J2EE Best Practices

This chapter describes the J2EE best practices. The topics include:

3.1 JSP Best Practices

This section describes JSP best practices. It includes the following topics:

3.1.1 Pre-Translate JSPs Before Deployment

You can use Oracle's ojspc tool to pre-translate the JSPs and avoid the translation overhead that has to be incurred when the JSPs are executed the first time. You can pre-translate the JSPs on the production system or before you deploy them. Also, pre-translating the JSPs allows you the option to deploy only the translated and compiled class files, if you choose not to expose and compromise the JSP source files.

3.1.2 Separate Presentation Markup From Java

Separating presentation markup such as HTML from Java code is a good practice to get better performance from your application. The following are a few tips:

- Use JavaBeans for the business logic and JSPs only for the view. Thus, JSPs should primarily contain logic for HTML (or other presentation markup) generation only.
- Use stylesheets when appropriate to provide even more separation of the aspects of HTML that a user can control better.
- JSPs containing a large amount of static content, including large amounts of HTML code that does not change at runtime, which may result in slow translation and execution. Use dynamic includes, or better, enable the external resource configuration parameter to put the static HTML into a Java resource file.

3.1.3 Use JSP Template Mechanism

Using the JSP code `out.print("<html>")` requires more resources than including static template text. For performance reasons, it is best to reserve the use of `out.print()` for dynamic text.

3.1.4 Set Sessions=False If Not Using Sessions

The default for JSPs is session="true". If your JSPs do not use any sessions, you should set session="false" to eliminate the overhead of creating and releasing these internal sessions created by the JSP runtime. To disable sessions, set the directive as follows:

```
<%@page session="false" %>
```

3.1.5 Always Invalidate Sessions When No Longer Used

Sessions add performance overhead to your Web applications. Each session is an instance of the `javax.servlet.http.HttpSession` class. The amount of memory used per session depends on the size of the session objects created.

If you use sessions, ensure that you explicitly cancel each session using the `invalidate()` method to release the memory occupied by each session when you no longer need it.

The default session timeout for OC4J is 30 minutes. You can change this for a specific application by setting the `<session-timeout>` parameter in the `<session-config>` element of `web.xml`.

3.1.6 Set Main_Mode Attribute To "justrun"

This attribute, found in `global-web-application.xml`, determines whether classes are automatically reloaded or JSPs are automatically recompiled. In a deployment environment set `main_mode` to `justrun`. The runtime dispatcher does not perform any timestamp checking, so there is no recompilation of JSPs or reloading of Java classes. This mode is the most efficient mode for a deployment environment where code is not expected to change.

If comparing timestamps is unnecessary, as is the case in a production deployment environment where source code does not change, you can avoid all timestamp comparisons and any possible retranslations and reloads by setting the

`main_mode` parameter to the value `justrun`. Using this value can improve the performance of JSP applications.

Note that before you set `main_mode` to `justrun`, make sure that the JSP is compiled at least once. You can compile the JSP by invoking it through a browser or by running your application (using the `recompile` value for `main_mode`). This assures that the JSP is compiled before you set the `justrun` flag.

3.1.7 Use Available JSP Tags In Tag Library

JSP tags make the JSP code cleaner, and more importantly, provide easy reuse. In some cases, there is also a performance benefit. Oracle9*i*AS ships with a very comprehensive JSP tag library that will meet most needs. In cases where custom logic is required or if the provided library is insufficient, you can build a custom tag library, if appropriate.

3.1.8 Minimize Context Switching Between Servlets and EJBs

Minimize context switching between different Enterprise JavaBeans (EJB) and servlet components especially when the EJB and Web container processes are different. If context switching is required, co-locate EJBs whenever possible.

3.1.9 Package JSP Files In EAR File For Deployment Rather Than Standalone

Oracle9*i*AS Release 2 supports deploying of JSP files by copying them to the appropriate location. This is very useful when developing and testing the pages. However, this is not recommended for releasing your JSP-based application for production. You should always package JSP files into an Enterprise Archive (EAR) file so that they can be deployed in a standard manner - even across multiple application servers.

3.1.10 Use Compile-Time Object Introspection

Developers should try to rely on compile-time object introspection on the beans and objects generated by the tag library instead of request-time introspection.

3.1.11 Choose Static Versus Dynamic Includes Appropriately

JSP pages have two different include mechanisms:

1. Static includes which have a page directive such as:

```
<%@ include file="filename.jsp" %>
```

2. Dynamic includes which have a page directive such as:

```
<jsp:include page="filename.jsp" flush="true" />
```

Static includes create a copy of the include file in the JSP. Therefore, it increases the page size of the JSP, but it avoids additional trips to the request dispatcher. Dynamic includes are analogous to function calls. Therefore, they do not increase the page size of the calling JSP, but they do increase the processing overhead because each call must go through the request dispatcher.

Dynamic includes are useful if you cannot determine which page to include until after the main page has been requested. Note that a page that can be dynamically included must be an independent entity, which can be translated and executed on its own.

3.1.12 Disable JSP Page Buffer If Not Used

In order to allow part of the response body to be produced before the response headers are set, JSPs can store the body in a buffer.

When the buffer is full or at the end of the page, the JSP runtime will send all headers that have been set, followed by any buffered body content. This buffer is also required if the page uses dynamic `contentType` settings, forwards, or error pages. The default size of a JSP page buffer is 8 KB. If you need to increase the buffer size, for example to 20KB, you can use the following JSP attribute and directive:

```
<%@page buffer="20kb" %>
```

If you are not using any JSP features that require buffering, you can disable it to improve performance; memory will not be used in creating the buffer, and output can go directly to the browser. You can use the following directive to disable buffering:

```
<%@ page buffer="none" %>
```

3.1.13 Use Forwards Instead of Redirects

For JSPs, you can pass control from one page to another by using forward or redirect, but forward is always faster. When you use forward, the forwarded target page is invoked internally by the JSP runtime, which continues to process the request. The browser is totally unaware that such an action has taken place.

When you use redirect, the browser actually has to make a new request to the redirected page. The URL shown in the browser is changed to the URL of the redirected page, but it stays the same in a forward operation.

Therefore, redirect is always slower than the forward operation. In addition, all request scope objects are unavailable to the redirected page because redirect involves a new request. Use redirect only if you want the URL to reflect the actual page that is being executed in case the user wants to reload the page.

3.1.14 Use JSP Tagged Cache

Using the Java Object Cache in JSP pages, as opposed to servlets, is particularly convenient because JSP code generation can save much of the development effort. OracleJSP provides the following tags for using the Java Object Cache:

- `ojsp:cache`
- `ojsp:cacheXMLObj`
- `ojsp:useCacheObj`
- `ojsp:invalidateCache`

Use the `ojsp:cacheXMLObj` or `ojsp:cache` tag to enable caching and specify cache settings. Use `ojsp:useCacheObj` to cache any Java serializable object. Use the `ojsp:invalidateCache` tag to invalidate a cache block. Alternatively, you can arrange invalidation through the `invalidateCache` attribute of the `ojsp:cacheXMLObj` or `ojsp:cache` tag.

3.1.15 Use well_known_taglib_loc To Share Tag Libraries

As an extension of standard JSP "well-known URI" functionality described in the JSP 1.2 specification, the OC4J JSP container supports the use of a shared tag library directory where you can place tag library JAR files to be shared across multiple Web applications. The benefits are:

- avoidance of duplication of tag libraries between applications
- allow easy maintenance as the TLDs can be in a single JAR file
- application size is minimized

OC4J JSP `well_known_taglib_loc` configuration parameter specifies the location of the shared tag library directory. The default location is `j2ee/home/jsp/lib/taglib/` under the *`ORACLE_HOME`* directory. If *`ORACLE_HOME`* is not defined, it is the current directory (from which the OC4J process was started).

The shared directory must be added to the server-wide `CLASSPATH` by specifying it as a library path element. The default location is set in the `application.xml` file in the OC4J configuration files directory (`j2ee/home/config` by default) and can be altered.

3.1.16 Use JSP-Timeout for Efficient Memory Utilization

Resource utilization is a key factor for any efficient application. Oracle9*i*AS 9.0.3 introduces the `<orion-web-app>` attribute `jsp-timeout` that can be specified in the OC4J `global-web-application.xml` file or `orion-web.xml` file. The `jsp-timeout` attribute specifies an integer value, in seconds, after which any JSP page will be removed from memory if it has not been requested. This frees up resources in situations where some pages are called infrequently. The default value is 0, for no timeout.

Like other attributes use the `<orion-web-app>` element of the OC4J `global-web-application.xml` file to apply to all applications in an OC4J instance. To set configuration values to a specific application, use the `<orion-web-app>` element of the deployment-specific `orion-web.xml` file.

3.1.17 Workarounds for the 64K Size Limit for the Generated Java Method

The Java Virtual Machine (JVM) limits the amount of code to 65536 bytes per Java method. Sometimes, as the JSPs grow larger, there is a possibility of hitting this limit. The following are some suggestions to workaround this limitation:

- As a general rule, design smaller JSPs for your web application.
- If your JSP uses tag libraries heavily, and if you are hitting the 64k limit, use the reduce_tag_code config parameter to reduce the size of generated code for custom tag usage. Note that this may impact performance.

3.2 Servlet Best Practices

This section describes servlet best practices. It includes the following topics:

3.2.1 Perform Costly One-Time Operation in Servlet init() Method

Use a servlet's `init()` method to perform any costly one-time initialization operations. Examples include:

1. Setting up resource pools.
2. Retrieving common data from a database that can be cached in the mid-tier to reduce warm-up time.

The `destroy()` method can be used to execute operations that release resources acquired in the `init()` method.

3.2.2 Improve Performance by Loading Servlet Classes at OC4J Startup

By default, OC4J loads a servlet when the first request for it is made. OC4J also allows you to load servlet classes when the JVM that runs the servlet is started. To do this, add the `<load-on-startup>` sub-element to the `<servlet>` element in the application's web`.xml` configuration file.

For example, add the `<load-on-startup>` as follows:

```
<servlet>
<servlet-name>viewsrc</servlet-name>
<servlet-class>ViewSrc</servlet-class>
<load-on-startup>
</servlet>
```

Using the load-on-startup facility increases the start-up time for your OC4J process but decreases first-request latency for servlets.

Using Oracle Enterprise Manager, you can also specify that OC4J load an entire Web module on startup. To specify that a Web module is to be loaded on startup, select the Web site Properties page for an OC4J instance, and then select the Load on Startup checkbox.

3.2.3 Analyze Servlet Duration for Performance Problems

It is useful to know the average duration for servicing servlet and JSP requests in your J2EE enterprise application. By understanding how long a servlet takes to service requests when the system is not under load, you can more easily determine the cause of a performance problem when the system is loaded. The average response time of a given servlet is reported in the metric `service.avg` 1 for that servlet. You should only examine this value after making many calls to the servlet so that any startup overhead such as class loading and database connection establishment is amortized.

As an example, suppose you have a servlet for which you notice the `service.avg` to be 32 milliseconds. And, suppose you notice a response time increase when your system is loaded but not CPU bound. When you examine the value of `service.avg`, you might find that the value is close to 32 ms, in which case you can assume the degradation is probably due to your system or application server configuration rather than your application. If, on the other hand, you notice that `service.avg` has increased significantly, you should look for the problem in your application. For example, multiple users of the application may be contending for the same resources, including but not limited to database connections.

See Also:

Oracle9i Application Server Performance Guide

3.2.4 Understand Server Request Load When Debugging

In debugging servlet and JSP problems, it is often useful to know how many requests your OC4J processes are servicing. If the problems are performance related, it is always helpful to know if they are aggravated by a high request load. You can track the requests for a particular OC4J instance using Oracle Enterprise Manager or by viewing an application's Web module metrics.

3.2.5 Find Large Servlets That Require a long Road Time When Debugging

You may find that a servlet application is especially slow the first time it is used after the server is started or that it is intermittently slow. It is possible that when this happens, the server is heavily loaded, and response times are suffering as a result. If there is no indication of a high load, which you can detect by monitoring your access logs, periodically monitoring CPU utilization, or by tracking the number of users that have active requests to the HTTP server(s) and OC4J instance(s), then you may have a large servlet that takes a long time to load.

You can see if you have a slow loading servlet by looking at `service.maxTime`, `service.minTime`, and `service.avg`. If the time to load the servlet is much longer than the time it takes to service the first request after loading, the first user that accesses the servlet after your system is started will feel the delay, and `service.maxTime` will be large. You can avoid this by configuring the system to initialize your servlet when it starts.

3.2.6 Watch for Unused Sessions When Debugging

You should regularly monitor your applications to look for unused sessions. It is easy to inadvertently write servlets that do not invalidate their sessions. Without access to application source code, you may not be aware that it could be causing problems for your production host(s), but sooner or later you may notice higher memory consumption than expected. You can check for unused sessions or sessions which are not being properly invalidated using the session metrics: `sessionActivation.time`, `sessionActivation.completed`, and `sessionActivation.active`.

3.2.7 Watch for Abnormal Session Usage When Debugging

The following is an example that shows an application that creates sessions but never uses them.

The following are metrics for a JSP under `/oc4j/<application>/WEBs/<context>`:

```
session.Activation.active:  500 ops
session.Activation.completed:  0 ops
```

This application created 500 sessions that are all still active. Possibly, this indicates that the application makes unnecessary use of the sessions. Over time, it will cause memory or CPU consumption problems.

A well-tuned application shows `sessionActivation.active` with a value that is less than sessionActivation.completed before the session time out. This indicates that the sessions are probably being used and

cleaned up.

Suppose you have a servlet that uses sessions effectively and invalidates them appropriately. Then, you might see a set of metrics such as the following:

```
session.Activation.active:  2 ops
session.Activation.completed:  500 ops
```

The fact that two sessions are active when more than 500 have been created and completed indicates that sessions are being invalidated after use.

3.2.8 Load Servlet Session Security Routines at Startup

OC4J uses the class `java.security.SecureRandom` for secure seed generation. The very first call to this method is time consuming. Depending on how your system is configured for security, this method may not be called until the very first request for a session-based servlet is received. One alternative is to configure the application to load on startup in the application's `web.xml` configuration file and to create an instance of `SecureRandom` during the class initialization of the application. The result will be a longer OC4J startup time in lieu of a delay in servicing the first request.

3.2.9 Retry Failed Transactions and Idempotent HttpServlet.doGet() Exactly Once

Retries are discouraged in general because if every `catch` block of an N frame `try..catch` stack performs M retries, the innermost method gets retried (MN)/2 times. This is likely to be perceived by the end user as a hang, and hangs are worse than receiving an error message.

If you could pick just one `try..catch` block to retry, it would be best to pick the outermost block. It covers the most code, and therefore, also covers the most exceptions. Of course, only idempotent operations should be retried. Transactions guarantee that database operations can be retried as long as the failed try results in a rollback and all finally blocks restore variables to a state consistent with the rolled back database state. Often, the case will be that a servlet's `doGet()` method will perform the retry, and a servlet's `doPost()` method will rollback any existing transaction and retry with a new transaction.

Other cases where a retry is warranted are:

- The semantics of a checked exception suggest a retry using a different method or different parameters. For example, `ShoppingCart.insert()` might throw an `ItemExists` exception, and this should be caught and `ShoppingCart.incrementQuantity()` should be tried.

- Several object replicas exist (usually in different processes). A failure of one (for example, a remote exception) could be caught and another replica could be tried. Retry only if the operation is idempotent. Most catch-all exceptions do not guarantee that all effects from the failed invocation are undone.

For example, if the database tier uses Oracle Real Application Clusters (ORAC), then a new connection may be to any available database server machine that mounts the desired database. For JDBC, the `DataSource.getConnection()` method is usually configured to pick among ORAC machines.

Also review the following:

- Do One-time Resource Allocation and Cleanup in init() and destroy() Methods.

3.2.9.1 Do One-time Resource Allocation and Cleanup in init() and destroy() Methods.

`init()` and `destroy()` methods are only called during the servlet initialization and destruction respectively.

3.2.10 Use HTTP Servlet.doPost() for Requests That Update Database

The HTTP specification states that the GET method should be idempotent and free of side effects. Proxies and caches along the route from client to mid-tier, as well as a user pressing the reload button, could cause the GET method at the mid-tier to be called more than once.

HTTP POST is not assumed to be idempotent. Browsers typically require client confirmation before another POST operation, and intermediate proxies/caches do not retry or cache the result of a POST. However, a failure may require the client to manually retry (press RELOAD or press BACK on the browser and then re-submit), which is not safe unless the update is idempotent.

Hence, it is important to use POST instead of GET for these kinds of updates. Some practices to be aware of are:

- Applications can warn users about potential duplicate requests. This can be implemented by encoding a unique request-id in a hidden form field and writing the request-id of each update request to the database. An update request first compares its request-id with those of already-processed requests in the database and warns the user about a potential duplicate if there is a match. Because the user may have intended to submit two separate and unique updates, the system cannot make duplicate suppression transparent.

Another good practice is to label non-idempotent submit buttons with advice against reloading or re-submitting the current page and provide instructions on which application level logs should be consulted should a failure occur.

Because this is easier to do than implementing request-ids, this is a more common practice.

3.2.11 Avoid Duplicating Libraries

Avoid duplicating copies of the same library at different location in your application server. Duplication of class libraries can lead to several classloading problems and may consume additional memory and disk space. If your class library is used by multiple applications, then you can put it at the application server level by using the `<library>` tag in `application.xml`. Or, use the `<parent>` attribute in `server.xml` to share libraries in two applications.

If you have a library that is shared between multiple modules in the same application, i.e. two web modules in the same EAR file, then use the WAR file manifest's CLASSPATH to share the class libraries between the modules instead of duplicating the libraries in the `WEB-INF/lib` for every module. In order to enable the CLASSPATH in a WAR file manifest, the following has to be defined in `orion-web.xml`:

```
<web-app-class-loader  include-war-manifest-class-path="true"  />
```

3.2.12 Use Resource Loading Appropriately

If you are using dynamic classloading or are loading a resource, for example, properties file in your application, use the correct loader.

If you call `Class.forName()`, always explicitly pass the loader returned by `Thread.currentThread().getContextClassLoader`.

If you are loading a properties file, use `Thread.currentThread().getContextClassLoader().getResourceAsStream()`.

3.3 Sessions Best Practices

This section describes session best practices. It includes the following topics:

- Persist Session State if Appropriate
- Replicate Sessions if Persisting is Not an Option
- Do Not Store Shared Resources in Sessions

3.3.1 Persist Session State if Appropriate

HTTP Sessions are used to preserve the conversation state with a browser. As such, they hold information, which if lost, could result in a client having to start over the conversation.

Hence, it is always safe to save the session state in database. However, this imposes a performance penalty. If this overhead is acceptable, then persisting sessions is indeed the best approach.

There are trade-offs when implementing state safety that affect performance, scalability, and availability. If you do not implement state-safe applications, then:

- A single JVM process failure will result in many user session failures. For example, work done shopping online, filling in a multiple page form, or editing a shared document will be lost, and the user will have to start over.
- Not having to load and store session data from a database will reduce CPU overhead, thus increasing performance.
- Having session data clogging the JVM heap when the user is inactive reduces the number of concurrent sessions a JVM can support, and thus decreases scalability. In contrast, a state safe application can be written so that session state exists in the JVM heap for active requests only, which is typically 100 times fewer than the number of active sessions.

To improve performance of state safe applications:

- Minimize session state. For example, a security role might map to detailed permissions on thousands of objects. Rather than store all security permissions as session state, just store the role id. Maintain a cache, shared across many sessions, mapping role id to individual permissions.
- Identify key session variables that change often, and store these attributes in a cookie to avoid database updates on most requests.

- Identify key session variables that are read often, and use `HttpSession` as a cache for that session data in order to avoid having to read it from the database on every request. You must manually synchronize the cache, which requires care to handle planned and unplanned transaction rollback.

3.3.2 Replicate Sessions if Persisting is Not an Option

For the category of applications where the HTTP session state information cannot be persisted and retrieved on each HTTP request (due to the performance overhead), OC4J provides an intermediate option - replication.

It can replicate the session state information across an island of servers (which are in the same cluster). This provides a performance improvement because the sessions remain in memory, and fault tolerance - because Oracle HTTP Server automatically routes the HTTP requests to a different server in the island, if the original OC4J (and the session it contains) is down.

Hence, the best practice here is to at least setup two servers in an island, so that they can back session state for each other.

3.3.3 Do Not Store Shared Resources in Sessions

Objects that are stored in the session objects will not be released until the session times out (or is invalidated). If you hold any shared resources that have to be explicitly released to the pool before they can be reused (such as a JDBC connection), then these resources may never be returned to the pool properly and can never be reused.

3.3.4 Set Session Timeout Appropriately

Set session timeout appropriately (`setMaxInactiveInterval()`) so that neither sessions timeout frequently nor does it live for ever this consuming memory.

3.3.5 Monitor Session Memory Usage

Monitor the memory usage for the data you want to store in session objects. Make sure there is sufficient memory for the number of sessions created before the sessions time out.

3.3.6 Always Use Islands, But Keep Island Size Small

Setting up an island of OC4J JVMs causes the sessions to be replicated across all JVMs. This provides better fault tolerance, since a server crash does not necessarily result in a lost session. Oracle9*i*AS automatically re-routes request to another server in the island - thus an end-user never finds out about a failure.

However, this replication overhead increases as more servers are added to the island. For example: if your session object requires 100KB per user, and there are 100 users per server. This results in a 10MB memory requirement for session replication per server. If you have 5 servers in an island, the memory requirement jumps five-fold. Since islands provide session replication, it is, in general, not prudent to exceed an island size beyond 3.

Hence, setting up multiple islands, with few servers in an island is a better choice compared to having a fewer number of larger sized islands.

3.3.7 Use a Mix of Cookie and Sessions

Typically, a cookie is set on the browser (automatically by the container), to track a user session. In some cases, this cookie may last a much longer duration than a single user session. (Example: one time settings, such as to determine the end-user's geographic location).

Thus, a cookie that persists on the client's disk could be used to save information valid for the long-term, while a server side session will typically include information valid for the short-term.

In this situation, the long-term cookie should be parsed on only the first request to the server - when a new session established. The session object created on the server should contain all the relevant information, so as not to require re-parsing the cookie on each request.

A new client side cookie should then be set that contains only an id to identify the server side session object. This is automatically done for any JSP page that uses sessions.

This gives performance benefit since the session object contents do not have to be re-created from the long-term cookie. The other option is of course to save the user settings in a database on the server, and have the user login. The unique userid can then be used to retrieve the contents from the database and store the information in a session.

3.3.8 Use Coarse Objects Inside HTTP Sessions

Oracle9*i*AS automatically replicates sessions when session object is updated. If a session object contains granular objects, for example a person's name), it results in too many update events to all the servers in the island.

Hence, it is recommended to use coarse objects, (for example the person object, as opposed to the name attribute), inside the session.

3.3.9 Use Transient Data in Sessions Whenever Appropriate

Oracle9*i*AS does not replicate transient data in a session across servers in the island. This reduces the replication overhead (and also the memory requirements). Hence, use transient type liberally.

3.3.10 Invalidate Sessions

The number of active users is generally quite small compared to the number of users on the system (ex. of the 100 users on a Web site, only 10 may actually be doing something).

A session is typically established for each user on the system, which costs memory.

Simple things - like a logout button - provide opportunity for quick session invalidation and removal. This avoids memory usage growth since the sessions on the system will be closer to the number of active users, as opposed to all those that have not timed out yet.

3.3.11 Miscellaneous Guidelines

- Use sessions as light-weight mechanism by verifying session creation state.
- Use cookies for long-standing sessions.
- Put recoverable data into sessions so that they can be recovered if the session is lost.
- Store non-recoverable data persistently (in file system or in database using JDBC). However, storing every data persistently is an expensive thing. Instead, one can save data in sessions and use `HttpSessionBindingListener` or other events to flush data into persistent storage during session close.

- Sticky vs Distributable Sessions
    - Distributable session data must be serializable, useful for failover, but are expensive, as the data has to be serialized & replicated among peer processes.
    - Sticky sessions affect load-balancing across multiple JVMs, but are less expensive as there is no state replication.

3.4 EJB Best Practices

This section describes EJB best practices. It includes the following topics:

3.4.1 Local vs. Remote vs. Message Driven EJB

EJBs can be local or remote. If you envision calls to an EJB to originate from the same container as the one running the EJB, local EJBs are better since they do not entail the marshalling, unmarshalling, and network communication overhead. The local beans also allow you to pass an object-by-reference, thus, improving performance further.

Remote EJBs allow clients to be on different machines and/or different application server instances to talk to them. In this case, it is important to use the value object pattern to improve performance by reducing network traffic.

If you choose to write an EJB, write a local EJB over a remote `EJBObject`. Since the only difference is in the exception on the `EJBObject`, almost all of the implementation bean code remains unchanged.

Additionally, if you do not have a need for making synchronous calls, message driven beans are more appropriate.

3.4.2 Decide EJB Use Judiciously

An EJB is a reusable component backed by component architecture with several useful services: persistence, transactions security, naming, etc. However, these additions make it "heavy."

If you just require abstraction of some functionality and are not leveraging the EJB container services, you should consider using a simple JavaBean, or implement the required functionality using JSPs or servlets.

3.4.3 Use Service Locator Pattern

Most J2EE services and/or resources require "acquiring" a handle to them via an initial Java Naming and Directory Interface (JNDI) call. These resources could be an `EJBHomeObject`, or, a JMS topic.

This results in expensive calls to the server machine to resolve the JNDI reference, even though the same client may have gone to the JNDI service for a different thread of execution to fetch the same data!

Hence, it is recommended to have a "Service Locator", which in some sense is a local proxy for the JNDI service, so that the client programs talk to the local service locator, which in turn talks to the real JNDI service, and that only if required.

The Java Object Cache bundled with the product may be used to implement this pattern.

This practice improves availability since the service locator can hide failures of the backend server or JNDI tree by having cached the lookup. Although this is only temporary since the results still have to be fetched.

Performance is also improved since trips to the back-end application server are reduced.

3.4.4 Cluster Your EJBs

OC4J in Oracle9*i*AS Release 2 provides a mechanism to cluster EJBs. Leveraging this mechanism gives significant benefits:

1. **Load Balancing**: The EJB client(s) are load balanced across the servers in the EJB cluster.
2. **Fault Tolerance**: The state (in case of stateful session beans) is replicated across the OC4J processes in the EJB cluster. If the proxy classes on the client cannot connect to an EJB server, they will attempt to connect to the next server in the cluster. The client does not see the failure.
3. **Scalability:** Since multiple EJB servers behaving as one can service many more requests than a single EJB server, a clustered EJB system is more scalable. The alternative is to have stand-alone EJB systems, with manual partitioning of clients across those servers. This is difficult to configure and does not have fault tolerance advantages.

3.4.5 Cluster Servlets and EJB into Identical Islands

Both servlets and EJBs in OC4J support session state replication, through islands.

An island is a group of servers configured for state replication. A servlet (or JSP) island could be different from an EJB island. Thus you could have a group of servers within an EJB island (sometimes also referred to as an EJB cluster), and a group of servers within a JSP/servlet island. However, this gets confusing and the benefits are fewer.

Hence, it is recommended to configure deployments so as to have a servlet island identical to an EJB island.

Note that to leverage EJB clustering fully, you will need to use remote EJBs, which have some performance implications over local EJBs (discussed in earlier best practice). If you use local EJBs and save a reference to them in a servlet (or JSP) session, when the session is replicated this reference is not valid. It is important to be aware of this trade-off.

3.4.6 Index Secondary Finder Methods

When finder methods, other than `findByPrimaryKey` and `findAll`, are created they may be extremely inefficient if appropriate indexes are not created that help to optimize execution of the SQL generated by the container.

3.4.7 Understand EJB Lifecycle

As a developer, it is imperative that you understand the EJB lifecycle. Many problems can be avoided by following the lifecycle and the expected actions during call backs more closely.

This is especially true with entity beans and stateful session beans. An example might be: in a small test environment during testing, a bean may never get passivated, and thus a mis-implementation (or non-implementation) of `ejbPassivate()` and `ejbActivate()` may not show up until later. Moreover, since these are not used for stateless beans, they may confuse new developers.

3.4.8 Use Deferred Database Constraints

For those constraints that may be invalid for a short time during a transaction but will be valid at transaction boundaries, use deferred database constraints. For example, if a column is not populated during an `ejbCreate()`, but will be set prior to the completion of the transaction, then you may want to set the not null constraint for that column to be deferred. This also applies to foreign key constraints that are mirrored by EJB relationships with EJB 2.0.

3.4.9 Create a Cache with Read Only EJBs

For those cases where data changes very slowly or not at all, and the changes are not made by your EJB application, read-only beans may make a very good cache. A good example of this is a country EJB. It is unlikely that it will change very often and it is likely that some degree of stale data is acceptable.

To do this:

1. Create read-only entity beans.
2. Set `exclusive-write-access="true"`.
3. Set the validity timeout to the maximum acceptable staleness of the data.

3.4.10 Pick an Appropriate Locking Strategy

It is critical that an appropriate locking strategy be combined with an appropriate database isolation mode for properly

performing and highly reliable EJB applications.

Use optimistic locking where the likelihood of conflict in updates is low. If a lost update is acceptable or cannot occur because of application design, use an isolation mode of read-committed. If the lost updates are problematic, use an isolation mode of serializable.

Use pessimistic locking where there is a higher probability of update conflicts. Use an isolation mode of read-committed for maximum performance in this case. Use read-only locking when the data will not be modified by the EJB application.

3.4.11 Understand and Leverage Patterns

With the wider industry adoption, there are several common (and generally) acceptable ways of solving problems with EJBs. These have been widely published in either books or discussion forums, etc. In some sense, these patterns are best practices for a particular problem. These should be researched and followed.

Here are some examples:

- Session Façade: Combines multiple entity bean calls into a single call on a session bean, thus reducing the network traffic.
- Message Façade: Use MDBs if you do not need a return status from your method invocation.
- Value Object Pattern: A value object pattern reduces the network traffic by combining multiple data values that are usually required to be together, into a single value object.

A full discussion on the large number of patterns available is outside the scope of this document, but the references section contains some useful books and/or Web sites on this subject.

3.4.12 When Using Entity Beans, Use Container Managed Aged Persistence Whenever Possible

Although there are some limitations to container-managed persistence (CMP), CMP has a number of benefits. One benefit is portability. With CMP, decisions like persistence mapping and locking model selection become a deployment activity rather than a coding activity. This allows deployment of the same application in multiple containers with no change in code. This is commonly not true for Bean Managed Persistence (BMP) since SQL statements and concurrency control must be written into the entity bean and are therefore specific to the container and/or the data store.

Another benefit is that, in general, J2EE container vendors provide quality of service (QoS) features such as locking model variations, lazy loading, and performance and scalability enhancements, which may be controlled via deployment configuration rather than by writing code. Oracle9*i*AS includes features such as read-only entity beans, minimal writing of changes, and lazy loading of relations, which would have to be built into code for BMP.

A third benefit of CMP is container-managed relationships. Through declarations, not unlike CMP field mapping, a CMP entity bean can have relationships between two entity beans managed by the container with no implementation code required from application developers.

Last but least, tools are available to aid in the creation of CMP entity beans so that minimal work is required from developers for persistence. This allows developers to focus on business logic, which allows them to be more efficient. JDeveloper9i is a perfect example where, through modeling tools and wizards, very little work is required to create CMP entity beans including creation of both the generic EJB descriptors and the Oracle9*i*AS specific descriptors.

Overall, there are cases where CMP does not meet the requirements of an application, but the development effort saved, and the optimizations that J2EE containers like OC4J provide make CMP much more attractive than BMP.

3.5 Data Access Best Practices

This section describes data access best practices. It includes the following topics:

- Set the Minimum Open Connections in Data Sources
- Setting the Cache Connection Inactivity Timeout in Data Sources
- Set the Wait for Free Connection Timeout in Data Sources
- Set the Connection Retry Interval in Data Sources
- Set the Maximum Number of Connection Attempts in Data Sources
- Use JDBC Connection Pooling and Connection Caching
- Use JDBC Statement Caching
- Avoid Using More Than One Database Connection Simultaneously in the Same Request
- Tune the Database and SQL Statements

3.5.1 Datasources Connections Caching and Handling

Connections must not be closed within `finalize()` methods. This can cause the connection cache to run out of connections to use, since the connection is not closed until the object that obtained it is garbage collected.

The current connection cache does not provide any mechanism to detect "abandoned" connections, reclaim them, and return them to the cache. All connections must be explicitly closed by the application.

If a connection is declared as static, then it is possible that the same connection object is used on different threads at the same time. Do not declare connections as static objects.

Use the `FIXED_WAIT_SCHEME` when using the connection cache, especially when writing Web applications. This guarantees enforcement of the `MaxLimit` on the connection cache as well as retrieval of a connection from the cache when a connection is returned to the cache.

Always use Connection Cache Timeouts such as `CacheInactivityTimeout` to close unused physical connections in the cache and cause "shrinking" of the cache, thus releasing valuable resources.

Also review the following:

- DataSource Connection Caching Strategies

3.5.1.1 DataSource Connection Caching Strategies

In order to minimize the lock up of resources for long periods of time but allow for recycling of connections from the connection cache, you should use the most appropriate strategy for obtaining and releasing connections as follows:

- Many clients, few connections - Open and close a connection in the same method that needs to use the connection. In order to ensure that connections are returned to the pool, all calls to this method should happen within try-catch, try-finally, or try-catch-finally blocks. This strategy is useful when you have a large number of clients sharing a few connections at the cost of the overhead associated with getting and closing each connection.
- Private client pool - Take advantage of the BMP life cycle. Get a connection within `setEntityContext()` and release the connection in `unsetEntityContext()`. Make connections available to all methods by declaring it a member instance.
- Combined strategy - You may take further advantage of BMP life cycle and implement a strategy which combines the two above.

3.5.2 Datasource Initialization

It is a good practice to put the JNDI lookup of a `DataSource` as part of the application initialization code, since DataSources are simply connection factories.

For example, when using servlets, it is a good idea to put the `DataSource` lookup code into the `init()` method of the servlet.

3.5.3 Disable Auto-Commit Mode for Better Performance

Auto-commit mode indicates to the database whether to issue an automatic commit operation after every SQL operation. Being in auto-commit mode can be expensive in terms of time and processing effort if, for example, you are repeating the same statement with different bind variables.

By default, new connection objects are in auto-commit mode. However, you can disable auto-commit mode with the `setAutoCommit()` method of the connection object (either `java.sql.Conection` or `oracle.jdbc.OracleConnection`).

For better application performance, disable auto-commit mode and use the `commit()` or `rollback()` method of the connection object to manually commit or rollback your transaction.

The following example illustrates how to do this. It assumes you have imported the `oracle.jdbc.*` and `java.sql.*` interfaces and classes.

```
//ds is a DataSource object
Connection conn = ds.getConnection();
// It's faster when auto commit is off
conn.setAutoCommit (false);
// Create a Statement
Statement stmt = conn.createStatement ();
...
```

3.5.4 Disable Escape Processing for Better Performance

Escape processing for SQL92 syntax is enabled by default, which results in the JDBC driver performing escape substitution before sending the SQL code to the database. If you want the driver to use regular Oracle SQL syntax, which is more efficient than SQL92 syntax and escape processing, then disable escape processing using the following statement:

```
stmt.setEscapeProcessing(false);
```

3.5.5 Defining Column Types

Defining column types provides the following benefits:

- Saves a roundtrip to the database server.
- Defines the datatype for every column of the expected result set.
- For `VARCHAR`, `VARCHAR2`, `CHAR` and `CHAR2`, specifies their maximum length.

The following example illustrates the use of this feature. It assumes you have imported the `oracle.jdbc.*` and `java.sql.*` interfaces and classes.

```
//ds is a DataSource object
Connection conn = ds.getConnection();
PreparedStatement pstmt = conn.prepareStatement("select  empno, ename, hiredate from emp");

//Avoid a roundtrip to the database and describe the columns
((OraclePreparedStatement)pstmt).defineColumnType(1,Types.INTEGER);

//Column #2 is a VARCHAR, we need to specify its max length
((OraclePreparedStatement)pstmt).defineColumnType(2,Types.VARCHAR,12);
((OraclePreparedStatement)pstmt).defineColumnType(3,Types.DATE);
ResultSet rset = pstmt.executeQuery();
while (rset.next())
System.out.println(rset.getInt(1)+","+rset.getString(2)+","+rset.getDate(3));
pstmt.close();
...
```

3.5.6 Prefetching Rows Improves Performance

Row prefetching improves performance by reducing the number of round trips to a database server. For most database-centric applications, Oracle recommends the use of row prefetching as much as possible. The recommended prefetch size is 10.

The following example illustrates the use of row prefetching. It assumes you have imported the `oracle.jdbc.*` and `java.sql.*` interfaces and classes.

```
//ds is a DataSource object Connection conn = ds.getConnection();

//Set the default row-prefetch setting for this connection
((OracleConnection)conn).setDefaultRowPrefetch(7);

//The following statement gets the default row-prefetch value for

//the connection, that is, 7 Statement stmt = conn.createStatement();

//Subsequent statements look the same, regardless of the row

//prefetch value. Only execution time changes.
ResultSet rset = stmt.executeQuery("SELECT ename FROM emp");
System.out.println( rset.next () );
while( rset.next () )
System.out.println( rset.getString (1) );

//Override the default row-prefetch setting for this

//statement
( (OracleStatement)stmt ).setRowPrefetch (2);
ResultSet rset = stmt.executeQuery("SELECT ename FROM emp");
System.out.println( rset.next () );
while( rset.next() )
System.out.println( rset.getString (1) );
stmt.close();
...
```

3.5.7 Update Batching Improves Performance

Update Batching sends a batch of operations to the database in one trip. When using it:

- Always disable auto-commit mode with Update Batching.
- Use a batch size of around 10.
- Don't mix the standard and Oracle models of Update Batching.

Also review the following:

- Oracle Update Batching
- Standard Update Batching

3.5.7.1 Oracle Update Batching

The following example illustrates how you use the Oracle Update Batching

feature. It assumes you have imported the `oracle.driver.*` interfaces.

```
//ds is a DataSource object
Connection conn = ds.getConnection();
//Always disable auto-commit when using update batching
conn.setAutoCommit(false);
PreparedStatement ps =
conn.prepareStatement("insert into dept values (?, ?, ?)");
//Change batch size for this statement to 3
((OraclePreparedStatement)ps).setExecuteBatch (3);
//--------#1------------
ps.setInt(1, 23);
ps.setString(2, "Sales");
ps.setString(3, "USA");
ps.executeUpdate(); //JDBC queues this for later execution
//--------#2------------
ps.setInt(1, 24);
ps.setString(2, "Blue Sky");
ps.setString(3, "Montana");
ps.executeUpdate(); //JDBC queues this for later execution
//--------#3------------
ps.setInt(1, 25);
ps.setString(2, "Applications");
ps.setString(3, "India");
ps.executeUpdate(); //The queue size equals the batch value of
3
//JDBC sends the requests to the database
//--------#1------------
ps.setInt(1, 26);
ps.setString(2, "HR");
ps.setString(3, "Mongolia");
ps.executeUpdate(); //JDBC queues this for later execution
((OraclePreparedStatement)ps).sendBatch(); // JDBC sends the
//queued request
conn.commit();
ps.close();
...
```

3.5.7.2 Standard Update Batching

This example uses the standard "Update Batching" feature. It assumes you have imported the `oracle.driver.*` interfaces.

```
//ds is a DataSource object
Connection conn = ds.getConnection();
//Always disable auto-commit when using update batching
conn.setAutoCommit(false);
Statement s = conn.createStatement();
s.addBatch("insert into dept values ('23','Sales','USA')");
s.addBatch("insert into dept values ('24','Blue
Sky','Montana')");
s.addBatch("insert into dept values
('25','Applications','India')");
//Manually execute the batch
s.executeBatch();
s.addBatch("insert into dept values ('26','HR','Mongolia')");
s.executeBatch();
conn.commit();
```

```
ps.close();
...
```

3.5.8 Use Emulated Data Sources for Better Performance

For speed and performance reasons emulated data sources are preferred over non-emulated ones.

A non-emulated datasource provides JDBC v2.0 compliance and additional capabilities such as XA which may not be required for all applications.

3.5.9 Use Emulated and Non-Emulated Data Sources Appropriately

Some of the performance related configuration options have different affects, depending on the type of the data source. OC4J supports two types of data sources, emulated and non-emulated.

The pre-installed default data source is an emulated data source. Emulated data sources are wrappers around Oracle data sources. If you use these data sources, your connections are extremely fast, because they do not provide full XA or JTA global transactional support. Oracle recommends that you use these data sources for local transactions or when your application requires access or update to a single database. You can use emulated data sources for Oracle or non-Oracle databases. You can use the emulated data source to obtain connections to different databases by changing the values of the url and connection-driver parameters.

The following is a definition of an emulated data source:

```
<data-source
class="com.evermind.sql.DriverManagerDataSource"
name="OracleDS"
location="jdbc/OracleCoreDS"
xa-location="jdbc/xa/OracleXADS"
ejb-location="jdbc/OracleDS"
connection-driver="oracle.jdbc.driver.OracleDriver"
username="scott"
password="tiger"
url="jdbc:oracle:thin:@localhost:5521:oracle"
inactivity-timeout="30"
/>
```

Non-emulated data sources are pure Oracle data sources. These are used by applications that want to coordinate access to multiple sessions within the same database or to multiple databases within a global transaction.

3.5.10 Use the EJB-Aware Location Specified in Emulated Data Sources

Each data source is configured with one or more logical names that allow you to identify the data source within J2EE applications. The `ejb-location` is the logical name of an EJB data source. In addition, use the `ejb-location` name to identify data sources for most J2EE applications, where possible, even when not using EJBs. The `ejb-location` only applies to emulated data sources. You can use this option for single phase commit transactions or emulated data sources.

Using the `ejb-location`, the data source manages opening a pool of connections, and manages the pool. Opening a connection to a database is a time-consuming process that can sometimes take longer than the operation of getting the data itself. Connection pooling allows client requests to have faster response times, because the applications do not need to wait for database connections to be created. Instead, the applications can reuse connections that are available in the connection pool.

Oracle recommends that you only use the `ejb-location` JNDI name in emulated data source definitions for retrieving the data source. For non-emulated data sources, you must use the location JNDI name.

3.5.11 Set the Maximum Open Connections in Data Sources

The `max-connections` option specifies the maximum number of open connections for a pooled data source. To improve system performance, the value you specify for the number `max-connections` depends on a combination of factors including the size and configuration of your database server, and the type of SQL operations that your application performs. The default value for `max-connections` and the handling of the maximum depends on the data source type, emulated or non-emulated.

For emulated data sources, there is no default value for `max-connections`, but the database configuration limits that affect the number of connections apply. When the maximum number of connections, as specified with `max-connections`, are all active, new requests must wait for a connection to be become available. The maximum time to wait is specified with wait-timeout.

For non-emulated data sources, there is a property, `cacheScheme`, that determines how max-connections is interpreted. The following lists the values for the `cacheScheme` property (`DYNAMIC_SCHEME` is the default value for `cacheScheme`).

FIXED_WAIT_SCHEME: In this scheme, when the maximum limit is reached, a request for a new connection waits until another client releases a connection.

FIXED_RETURN_NULL_SCHEME: In this scheme, the maximum limit cannot be exceeded. Requests for connections when the maximum has already been reached return null.

For some applications you can improve performance by limiting the number of connections to the database (this causes the system to queue requests in the mid-tier).

For example, for one application that performed a combination of updates and complex parallel queries into the same database table, performance was improved by over 35% by reducing the maximum number of open connections to the database by limiting the value of `max-connections`.

3.5.12 Set the Minimum Open Connections in Data Sources

The `min-connections` option specifies the minimum number of open connections for a pooled data source.

For applications that use a database, performance can improve when the data source manages opening a pool of connections, and manages the pool. This can improve performance because incoming requests don't need to wait for a database connection to be established; they can be given a connection from one of the available connections, and this avoids the cost of closing and then reopening connections.

By default, the value of min-connections is set to 0. When using connection pooling to maintain connections in the pool, specify a value for `min-connections` other than 0.

For emulated and non-emulated data sources, the `min-connections` option is treated differently.

For emulated data sources, when starting up the initial `min-connections` connections, connections are opened as they are needed and once the `min-connections` number of connections is established, this number is maintained.

For non-emulated data sources, after the first access to the data source, OC4J then starts the `min-connections` number of connections and maintains this number of connections.

Limiting the total number of open database connections to a number your database can handle is an important tuning consideration. You should check to make sure that your database is configured to allow at least as large a number of open connections as the total of the values specified for all the data sources `min-connections` options, as specified in all the applications that access the database.

3.5.13 Setting the Cache Connection Inactivity Timeout in Data Sources

The `inactivity-timeout` specifies the time, in seconds, to cache unused connections before closing them.

To improve performance, you can set the `inactivity-timeout` to a value that allows the data source to avoid dropping and then re-acquiring connections while your J2EE application is running.

The default value for the `inactivity-timeout` is 60 seconds, which is typically too low for applications that are frequently accessed, where there may be some inactivity between requests. For most applications, to improve performance, Oracle recommends that you increase the `inactivity-timeout` to 120 seconds.

To determine if the default `inactivity-timeout` is too low, monitor your system. If you see that the number of database connections grows and then shrinks during an idle period, and grows again soon after that, you have two options: you can increase the `inactivity-timeout`, or you can increase the min-connections.

3.5.14 Set the Wait for Free Connection Timeout in Data Sources

The wait-timeout specifies the number of seconds to wait for a free connection if the connection pool does not contain any available connections (that is, the number of connections has reached the limit specified with `max-connections` and they are all currently in use).

If you see connection timeout errors in your application, increasing the `wait-timeout` can prevent the errors. The default wait-timeout is 60 seconds.

If database resources, including memory and CPU are available and the number of open database connections is approaching max-connections, you may have limited `max-connections` too stringently. Try increasing `max-connections` and monitor the impact on performance. If there are not additional machine resources available, increasing `max-connections` is not likely to improve performance.

You have several options in the case of a saturated system:

- Increase the allowable wait-timeout.
- Evaluate the application design for potential performance improvements.
- Increase the system resources available and then adjust these parameters.

3.5.15 Set the Connection Retry Interval in Data Sources

The `connection-retry-interval` specifies the number of seconds to wait before retrying a connection when a connection attempt fails.

If the `connection-retry-interval` is set to a small value, or a large number of connection attempts is specified with `max-connect-attempts` this may degrade performance if there are many retries performed without obtaining a connection.

The default value for the `connection-retry-interval` is 1 second.

3.5.16 Set the Maximum Number of Connection Attempts in Data Sources

The `max-connect-attempts` option specifies the maximum number of times to retry making a connection. This option is useful to control when the network is not stable, or the environment is unstable for any reason that sometimes makes connection attempts fail.

If the `connection-retry-interval` option is set to a small value, or a large number of connection attempts is specified with `max-connect-attempts` this may degrade performance if there are many retries performed without obtaining a connection.

The default value for `max-connect-attempts` is 3.

3.5.17 Use JDBC Connection Pooling and Connection Caching

Constant creation and destruction of resource objects can be very expensive in Java. Oracle suggests using a resources pool to share resources that are expensive to create. The JDBC connections are one of the most common resources used in any Web application that requires database access. They are also very expensive to create. Oracle has observed overhead from hundreds of milliseconds to seconds (depending on the load) in establishing a JDBC connection on a mid-size system with 4 CPUs and 2 GB memory.

In JDBC 2.0, a connection-pooling API allows physical connections to be reused. A pooled connection represents a physical connection, which can be reused by multiple logical connections. When a JDBC client obtains a connection

through a pooled connection, it receives a logical connection. When the client closes the logical connection, the pooled connection does not close the physical connection. It simply frees up resources, clears the state, and closes any statement objects associated with the instance before the instance is given to the next client. The physical connection is released only when the pooled connection object is closed directly.

The term pooling is extremely confusing and misleading in this context. It does not mean there is a pool of connections. There is just one physical connection, which can be serially reused. It is still up to the application designer to manage this pooled connection to make sure it is used by only one client at a time.

To address this management challenge, Oracle's extension to JDBC 2.0 also includes connection caching, which helps manage a set of pooled connections. It allows each connection cache instance to be associated with a number of pooled connections, all of which represent physical connection to the same database and schema. You can use one of Oracle's JDBC connection caching schemes (dynamic, fixed with no wait, or fixed wait) to determine how you want to manage the pooled connections, or you can use the connection caching APIs to implement your own caching mechanisms.

3.5.18 Use JDBC Statement Caching

Use JDBC statement caching to cache a JDBC `PreparedStatement` or `OracleCallableStatement` that is used repeatedly in the application to:

- prevent repeated statement parsing and recreation
- reduce the overhead of repeated cursor creation

The performance gain will depend on the complexity of the statement and how often the statement has to be executed. Since each physical connection has its own statement cache, the advantage of using statement caching with a pool of physical connections may vary. That is, if you execute a statement in a first connection from a pool of physical connections, it will be cached with that connection. If you later get a different physical connection and want to execute the same statement, then the cache does you no good.

See Also:

Oracle JDBC Developer's Guide and Reference

3.5.19 Avoid Using More Than One Database Connection Simultaneously in the Same Request

Using more than one database connection simultaneously in a request can cause a deadlock in the database. This is most common in JSPs. First, a JSP will get a database connection to do some data accessing. But then, before the JSP

commits the transaction and releases the connection, it invokes a bean which gets its own connection for its database operations. If these operations are in conflict, they can result in a deadlock.

Furthermore, you cannot easily roll back any related operations if they are done by two separate database connections in case of failure.

Unless your transaction spans multiple requests or requires some complex distributed transaction support, you should try to use just one connection at a time to process the request.

3.5.20 Tune the Database and SQL Statements

Current Web applications are still very database-centric. From 60% to 90% of the execution time on a Web application can be spent in accessing the database. No amount of tuning on the mid-tier can give significant performance improvement if the database machine is saturated or the SQL statements are inefficient.

Monitor frequently executed SQL statements. Consider alternative SQL syntax, use PL/SQL or bind variables, pre-fetch rows, and cache rowsets from the database to improve your SQL statements and database operations. See Oracle's Server Tuning Guide for more information.

Web applications often access a database at the backend. One must carefully optimize handling of database resources, since a large number of concurrent users and high volumes of data may be involved. Database performance tuning can be divided into two categories:

- Tuning of SQL tables and statements
- Tuning of JDBC calls to access the SQL database

Also refer to the following JDBC tuning topics:

- JDBC Tuning
- JDBC Connection Caching
- JDBC Statement Caching
- JDBC Cached Rowsets

3.5.20.1 JDBC Tuning

JDBC objects such Connections, Statements, and Result Sets are quite often used for database access in Web applications. Frequent creation & destruction of these objects can be quite detrimental to the performance and scalability of the application as these objects are quite heavy-weight. So it is always desirable to cache these JDBC resources.

3.5.20.2 JDBC Connection Caching

- Reuse database connections thus avoiding frequent session creations and tear-downs.
- EJBs, servlets, JSPs can use/share the connection cache within a JVM.
- Create at startup as singleton object so that they can be shared across multiple requests.

3.5.20.3 JDBC Statement Caching

- Avoids cursor creation and teardown
- Avoid cursor parsing
- Two types of statement caching:
    - Implicit: Saves Metadata of cursor but clears the State and Data content of the cursor across calls
    - Explicit: Saves Metadata, Data, and State of the cursor across calls
- Can be used with Pooled Connection and Connection Cache
- For example: `conn.setStmtCacheSize`(<cache-size>)

3.5.20.4 JDBC Cached Rowsets

- Result set implementation that is disconnected, serializable, and scrollable
- Free up connections and cursors faster
- Local scrolling on cached data
- Specially useful for:
    - small read-only data set
    - scrolling for long time

3.6 Java Message Service Best Practices

This section describes Java message service (JMS) best practices. It include the following topics:

3.6.1 Set the Correct time_to_live Value

JMS message expiration is set in the `JMSExpiration` header field. If this value is set to zero (the default), then the message will never expire. If the amount of used table space (memory for OC4J) is a concern, then optimally setting the `time_to_live` parameter will keep messages from accumulating. This is especially true in the publish-subscribe domain where messages may sit forever waiting for the final durable subscriber to return to retrieve the message.

3.6.2 Do Not Grant Execute Privilege of the AQ PL/SQL Package to a User or Role While There Are Outstanding OJMS Session Blocking on a Dequeue Operation

This might cause the granting operation to be blocked and even time-out. Granting calls should be executed before other OJMS operations.

Another way to avoid the blocking or time out is to grant roles instead of granting specific privileges to the user directly. AQ has an `AQ_ADMINISTRATOR_ROLE` that can be used, or users may create their own tailored role. You can then grant the execute privilege of a PL/SQL package to this role. Provided the role was created before hand, the granting of the role to the user does not require a lock on the package. This will allow the granting of the role to be executed concurrently with any other OJMS operation.

3.6.3 Close JMS Resources No Longer Needed

When JMS objects like JMS connections, JMS sessions, and JMS consumers are created, they acquire and hold on to server-side database and client-side resources. If JMS programs do not close JMS objects when they are done using them either during the normal course of operation or at shutdown, then database and client-side resources held by JMS objects are not available for other programs to use. The JVM implementation does not guarantee that finalizers will kick in and clean-up JMS object held resources in a timely fashion when the JMS program terminates.

3.6.4 Reuse JMS Resources Whenever Possible

JMS objects like JMS connections are heavy weight and acquire database resources not unlike JDBC connection pools. Instead of creating separate JMS connections based on coding convenience, it is recommended that a given JMS client program create only one JMS connection against a given database instance for a given connect string and reuse this JMS connection by creating multiple JMS sessions against it to perform concurrent JMS operations.

JMS administrable objects like queues, queue tables, durable subscribers are costly to create and lookup. This is because of the database round trips and in some cases, JDBC connection creation and teardown overhead. It is recommended that JMS clients cache JMS administrable objects once they are created or looked up and reuse them rather than create or look them up each time the JMS client wants to enqueue or dequeue a message. The Oracle9*i*AS Java Object Cache could be used to facilitate this caching.

3.6.5 Use Debug Tracing to Track Down Problems

OJMS allows users to turn debug tracing by setting `oracle.jms.traceLevel` to values between 1 and 5 (1 captures fatal errors only and 5 captures all possible trace information including stack traces and method entries and exits). Debug tracing allows one to track down silent or less understood error conditions.

3.6.6 Understand Handle/Interpret JMS Thrown Exceptions

OJMS is required by the JMS specification to throw particular JMS defined exceptions when certain error/exception conditions occur. In some cases the JMS specification allows or expects OJMS to throw runtime exceptions when certain conditions occur. The JMS client program should be coded to handle these conditions gracefully.

The catch all JMS exception, `JMSException`, that OJMS is allowed to throw in certain error/exception cases provides information as to why the error/exception occurred as a linked exception in the `JMSException`. JMS programs should be coded to obtain and interpret the linked exception in some cases.

For instance, when resources like processes, cursors, or tablespaces run out or when database timeouts/deadlocks occur, SQL exceptions are thrown by the backend database, which are presented to the JMS client program as linked SQL exceptions to the catch all `JMSException` that is thrown. It would be useful for JMS programs to log or interpret the ORA error numbers and strings so that the administrator of the database can take corrective action.

The code segment below illustrates a way to print both the `JMSException` and its linked Exception:

try

```
{...}
catch (JMSException jms_ex)
{
    jms_ex.printStackTrace();
    if (jms_ex.getLinkedException()  != null)
         jms_ex.getLinkedException().printStackTrace();
}
```

3.6.7 Ensure You Can Connect to the Server Machine and Database From the Client Machine

When debugging JMS connection creation problems or problems with receiving asynchronous messages/notifications make sure that you can:

- Ping the database using `tnsping`
- Connect to the database with its connect string using sqlplus
- Are able to resolve the name or the IP address of the server box from the client (by using a simple program that accesses a socket) and vice versa

3.6.8 Tune Your Database Based on Load

OJMS performance is greatly improved by proper database tuning. OJMS performance is dependent on AQ enqueue/dequeue performance. AQ performance will not scale even if you run the database on a box with better physical resources unless the database is tuned to make use of those physical resources.

3.6.9 Make Sure You Tune the OracleOCIConnectionPool

If a JDBC OCI driver is specified when creating a JMS connection, OJMS creates an `OracleOCIConnectionPool` instance from which to obtain the JDBC OCI connections. Depending on the number of JMS session instances that need to be created against the JMS connection and the number of blocking receives that are expected to be performed at a given time against the given JMS connection, the underlying `OracleOCIConnectionPool` instance can be tuned. This is because the `OracleOCIConnectionPool` instance is not self-tuning and is created with a default maximum number of logical connections that can be created from it. Each blocking receive holds onto a logical JDBC OCI connection, and this connection is not available to share. The JMS developer/application, depending on load, can tune the `OracleOCIConnectionPool` instance on-the-fly by obtaining a handle to the `OracleOCIConnectionPool` instance and using its administrative API's.

3.7 Web Services Best Practices

This section describes Web services best practices. It includes the following topics:

- Create Stateless Web Services Instead of Stateful Web Services Whenever Possible
- UDDI Best Practices

3.7.1 Create Stateless Web Services Instead of Stateful Web Services Whenever Possible

For interactive Web sites, one or a combination of three common techniques is typically used to maintain server-side state between page serves: Cookies, URL injection or embedding state (reference) information in hidden form fields. Unfortunately, none of these techniques work reliably with XML Web services. The Web services core technology stack (Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL)) does not define how to maintain state across multiple client invocations. Vendors who support stateful Web services implement this feature using HTTP cookies and Session ID, but using them comes with its own set of challenges.

First, cookies are an optional feature of HTTP and a client can freely choose either not to support them at all or to return them inconsistently (possibly consulting the user for permission). While this may sometimes be desirable for interactive Web sites because of privacy concerns, XML Web services are entirely different by nature because they are a method for coupling systems and not merely a presentation technique. The fact that support for cookies is entirely optional creates a big problem for Web services.

Another problem is that cookies are transport dependent and are valid only for a distinct peer-to-peer connection. Cookies do not work with routing, they do not work with programmatic redirects to Web service replicas (for wide-area load balancing), and they do not work with any transport other than HTTP.

Also, cookie management becomes a major hassle when Web services call other Web services. The cookies returned by any subordinate Web services must be explicitly stored in the main Web service's state, which is, in turn,

referenced by a cookie returned to the client. This is even more difficult to manage when the business code that is called from the main Web service and that calls subordinate Web services itself, isn't aware that it is called from within a Web services environment.

3.7.2 UDDI Best Practices

Since the UDDI specification allows for various types of services to be published (for example, phone, fax, directory services) in addition to Web services, there is no defined relationship between a WSDL document and its representation in the registry. The document "Using WSDL in a UDDI Registry" (http://uddi.org/bestpractices.html) recommends a standard convention for representing WSDL using the UDDI information model. For example, a WSDL document may contain service access details because the UDDI model separates service implementation description from access details; the WSDL document should be modified to remove service access details to produce just an interface WSDL document. Service access information can be stored in a bindingTemplate and the URL for the interface WSDL document can be referenced in the overviewDoc entry in a new wsdlSpec tModel. The bindingTemplate can then reference the new tModel; this allows multiple access points to be registered for the same interface WSDL.

There are also other published UDDI.org technical notes recommended for designing applications and configuring registry taxonomies at:

`http://uddi.org/technotes.html`

The description includes:

- Using WSCL in a UDDI Registry 1.02
- Versioning Taxonomy and Identifier Systems
- Providing a Taxonomy for Use in UDDI Version 2

Review the following UDDI best practices:

- Invocation Patterns
- Taxonomy Development

3.7.2.1 Invocation Patterns

In addition to design-time querying when developing a client application, the UDDI registry can also be used for dynamic, run-time querying of Web services for invocation. This is useful for an application that needs to reliably

deliver service invocation responses, for example, if the service provider has changed access points or decided to refer all requests to an affiliate provider.

This can be accomplished by caching binding information in the client application upon the initial query for the Web service, and then only re-querying if the service invocation fails. The client application needs to cache the bindingTemplate, and retrieve the associated tModel that references the Web service (i.e. `wsdlSpec tModel`); the application will use this information for subsequent service invocations. Upon an invocation failure, the application should then use the `bindingKey` value and `get_bindingTemplate` call to retrieve a current version of the `bindingTemplate`. If this `bindingTemplate` differs from the cached one, replace the cached version with the new one and retry the call; if the call still fails, return an error. If the `bindingTemplate` is identical to the cached version, return an error, as the service provider needs to be contacted to update the access information in the registry.

Another example for run-time UDDI access is finding an optimal access point for a particular Web service, based on geographic-based metadata or uptime for that service. The client could find all access points that implement a particular Web service, and from that list, cull only the ones that are closest in physical proximity, or that have a certain guarantee on uptime. Another example using the same methodology is that a client could also gather all the responses for a particular Web service by cycling through the associated access points.

3.7.2.2 Taxonomy Development

An important consideration in publishing Web services to a UDDI registry is classification; without this, service descriptions cannot be realistically searched for and retrieved, based on some required criteria. The UDDI specification itself does not specify any set taxonomies; it is up to the UDDI registry host to decide which taxonomies to specify. For example, the Universal Business Registry (UBR) includes four major taxonomies by which business entities, services, bindingTemplate, and tModels can be classified; they are the uddi-org types, NAICS, ISO 3166, and UNSPSC taxonomies. Each taxonomy type is defined as a tModel, while the valid categories and their IDs are defined using the specific UDDI registry vendor's available tools.

There are two types of categorizations specified by UDDI: checked and unchecked. It is useful to understand the trade-offs between the two, and which one to implement for a given registry.

A checked categorization indicates that the registry will validate any keyValue associated with that categorization for any publish or inquiry API call. If that value is not part of the categorization, the API message will return an error message saying that the value is invalid. This type of categorization can help reduce the occurrence of garbage data from being published; also, having a checked categorization allows browsers and tools to present these categories to the user. The downsides of checked categorizations are that effort is required to develop the taxonomy, and they can potentially change over time.

An unchecked categorization has no a priori knowledge of valid category values; all inquiry and publishing API calls using this categorization will succeed and not be validated. The advantage of this is that developing and maintaining taxonomies are outside the scope of the registry. Of course, using this type of categorization allows the user to more

likely introduce garbage data into the registry, as well as preventing UDDI registry administration tools from presenting a category hierarchy listing to the user.

Ideally, and by most what can be read about proper XML Web service architecture nowadays, XML Web services are implemented in an entirely stateless manner. This common recommendation is based on the fact that today the term "XML Web Services" is largely equivalent to the combination of XML and the web's core protocol HTTP. HTTP is a fully stateless protocol and the surrounding infrastructure works best of the code that's driven by HTTP is implemented in an HTTP-aware manner.
