

Application Design Guidelines for Storing Session State

Sun™ ONE Application Server

Version 7, Enterprise Edition

817-2162-10
September 2003

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 U.S.A.

Copyright © 2003 Sun Microsystems, Inc. All rights reserved.

THIS SOFTWARE CONTAINS CONFIDENTIAL INFORMATION AND TRADE SECRETS OF SUN MICROSYSTEMS, INC. USE, DISCLOSURE OR REPRODUCTION IS PROHIBITED WITHOUT THE PRIOR EXPRESS WRITTEN PERMISSION OF SUN MICROSYSTEMS, INC. U.S. Government Rights - Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements. Use is subject to license terms.

This distribution may include materials developed by third parties.

Sun, Sun Microsystems, the Sun logo, Java, Sun™ ONE, the Java Coffee Cup logo and the Sun™ ONE logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

This product is covered and controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists is strictly prohibited.

Copyright © 2003 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, Etats-Unis. Tous droits réservés.

CE LOGICIEL CONTIENT DES INFORMATIONS CONFIDENTIELLES ET DES SECRETS COMMERCIAUX DE SUN MICROSYSTEMS, INC. SON UTILISATION, SA DIVULGATION ET SA REPRODUCTION SONT INTERDITES SANS L'AUTORISATION EXPRESSE, ÉCRITE ET PRÉALABLE DE SUN MICROSYSTEMS, INC. Droits du gouvernement américain, utilisateurs gouvernementaux - logiciel commercial. Les utilisateurs gouvernementaux sont soumis au contrat de licence standard de Sun Microsystems, Inc., ainsi qu'aux dispositions en vigueur de la FAR (Federal Acquisition Regulations) et des suppléments à celles-ci. L'utilisation est soumise aux termes de la Licence.

Cette distribution peut comprendre des composants développés par des tierces parties.

Sun, Sun Microsystems, le logo Sun, Java, Sun™ ONE, le logo Java Coffee Cup et le logo Sun™ ONE sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays.

UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

Ce produit est soumis à la législation américaine en matière de contrôle des exportations et peut être soumis à la réglementation en vigueur dans d'autres pays dans le domaine des exportations et importations. Les utilisations, ou utilisateurs finaux, pour des armes nucléaires, des missiles, des armes biologiques et chimiques ou du nucléaire maritime, directement ou indirectement, sont strictement interdites. Les exportations ou réexportations vers les pays sous embargo américain, ou vers des entités figurant sur les listes d'exclusion d'exportation américaines, y compris, mais de manière non exhaustive, la liste de personnes qui font objet d'un ordre de ne pas participer, d'une façon directe ou indirecte, aux exportations des produits ou des services qui sont régis par la législation américaine en matière de contrôle des exportations et la liste de ressortissants spécifiquement désignés, sont rigoureusement interdites.

Contents

About This Guide	5
Who Should Use This Guide	5
How This Guide is Organized	6
Using the Documentation	6
Documentation Conventions	8
General Conventions	9
Conventions Referring to Directories	10
Product Support	10
 Chapter 1 Introduction	 11
 Chapter 2 J2EE References	 15
Problem Scenario for Stateful Session Beans	15
Use of ServiceLocator for Factory Objects	17
General Rule for Service Objects	17
EJB Reference Types	18
Managing Stateless Session Bean References	18
Managing Stateful Session Bean References	19
New Applications	19
Existing Applications	19
Managing Entity Bean References	21
Managing JMS References	22
Other Reference Types	24
NamingContext	25
JDBC Connections	25
UserTransaction References	25
J2EE Connector References	26

File and Network References	26
Impact of Code Changes	26
Appendix A Code Samples	27
WebServiceLocator	27
ServiceLocatorException	34
Index	35

About This Guide

The Sun™ Open Network Environment (Sun ONE) Application Server 7, Enterprise Edition provides strategies for HTTP session availability. However, it does not provide strategies for automatic failover of EJB references, Java Message Service (JMS) references, NamingContext, UserTransaction, and J2EE connector references.

The guidelines discussed in this document are based on a portable Java™ 2 Platform, Enterprise Edition (J2EE) design.

This preface includes the following sections:

- [Who Should Use This Guide](#)
- [How This Guide is Organized](#)
- [Using the Documentation](#)
- [Documentation Conventions](#)
- [Product Support](#)

Who Should Use This Guide

This document outlines recommendations for application architects and developers.

This guide assumes you are familiar with the following:

- Installation of enterprise-level software products
- UNIX® operating system
- Client/server programming model

- Internet and World Wide Web

How This Guide is Organized

This document provides:

- Description and brief background
- Problem Scenario
- Solution or recommendations
- Code snippet that demonstrates the recommendations (wherever applicable)
- Performance benefits of the recommendation (wherever applicable)

In addition to the recommendations discussed in this document, application architects and developers must be aware of, and use Java™ technology best practices.

Using the Documentation

The Sun ONE Application Server 7, Enterprise Edition manuals are available as online files in Portable Document Format (PDF) and Hypertext Markup Language (HTML).

The following table lists tasks and concepts described in the Sun ONE Application Server manuals. The left column lists the tasks and concepts, and the right column lists the corresponding document

Table 1 Sun ONE Application Server Documentation Roadmap

For information about	See the following
Late-breaking information about the software and the documentation.	<i>Release Notes</i>
Comprehensive, table-based summary of supported hardware, operating system, JDK, and JDBC/RDBMS.	<i>Platform Summary</i>
Sun ONE Application Server 7 overview, features available with each product edition.	<i>Product Overview</i>
Diagrams and descriptions of server architecture, benefits of the Sun ONE Application Server architectural approach.	<i>Server Architecture</i>
New enterprise, developer, and operational features of Sun ONE Application Server 7.	<i>What's New</i>

Table 1 Sun ONE Application Server Documentation Roadmap (*Continued*)

For information about	See the following
How to get started with the Sun ONE Application Server 7 product. Includes new features, architectural overview, and sample application tutorial.	<i>Getting Started Guide</i>
Installing the Sun ONE Application Server software and its components, such as sample applications, the Administration interface, and the high-availability components. Instructions for implementing a basic high-availability configuration are included.	<i>Installation Guide</i>
Evaluating your system needs and enterprise to ensure that you deploy Sun ONE Application Server in a manner that best suits your site. General issues and concerns that you must be aware of when deploying an application server are also discussed.	<i>System Deployment Guide</i>
Best practices for HTTP session availability that application architects and developers can use.	<i>Application Design Guidelines for Storing Session State</i>
Creating and implementing Java™ 2 Platform, Enterprise Edition (J2EE™ platform) applications intended to run on the Sun ONE Application Server 7 that follow the open Java standards model for J2EE components such as servlets, Enterprise JavaBeans™ (EJBs™), and JavaServer Pages™ (JSPs™). Includes general information about application design, developer tools, security, assembly, deployment, debugging, and creating lifecycle modules. A comprehensive Sun ONE Application Server glossary is included.	<i>Developer's Guide</i>
Creating and implementing J2EE web applications that follow the Java™ Servlet and JavaServer Pages (JSP) specifications on the Sun ONE Application Server 7. Discusses web application programming concepts and tasks, and provides sample code, implementation tips, and reference material. Topics include results caching, JSP precompilation, session management, security, deployment, SHTML, and CGI.	<i>Developer's Guide to Web Applications</i>
Creating and implementing J2EE applications that follow the open Java standards model for enterprise beans on the Sun ONE Application Server 7. Discusses Enterprise JavaBeans (EJB) programming concepts and tasks, and provides sample code, implementation tips, and reference material. Topics include container-managed persistence, read-only beans, and the XML and DTD files associated with enterprise beans.	<i>Developer's Guide to Enterprise JavaBeans Technology</i>
Creating Application Client Container (ACC) clients that access J2EE applications on the Sun ONE Application Server 7.	<i>Developer's Guide to Clients</i>
Creating web services in the Sun ONE Application Server environment.	<i>Developer's Guide to Web Services</i>
Java™ Database Connectivity (JDBC™), transaction, Java Naming and Directory Interface™ (JNDI), Java™ Message Service (JMS), and JavaMail™ APIs.	<i>Developer's Guide to J2EE Services and APIs</i>
Creating custom NSAPI plug-ins.	<i>Developer's Guide to NSAPI</i>

Table 1 Sun ONE Application Server Documentation Roadmap (*Continued*)

For information about	See the following
Information and instructions on the configuration, management, and deployment of the Sun ONE Application Server subsystems and components, from both the Administration interface and the command-line interface. Topics include cluster management, the high-availability database, load balancing, and session persistence. A comprehensive Sun ONE Application Server glossary is included.	<i>Administrator's Guide</i>
Editing Sun ONE Application Server configuration files, such as the <code>server.xml</code> file.	<i>Administrator's Configuration File Reference</i>
Configuring and administering security for the Sun ONE Application Server operational environment. Includes information on general security, certificates, and SSL/TLS encryption. HTTP server-based security is also addressed.	<i>Administrator's Guide to Security</i>
Configuring and administering service provider implementation for J2EE™ Connector Architecture (CA) connectors for the Sun ONE Application Server 7. Topics include the Administration Tool, Pooling Monitor, deploying a JCA connector, and sample connectors and sample applications.	<i>J2EE CA Service Provider Implementation Administrator's Guide</i>
Migrating your applications to the new Sun ONE Application Server 7 programming model, specifically from iPlanet Application Server 6.x and from Netscape Application Server 4.0. Includes a sample migration.	<i>Migrating and Redploying Server Applications Guide</i>
How and why to tune your Sun ONE Application Server to improve performance.	<i>Performance Tuning Guide</i>
Information on solving Sun ONE Application Server problems.	<i>Troubleshooting Guide</i>
Messages that you may encounter while running Sun ONE Application Server 7. Includes a description of the likely cause and guidelines on how to address the condition that caused the message to be generated.	<i>Error Message Reference</i>
Utility commands available with the Sun ONE Application Server; written in manpage style.	<i>Utility Reference Manual</i>
Using the Sun™ Open Net Environment (Sun ONE) Message Queue software.	The Sun ONE Message Queue documentation at: <code>http://docs.sun.com/db?p=prod/sl.s1msgqu</code>

Documentation Conventions

This section describes the types of conventions used throughout this guide:

- [General Conventions](#)
- [Conventions Referring to Directories](#)

General Conventions

The following general conventions are used in this guide:

- **File and directory paths** are given in UNIX[®] format (with forward slashes separating directory names).

- **URLs** are given in the format:

`http://server.domain/path/file.html`

In these URLs, *server* is the server name where applications are run; *domain* is your Internet domain name; *path* is the server's directory structure; and *file* is an individual filename. Italic items in URLs are placeholders.

- **Font conventions** include:
 - The monospace font is used for sample code and code listings, API and language elements (such as function names and class names), file names, pathnames, directory names, and HTML tags.
 - *Italic* type is used for code variables.
 - *Italic* type is also used for book titles, emphasis, variables and placeholders, and words used in the literal sense.
 - **Bold** type is used as either a paragraph lead-in or to indicate words used in the literal sense.
- **Installation root directories** for most platforms are indicated by *install_dir* in this document. Exceptions are noted in [“Conventions Referring to Directories” on page 10](#).

By default, the location of *install_dir* on **most** platforms is:

`/opt/SUNWappserver7`

For the platforms listed above, *default_config_dir* and *install_config_dir* are identical to *install_dir*. See [“Conventions Referring to Directories” on page 10](#) for exceptions and additional information.

- **Instance root directories** are indicated by *instance_dir* in this document, which is an abbreviation for the following:

`default_config_dir/domains/domain/instance`

Conventions Referring to Directories

By default, when using the Solaris™ 8 and 9 installation, the application server files are spread across several root directories. These directories are described in this section.

- **For Solaris 8 and 9 installations**, this guide uses the following document conventions to correspond to the various default installation directories provided:
 - *install_dir* refers to `/opt/SUNWappserver7`, which contains the static portion of the installation image. All utilities, executables, and libraries that make up the application server reside in this location.
 - *default_config_dir* refers to `/var/opt/SUNWappserver7/domains` which is the default location for any domains that are created.
 - *install_config_dir* refers to `/etc/opt/SUNWappserver7/config`, which contains installation-wide configuration information such as licenses and the master list of administrative domains configured for this installation.

Product Support

If you have general feedback on the product or documentation, please send this to appserver-feedback@sun.com.

If you have problems with your system, contact customer support using one of the following mechanisms:

- The online support web site at:
<http://www.sun.com/supporttraining/>
- The telephone dispatch number associated with your maintenance contract

Please have the following information available prior to contacting support. This helps to ensure that our support staff can best assist you in resolving problems:

- Description of the problem, including the situation where the problem occurs and its impact on your operation
- Machine type, operating system version, and product version, including any patches and other software that might be affecting the problem
- Detailed steps on the methods you have used to reproduce the problem
- Any error logs or core dumps

Introduction

This chapter provides some background information on how HTTP session state is managed by the Sun ONE Application Server.

Sun ONE Application Server uses a high-availability database (HADB), to make the state of HTTP session objects highly available and resilient against failures. All serializable objects stored in `HttpSession` are persisted and made available to other server instances in the event a server instance fails. Since some EJB references and references to other serializable J2EE components such as `InitialContexts` and `DataSources` are serializable, the application developer must understand how these references behave when a failover occurs.

This document provides guidelines for working with these reference types in the Sun ONE Application Server.

NOTE Although reference failover is supported for a particular object type, it is not recommended that the reference be stored in a session. More information on the working of each reference type are provided in the specific reference types section of this document.

The following table lists various J2EE object types and provides information on whether reference failover is supported.

The left column lists the EJB reference types, the second column indicates if Home reference is supported or not, the third column indicates if the Object reference is supported or not, and the right-most column indicates if it is recommended or not.

Table 1-1 EJB Reference Types

EJB Reference Types (Local or Remote¹)	Home Reference Supported	Object Reference Supported	Recommended
Stateless Session Bean	Yes	Yes	No
Stateful Session Bean	Yes	No	No ²
Entity Bean	Yes	Yes	Home - No Object - Yes

1. In the Sun ONE Application Server, remote EJB references are co-located in the same virtual machine as EJB references.

2. Stateful Session bean object reference types would be recommended for HttpSession if stateful session bean reference failover was supported by the Sun ONE Application Server.

The left column lists the JMS reference types, the second column indicates if the JMS reference type is supported or not, the third column indicates if it is recommended or not.

Table 1-2 JMS Reference Types

JMS References	Supported	Recommended
ConnectionFactory	No	No
Connection	No	No
Session	No	No
Destination	No	No
MessageProducer	No	No
MessageConsumer	No	No

The left column lists the J2EE reference types, the second column indicates if the J2EE reference type is supported or not, the third column indicates if it is recommended or not.

Table 1-3 J2EE References Types

Other J2EE Reference Types	Supported	Recommended
InitialContext	Yes	No
UserTransaction	Yes. If the instance that fails is never restarted, any global transactions are lost and may not be correctly rolled back or committed.	No
JDBC	No	No
J2EE Connector	Depends on the Connector	Only if required by the connector.

So of all the reference types listed above, only references to stateful session beans and entity beans should be stored in the session. All the other referenced object types are stateless in nature and must not be held in a user's session from one web request to the next. Since the Sun ONE Application Server does not provide failover support for stateful session beans, if failover is a requirement in your application, the only reference type that should be stored in `HttpSession` is an entity bean reference.

J2EE References

This chapter discusses limitations when failover of a HTTP session state occurs and suggests solutions.

This chapter includes the following topics:

- [Problem Scenario for Stateful Session Beans](#)
- [Use of ServiceLocator for Factory Objects](#)
- [General Rule for Service Objects](#)
- [EJB Reference Types](#)
- [Managing JMS References](#)
- [Other Reference Types](#)
- [Impact of Code Changes](#)

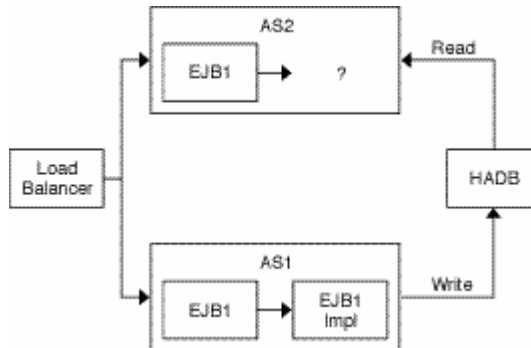
Problem Scenario for Stateful Session Beans

Consider a cluster set up with two Sun ONE Application Server instances, AS1, and AS2. The application server is configured to persist HTTP session state to the session store. An application is deployed to both the instances, and the load balancer is configured to route requests for this application to the two instances. The cluster is servicing requests for this application. A web client starts a new session, SESS1. The load balancer assigns the SESS1 session to AS1.

When the first request arrives from the web client, the application creates a stateful bean and stores a reference to this stateful session bean as an attribute in the `HttpSession` object, so that it can be accessed on ensuing requests. At the end of the request, AS1 stores all session states for SESS1 into the session store, including these references.

Subsequently, AS1 becomes unavailable due to some unforeseen failure. The load balancer detects that AS1 is unavailable, and begins routing all requests for SESS1 to AS2. The reference fails to work in the new container because the referenced stateful session bean running in the AS1 EJB container is no longer available in the new AS2 execution environment for SESS1.

Figure 2-1 Problem Scenario



In the absence of such automatic mechanisms to handle proper reconstitution of references (of the type discussed earlier), it is possible to handle the problem in the application tier.

These guidelines do *not* cover more catastrophic failure scenarios such as the EIS backend becoming unreachable or the unavailability of the JMS provider, and so on. It is assumed that the services that these references are directed to, are available even after any failures in the application server tier.

The pattern discussed in this document illustrates how, with minimal modifications to an application, stateful session bean references that need to be reconstructed after the failover of a session from one server instance to another can be located and managed appropriately.

Note that there is minimum risk involved in implementing and using the solution presented in this document. If the solution is applied incorrectly, exceptions are transmitted to the client just as if the solution was not implemented at all.

NOTE Future versions may automatically reconstitute stateful session bean references correctly, and the guidelines proposed in this document will still be usable. They will also be in full working condition, although it will no longer be necessary in its full scope.

Use of ServiceLocator for Factory Objects

The example code in this document uses the `ServiceLocator` pattern for accessing all factory objects (such as `EJBHome` objects). This pattern hides some of the complexity involved with looking up factory objects but its use is completely optional to the solution.

A brief summary of the `ServiceLocator` pattern is provided in the following section. For more information visit the following web site.

[Sun Java Center J2EE Patterns](#)

The `ServiceLocator` provides access to all the factory objects. Since these factory objects can be safely shared by multiple user sessions, the `ServiceLocator` may be implemented as a singleton (one `ServiceLocator` per classloader).

The `ServiceLocator` design pattern centralizes distributed service object lookups and point of control, and may act as a cache that eliminates redundant lookups. This pattern is used to cache all the factory objects, such as `Home` objects for enterprise beans.

The `ServiceLocator` class maintains a table of pairs (JNDI Names, and Home References). The class is used in the following manner:

```
ServiceLocator sl = ServiceLocator.getInstance(); // get Singleton
EJBLocalHome home = sl.getLocalHome("java:comp/env/ejb/Account");
```

When `sl.getLocalHome()` is called, the `ServiceLocator` checks the table to see if the corresponding `Home` reference already exists. If it does not, it creates it by using the supplied Java Naming and Directory Service (JNDI) name, and saves it in the table for subsequent use.

For more information on the implementation of `ServiceLocator`, see [Appendix A, "Code Samples."](#)

General Rule for Service Objects

Most application programmers take advantage of `HttpSession` to avoid performing repeated lookup and create calls. Since most of these references support concurrent use, the recommended practice is to store common references in the static storage of a front controller servlet.

For more information on using a front controller servlet, visit the following web site.

[Sun Java Center J2EE Patterns](#)

In the absence of a controller servlet the reference should be stored in a class member variable, or if the request spans multiple servlets, as an attribute of the `HttpServletRequest`. Specific examples are provided in relevant sections below.

EJB Reference Types

This section includes the following topics:

- [Managing Stateless Session Bean References](#)
- [Managing Stateful Session Bean References](#)
- [Managing Entity Bean References](#)

Managing Stateless Session Bean References

Stateless session beans do not hold any state specific to the current client session, and hence they can be safely discarded and recreated any time. The references to these enterprise beans should not be stored directly in the `HttpSession` but instead should be stored in a class member variable as shown in the following code example:

```
HelloLocal hello;

public void jspInit() {
    try {
        InitialContext ic = new InitialContext();
        Object obj = ic.lookup("java:comp/env/ejb/HelloLocal");
        HelloLocalHome home = (HelloLocalHome) obj;
        hello = home.create();
        ...
    }
}
```

Managing Stateful Session Bean References

A stateful session bean's state must be persisted if you expect the bean to survive a session failover. However, Sun ONE Application Server does not support stateful session bean failover in this release. This section discusses issues and suggests alternatives for handling stateful session bean recovery.

New Applications

For new application development, avoid the use of stateful session beans. Simple Java objects or entity beans are recommended. The Java object solution is explained in [“Recovering from Failover” on page 20](#).

Existing Applications

Existing applications which use stateful session beans that are being ported to the Sun ONE Application Server will most likely need some level of code modifications. In non-failover situations, your application functions properly without code modifications.

Failover Behavior

Stateful session beans in the Sun ONE Application Server cannot survive a session failover. When a session failover occurs, your application may be prone to unexpected behavior or, null pointer exceptions. When an attempt is made to retrieve a stateful session bean from the session on its new server instance, a null is returned. In the worst case scenario, if you are assuming your reference to be valid in subsequent requests after it is created, null pointer exceptions are returned in a session failover situation. You have several options to address this issue, depending on your application requirements.

Detecting Failover

At the most basic level, you need to test for null after each fetch of a stateful session bean reference from the session.

```
if (session.getAttribute(COUNT_OBJECT_NAME) == null)
{
    /* A failover has occurred. Determine how you want to proceed.
    This is application dependent. Your options include recreating
    the bean, notifying the user that a problem has occurred and they
    need to start over, or both.*/
}
```

Recovering from Failover

If your application requires a more robust solution, it is recommended that you convert your stateful session beans to serializable Java objects and manage those objects in `HttpSession`. This is a straightforward exercise. As an example, start with a simple stateful session bean, `Count`. The following code is a bean class code:

```
public class CountBean public class CountBean
    implements javax.ejb.SessionBean {
    private javax.ejb.SessionContext context;

    // The current counter is our conversational state.
    public int val;
    public int count() {
        return ++val;
    }
    public void ejbCreate(int val) {
        this.val = val;
    }
}
```

To convert this class to a serializable Java object, simply place the `ejbCreate` method code into the Java object's constructor. Business methods such as `count()` can be copied, as- is, in the Java object. The results in this case appears as follows:

```
public class Count implements java.io.Serializable {
    // The current counter is our conversational state.
    public int val;

    /** Creates a new instance of Count */
    public Count(int val) {
        this.val = val;
    }
    public int count() {
        return ++val;
    }
}
```

Create the `Count` object from the web client as follows:

```
count = new Count(iInitialValue);
session.setAttribute(COUNT_OBJECT_NAME, count);
```

On subsequent requests, you can comfortably retrieve the count object and use it as shown below:

```
Count count = (Count) session.getAttribute(COUNT_OBJECT_NAME);
int countVal = count.count();
```

However, there is an issue with this approach if you rely on the session synchronization interface inside your stateful session bean. `HttpSession` instances do not have a mechanism that allows them to receive transaction notifications. This means that any data that is modified in an `HttpSession` during a transaction is not reverted if the current transaction is rolled back.

Managing Entity Bean References

The entity bean references are correctly reconstructed when a session is recreated in a new container. For this reason, there is no change required to be made and the entity bean code works as-is.

Although entity bean references do not carry the state associated with a user's session, it may be desirable to work with an entity bean spanning multiple web requests. Instead of finding the bean at the start of every web request, the reference to the bean object can be stored in the session for quick and easy access for future requests. Since the entity bean's home object is stateless and supports concurrent use, it should be stored in a class member variable rather than the session.

```
ProductLocalHome sharedHome;

public void jspInit() {
    try {
        InitialContext ic = new InitialContext();
        sharedHome = (ProductLocalHome)
            ic.lookup("java:comp/env/ejb/MyProduct");
        ...
    }
}

// Now in the body of the JSP or Servlet:
Product product = sharedHome.findByPrimaryKey(productId);
```

```

session.setAttribute("productBean", product);

// Then on future requests, simply:

Product product = session.getAttribute("productBean");

```

Managing JMS References

Java Messaging Service (JMS) provides two interfaces for administered objects: `JMS ConnectionFactory` and `JMS Destination`. Both these interfaces encapsulate JMS provider specific information configured at the container level. The `JMS ConnectionFactory` interface is used by a JMS client to create an authenticated, physical JMS Connection to the JMS provider's server. The `JMS Destination` interface is used by a JMS client to identify physical destinations (of either `Topic` or `Queue` domain) used by the JMS message service. The destinations, connection factories and the connections issued from the `ConnectionFactory` all support concurrent use and therefore may be used by simultaneous request processing threads in the web container. JMS client code in a web application ultimately access these administered objects through standard JNDI lookup code. For these reasons, the recommended practice is to use the `ServiceLocator` pattern. For more information see, [Appendix A, "Code Samples."](#)

Another technique for using JMS administered references in a web application is to store common references in the static storage of a controller servlet.

For more information on using a front controller servlet, visit the following web site.

[Sun Java Center J2EE Patterns](#)

Using the `javax.servlet.Servlet.init(javax.servlet.ServletConfig)` method as a predictable singleton across a web application's life cycle, you may acquire the required JMS administered objects using `ServiceLocator` pattern and cache these in the static storage of the servlet. Because `JMS ConnectionFactory` and `Destination` may be used concurrently, multiple concurrent request threads in the web application may use these static references directly.

The application may reliably use the static references regardless of the web container instance in the Sun ONE Application Server cluster that a request is processed on.

It may be possible to set a `JMS ConnectionFactory` or `Destination` reference as a `HttpSession` attribute, but there is no value added in doing so.

The only other JMS interface which may be used concurrently is a JMS Connection. Connections are issued by a `JMSConnectionFactory`. Although the JMS Connections may be used concurrently (shared by multiple concurrent requests), a JMS session created by the `Connection` does not support concurrent usage. The JMS sessions and the related interfaces that the session manages (including JMS messages, `MessageConsumers`, `MessageProducers` and transactions) are normally scoped to a single request in the web container. Individual requests of a web application normally use a short-lived JMS session lasting no longer than the scope of the request.

In most cases a web application that needs the functionality of sending a JMS message may just issue a common JMS Connection for all requests to share. With the thread and input/output resources required of a JMS Connection, and the ability of the JMS Connection to be shared naturally by concurrent requests, using a shared connection approach works well. For each logical JMS Connection that the application requires, a new connection is created from the correct `ConnectionFactory`. A proven approach for implementing this is to issue the shared connection during `javax.servlet.Servlet.init(javax.servlet.ServletConfig)` and then store the shared connection in public static storage of the controller Servlet. See the following code sample.

```
public class SampleServlet extends javax.servlet.http.HttpServlet
{
    public SampleServlet() {
        super();
    }

    private static javax.jms.QueueConnectionFactory
    queueConnFactory;

    private static javax.jms.Queue queue;

    private static javax.jms.QueueConnection sharedQueueConn;

    public void init(javax.servlet.ServletConfig config)
        throws javax.servlet.ServletException {
        super.init(config);
        try {
            WebServiceLocator sl = WebServiceLocator.getInstance();
            queueConnFactory = sl.getQueueConnectionFactory(
```

```

        "java:comp/env/jms/FooQueueConnectionFactory");
    queue = sl.getQueue("java:comp/env/jms/FooQueue");
    sharedQueueConn =
        queueConnFactory.createQueueConnection();
}
catch(ServiceLocatorException sle) {
    throw new javax.servlet.ServletException(
        "Failure bootstrapping common references",sle);
}
catch(javax.jms.JMSException jmse) {
    throw new javax.servlet.ServletException(
        "Failure issuing common JMS connection",jmse);
}
}
}

```

NOTE JMS `MessageListeners` are not capable of being `HttpSession` attributes (see JMS 4.4.6). This is due to concurrency limitations and requirements on the JMS session; the intent for J2EE containers is to use Message Driven Beans (MDBs) for asynchronous message consumption.

Other Reference Types

This section includes the following topics:

- [NamingContext](#)
- [JDBC Connections](#)
- [UserTransaction References](#)
- [J2EE Connector References](#)
- [File and Network References](#)

NamingContext

A `NamingContext` object should never be stored in the `HttpSession`. It should always be created by calling a new `InitialContext()`.

JDBC Connections

In general, the JDBC connections should not be held beyond a single request. The recommended programming practice is to open and close the connection per Web request, or per method call for enterprise bean methods.

This is important because the application server manages the connections through a connection pool. Since the number of JDBC connections is a precious resource managed by the container, saving the connection for later use in the session precludes the possibility of its reuse by some other session.

Sometimes, this is done to iterate over a large result, set over multiple web requests. However, this is bad programming practice. Instead, convert the result set to a `CachedRowSet` and save it in `HttpSession` for subsequent use. While doing this consider setting the limits on the size of `CachedRowSet` so that it does not become too large.

NOTE	It is generally a bad programming practice to store the connection in the <code>HttpSession</code> object since it interferes with the connection pool managed by the application server.
-------------	---

Good practice requires pulling a connection from the connection pool for each request, to maximize the scalability of the connection pool.

UserTransaction References

A `UserTransaction` object should *not* be stored in the `HttpSession`.

According to the J2EE specifications, it is illegal to start a transaction in one Web request and commit (or roll-back) it in a subsequent Web request. Hence, the `UserTransaction` object should not be stored in the `HttpSession` and reused in a subsequent Web request to implement a long-running transaction. A `UserTransaction` object does not hold any session specific state. All the state

corresponding to the currently active transaction is stored in the thread that is servicing the request. The application developers should consider using the `getUserTransaction` method provided by the `ServiceLocator`. The `ServiceLocator` object should never be stored in the session.

J2EE Connector References

J2EE Connector references are stateless references and should not be stored in `HttpSession`. However, they do support concurrent use and can be stored in a class member variable. The approach to be used here is identical to that used for stateless session beans.

File and Network References

If you are storing open files and network connections in HTTP sessions, then you need to declare these references as transient. This is something you need to do regardless of whether you are working with a distributed container; because even in a container that is not distributed these references are lost during activation/passivation since they are not serializable.

Prior to each use, test the reference. If the reference is null, you can assume that a passivation or failover has occurred and you must now re-establish the reference.

Impact of Code Changes

Future versions of Sun ONE Application Server may provide more complete support for failover of J2EE applications. The code changes recommended in this document are not specific to Sun ONE Application Server. You should be able to run the code without modifying the code in future Sun ONE Application Server versions.

Code Samples

This appendix provides code examples for the following:

- [WebServiceLocator](#)
- [ServiceLocatorException](#)

WebServiceLocator

```
package com.sun.j2ee.blueprints.web;

import java.util.*;
import java.net.*;
import javax.ejb.*;
import javax.jms.*;
import javax.naming.*;
import javax.rmi.*;
import javax.sql.*;
import javax.transaction.*;

/**
 * This class is an implementation of the Service Locator pattern. It is
 * used to looukup resources such as EJBHomes, JMS Destinations, etc.
 * This implementation uses the "singleton" strategy and also the "caching"
 * strategy.
```

```

* This implementation is intended to be used on the web tier and
* not on the ejb tier.
* This class should never be serialized or saved in HttpSession.
*/

```

```

public class WebServiceLocator {
    private InitialContext ic;

    //used to hold references to EJBHomes/JMS Resources for re-use
    private Map cache = Collections.synchronizedMap(new HashMap());

    private static WebServiceLocator instance = new WebServiceLocator();

    public static WebServiceLocator getInstance() {
        return instance;
    }

    private WebServiceLocator() throws ServiceLocatorException {
        try {
            ic = new InitialContext();
        } catch (Exception e) {
            throw new ServiceLocatorException(e);
        }
    }

    /**
    * will get the ejb Local home factory. If this ejb home factory has already
    * been clients need to cast to the type of EJBHome they desire
    *
    * @return the EJB Home corresponding to the homeName
    */
}

```

```

public EJBLocalHome getLocalHome(String jndiHomeName) throws
ServiceLocatorException {
    EJBLocalHome home = (EJBLocalHome) cache.get(jndiHomeName);
    if (home == null) {
        try {
            home = (EJBLocalHome) ic.lookup(jndiHomeName);
            cache.put(jndiHomeName, home);
        } catch (Exception e) {
            throw new ServiceLocatorException(e);
        }
    }
    return home;
}

/**
 * will get the ejb Remote home factory. If this ejb home factory has already
 * been clients need to cast to the type of EJBHome they desire
 *
 * @return the EJB Home corresponding to the homeName
 */
public EJBHome getRemoteHome(String jndiHomeName, Class className) throws
ServiceLocatorException {
    EJBHome home = (EJBHome) cache.get(jndiHomeName);
    if (home == null) {
        try {
            Object objref = ic.lookup(jndiHomeName);
            Object obj = PortableRemoteObject.narrow(objref, className);
            home = (EJBHome)obj;
            cache.put(jndiHomeName, home);
        } catch (Exception e) {
            throw new ServiceLocatorException(e);
        }
    }
}

```

```

        }
    }
    return home;
}

/**
 * @return the factory for the factory to get queue connections from
 */
public QueueConnectionFactory getQueueConnectionFactory(String
qConnFactoryName)
throws ServiceLocatorException {
    QueueConnectionFactory factory = (QueueConnectionFactory)
cache.get(qConnFactoryName);
    if (factory == null) {
        try {
            factory = (QueueConnectionFactory) ic.lookup(qConnFactoryName);
            cache.put(qConnFactoryName, factory);
        } catch (Exception e) {
            throw new ServiceLocatorException(e);
        }
    }
    return factory;
}

/**
 * @return the Queue Destination to send messages to
 */
public Queue getQueue(String queueName) throws ServiceLocatorException {
    Queue queue = (Queue) cache.get(queueName);
    if (queue == null) {
        try {

```

```

        queue =(Queue)ic.lookup(queueName);
        cache.put(queueName, queue);
    } catch (Exception e) {
        throw new ServiceLocatorException(e);
    }
}

return queue;
}

/**
 * This method helps in obtaining the topic factory
 * @return the factory for the factory to get topic connections from
 */
public TopicConnectionFactory getTopicConnectionFactory(String
topicConnFactoryName) throws ServiceLocatorException {
    TopicConnectionFactory factory = (TopicConnectionFactory)
        cache.get(topicConnFactoryName);
    if (factory == null) {
        try {
            factory = (TopicConnectionFactory)
                ic.lookup(topicConnFactoryName);
            cache.put(topicConnFactoryName, factory);
        } catch (Exception e) {
            throw new ServiceLocatorException(e);
        }
    }
    return factory;
}

/**
 * This method obtains the topic itself for a caller
 * @return the Topic Destination to send messages to
 */
public Topic getTopic(String topicName) throws ServiceLocatorException {

```

```

    Topic topic = (Topic) cache.get(topicName);
    if (topic == null) {
        try {
            topic = (Topic)ic.lookup(topicName);
            cache.put(topicName, topic);
        } catch (Exception e) {
            throw new ServiceLocatorException(e);
        }
    }
    return topic;
}

/**
 * This method obtains the datasource itself for a caller
 * @return the DataSource corresponding to the name parameter
 */
public DataSource getDataSource(String dataSourceName) throws
ServiceLocatorException {
    DataSource dataSource = (DataSource) cache.get(dataSourceName);
    if (dataSource == null) {
        try {
            dataSource = (DataSource)ic.lookup(dataSourceName);
            cache.put(dataSourceName, dataSource );
        } catch (Exception e) {
            throw new ServiceLocatorException(e);
        }
    }
    return dataSource;
}

/**
 * This method obtains the UserTransaction itself for a caller
 * @return the UserTransaction corresponding to the name parameter

```



```

*/

public UserTransaction getUserTransaction(String utName) throws
ServiceLocatorException {
    try {
        return (UserTransaction) ic.lookup(utName);
    } catch (Exception e) {
        throw new ServiceLocatorException(e);
    }
}

/**
 * @return the URL value corresponding
 * to the env entry name.
 */

public URL getUrl(String envName) throws ServiceLocatorException {
    try {
        return (URL)ic.lookup(envName);
    } catch (Exception e) {
        throw new ServiceLocatorException(e);
    }
}

/**
 * @return the boolean value corresponding
 * to the env entry such as SEND_CONFIRMATION_MAIL property.
 */

public boolean getBoolean(String envName) throws ServiceLocatorException {
    try {
        return ((Boolean)ic.lookup(envName)).booleanValue();
    } catch (Exception e) {
        throw new ServiceLocatorException(e);
    }
}

```

```

/**
 * @return the String value corresponding
 * to the env entry name.
 */
public String getString(String envName) throws ServiceLocatorException {
    try {
        return (String)ic.lookup(envName);
    } catch (Exception e) {
        throw new ServiceLocatorException(e);
    }
}
}

```

ServiceLocatorException

```
package com.sun.j2ee.blueprints.web;
```

```

/**
 * This class implements an exception which can wrapped a lower-level exception.
 */
public class ServiceLocatorException extends RuntimeException {
    public ServiceLocatorException() {}
    public ServiceLocatorException(String msg) { super(msg); }
    public ServiceLocatorException(String msg, Throwable cause) {
        super(msg, cause); // will not work with JDK 1.3
    }
    public ServiceLocatorException(Throwable cause) {
        super(cause); // will not work with JDK 1.3
    }
}

```

Index

A

- application
 - robust solution [20](#)
- asynchronous message consumption [24](#)

C

- CachedRowSet [25](#)
- class member variable [18](#)
- code changes [26](#)
- common references
 - static storage [17](#)
- connection pool [25](#)
 - scalability [25](#)
- container level [22](#)
- controller servlet [18](#)

D

- document
 - organization [6](#)

E

- enterprise bean methods [25](#)

- entity bean references [21](#)
- entity beans [19](#)

F

- factory objects
 - multiple user sessions [17](#)
 - ServiceLocator [17](#)
- failures
 - EIS backend [16](#)
 - unavailability of JMS provider [16](#)
- front controller servlet [17](#)
- future versions
 - automatically reconstitute [16](#)

H

- HTTP session
 - background information [11](#)
 - high-availability database [11](#)
 - problem scenario [15](#)
 - solution [15](#)

I

- implementation

risk [16](#)
InitialContext() [25](#)

J

J2EE containers [24](#)
J2EE specifications [25](#)
Java object's constructor [20](#)
Java objects [19](#)
JDBC connections [25](#)
JMS administered references [22](#)
 controller servlet [22](#)
JMS client [22](#)
JMS connection [23](#)
JMS ConnectionFactory [22](#)
JMS Destination [22](#)
JMS interface [23](#)
JMS message service [22](#)
JMS references [22](#)

L

limitations
 HTTP session [15](#)

M

multiple concurrent requests [23](#)

N

NamingContext [25](#)
null pointer exceptions [19](#)

P

pre-requisite [5](#)

S

serializable Java objects [20](#)
ServiceLocator
 object lookups [17](#)
 table of pairs [17](#)
session failover [19](#)
singleton [17](#)
stateful session bean [19](#)
stateful session bean recovery [19](#)
stateful session beans [19](#)
stateless session beans [18](#)
static references [22](#)
static storage [17](#)
Sun customer support [10](#)

T

table of pairs
 home references [17](#)
 JNDI names [17](#)

U

UserTransaction object [25](#)