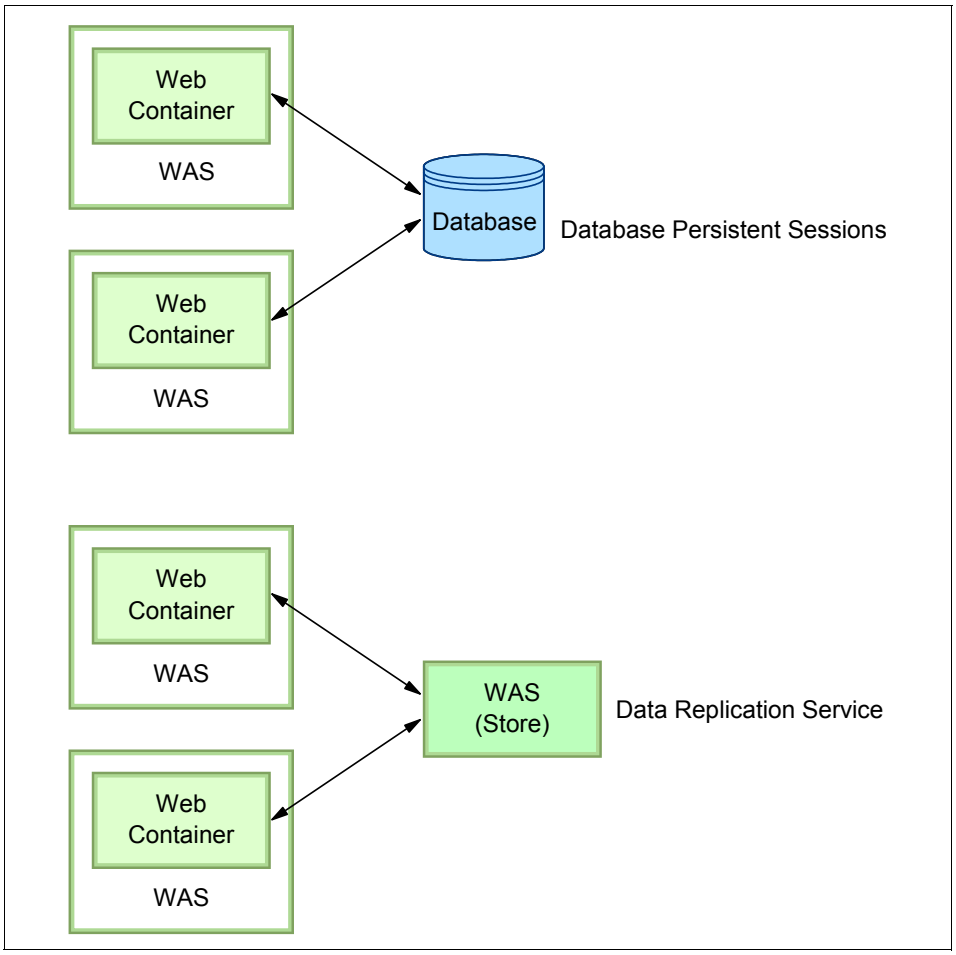

Figure 5 Persistent session options

All information stored in a persistent session store must be serialized. As a result, all of the objects held by a session must implement java.io.Serializable if the session needs to be stored in a persistent session store.

In general, consider making all objects held by a session serialized, even if immediate plans do not call for the use of persistent session management. If the Web site grows, and persistent session management becomes necessary, the transition between local and persistent management occurs transparently to the application if the sessions only hold serialized objects. If not, a switch to persistent session management requires coding changes to make the session contents serialized.

Persistent session management does not impact the session API, and Web applications require no API changes to support persistent session management. However, as mentioned previously, applications storing unserializable objects in their sessions require modification before switching to persistent session management.

If you use database persistence, using multi-row sessions becomes important if the size of the session object exceeds the size for a row, as permitted by the WebSphere session manager. If the administrator requests multi-row session support, the WebSphere session manager breaks the session data across multiple rows as needed. This allows WebSphere to support large session objects. Also, this provides a more efficient mechanism for storing and retrieving session contents under certain circumstances. See “Single and multi-row schemas (database persistence)” on page 44 for information about this feature.

Using a cache lets the session manager maintain a cache of most recently used sessions in memory. Retrieving a user session from the cache eliminates a more expensive retrieval from the persistent store. The session manager uses a *least recently used* scheme for removing objects from the cache. Session data is stored to the persistent store based on your selections for write frequency and write option.

Enabling database persistence

We assume in this section that the following tasks have already been completed before enabling database persistence:

1. Create a session database.
2. (z/OS® DB2®) Create a table for the session data. Name the table SESSIONS. If you choose to use another name, update the Web container custom property SessionTableName value to the new table name. Grant ALL authority for the server region user ID to the table. An example of creating the table can be found in the Information Center at:

   http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.nd.multiplatform.doc/info/ae/ae/tprs_db2tzos.html
3. In distributed environments, the session table will be created automatically for you when you define the data source for the database as the session management table; however, if you want to use a page (row) size greater than 4 KB, you will need to create the tablespace manually. An example of creating the tablespace can be found in the Information Center at:

   http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.nd.multiplatform.doc/info/ae/ae/tprs_db2t.html
4. Create a JDBC™ provider and data source for the database.

   The data source should be non-XA enabled. and must be a non-JTA enabled data source.

   The JNDI name will be used to specify the database for persistence (in this example, jdbc/Sessions).

   A summary of the data source selections for a DB2 database on z/OS is shown at the end of the wizard (Figure 6).

Options	Values
Scope	cells:WTCell
Data source name	SessionDb
JNDI name	jdbc/Sessions
JDBC provider name	DB2 Universal JDBC Driver Provider
Description	One-phase commit DB2 JCC provider that supports JT Data sources that use this provider support only 1-ph commit processing, unless you use driver type 2 with application server for z/OS. If you use the application for z/OS, driver type 2 uses RRS and supports 2-pha processing.
Class path	${DB2UNIVERSAL_JDBC_DRIVER_PATH}/db2jcc.jar ${UNIVERSAL_JDBC_DRIVER_PATH}/db2jcc_license_cu ${DB2UNIVERSAL_JDBC_DRIVER_PATH}/db2jcc_licens
${DB2UNIVERSAL_JDBC_DRIVER_PATH}	/pp/db2v8/UK39204/jcc/classes
${UNIVERSAL_JDBC_DRIVER_PATH}	
Native path	${DB2UNIVERSAL_JDBC_DRIVER_NATIVEPATH}
${DB2UNIVERSAL_JDBC_DRIVER_NATIVEPATH}	/pp/db2v8/UK39204/jcc/lib
Implementation class name	com.ibm.db2.jcc.DB2ConnectionPoolDataSource
Driver type	2
Database name	DB8X
Server name	wtsc04.itso.ibm.com
Port number	33760
Use this data source in container managed persistence (CMP)	true
Component-managed authentication alias	WTDmNode/jdbc/db8x
Mapping-configuration alias	(none)
Container-managed authentication alias	WTDmNode/jdbc/db8x

Figure 6 Data source creation summary

Note: The following example illustrates the steps to enable database persistence at the application server level. Session management settings can also be performed at the enterprise application level and the Web application level.

To enable database persistence, repeat the following steps for each application server:

1. Select **Servers** → **Server Types** → **WebSphere application servers**.
2. Select the server.
3. Click **Session management** under the Container Settings section.
4. Click **Distributed environment settings**.
5. Select **Database** and click the **Database** link. See Figure 7.

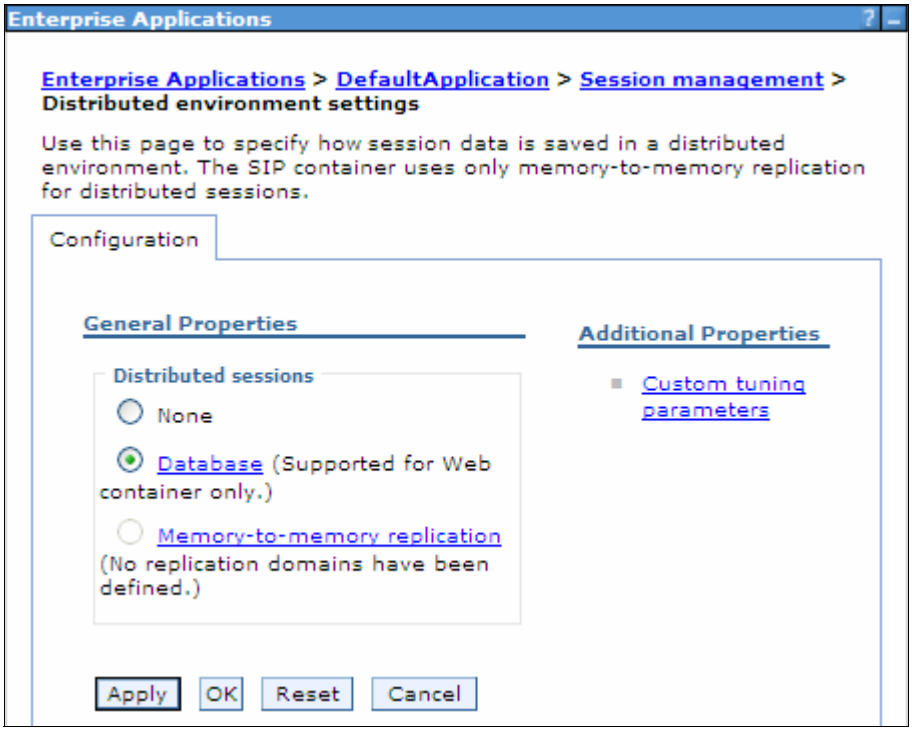

Figure 7 Distributed Environment Setting (database)

6. Enter the database information:
   - **(z/OS)** Enter the data source JNDI name (Figure 8).

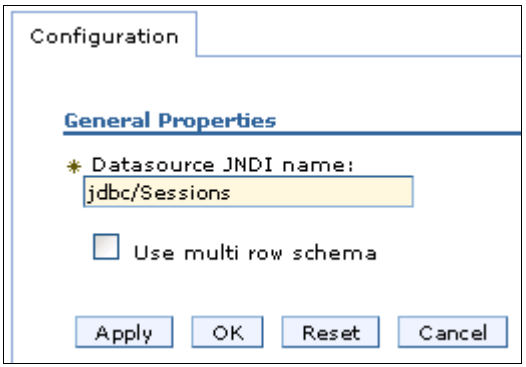

Figure 8 Data Source name

Click **OK**.

   - **In distributed platforms**:

     Enter the information required to access the database (Figure 9).

     If you are using DB2 and you anticipate requiring row sizes greater than 4 KB, select the appropriate value from the DB2 row size menu. If the DB2 row size is other than 4 KB, you are required to enter the name of tablespace. See “Larger DB2 page sizes and database persistence” on page 43.

     If you intend to use a multi-row schema, select **Use Multi row schema**. See “Single and multi-row schemas (database persistence)” on page 44 for more information.

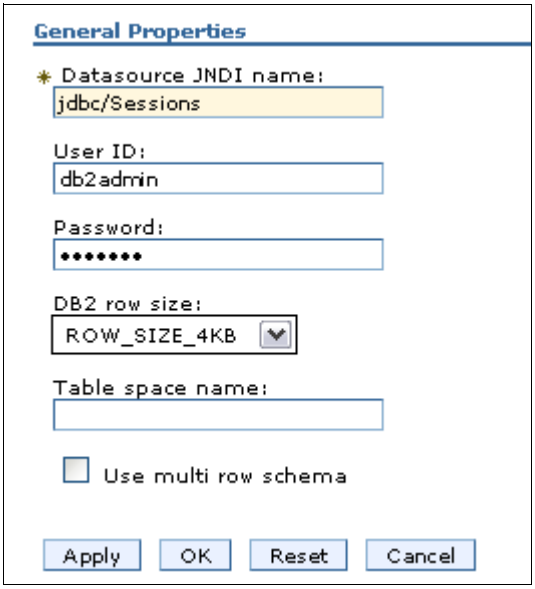

Figure 9 Database settings for session persistence

Click **OK**.

7. Click **OK** to accept the changes to the distribute session configuration.
8. Repeat this for each server in the cluster and save the configuration changes, synchronize them with the servers, and restart the application servers.

Memory-to-memory replication

Memory-to-memory replication uses the data replication service to replicate data across many application servers in a cluster without using a database. Using this method, sessions are stored in the memory of an application server, providing the same functionality as a database for session persistence. Separate threads handle this functionality within an existing application server process.

The data replication service is an internal WebSphere Application Server component. In addition to its use by the session manager, it is also used to replicate dynamic cache data and stateful session beans across many application servers in a cluster.

Using memory-to-memory replication eliminates the overhead and cost of setting up and maintaining a real-time production database. It also eliminates the single point of failure that can occur with a database. Session information between application servers is encrypted.

Note: Memory-to-memory replication requires the HA manager to be active. HA manager is active by default, but can be disabled. For more information, see *When to use a high availability manager* at:

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm.websphere.nd.multiplatform.doc/info/ae/ae/crun_ha_ham_required.html

Data replication service modes

The memory-to-memory replication function is accomplished by the creation of a data replication service instance in an application server that communicates to other data replication service instances in remote application servers.

You can set up a replication service instance to run in three possible modes:

- Server mode:

  In this mode, a server only stores backup copies of other application server sessions. It does not send copies of sessions created in that particular server.

- Client mode:

  In this mode, a server only broadcasts or sends copies of the sessions it owns. It does not receive backup copies of sessions from other servers.

- Both mode:

  In this mode, the server simultaneously sends copies of the sessions it owns, and acts as a backup table for sessions owned by other application servers.

You can select the replication mode of server, client, or both when configuring the session management facility for memory-to-memory replication. The default is both modes.

With respect to mode, these are the primary examples of memory-to-memory replication configuration:

- Peer-to-peer replication
- Client/server replication

Although the administrative console allows flexibility and additional possibilities for memory-to-memory replication configuration, only these configurations are officially supported.

There is a single replica in a cluster by default. You can modify the number of replicas through the replication domain.

Peer-to-peer topology

Figure 10 shows an example of peer-to-peer topology. Each application server stores sessions in its own memory. It also stores sessions to and retrieves sessions from other application servers. Each application server acts as a client by retrieving sessions from other application servers. Each application server acts as a server by providing sessions to other application servers.

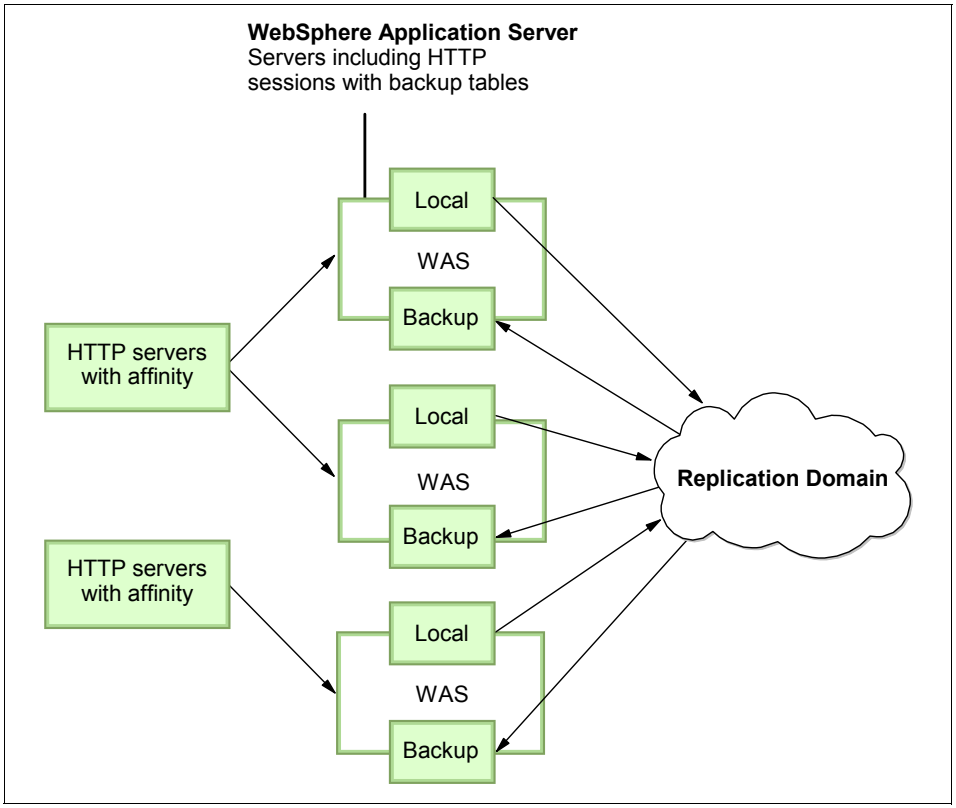

Figure 10 Example of peer-to-peer topology

The basic peer-to-peer (both mode) topology is the default configuration and has a single replica. However, you can also add additional replicas by configuring the replication domain.

In this basic peer-to-peer topology, each application server can:

- Host the Web application leveraging the HTTP session
- Send changes to the HTTP session that it owns
- Receive backup copies of the HTTP session from all of the other servers in the cluster

This configuration represents the most consolidated topology, where the various system parts are collocated and requires the fewest server processes. When using this configuration, the most stable implementation is achieved when each node has equal capabilities (CPU, memory, and so on), and each handles the same amount of work.

The advantage of this topology is that no additional processes and products are required to avoid a single point of failure. This reduces the time and cost required to configure and maintain additional processes or products.

One of the disadvantages of this topology is that it can consume large amounts of memory in networks with many users, because each server has a copy of all sessions. For example, assuming that a single session consumes 10 KB and one million users have logged into the system, each application server consumes 10 GB of memory in order to keep all sessions in its own memory. Another disadvantage is that every change to a session must be replicated to all application servers. This can cause a performance impact.

Client/server topology

Figure 11 shows an example of client/server topology. In this setup, application servers act as either a replication client or a server. Those that act as replication servers store sessions in their own memory and provide session information to clients. They are dedicated replication servers that just store sessions but do not respond to the users' requests. Client application servers send session information to the replication servers and retrieve sessions from the servers. They respond to user requests and store only the sessions of the users with whom they interact.

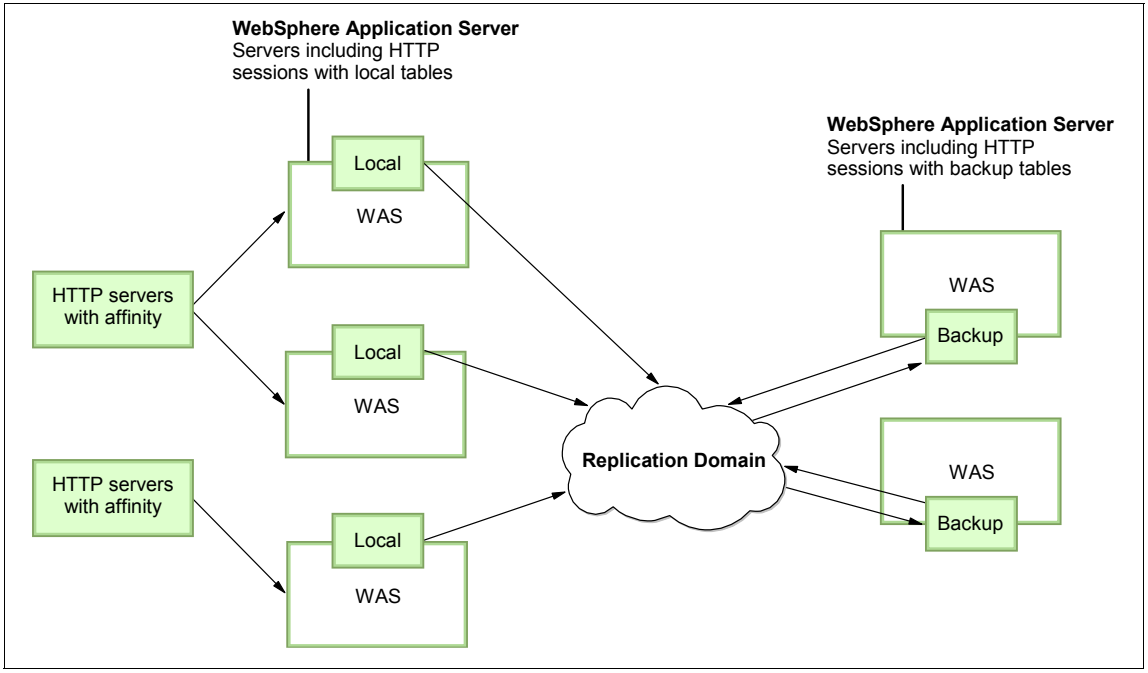

Figure 11 Example of client/server topology

The advantage of this topology is that it clearly distinguishes the role of client and server. Only replication servers keep all sessions in their memory and only the clients interact with users. This reduces the consumption of memory on each application server and reduces the performance impact, because session information is only sent to the servers.

You can recycle a backup server without affecting the servers running the application, When there are two or more backups, failure recovery is possible. Conversely, you can recycle an application server without losing the backup data.

When running Web applications on lower-end hardware, you can choose to have one or two more powerful computers that have the capacity to run a couple of session managers in replication server mode, allowing you to reduce the load on the Web application hardware.

One of the disadvantages of this topology is that additional application servers have to be configured and maintained over and above those that interact with users. We recommend that you have multiple replication servers configured to avoid a single point of failure.

Replication domain

The memory-to-memory replication function is accomplished by the creation of a data replication service instance in an application server that communicates to other data replication service instances in remote application servers. You must configure this data replication service instance as a part of a replication domain.

Data replication service instances on disparate application servers that replicate to one another must be configured as a part of the same domain. You must configure all session managers connected to a replication domain to have the same topology. If one session manager instance in a domain is configured to use the client/server topology, then the rest of the session manager instances in that domain must be a combination of servers configured as Client only and Server only.

If one session manager instance is configured to use the peer-to-peer topology, then all session manager instances must be configured as both client and server. For example, a server-only data replication service instance and a both client and server data replication service instance cannot exist in the same replication domain. Multiple data replication service instances that exist on the same application server due to session manager memory-to-memory configuration at various levels that are configured to be part of the same domain must have the same mode.

You should create a separate replication domain for each consumer. For example, create one replication domain for session manager and another replication domain for dynamic cache. The only situation where you should configure one replication domain is when you configure session manager replication and stateful session bean failover. Using one replication domain in this case ensures that the backup state information of HTTP sessions and stateful session beans are on the same application servers.

New in V7: A replication domain created with WebSphere Application Server V6.1 is referred to as a *multi-broker domain*. This type of replication domain consists of replicator entries. This is deprecated in WebSphere Application Server V7 and supported only for backward compatibility. Multi-broker replication domains do not communicate with each other, so migrate any multi-broker replication domains to the new data replication domains. You cannot create a multi-broker domain or replicator entries in the administrative console of WebSphere Application Server V7.

Enabling memory-to-memory replication

We assume in this section that the following tasks have already been completed before enabling data for the replication service:

1. You have created a cluster consisting of at least two application servers.

   In this example, we are working with a cluster called MyCluster. It has two servers, server1 and server2.

2. You have installed applications to the cluster.

Note: This example illustrates setting up the replication domain and replicators after the cluster has been created. You also have the option of creating the replication domain and the replicator in the first server in the cluster when you create the cluster.

To enable memory-to-memory replication, do the following steps:

1. Create a replication domain to define the set of replicator processes that communicate with each other.

   a. Select **Environment** → **Replication domains.** Click **New**. See Figure 12, and enter information in the fields.

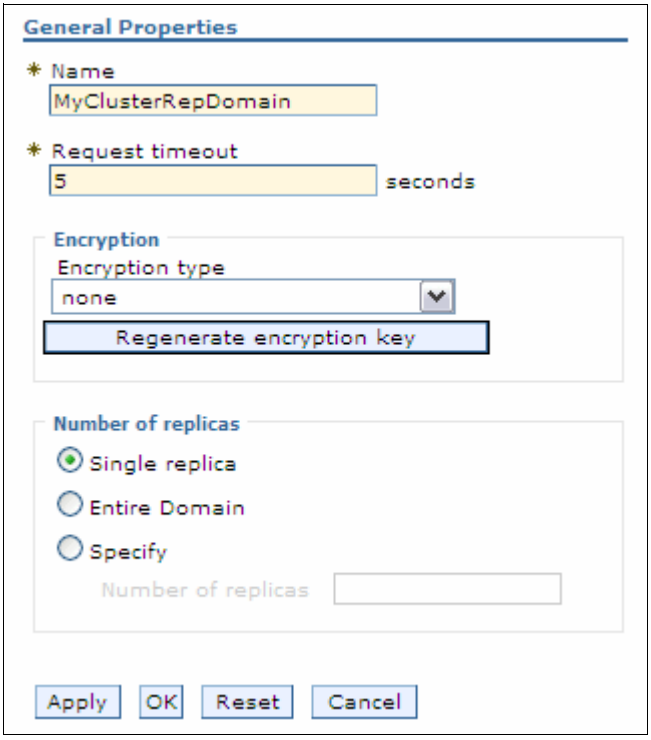

Figure 12 Create a replication domain

– Name:

  At a minimum, you need to enter a name for the replication domain. The name must be unique within the cell. In this example, we used MyClusterRepDomain as the name, and defaults are used for the other properties.

– Encryption:

  Encrypted transmission achieves better security but can impact performance. If DES or TRIPLE_DES is specified, a key for data transmission is generated. We recommend that you generate a key by clicking the **Regenerate encryption key** button periodically to enhance security.

– Number of replicas:

A single replica allows you to replicate a session to only one other server. This is the default. When you choose this option, a session manager picks another session manager connected to the same replication domain with which to replicate the HTTP session during session creation. All updates to the session are replicated to that single server. This option is set at the replication domain level.

When this option is set, every session manager connected to this replication domain creates a single backup copy of the HTTP session state information on a backup server.

Alternatively, you can replicate to every application server that is configured as a consumer of the replication domain with the **Entire Domain** option or to a specified number of replicas within the domain.

b. Click **Apply**.

c. Click **OK**.

d. Save the configuration changes.

2. Configure the cluster members.

Repeat the following steps for each application server:

a. Select **Servers** → **Server Types** → **WebSphere application servers**.

b. Click the application server name. In this example, **server1** and **server2** are selected as application servers respectively.

c. Click **Session management** in the Container settings section.

d. Click **Distributed environment settings**.

e. Click on **Memory-to-memory replication** (Figure 13).

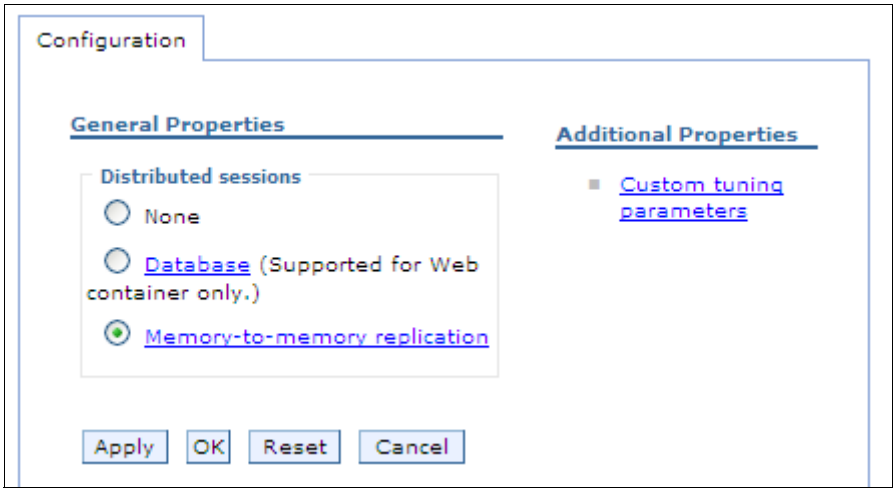

Figure 13 Distributed environment settings

f. Choose a replicator domain. See Figure 14.

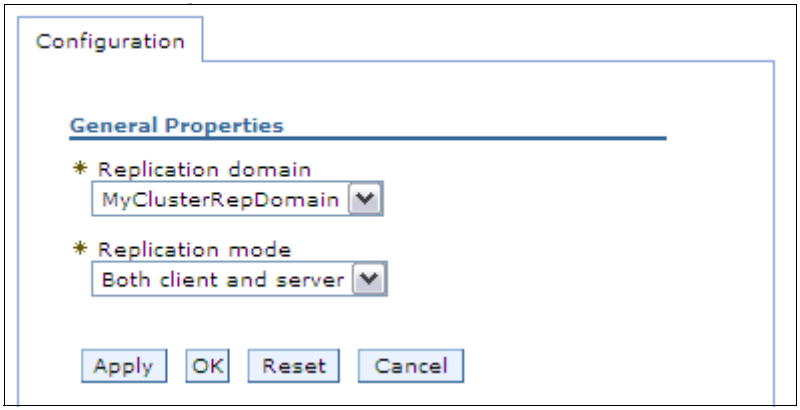

Figure 14 Data replication service settings

Select the replication topology by specifying the replication mode. Selecting **Both client and server** identifies this as a peer-to-peer topology. In a client/server topology, select **Client only** for application servers that will be responding to user requests. Select **Server only** for those that will be used as replication servers.

g. Click **OK**.

3. Repeat the previous steps for the rest of the application servers in the cluster.

4. Save the configuration and restart the cluster. You can restart the cluster by selecting **Servers** → **Clusters** → **WebSphere application server clusters**. Check the cluster, and click **Stop**. After the messages indicate the cluster has stopped, click **Start**.

HTTP session replication in the z/OS controller

WebSphere Application Server V7 for z/OS can store replicated HTTP session data in the controller and replicate data to other WebSphere Application Servers. HTTP session data stored in a controller is retrievable by any of the servants of that controller. HTTP session affinity is still associated to a particular servant; however, if that servant should fail, any of the other servants can retrieve the HTTP session data stored in the controller and establish a new affinity.

The capability of storing HTTP sessions in the controller can also be enabled in unmanaged application servers on z/OS. When this capability is enabled, servants store the HTTP session data in the controller for retrieval when a servant fails which is similar to managed servers. HTTP session data stored in the controller of an unmanaged application server is not retrievable by other application servers and is not replicated to other application servers.

To store HTTP session data in the controller in an unmanaged application server:

1. Select to **Servers** → **Server Types** → **WebSphere application servers** → ***server_name***.
2. Under Server Infrastructure, click **Java and Process Management** → **Process Definition** → **Servant** → **Java Virtual Machine** → **Custom Properties**.
3. In the name field put `HttpSessionEnableUnmanagedServerReplication` and `true` on value field. (Figure 15 on page 36).

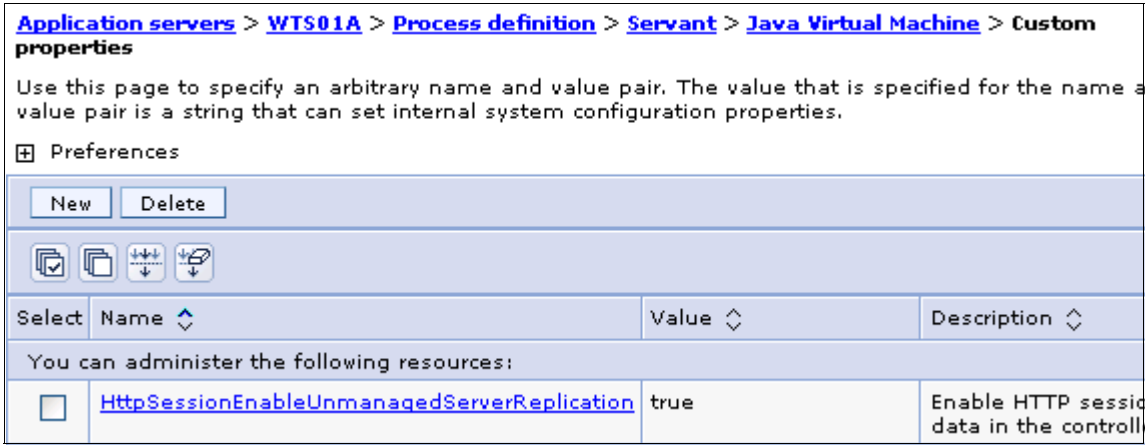

Figure 15 Enable HTTP session in the controller

4. Save and restart the application server.

Session management tuning

Performance tuning for session management persistence consists of defining the following conditions:

- How often session data is written (write frequency settings).
- How much data is written (write contents settings).
- When the invalid sessions are cleaned up (session cleanup settings).

These settings are set in the Custom tuning parameters found under the Additional properties section for session management settings. Several combinations of these settings are predefined and available for selection, or you can customize them. These options are available on **Servers** → **Server Types** → **WebSphere application servers** → ***server_name*** → **Session management** → **Distributed environment settings** → **Custom Tuning parameters**.

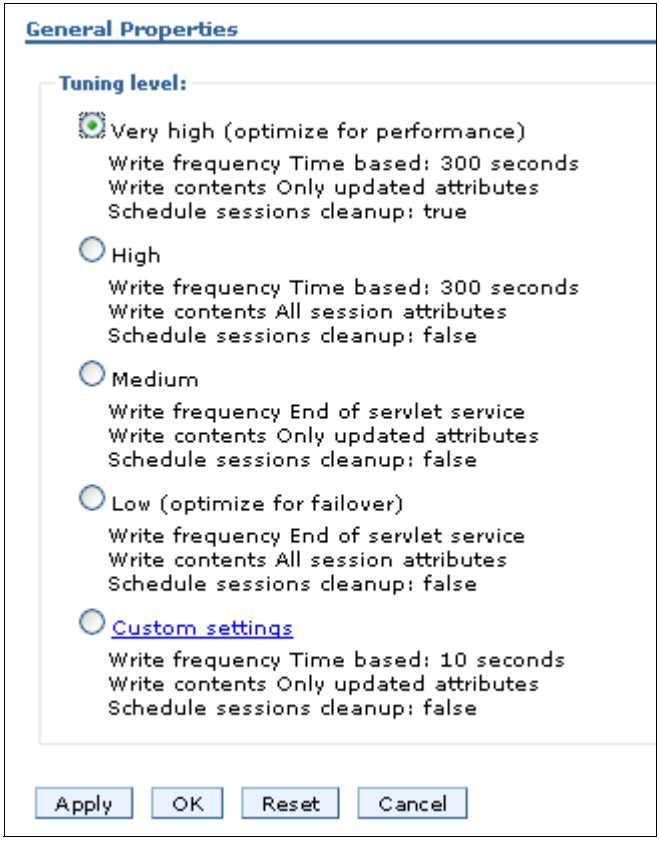

Figure 16 Distributed session management tuning levels

Custom settings

You can customize the settings by selecting the **custom settings option**. The settings can be seen in Figure 17.

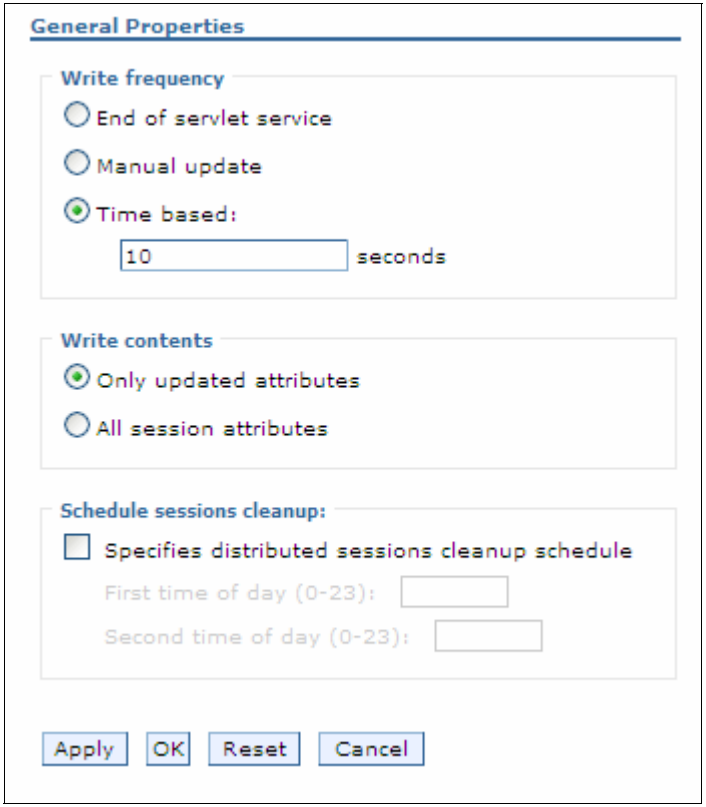

Figure 17 Session management tuning parameters

The following sections go into more detail on these custom settings.

Writing frequency settings

You can select from three different settings that determine how often session data is written to the persistent data store:

- End of servlet service:

  If the session data has changed, it will be written to the persistent store after the servlet finishes processing an HTTP request.

- Manual update:

  The session data will be written to the persistent store when the sync() method is called on the IBMSession object.

- Time-based:

  The session data will be written to the persistent store based on the specified write interval value.

Note: The **last access time** attribute is always updated each time the session is accessed by the servlet or JSP, whether the session is changed or not. This is done to make sure the session does not time out:

- If you choose the end of servlet service option, each servlet or JSP access will result in a corresponding persistent store update of the last access time.
- If you select the manual update option, the update of the last access time in persistent store occurs on sync() call or at later time.
- If you use time-based updates, the changes are accumulated and written in a single transaction. This can significantly reduce the amount of I/O to the persistent store.

See “Reducing persistent store I/O” on page 56 for options to change this database update behavior.

Consider an example where the Web browser accesses the application once every five seconds:

- In End of servlet service mode, the session would be written out every five seconds.
- In Manual update mode, the session would be written out whenever the servlet issues IBMSession.sync(). It is the responsibility of the servlet writer to use the IBMSession interface instead of the HttpSession Interface and the servlets/JSPs must be updated to issue the sync().
- In Time-based mode, the servlet or JSP need not use the IBMSession class nor issue IBMSession.sync(). If the write interval is set to 120 seconds, then the session data is written out at most every 120 seconds.

End of servlet service

When the write frequency is set to the end of servlet service option, WebSphere writes the session data to the persistent store at the completion of the HttpServlet.service() method call. The write content settings determine output.

Manual update

In manual update mode, the session manager only sends changes to the persistent data store if the application explicitly requests a save of the session information.

Note: Manual updates use an IBM extension to HttpSession that is not part of the Servlet 2.5 API.

Manual update mode requires an application developer to use the IBMSession class for managing sessions. When the application invokes the sync() method, the session manager writes the modified session data and last access time to the persistent store. The session data written to the persistent store 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 database. The session manager will only update the last access time in the persistent store asynchronously, at later time. Example 5 shows how the IBMSession class can be used to manually update the persistent store.

Example 5 Using IBMSession for manual update of the persistent store

```
public void service (HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException
{
   // Use the IBMSession to hold the session information
   // We need the IBMSession object because it has the manual update
   // method sync()
   com.ibm.websphere.servlet.session.IBMSession session =

(com.ibm.websphere.servlet.session.IBMSession)req.getSession(true);

   Integer value = 1;

   //Update the in-memory session stored in the cache
   session.putValue("MyManualCount.COUNTER", value);

   //The servlet saves the session to the persistent store
   session.sync();
}
```

This interface gives the Web application developer additional control over when and if session objects go to the persistent data store. If the application does not invoke the sync() method, and manual update mode is specified, the session updates go only to the local session cache, not the persistent data store. Web developers use this interface to reduce unnecessary writes to the session database, and thereby to improve overall application performance.

All servlets in the Web application server must perform their own session management in manual update mode.

Time-based writes to the session database

Using the time-based write option will write session data to the persistent store at a defined write interval. The reasons for implementing time-based write lies in the changes introduced with the Servlet 2.2 API. The Servlet 2.2 specification introduced two key concepts:

- It limits the scope of a session to a single Web application.
- It both explicitly prohibits concurrent access to an HttpSession from separate Web applications, and allows for concurrent access within a given JVM.

Because of these changes, WebSphere provides the session affinity mechanism that assures an HTTP request is routed to the Web application handling its HttpSession. This assurance still holds in a WLM environment when using persistent HttpSessions. This means that the necessity to immediately write the session data to the persistent store can now be relaxed somewhat in these environments, as well as non-clustered environments, because the persistent store is used now only for failover and session cache full scenarios.

With this in mind, it is now possible to gain potential performance improvements by reducing the frequency of persistent store writes.

Note: Time-based writes requires session affinity for session data integrity.

The following details apply to time-based writes:

- The expiration of the write interval does not necessitate a write to the persistent store unless the session has been touched (getAttribute/setAttribute/removeAttribute was called since the last write).
- If a session write interval has expired and the session has only been retrieved (request.getSession() was called since the last write), then the last access time will be written to the persistent store regardless of the write contents setting.
- If a session write interval has expired and the session properties have been either accessed or modified since the last write, then the session properties will be written in addition to the last access time. Which session properties get written is dependent on the write contents settings.
- Time-based write allows the servlet or JSP to issue IBMSession.sync() to force the write of session data to the database.
- If the time between session servlet requests for a particular session is greater than the write interval, then the session effectively gets written after each service method invocation.

- The session cache should be large enough to hold all of the active sessions. Failure to do this will result in extra persistent store writes, because the receipt of a new session request can result in writing out the oldest cached session to the persistent store. To put it another way, if the session manager has to remove the least recently used HttpSession from the cache during a full cache scenario, the session manager will write that HttpSession using the Write contents settings upon removal from the cache.
- The session invalidation time must be at least twice the write interval to ensure that a session does not inadvertently get invalidated prior to getting written to the persistent store.
- A newly created session will always be written to the persistent store at the end of the service method.

Writing content settings

These options control what is written. See “Contents written to the persistent store using a database” on page 45 before selecting one of the options, because there are several factors to decide. The options available are:

- Only updated attributes are written to the persistent store.
- All session attributes are written to the persistent store.

Session cleanup settings

WebSphere allows the administrator to defer (to off-hours) the clearing of invalidated sessions from the persistent store. *Invalidated sessions* are sessions that are no longer in use and timed out. For more information, see “Invalidating sessions” on page 49.

Select **Specifies distributed sessions cleanup schedule** to enable this option.

The cleanup can be done either once or twice a day. The following fields are available:

- First time of day (0-23) is the first hour, during which the invalidated persistent sessions will be cleared from the persistent store. This value must be a positive integer between 0 and 23.
- Second time of day (0-23) is the second hour, during which the invalidated persistent sessions will be cleared from the persistent store. This value must be a positive integer between 0 and 23.

Consider using scheduled invalidation for intranet-style applications that have a somewhat fixed number of users wanting the same HTTP session for the whole business day.

Larger DB2 page sizes and database persistence

WebSphere supports 4 KB, 8 KB, 16 KB, and 32 KB page sizes, and can have larger varchar for bit data columns of 7 KB, 15 KB, or 31 KB. Using this performance feature, we see faster persistence for HttpSessions of sizes of 7 KB to 31 KB in the single-row case, or attribute sizes of 4 KB - 31 KB in the multi-row case.

Enabling the use of greater than 4K page size involves dropping any existing table created with a 4 KB buffer pool and tablespace. This also applies if you subsequently change between 4 KB, 8 KB, 16 KB, or 32 KB.

To use a page size other than the default 4 KB, do the following steps:

1. If the SESSIONS table already exists, drop it from the DB2 database:

   ```
   DB2 connect to session
   DB2 drop table sessions
   ```

2. Create a new DB2 buffer pool and tablespace, specifying the same page size (8 KB, 16 KB, or 32 KB) for both, and assign the new buffer pool to this tablespace. Example 6 shows simple steps for creating an 8 KB page.

Example 6 Creating an 8K page size

```
DB2 connect to session
DB2 CREATE BUFFERPOOL sessionBP SIZE 1000 PAGESIZE 8K
DB2 connect reset
DB2 connect to session
DB2 CREATE TABLESPACE sessionTS PAGESIZE 8K MANAGED BY SYSTEM USING
('D:\DB2\NODE0000\SQL00005\sessionTS.0') BUFFERPOOL sessionBP
DB2 connect reset
```

   Refer to the DB2 product documentation for details.

3. Configure the correct tablespace name and page size, and DB2 row size, in the session management database configuration. See Figure 9 on page 25.

Restart WebSphere. On startup, the session manager creates a new SESSIONS table based on the page size and tablespace name specified.

Single and multi-row schemas (database persistence)

When using the single-row schema, each user session maps to a single database row. This is WebSphere's default configuration for persistent session management. With this setup, there are hard limits to the amount of user-defined, application-specific data that WebSphere Application Server can access.

When using the multi-row schema, each user session maps to multiple database rows, with each session attribute mapping to a database row.

In addition to allowing larger session records, using a multi-row schema can yield performance benefits, as discussed in "Multirow persistent sessions: Database persistence" on page 56.

Switching from single-row to multi-row schema

To switch from single-row to multi-row schema for sessions, do the following steps:

1. Modify the session manager properties to switch from single to multi-row schema. Select the **Use multi row schema** on **Servers** → **Server Types** → **WebSphere application servers** → ***server_name*** → **Session management** → **Distributed environment settings** → **Database settings**.
2. Manually drop the database table or delete all the rows in the session database table. To drop the table:
   a. Determine which data source the session manager is using. This is set in the session management distributed settings window. See "Enabling database persistence" on page 21.
   b. Look up the database name in the data source settings. Find the JDBC provider, then the data source. The database name is in the custom settings.
   c. Use the database facilities to connect to the database and drop it.
3. Restart the application server or cluster.

Design considerations

Consider configuring direct, single-row usage to one database and multi-row usage to another database while you verify which option suits your application's specific needs. You can do this by switching the data source used, then monitoring the performance. Table 2 provides an overview.

Table 2 Single versus multi-row schemas

Programming issue	Application scenario
Reasons to use single-row	► You can read/write all values with just one record read/write. ► This takes up less space in a database, because you are guaranteed that each session is only one record long.
Reasons *not* to use single-row	► There is a 2 MB limit of stored data per session. The sum of sizes of all session attributes is limited to 2 MB.
Reasons to use multi-row	► The application can store an unlimited amount of data. You are limited only by the size of the database and a 2 MB-per-record limit. The size of each session attribute can be 2 MB. ► The application can read individual fields instead of the whole record. When large amounts of data are stored in the session but only small amounts are specifically accessed during a given servlet's processing of an HTTP request, multi-row sessions can improve performance by avoiding unneeded Java object serialization.
Reasons *not* to use multi-row	► If data is small in size, you probably do not want the extra overhead of multiple row reads when everything could be stored in one row.

In the case of multi-row usage, design your application data objects so they do not have references to each other. This is 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 multi-row 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 from that stored. A and B behave as independent objects.

Contents written to the persistent store using a database

WebSphere supports two modes for writing session contents to the persistent store:

► Only updated attributes:

Write only the HttpSession properties that have been updated via setAttribute() and removeAttribute().

- All session attributes:

  Write all the HttpSession properties to the database.

These settings are found in the tuning parameters for distributed environment settings (select **custom settings**).

When a new session is initially created with either of the above two options, the entire session is written, including any Java objects bound to the session. When using database persistence, the behavior for subsequent servlet or JSP requests for this session varies depending on whether the single-row or multi-row database mode is in use:

- In single-row mode, choose from the following options:
  - Only updated attributes:

    If any session attribute has been updated, through setAttribute or removeAttribute, then all of the objects bound to the session will be written to the database.
  - All session attributes:

    All bound session attributes will be written to the database.
- In multi-row mode, choose from the following options:
  - Only updated attributes

    Only the session attributes that were specified via setAttribute or removeAttribute will be written to the database.
  - All session attributes

    All of the session attributes that reside in the cache will be written to the database. If the session has never left the cache, then this should contain all of the session attributes.

By using the All session attributes mode, servlets and JSPs can change Java objects that are attributes of the HttpSession without having to call setAttribute() on the HttpSession for that Java object in order for the changes to be reflected in the database.

Using the All session attributes mode provides some flexibility to the application programmer and protects against possible side effects of moving from local sessions to persistent sessions.

However, using All session attributes mode can potentially increase activity and be a performance drain. Individual customers will have to evaluate the pros and cons for their installation. It should be noted that the combination of All session attributes mode with time-based write could greatly reduce the performance penalty and essentially give you the best of both worlds.

As shown in Example 7 and Example 8, the initial session creation contains a setAttribute, but subsequent requests for that session do not need to use setAttribute.

Example 7 Initial servlet

```
HttpSession sess = request.getSession(true);
myClass myObject = new myClass();
myObject.someInt = 1;
sess.setAttribute("myObject", myObject);   // Bind object to the session
```

Example 8 Subsequent servlet

```
HttpSession sess = request.getSession(false);
myObject =  sess.getAttribute("myObject");   // get bound session object
myObject.someInt++;  // change the session object
// setAttribute() not needed with write "All session attributes" specified
```

Example 9 and Example 10 show setAttribute is still required even though the write all session attributes option is enabled.

Example 9 Initial servlet

```
HttpSession sess = request.getSession(true);
String myString = new String("Initial Binding of Session Object");
sess.setAttribute("myString", myString);   // Bind object to the session
```

Example 10 Subsequent servlet

```
HttpSession sess = request.getSession(false);
String myString  = sess.getAttribute("myString"); // get bound session object
...
myString = new String("A totally new String");  // get a new String object
sess.setAttribute("myString", myString);   // Need to bind the object to the
session since a NEW Object is used
```

HttpSession set/getAttribute action summary

Table 3 summarizes the action of the HttpSession setAttribute and removeAttribute methods for various combinations of the row type, write contents, and write frequency session persistence options.

Table 3 Write contents versus write frequency

Row type	Write contents	Write frequency	Action for setAttribute	Action for removeAttribute
Single-row	Only updated attributes	End of servlet service / sync() call with Manual update	If any of the session data has changed, then write all of this session's data from cache.[1]	If any of the session data has changed, then write all of this session's data from cache.[1]
Single-row	Only updated attributes	Time-based	If any of the session data has changed, then write all of this session's data from cache.[1]	If any of the session data has changed, then write all of this session's data from cache.[1]
	All session attributes	End of servlet service / sync() call with Manual update	Always write all of this session's data from cache.	Always write all of this session's data from cache.
		Time-based	Always write all of this session's data from cache.	Always write all of this session's data from cache.
Multi-row	Only updated attributes	End of servlet service / sync() call with Manual update	Write only thread-specific data that has changed.	Delete only thread-specific data that has been removed.
		Time-based	Write thread-specific data that has changed for *all* threads using this session.	Delete thread-specific data that has been removed for *all* threads using this session.
	All session attributes	End of servlet service / sync() call with Manual update	Write all session data from cache.	Delete thread-specific data that has been removed for *all* threads using this session.
		Time-based	Write all session data from cache.	Delete thread-specific data that has been removed for *all* threads using this session.

[1] When a session is written to the database while using single-row mode, *all* of the session data is written. Therefore, no database deletes are necessary for properties removed with removeAttribute(), because the write of the entire session does not include removed properties.

Multi-row mode has the notion of thread-specific data. *Thread-specific data* is defined as session data that was added or removed while executing under this

thread. If you use End of servlet service or Manual update modes and enable **Only updated attributes**, then only the thread-specific data is written to the database.

Invalidating sessions

This section discusses invalidating sessions when the user no longer needs the session object, for example, when the user has logged off a site. Invalidating a session removes it from the session cache, as well as from the persistent store.

WebSphere offers three methods for invalidating session objects:

- Programmatically, you can use the invalidate() method on the session object. If the session object is accessed by multiple threads in a Web application, be sure that none of the threads still have references to the session object.
- An invalidator thread scans for timed-out sessions every n seconds, where n is configurable from the administrative console. The session timeout setting is in the general properties of the session management settings.
- For persistent sessions, the administrator can specify times when the scan runs. This feature has the following benefits when used with persistent session:
  - Persistent store scans can be scheduled during periods that normally have low demand. This avoids slowing down online applications due to contention in the persistent store.
  - When this setting is used with the End of servlet service write frequency option, WebSphere does not have to write the last access time with every HTTP request. The reason is that WebSphere does not have to synchronize the invalidator thread's deletion with the HTTP request access.

  You can find the schedule sessions cleanup setting in the Session management settings under the Custom tuning parameters for distributed environments (select **custom settings**).

  If you are going to use session cleanup, be aware of these considerations:

  - HttpSession timeouts are not enforced. Instead, all invalidation processing is handled at the configured invalidation times.
  - With listeners, described in “Session listeners” on page 50, processing is potentially delayed by this configuration. Session cleanup scheduling is not recommended if listeners are used.

Session listeners

Listener classes are defined to listen for state changes of a session and its attributes. This allows greater control over interactions with sessions, leading programmers to monitor creation, deletion, and modification of sessions. Programmers can perform initialization tasks when a session is created, or clean up tasks when a session is removed. It is also possible to perform some specific tasks for the attribute when an attribute is added, deleted, or modified.

The following the Listener interfaces are used to monitor the events associated with the HttpSession object:

- javax.servlet.http.HttpSessionListener:

  Use this interface to monitor creation and deletion, including session timeout, of a session.

- javax.servlet.http.HttpSessionAttributeListener:

  Use this interface to monitor changes of session attributes, such as add, delete, and replace.

- javax.servlet.http.HttpSessionActivationListener:

  This interface monitors activation and passivation of sessions. This interface is useful to monitor if the session exists, whether in memory or not, when persistent session is used.

Table 4 is a summary of the interfaces and methods.

Table 4 Listener interfaces and their methods

Target	Event	Interface	Method
session	create	HttpSessionListener	sessionCreated()
	destroy	HttpSessionListener	sessionDestroyed()
	activate	HttpSessionActivationListener	sessionDidActivate()
	passivate	HttpSessionActivationListener	sessionWillPassivate()
attribute	add	HttpSessionAttributeListener	attributeAdded()
	remove	HttpSessionAttributeListener	attributeRemoved()
	replace	HttpSessionAttributeListener	attributeReplaced()

For more information, see J*ava Platform Enterprise Edition, v 5.0 API Specifications* at:

http://java.sun.com/javaee/5/docs/api/

Session security

WebSphere Application Server V7 maintains the security of individual sessions. When session manager integration with WebSphere security is enabled, the session manager checks the user ID of the HTTP request against the user ID of the session held within WebSphere. This check is done as part of the processing of the request.getSession() function. If the check fails, WebSphere throws an com.ibm.websphere.servlet.session.UnauthorizedSessionRequestException exception. If it succeeds, the session data is returned to the calling servlet or JSP.

Session security checking works with the standard HttpSession. The identity or user name of a session can be accessed through the com.ibm.websphere.servlet.session.IBMSession interface. An unauthenticated identity is denoted by the user name *anonymous*.

The session manager uses WebSphere's security infrastructure to determine the authenticated identity associated with a client HTTP request that either retrieves or creates a session.

Security integration rules for HTTP sessions

Session management security has the following rules:

- Sessions in unsecured pages are treated as accesses by the anonymous user.
- Sessions created in unsecured pages are created under the identity of that anonymous user.
- Sessions in secured pages are treated as accesses by the authenticated user.
- 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 unsecure page. Do not mix access to secure and unsecure pages.

Table 5 lists possible scenarios when security integration is enabled, where outcomes depend on whether the HTTP request was authenticated and whether a valid session ID and user name was passed to the session manager.

Table 5 HTTP session security

Request session ID/ user name.	**Unauthenticated HTTP request is used to retrieve the session.**	**Authenticated HTTP request is used to retrieve the session. The user ID in the HTTP request is FRED.**
No session ID was passed in for this request, or the ID is for a session that is no longer valid.	A new session is created. The user name is anonymous.	A new session is created. The user name is FRED.
A valid session ID is received. The current session user name is anonymous.	The session is returned.	The session is returned. The session manager changes the user name to FRED.
A valid session ID is received. The current session user name is FRED.	The session is not returned. UnauthorizedSession-RequestException is thrown.[1]	The session is returned.
A valid session ID is received. The current session user name is BOB.	The session is not returned. UnauthorizedSession-RequestException is thrown.[1]	The session is not returned. UnauthorizedSession-RequestException is thrown.[1]
[1] com.ibm.websphere.servlet.session.UnauthorizedSessionRequestException is thrown to the servlet or JSP.		

See “General properties for session management” on page 12 for more information about the security integration setting.

Session performance considerations

This section includes guidance for developing and administering scalable, high-performance Web applications using WebSphere Application Server V7 session support.

Session size

Large session objects pose several problems for a Web application. If the site uses session caching, large sessions reduce the memory available in the WebSphere instance for other tasks, such as application execution.

For example, assume that a given application stores 1 MB of information for each user session object. If 100 users arrive over the course of 30 minutes, and assume the session timeout remains at 30 minutes, the application server instance must allocate 100 MB just to accommodate the newly arrived users in the session cache:

```
1 MB for each user session * 100 users = 100 MB
```

Note this number does not include previously allocated sessions that have not timed out yet. The memory required by the session cache could be considerably higher than 100 MB.

Web developers and administrators have several options for improving the performance of session management:

- Reduce the size of the session object.
- Reduce the size of the session cache.
- Add additional application servers.
- Invalidate unneeded sessions.
- Increase the memory available.
- Reduce the session timeout interval.

Reducing session object size

Web developers must consider carefully the information kept by the session object:

- Remove information easily obtained or easily derived to help keep the session object small.
- Remove unnecessary, unneeded, or obsolete data from the session.
- Consider whether it would be better to keep a certain piece of data in an application database rather than in the HTTP session. This gives the developer full control over when the data is fetched or stored and how it is combined with other application data. Web developers can leverage the power of SQL if the data is in an application database.

Reducing object size becomes particularly important when persistent sessions are used. Serializing a large amount of data and writing it to the persistent store requires significant WebSphere performance overhead. Even if the option to write only updated attributes is enabled, if the session object contains large Java objects or collections of objects that are updated regularly, there is a significant performance penalty in persisting these objects. This penalty can be reduced by using time-based writes.

Notes: In general, you can obtain the best performance with session objects that are less than 2 KB in size. When the session object exceeds 4-5 KB, you can expect a significant decrease in performance.

Even if session persistence is not an issue, minimizing the session object size will help to protect your Web application from scale-up disasters as user numbers increase. Large session objects will require more and more JVM memory, leaving no room to run servlets.

See “Larger DB2 page sizes and database persistence” on page 43 to learn how WebSphere can provide faster persistence of larger session objects when using DB2.

Session cache size

The session manager allows administrators to change the session cache size to alter the cache’s memory footprint. By default, the session cache holds 1000 session objects. By lowering the number of session objects in the cache, the administrator reduces the memory required by the cache.

However, if the user’s session is not in the cache, WebSphere must retrieve it from either the overflow cache (local caching), or the session store (for persistent sessions). If the session manager must retrieve persistent sessions frequently, the retrievals can impact overall application performance.

WebSphere maintains overflowed local sessions in memory, as discussed in “Local sessions” on page 10. Local session management with cache overflow enabled allows an unlimited number of sessions in memory. To limit the cache footprint to the number of entries specified in session manager, use persistent session management, or disable the overflow.

Note: When using local session management without specifying the Allow overflow property, a full cache will result in the loss of user session objects.

Creating additional application servers

WebSphere also gives the administrator the option of creating additional application servers. Creating additional instances spreads the demand for memory across more JVMs, thus reducing the memory burden on any particular instance. Depending on the memory and CPU capacity of the machines involved, the administrator can add additional instances within the same machine. Alternatively, the administrator can add additional machines to form a hardware cluster, and spread the instances across this cluster.

Note: When configuring a cluster, session affinity routing provides the most efficient strategy for user distribution within the cluster, even with session persistence enabled. With cluster members, the Web server plug-in provides affinity routing among cluster member instances.

Invalidating unneeded sessions

If the user no longer needs the session object, for example, when the user has logged out of the site, it should be invalidated. Invalidating a session removes it from the session cache, as well as from the session database. For more information, see “Invalidating sessions” on page 49.

Increasing available memory

WebSphere allows the administrator to increase an application server’s heap size. By default, WebSphere allocates 256 MB as the maximum heap size. Increasing this value allows the instance to obtain more memory from the system, and thus hold a larger session cache.

A practical limit exists, however, for an instance heap size. The machine memory containing the instance needs to support the heap size requested. Also, if the heap size grows too large, the length of the garbage collection cycle with the JVM might impact overall application performance. This impact has been reduced with the introduction of multi-threaded garbage collection.

Session timeout interval

By default, each user receives a 30 minute interval between requests before the session manager invalidates the user’s session. Not every site requires a session timeout interval this generous. By reducing this interval to match the requirements of the average site user, the session manager purges the session from the cache and the persistent store, if enabled, more quickly.

Avoid setting this parameter too low and frustrating users. In some cases where the persistent store contains a large number of entries, frequent execution of the timeout scanner reduces overall performance. In cases where the persistent store contains many session entries, avoid setting the session timeout so low

that it triggers frequent, expensive scans of the persistent store for timed-out sessions. Alternatively, the administrator should consider schedule-based invalidation where scans for invalid object can be deferred to a time that normally has low demand. See “Invalidating sessions” on page 49.

Reducing persistent store I/O

From a performance point of view, the Web developer’s considerations are the following:

- Optimize the use of the HttpSession within a servlet. Only store the minimum amount of data required in HttpSession. Data that does not have to be recovered after a cluster member fails or is shut down can be best kept elsewhere, such as in a hash table. Recall that HttpSession is intended to be used as a *temporary* store for state information between browser invocations.
- Specify session=false in the JSP directive for JSPs that do not need to access the session object.
- Use time-based write frequency mode. This greatly reduces the amount of I/O, because the persistent store updates are deferred up to a configurable number of seconds. Using this mode, all of the outstanding updates for a Web application are written periodically based on the configured write interval.
- Use the Schedule sessions cleanup option. When using the End of servlet service write frequency mode, WebSphere does not have to write out the last access time with every HTTP request. This is because WebSphere does not have to synchronize the invalidator thread's deletion with the HTTP request’s access.

Multirow persistent sessions: Database persistence

When a session contains multiple objects accessed by different servlets or JSPs in the same Web application, multi-row session support provides a mechanism for improving performance. Multi-row session support stores session data in the persistent session database by Web application and value. Table 6 shows a simplified representation of a multi-row database table.

Table 6 Simplified multi-row session representation

Session ID	Web application	Property	Small value	Large value
DA32242SSGE2	ShoeStore	ShoeStore.First.Name	Alice	
DA32242SSGE2	ShoeStore	ShoeStore.Last.Name	Smith	

Session ID	Web application	Property	Small value	Large value
DA32242SSGE2	ShoeStore	ShoeStore.Big.String		A big string....

In this example, if the user visits the ShoeStore application, and the servlet involved needs the user's first name, the servlet retrieves this information through the session API. The session manager brings into the session cache only the value requested. The ShoeStore.Big.String item remains in the persistent session database until the servlet requests it. This saves time by reducing both the data retrieved and the serialization overhead for data the application does not use.

After the session manager retrieves the items from the persistent session database, these items remain in the in-memory session cache. The cache accumulates the values from the persistent session database over time as the various servlets within the Web application request them. With WebSphere's session affinity routing, the user returns to this same cached session instance repeatedly. This reduces the number of reads against the persistent session database, and gives the Web application better performance.

How session data is written to the persistent session database has been made configurable in WebSphere. For information about session updates using single and multi-row session support, see "Single and multi-row schemas (database persistence)" on page 44. Also see "Contents written to the persistent store using a database" on page 45.

Even with multi-row session support, Web applications perform best if the overall contents of the session objects remain small. Large values in session objects require more time to retrieve from the persistent session database, generate more network traffic in transit, and occupy more space in the session cache after retrieval.

Multi-row session support provides a good compromise for Web applications requiring larger sessions. However, single-row persistent session management remains the best choice for Web applications with small session objects. Single-row persistent session management requires less storage in the database, and requires fewer database interactions to retrieve a session's contents (all of the values in the session are written or read in one operation). This keeps the session object's memory footprint small, as well as reducing the network traffic between WebSphere and the persistent session database.

Note: Avoid circular references within sessions if using multi-row session support. The multi-row session support does not preserve circular references in retrieved sessions.

Managing your session database connection pool

When using persistent session management, the session manager interacts with the defined database through a WebSphere Application Server data source. Each data source controls a set of database connections known as a connection pool. By default, the data source opens a pool of no more than 10 connections. The maximum pool size represents the number of simultaneous accesses to the persistent session database available to the session manager.

For high-volume Web sites, the default settings for the persistent session data source might not be sufficient. If the number of concurrent session database accesses exceeds the connection pool size; the data source queues the excess requests until a connection becomes available. Data source queuing can impact the overall performance of the Web application (sometimes dramatically).

For best performance, the overhead of the connection pool used by the session manager needs to be balanced against the time that a client can spend waiting for an occupied connection to become available for use. By definition, a connection pool is a *shared* resource, so in general the best performance is realized typically with a connection pool that has significantly fewer connections than the number of simultaneous users.

A large connection pool does not necessarily improve application performance. Each connection represents memory overhead. A large pool decreases the memory available for WebSphere to execute applications. Also, if database connections are limited because of database licensing issues, the administrator must share a limited number of connections among other Web applications requiring database access as well. This is one area where performance tuning tests are required to determine the optimal setting for a given application.

As discussed above, session affinity routing combined with session caching reduces database read activity for session persistence. Likewise, manual update write frequency, time-based write frequency, and multi-row persistent session management reduce unnecessary writes to the persistent database. Incorporating these techniques can also reduce the size of the connection pool required to support session persistence for a given Web application.

Prepared statement caching is a connection pooling mechanism that can be used to further improve session database response times. A cache of previously prepared statements is available on a connection. When a new prepared statement is requested on a connection, the cached prepared statement is returned, if available. This caching reduces the number of costly prepared statements created, which improves response times. In general, base the prepared statement cache size on the following considerations:

- The smaller of:
  - Number of concurrent users
  - Connection pool size
- The number of different prepared statements

With 50 concurrent users, a connection pool size of 10, and each user using two statements, a select and an insert, the prepared statement cache size should be at least 10 x 2 = 20 statements. Check the Information Center for more details at:

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.nd.multiplatform.doc/info/ae/ae/udat_jdbcdatasorprops.html

Session database tuning

While the session manager implementation in WebSphere provides for a number of parameters that can be tuned to improve performance of applications that utilize HTTP sessions, maximizing performance requires tuning the underlying session persistence table. WebSphere provides a first step by creating an index for the sessions table when creating the table. The index is composed of the session ID, the property ID for multi-row sessions, and the Web application name.

While most database managers provide a great deal of capability in tuning at the table or tablespace level, creating a separate database or instance provides the most flexibility in tuning. Proper tuning of the instance and database can improve performance by 5% or more over that which can be achieved by simply tuning the table or tablespace.

While the specifics vary, depending on the database and operating system, in general, tune and configure the database as appropriate for a database that experiences a great deal of I/O. The database administrator (DBA) should monitor and tune the database buffer pools, database log size, and write frequency. Additionally, maximizing performance requires striping the database or instance across multiple disk drives and disk controllers, and utilizing any hardware or OS buffering available to reduce disk contention.

Stateful session bean failover

Stateful session bean utilizes the functions of the data replication service and workload management.

Each EJB™ container provides a method for stateful session beans to fail over to other servers. This enables you to specify whether failover occurs for the stateful session beans at the EJB module level or container level. You can also override the parent object's stateful session bean replication settings from the module level.

Enabling stateful session bean failover

Depending on the requirement, you might not want to enable failover for every single stateful session bean installed in the EJB container. You can set or override the EJB container settings at either the application or EJB module level. You can either enable or disable failover at each of these levels. For example, consider the following situations:

- You want to enable failover for all applications except for a single application. To do this, you enable failover at the EJB container level and override the setting at the application level to disable failover on the single application.
- You want to enable failover for a single, installed application. To do this, disable failover at the EJB container level and then override the setting at the application level to enable failover on the single application.
- You want to enable failover for all applications except for a single module of an application. To do this, enable failover at the EJB container level, then override the setting at the module application level to disable failover on the single module.
- You want to enable failover for a single, installed EJB module. To do this, disable failover at the EJB container level and then override the setting at the EJB module level to enable failover on the single EJB module.

EJB container stateful session bean failover properties

To access stateful session bean failover properties at the EJB container level from the administrative console:

1. Select **Servers** → **Server Types** → **WebSphere application servers**.
2. Click the application server.
3. In the Container Settings section of the Configuration tab, click **EJB Container Settings** → **EJB container**.

4. In the General Properties section, check **Enable stateful session bean failover using memory-to-memory replication**.

   This check box is disabled until you define a replication domain. This selection has a hyperlink to help you configure the replication settings. If no replication domains are configured, the link takes you to a window where you can create one. If at least one domain is configured, the link takes you to a window where you can select the replication settings to be used by the EJB container. See Figure 18.

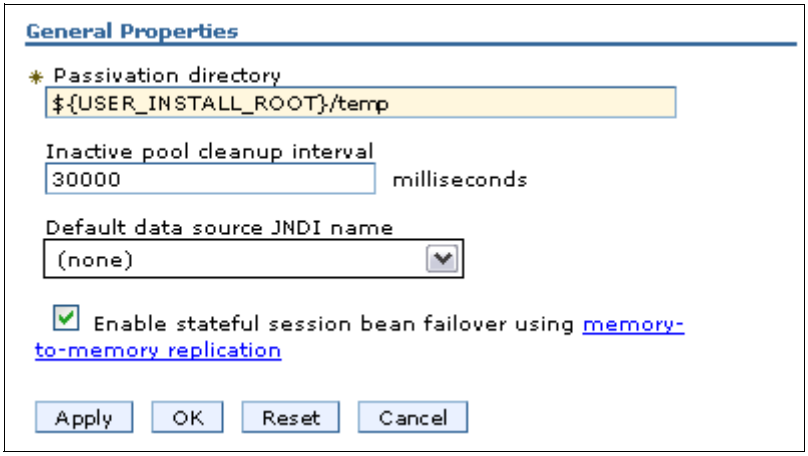

Figure 18 Stateful session bean failover settings at the container level

EJB module stateful session bean failover properties

To access stateful session bean failover properties at the EJB module level from the administrative console:

1. Select **Applications** → **Application Types** → **WebSphere enterprise applications**.
2. Click the application.
3. In the Enterprise Java Bean Properties section of the Configuration tab, click **Stateful session bean failover settings**. See Figure 19.

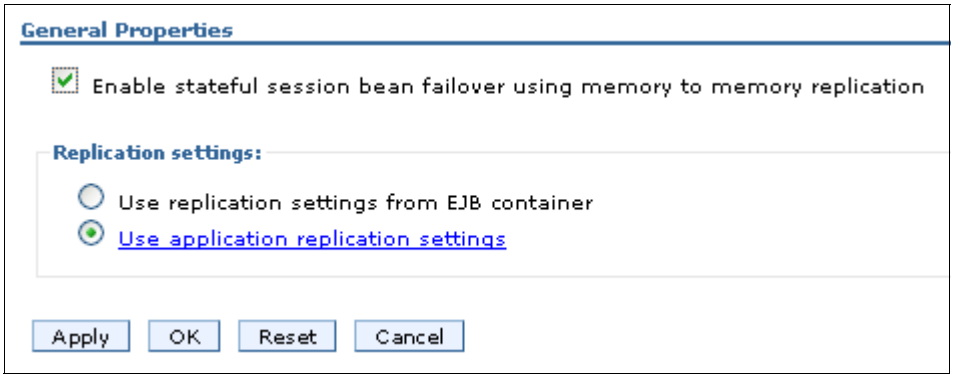

Figure 19 Stateful session bean failover settings at the module level

- Enable stateful session bean failover:

  Check this check box to enable stateful session bean failover. If you want to disable the failover, clear this check box.

- Use replication settings from EJB container:

  If you select this option, any replication settings defined for this application are ignored.

  Important: If you use this radio button, then you must configure memory to memory replication at the EJB container level. Otherwise, the settings on this window are ignored by the EJB container during server startup and the EJB container will log a message indicating that stateful session bean failover is not enabled for this application.

- Use application replication settings:

  If you select this option, you override the EJB container settings. This button is disabled until you define a replication domain. This selection has a hyperlink to help you configure the replication settings. If no replication domains are configured, the link takes you to a window to create one. If at least one domain is configured, the link takes you to a window where you can select the replication settings to be used by the application.

4. Select your choice of replication settings from:
   - Use replication settings from EJB container
   - Use application replication settings using memory-to-memory replication
5. Select **OK**.

On WebSphere Application Server V7 for z/OS, the stateful session bean failover among servants can be enabled. Failover only occurs between the servants of a given unmanaged server. If an unmanaged z/OS server has only one servant, then enabling failover has no effect. An unmanaged z/OS server that has failover enabled does not fail over to another unmanaged z/OS server. To enable this feature, consult the instructions in the Information Center at:

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.nd.multiplatform.doc/info/ae/ae/tejb_sfsbfzos.html

Stateful session bean failover consideration

In the following sections we present a few considerations when using the stateful session bean failover feature.

Stateful session bean activation policy with failover enabled

At application assembly, you can specify an activation policy to use for stateful session beans. It is important to consider that the only time the EJB container prepares for failover by replicating the stateful session bean data using DRS, is when the stateful session bean is passivated. If you configure the bean with an activate once policy, the bean is essentially never passivated. If you configure the activate at transaction boundary policy, the bean is passivated whenever the transaction that the bean is enlisted in completes.

For stateful session bean failover to be useful, the **activate at transaction boundary policy** is required. Rather than forcing you to edit the deployment descriptor of every stateful session bean and reinstall the bean, the EJB container simply ignores the configured activation policy for the bean when you enable failover. The container automatically uses the activate at transaction boundary policy.

Container or bean managed units of work

The relevant units of work in this case are transactions and activity sections. WebSphere Application Server supports stateful session bean failover for:

- Container managed transactions (CMT)
- Bean managed transactions (BMT)
- Container managed activity sessions (CMAS)
- Bean managed activity sessions (BMAS)

In the container-managed cases, preparation for failover only occurs if a request for an enterprise bean method invocation fails to connect to the server. Failover does not take place if the server fails after a request is sent to it and has been acknowledged.

When a failure occurs in the middle of a request or unit of work, WLM cannot safely fail over to another server without some compensation code being executed by the application. When that happens, the application receives a Common Object Request Broker Architecture (CORBA) exception and minor code telling it that transparent failover could not occur because the failure happened during execution of a unit of work.

The application should be written to check for the CORBA exception and minor code, and compensate for the failure. After the compensation code executes, the application can retry the requests and, if a path exists to a backup server, WLM routes the new request to a new primary server for the stateful session bean.

The same is true for bean-managed units of work, transactions, or activity sessions. However, bean managed work introduces a new possibility that needs to be considered.

For bean managed units of work, the failover process is not always able to detect that a BMT or BMAS started by a stateful session bean method has not completed. Thus, it is possible that failover to a new server can occur despite the unit of work failing during the middle of a transaction or session. Because the unit of work is implicitly rolled back, WLM behaves as thought it is safe to transparently fail over to another server, when in fact some compensation code might be required. When this happens, the EJB container detects this on the new server and initiates an exception. This exception occurs under the following scenario:

1. A method of a stateful session bean using bean-managed transaction or activity session calls begin on a UserTransaction it obtained from the SessionContext. The method does some work in the started unit of work, but does not complete the transaction or session before returning to the caller of the method.

2. During post invocation of the method started in step 1, the EJB container suspends the work started by the method. This is the action required by EJB specification for bean managed units of work when the bean is a stateful session bean.
3. The client starts several other methods on the stateful session bean. Each invocation causes the EJB container to resume the suspended transaction or activity session, dispatch the method invocation, and then suspend the work again before returning to the caller.
4. The client calls a method on the stateful session bean that completes the transaction or session started in step 1.

This scenario depicts a *sticky* bean-managed unit of work. The transaction or activity session sticks around for more than a single stateful session bean method. If an application uses a sticky BMT or BMAS, and the server fails after a sticky unit of work completes and before another sticky unit of work starts, failover is successful. However, if the server fails before a sticky transaction or activity session completes, the failover is not successful. Instead, when the failover process routes the stateful session bean request to a new server, the EJB container detects that the failure occurred during an active, sticky transaction or activity session. At that time, the EJB container initiates an exception.

Essentially, this means that failover for both container-managed and bean-managed units of work is not successful if the transaction or activity session is still active. The only real difference is the exception that occurs.

Application design considerations

Consider the following recommendations when designing applications that use the stateful session bean failover process:

- To avoid the possibility described in the section above, you are encouraged to write your application to configure stateful session beans to use container-managed transactions (CMT) rather than bean-managed transactions (BMT).
- If you want immediate failover, and your application creates either an HTTP session or a stateful session bean that stores a reference to another stateful session bean, then the administrator must ensure the HTTP session and stateful session bean are configured to use the same replication domain.
- Do not use a local and a remote reference to the same stateful session bean.

  The Java EE 5 specification has additional requirements for Http Sessions that require the Http Session state objects to be able to contain local references to EJBs.

Normally a stateful session bean instance with a given primary key can only exist on a single server at any given moment in time. Failover might cause the bean to be moved from one server to another, but it never exists on more than one server at a time. However, there are some unlikely scenarios that can result in the same bean instance, the same primary key, existing on more than one server concurrently. When that happens, each copy of the bean is unaware of the other, and no synchronization occurs between the two instances to ensure they have the same state data. Thus, your application receives unpredictable results.

Note: To avoid this situation, you must remember that with failover enabled, your application should never get both a local (EJBLocalObject) and remote (EJBObject) reference to the same stateful session bean instance.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

This document REDP-4580-00 was created or updated on October 12, 2009.

Send us your comments in one of the following ways:
- Use the online **Contact us** review Redbooks form found at:
  ibm.com/redbooks
- Send your comments in an email to:
  redbook@us.ibm.com
- Mail your comments to:
  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099, 2455 South Road
  Poughkeepsie, NY 12601-5400 U.S.A.

Trademarks