

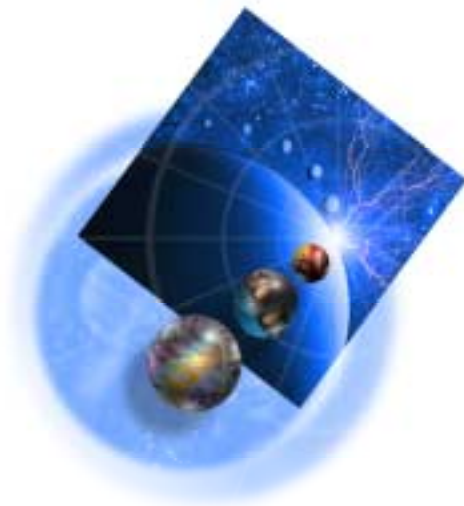
White paper

IBM
WebSphere



IBM WebSphere Application Server Standard and Advanced Editions

*WebSphere Application Server
Development Best Practices for
Performance and Scalability*



by Harvey W. Gunther

Document Revision Control: 1.1.0

Date: September 7, 2000

**WebSphere Application Server
White Paper
Best Practices for Developing High Performance Web and Enterprise Applications**

Intended Audience

This paper is intended for anyone responsible for the development and deployment of high performance and scalable IBM® WebSphere® Application Server applications, including application architects, developers, and their managers.

Acknowledgements

The following people from WebSphere Performance provided great support and guidance: Gennaro (Jerry) Cuomo, Ruth Willenborg, Carolyn Norton, Michael Fraenkel, Stan Cox, Carmine Greco, Brian Martin, Ron Bostick, Chris Forte, and Charlie Bradley. I am also grateful for the assistance that I received from Tricia York from Information Development for Web Solutions, Tom Alcott from WebSphere Sales Technical Support, Keys Botzum, Kyle Brown and Mike Curtin from AIM Services, Scott Snyder from WebSphere Development and David Williams from VisualBuilder.

Thanks also to Dan Ellentuck, Don Fracapane, and Maria Mosca from Columbia University for verifying that this was a worthwhile endeavor.

**WebSphere Application Server
White Paper
Best Practices for Developing High Performance Web and Enterprise Applications**

WebSphere Application Server Best Practices – Overview

This white paper describes development best practices for both Web applications containing servlets, JavaServer™ (JSP™) files, and JDBC connections, and enterprise applications containing EJB components. The table below lists the Best Practices by category assigning each practice a relative importance rating from 1 to 5. The ratings are based on degree of performance impact and on frequency of occurrence.

Category	Best Practice Number and Description	Importance
Servlets	1. Do not store large object graphs in HttpSession	2
Servlets	2. Release HttpSession when finished	3
JSP Files	3. Do not create HttpSession in JSPs by default	4
Servlets	4. Minimize synchronization in Servlets	2
Servlets	5. Do not use SingleThreadModel	5
All web and enterprise application components	6. Use JDBC connection pooling	3
All web and enterprise application components	7. Reuse datasources for JDBC connections	1
All web and enterprise application components	8. Release JDBC resources when done	3
Servlets	9. Use the HttpServlet Init method to perform expensive operations that need only be done once	4
All web and enterprise application components	10. Minimize use of System.out.println	2
All web and enterprise application components	11. Avoid String concatenation "+="	1
Enterprise beans	12. Access entity beans from session beans	3
Enterprise beans	13. Reuse EJB homes	1
Enterprise beans	14. Use "Read-Only" methods where appropriate	3

**WebSphere Application Server
White Paper
Best Practices for Developing High Performance Web and Enterprise Applications**

Enterprise beans	15. Reduce the transaction isolation level where appropriate	5
Enterprise beans	16. EJBs and Servlets - same JVM - "No local copies"	2
Enterprise beans	17. Remove stateful session beans when finished	2
Servlets	18. Don't use Beans.instantiate() to create new bean instances	3

To ease navigation through this document you can hyperlink directly to a specific "Best Practice" from the table below. You can then hyperlink back to this page through tag, [\[TOP\]](#), next to the Best Practice.

For each Best Practice, this document provides:

1. A description and brief background
2. **A code snippet for code to be avoided (as applicable)**
3. **A code snippet demonstrating the Best Practice¹ (as applicable).**
4. Performance comparison to illustrate the benefit of the Best Practice² (as applicable).

In addition to these Best Practices for high performance scalable web and enterprise applications, developers should also be aware of and use good Java™ language performance constructs. See the [Bibliography – Additional References](#) section of this paper for recommended for Java™ language performance reference texts.

¹ Although the code samples work and have been tested, they are designed solely to illustrate the applicable best practice. They are not designed for use in actual applications, which are by definition more complicated.

² The examples used in this paper are designed to illustrate the impact of the best practice. The examples have no other application functionality to dilute such an impact. It is unlikely that the changes discussed here will result in the same absolute performance impacts in actual applications, which are inherently more complex.

[\[TOP\]](#)

Best Practice 1 Do not store large object graphs in HttpSession

Large applications require using persistent HttpSession. However, there is a cost. An HttpSession must be read by the servlet whenever it is used and rewritten whenever it is updated. This involves serializing the data and reading it from and writing it to a database. In most applications, each servlet requires only a fraction of the total session data. However, by storing the data in the HttpSession as one large object graph, an application forces WebSphere Application Server to process the entire HttpSession object each time.

WebSphere Application Server has HttpSession configuration options that can optimize the performance impact of using persistent HttpSession. The HttpSession configuration options are discussed in detail in the WebSphere Application Server documentation. Also consider alternatives to storing the entire servlet state data object graph in the HttpSession. Using your own JDBC connection alternative is described and discussed below.

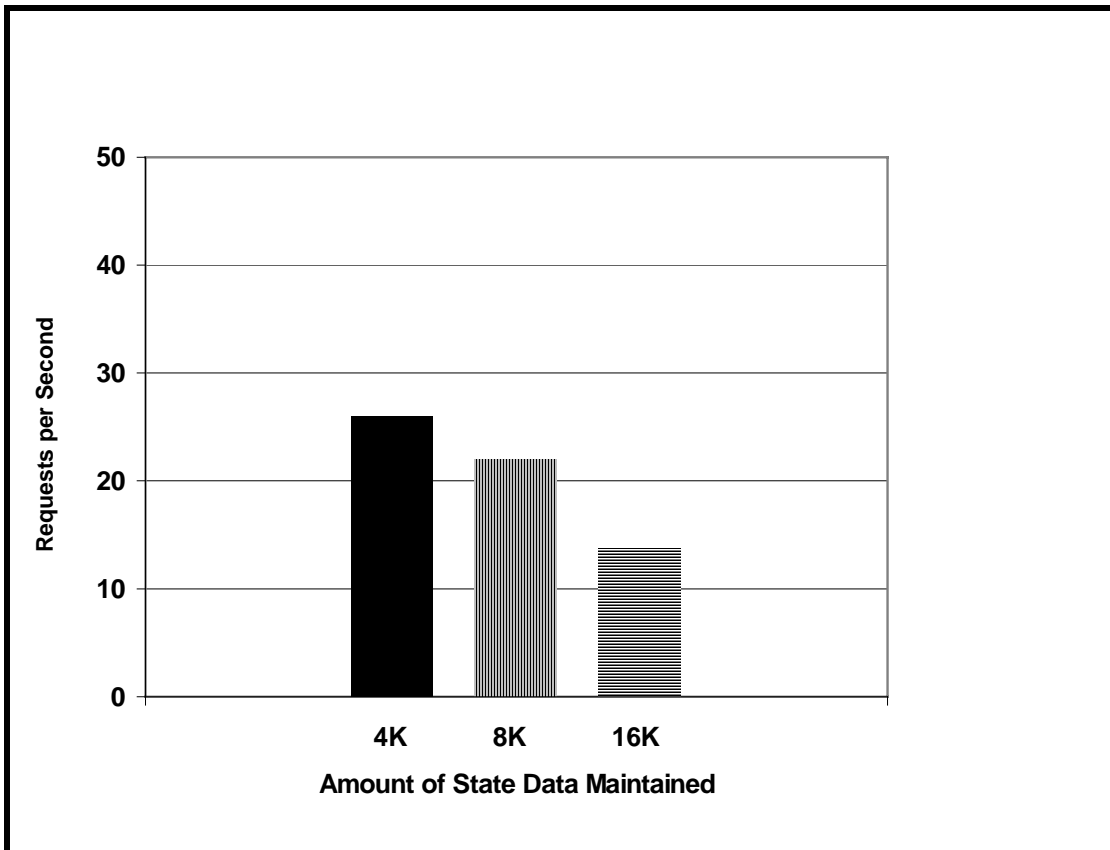
Please don't forget to explicitly invalidate the HttpSession when you are finished with it. See Best Practice 2 [Release HttpSession when finished](#) for more detail.

When configuring persistent sessions in WebSphere Application Server, use a dedicated data source. To avoid contention for JDBC connections, don't reuse an application DataSource or the WebSphere Application Server repository for persistent session data.

Figure 1a compares relative performance of a sample application with a single object of different sizes. As the size of the objects stored in the HttpSession increases, throughput decreases, in large part due to the serialization cost.

**WebSphere Application Server
White Paper
Best Practices for Developing High Performance Web and Enterprise Applications**

Figure 1a - Performance Impact - HTTP Session Single Object



JDBC alternative to Using HttpSession Data for Storing Servlet State Data

In most applications, each servlet needs only a fraction of the entire application's state data. As an alternative to storing the entire object graph in the HttpSession, use JDBC for partitioning and maintaining the state data needed by each servlet in the application.

A sample JDBC solution maintains the state data needed by each servlet as a separate row in an application-maintained JDBC datasource. The primary keys for each row (the data for each servlet) are stored as separate attributes in the HttpSession. The use of the HttpSession is reduced to the few strings needed to locate the data.

Figure 1b shows the code for this alternative.

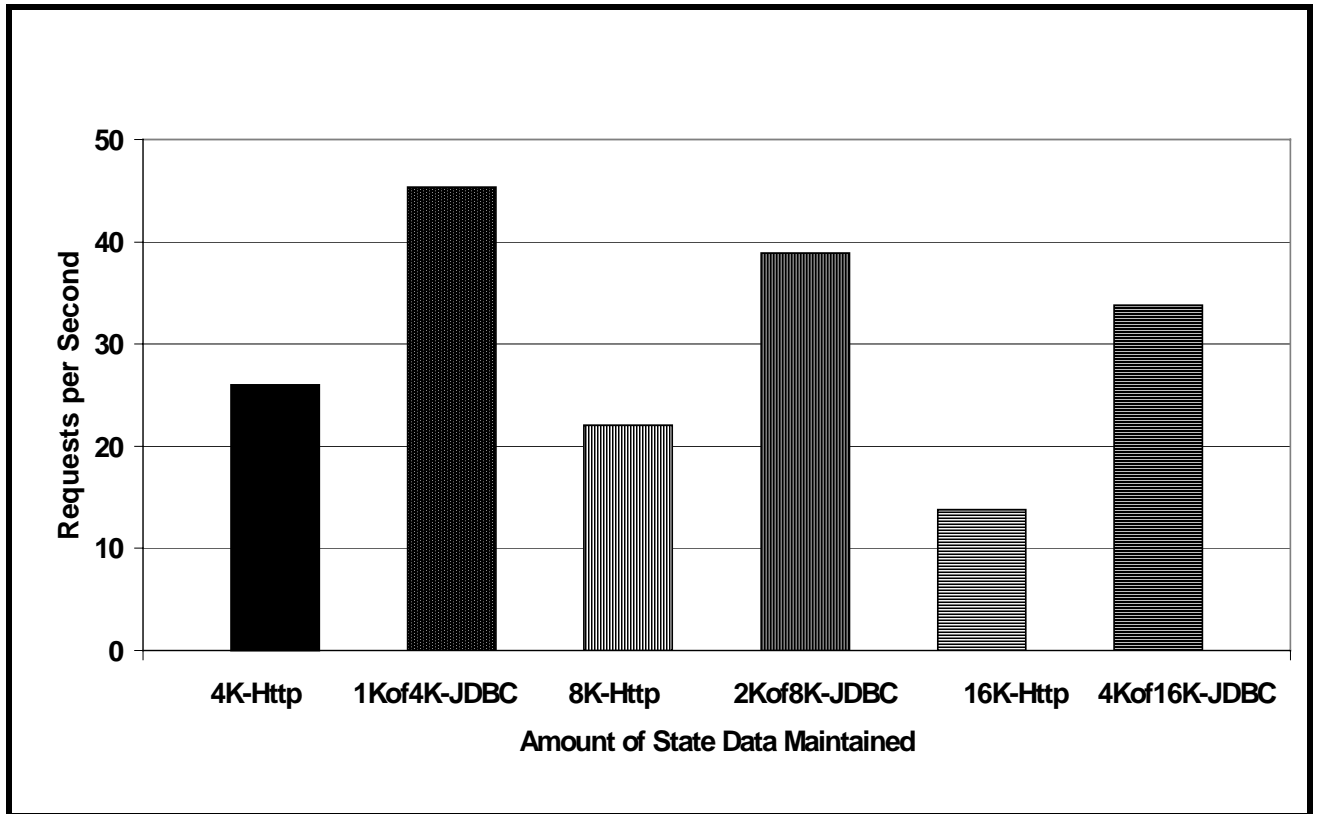
Figure 1b - JDBC Session Data Alternative

```
// Get the session data value
try {
    conn = getPooledConnection(); // Uses DataSources to Get JDBC Connection See Best Practice
    conn.setAutoCommit(false);
    sessionKey = (String) session.getValue(TOP_KEY);
    // Session Key Does Not Exist - This is a new Session Build the Data
    if(sessionKey == null){
        sessionKey = UniqueValue.getUniqueValue();
        session.putValue(TOP_KEY,sessionKey);
        sessionData = new SessionLargeObjectCollection(NUM_REPS);
        psi = conn.prepareStatement(INSERTSTMT);
        psi.setString(1,sessionKey);
        psi.setBytes(2,convertToBytes(sessionData)); // Serialize and Write Out
        psi.executeUpdate();
        sessionKey = UniqueValue.getUniqueValue();
        session.putValue(Integer.toString(i+1),sessionKey);
    }
    else { // The Session Does Exist Retrieve the Data to Generate Load
        sessionKey = (String)session.getValue(TOP_KEY);
        pss = conn.prepareStatement(SELECTSTMT);
        pss.setString(1,sessionKey);
        rs = pss.executeQuery();
        if(rs.next()){ // De-serialize and Read In
            sessionData = convertBytesToSessionObjectCollections(rs.getBytes("SAVESERIALIZEDDATA"));
        }
        // Update the Data to Generate Load
        psu = conn.prepareStatement(UPDATESTMT);
        psu.setString(2,sessionKey);
        psu.setBytes(1,convertToBytes(sessionData)); // Serialize and Write Out
        psu.executeUpdate();
    }
    conn.commit();
}
catch (Exception e){ // Handle Errors}
finally
{ // Close Connections and Statements - See Best Practice
}
```

**WebSphere Application Server
White Paper
Best Practices for Developing High Performance Web and Enterprise Applications**

Figure 1c shows the performance improvement of partitioning the servlet state data and using JDBC over maintaining servlet state data as a single object in the HttpSession.

Figure 1c - Performance Impact – Using JDBC to store servlet state data instead of HttpSession



JDBC Alternative – Cleanup Issues

The homegrown servlet state data framework illustrated in the example alternative creates records in a user database. At some point, these records need to be deleted through a cleanup process. Here are two good alternatives for this purpose:

- **Periodic Offline: Non-WebSphere Process.** On UNIX® systems, this would be a CRON job. On Windows® NT® or Windows 2000, it would be a scheduled function. In either case, it would delete records after a certain period of time depending on the application. In this example, the database is keyed on servlet data time of creation.

**WebSphere Application Server
White Paper
Best Practices for Developing High Performance Web and Enterprise Applications**

- HttpSessionBindingListener. This homegrown alternative stores the servlet state data in an application - specific database. It stores the primary keys for the servlet state data records in the HttpSession. When the HttpSession is about to be destroyed, the javax.servlet.http.HttpSessionBindingEvent for valueUnbound could direct a delete of the servlet state data. If the lifetime of servlet state data is longer than that of the HttpSession, use the first alternative instead.

[\[TOP\]](#)

Best Practice 2 Release HttpSessions when finished

HttpSession objects live inside the WebSphere servlet engine until:

- The application explicitly and programmatically releases it using the API, **javax.servlet.http.HttpSession.invalidate ()**; quite often, programmatic invalidation is part of an application logout function. Figure 2a provides an example.
- WebSphere Application Server destroys the allocated HttpSession when it expires (by default, after 1800 seconds or 30 minutes). WebSphere Application Server can only maintain a certain number of HttpSessions in memory. When this limit is reached, WebSphere Application Server serializes and swaps the allocated HttpSession to disk. In a high volume system, the cost of serializing many abandoned HttpSessions can be quite high.

Figure 2a - Explicit HttpSession Invalidation

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ApplicationLogoutServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        HttpSession mySession = request.getSession(false);
        if(mySession != null)
        {
            // Invalidate the Session Here !!!!!
            mySession.invalidate();
            // Invalidate the Session Here !!!!!
        }
        //-----
        //
        // Some other Application Logoff Processing and Output Reply Back
        // to Browser
        //-----
    }
}
```

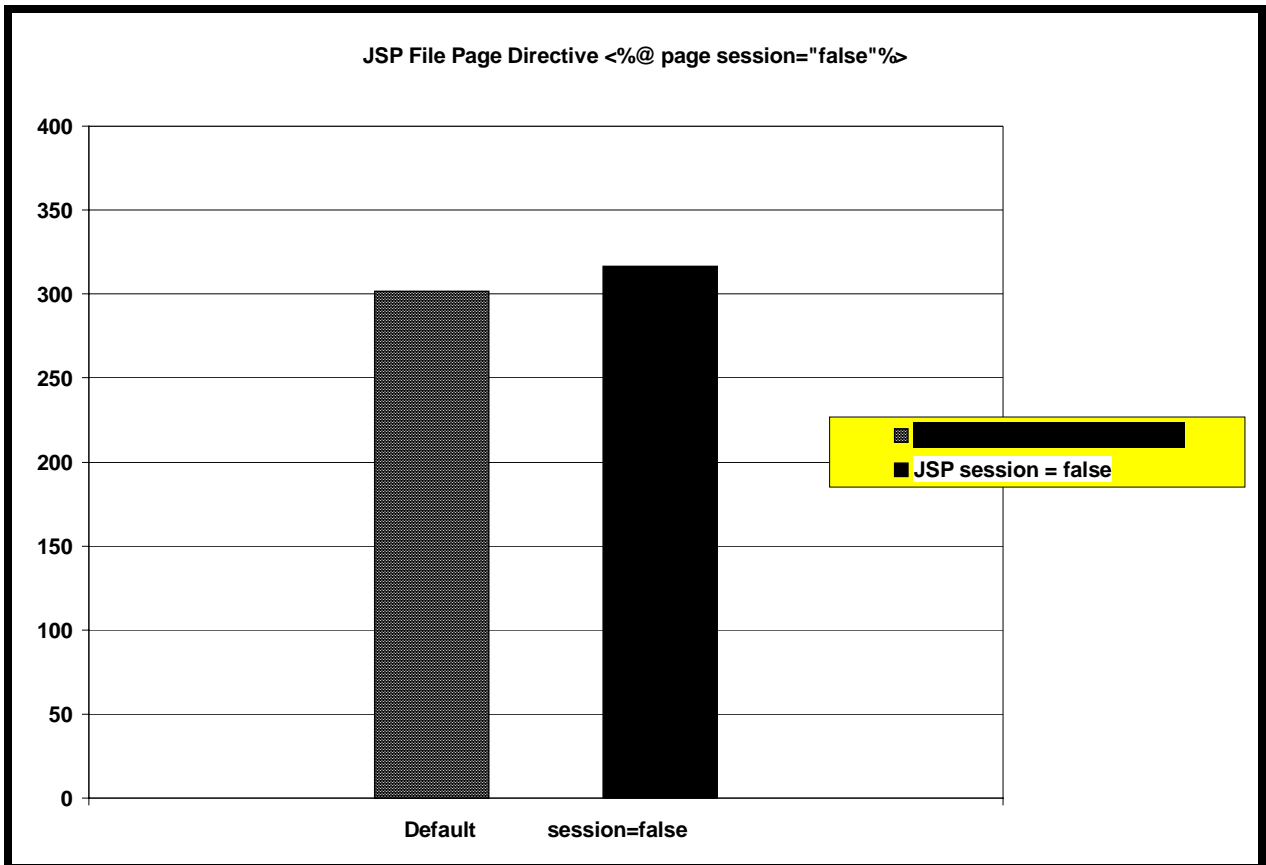
[\[TOP\]](#)

Best Practice 3 Do not create HttpSession in JSPs by default

By default, JSP files create HttpSession. This is in compliance with J2EE™ to facilitate the use of JSP implicit objects, which can be referenced in JSP source and tags without explicit declaration. HttpSession is one of those objects. If you do not use HttpSession in your JSP files then you can save some performance overhead with the following JSP page directive:

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

Figure 3 - Performance Impact - Avoiding JSP File HttpSession By Default



[\[TOP\]](#)

Best Practice 4 Minimize synchronization in Servlets

Servlets are multi-threaded. Servlet-based applications have to recognize and handle this. However, if large sections of code are synchronized, an application effectively becomes single threaded, and throughput decreases.

The code in Figure 4a synchronizes the major code path of the servlet's processing to protect a servlet instance variable, "numberOfRows." The code in Figure 4b moves the lock to a servlet instance variable and out of the critical code path. See Figure 4c for the performance comparison.

Using `javax.servlet.SingleThreadModel`, is yet another way to protect updateable servlet instance variables, but should be avoided. See Best Practice 5 [Don't Use SingleThreadModel](#) for more detail.

Figure 4a - Locking the Major Code Path: Excessive Synchronization

```
public class BpAllBadThingsServletsU1a extends HttpServlet
{
    private int numberOfRows = 0;
    private javax.sql.DataSource ds = null;

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        Connection conn = null;
        ResultSet rs = null;
        PreparedStatement pstmt = null;
        int startingRows;

        try
        {
            synchronized(this) // Locks out Most of the Servlet Processing
            {
                startingRows = numberOfRows;
                String employeeInformation = null;
                conn = ds.getConnection("db2admin","db2admin");
                pstmt = conn.prepareStatement("select * from db2admin.employee");
                rs = pstmt.executeQuery();
            }
        }
    }
}
```

Figure 4b - Better Approach - Don't Lock The Major Code Path

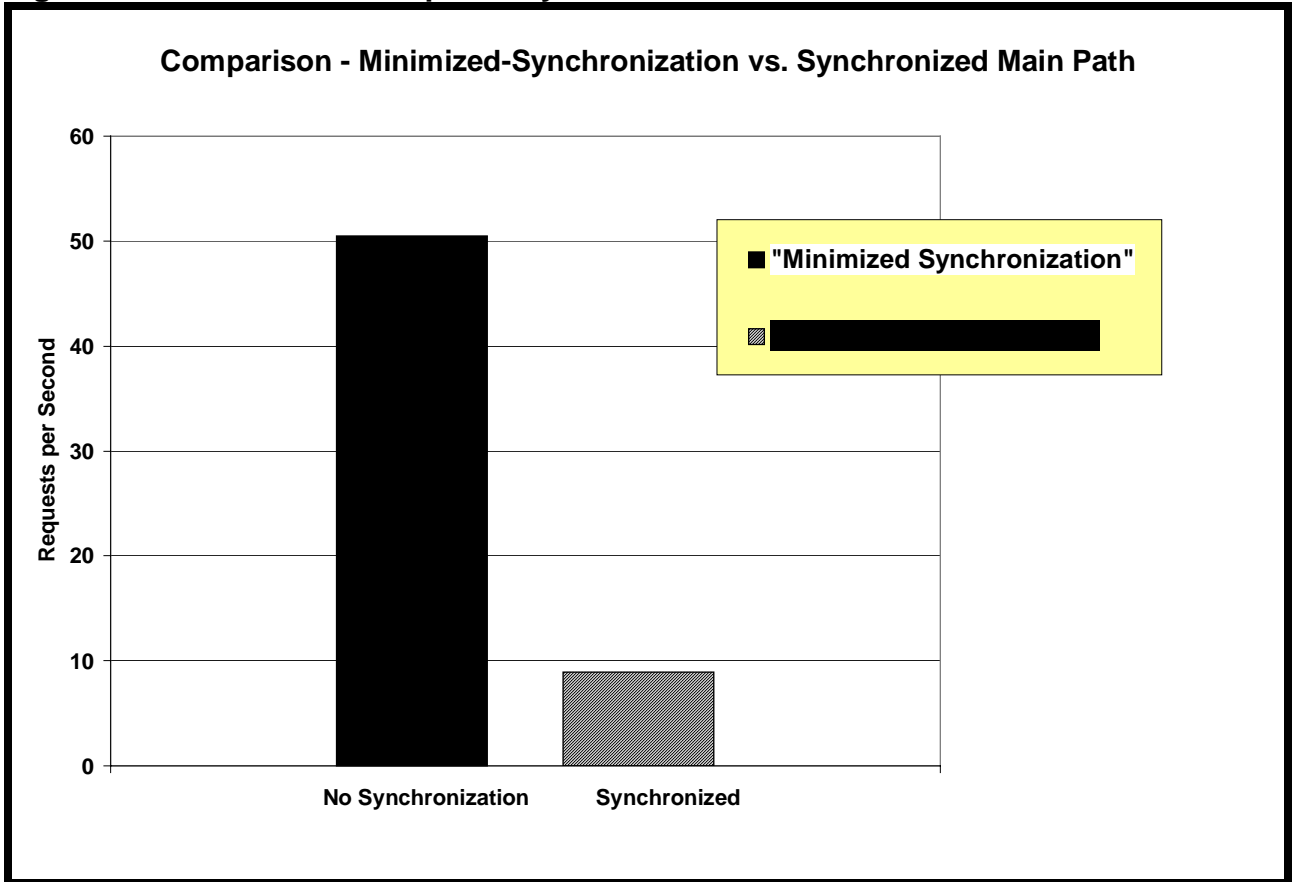
```
public class BpAllBadThingsServletsV1b extends HttpServlet
{
    private int numberOfRows = 0;
    private javax.sql.DataSource ds = null;

    private Object lockObject = new Object();

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        Connection conn = null;
        ResultSet rs = null;
        PreparedStatement pStmt = null;
        int startingRows = 0;

        // Lock Here if we Must - Much Less Impact
        synchronize(lockObject)
        {
            startingRows = numberOfRows;
        }
        try
        {
            String employeeInformation = null;
            conn = ds.getConnection("db2admin","db2admin");
            pStmt = conn.prepareStatement("select * from db2admin.employee");
            rs = pStmt.executeQuery();
        }
    }
}
```

Figure 4c – Performance Impact - Synchronization



[\[TOP\]](#)

Best Practice 5 Do not use SingleThreadModel

SingleThreadModel is a tag interface that a servlet can implement to transfer its re-entrancy problem to the servlet engine. As such, `javax.servlet.SingleThreadModel` is part of the J2EE specification. The WebSphere servlet engine handles the servlet's re-entrancy problem by creating separate servlet instances for each user. Because this causes a great amount of system overhead, `SingleThreadModel` should be avoided.

Developers typically use `javax.servlet.SingleThreadModel` to protect updateable servlet instances in a multithreaded environment. The better approach is to avoid using servlet instance variables that are updated from the servlet's service method.

Figure 5a – AVOID THIS!!! - `javax.servlet.SingleThreadModel`

```
public class BpAllBadThingsServletsV1c extends HttpServlet implements SingleThreadModel
{
    private int numberOfRows = 0;
    private javax.sql.DataSource ds = null;

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        Connection conn = null;
        ResultSet rs = null;
        PreparedStatement pStmt = null;
        int startingRows = numberOfRows;

        try
        {
            String employeeInformation = null;
            conn = ds.getConnection("db2admin","db2admin");
            pStmt = conn.prepareStatement("select * from db2admin.employee");
            rs = pStmt.executeQuery();
        }
    }
}
```

[\[TOP\]](#)

Best Practice 6 Use JDBC connection pooling

To avoid the overhead of acquiring and closing JDBC connections, WebSphere Application Server provides JDBC connection pooling based on JDBC 2.0. Servlets should use WebSphere Application Server JDBC connection pooling instead of acquiring these connections directly from the JDBC driver. WebSphere Application Server JDBC connection pooling involves the use of `javax.sql.DataSources`. Refer to Best Practice 7 [Reuse DataSources for JDBC Connections](#) for the correct handling of JDBC `javax.sql.DataSources`.

Figure 6a shows the wrong way to obtain JDBC connections. Figure 6b shows the correct way to obtain them, and Figure 6c shows the contrast in throughput.

Figure 6a – The Wrong Way to Obtain JDBC Connections

```
public class BpAllBadThingsServletV0a extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        Connection conn = null;
        ResultSet rs = null;
        PreparedStatement pStmt = null;

        try
        {
            // THIS IS THE WRONG WAY TO DO THIS!!!!!!
            conn = DriverManager.getConnection("jdbc:db2:SAMPLE","db2admin","db2admin");

            String employeeInformation = null;

            pStmt = conn.prepareStatement("select * from db2admin.employee");
            rs = pStmt.executeQuery();
        }
    }
}
```

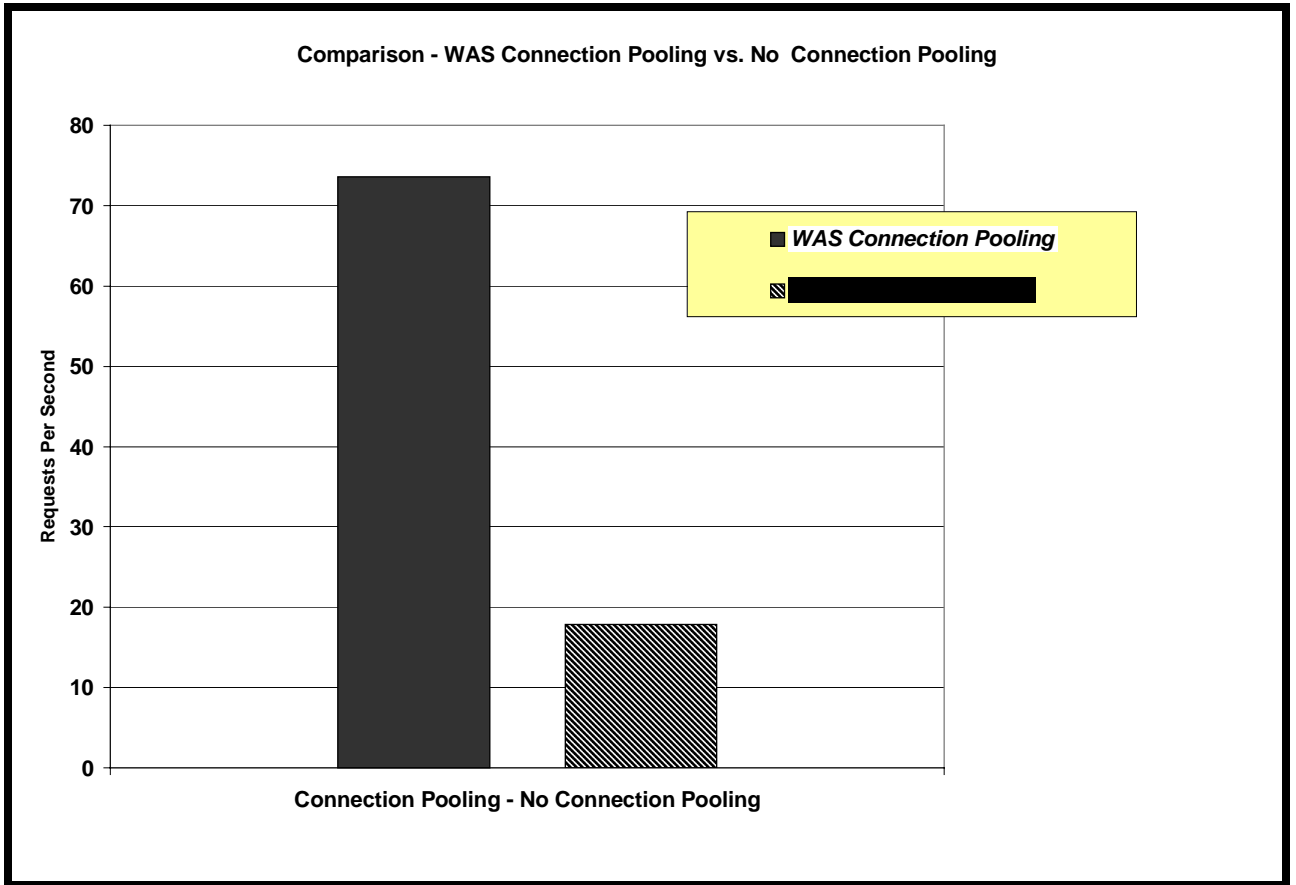
Figure 6b - The Right Way to Obtain JDBC Connections

```
public class BpAllBadThingsServletsV5 extends HttpServlet
{
    // Caching the DataSource - It is obtained in the Servlet.init() method
    private javax.sql.DataSource ds = null;

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        Connection conn = null;
        ResultSet rs = null;
        PreparedStatement pStmt = null;

        try
        {
            String employeeInformation = null;
            conn = ds.getConnection("db2admin","db2admin");
            pStmt = conn.prepareStatement("select * from db2admin.employee");
            rs = pStmt.executeQuery();
        }
    }
}
```

Figure 6c - Performance Impact – Using Connection Pooling



[\[TOP\]](#)

Best Practice 7 Reuse datasources for JDBC connections

JDBC Connection Pooling was discussed in Best Practice 6, [Use JDBC connection pooling](#). In WebSphere Application Server, as defined in JDBC 2.0, servlets acquire JDBC connections from a `javax.sql.DataSource` defined for the database.

A `javax.sql.DataSource` is obtained from WebSphere Application Server through a JNDI naming lookup. Avoid the overhead of acquiring a `javax.sql.DataSource` for each SQL access. This is an expensive operation that will severely impact the performance and scalability of the application. Instead, servlets should acquire the `javax.sql.DataSource` in the `Servlet.init()` method (or some other thread-safe method) and maintain it in a common location for reuse.

Figure 7a shows the incorrect way to obtain a `javax.sql.DataSource`". Figure 7b shows the correct way to obtain a `javax.sql.DataSource`. Figure 7c shows the contrast in performance between the two.

Figure 7a - The Wrong Way to Acquire a Data Source

```
public class BpAllBadThingsServletsU2a extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        Connection conn = null;
        ResultSet rs = null;
        PreparedStatement pstmt = null;
        javax.sql.DataSource ds = null;

        try
        {
            java.util.Hashtable env = new java.util.Hashtable();
            env.put(Context.INITIAL_CONTEXT_FACTORY, "com.ibm.ejs.ns.jndi.CNInitialContextFactory");

            ctx = new InitialContext(env);
            ds = (DataSource)ctx.lookup("jdbc/SAMPLE");
            ctx.close();

            conn = ds.getConnection("db2admin", "db2admin");

            pstmt = conn.prepareStatement("select * from hgunther.employee");
            rs = pstmt.executeQuery();
        }
    }
}
```

Figure 7b - The Correct Way to Obtain and Reuse a javax.sql.DataSource

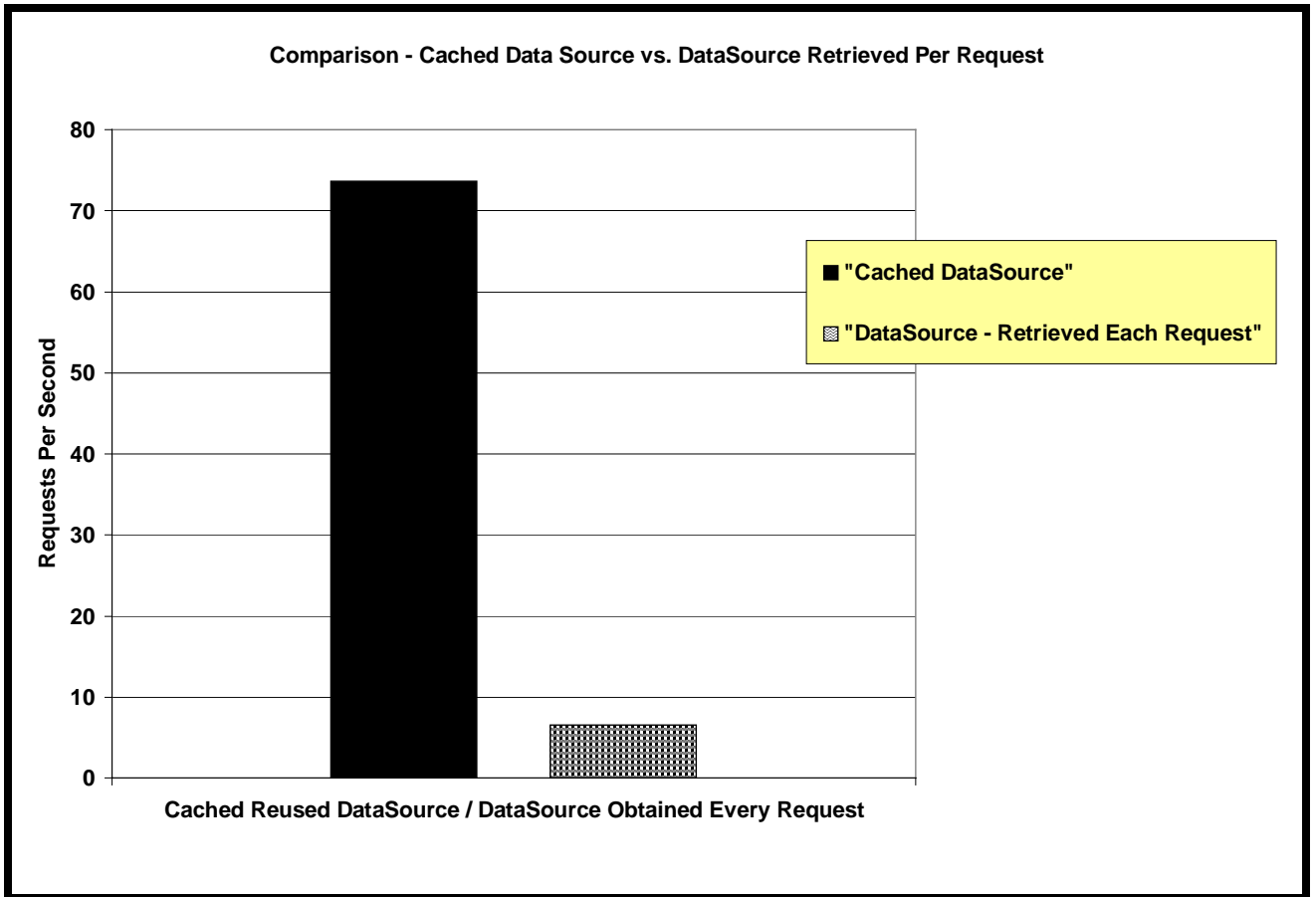
```
public class BpAllBadThingsServletsV5 extends HttpServlet
{
    // Caching the DataSource - It is obtained in the Servlet.init() method
    private javax.sql.DataSource ds = null;

    // This Happens Once and is Reused
    public void init(ServletConfig config) throws ServletException
    {
        super.init(config);
        Context ctx = null;
        try
        {
            java.util.Hashtable env = new java.util.Hashtable();
            env.put(Context.INITIAL_CONTEXT_FACTORY, "com.ibm.ejs.ns.jndi.CNInitialContextFactory");

            ctx = new InitialContext(env);
            ds = (javax.sql.DataSource)ctx.lookup("jdbc/SAMPLE");
            ctx.close();
        }
        catch(Exception es)
        {
            es.printStackTrace();
        }
    }
}
```

WebSphere Application Server
White Paper
Best Practices for Developing High Performance Web and Enterprise Applications

Figure 7c – Performance Impact - Using javax.sql.DataSources Correctly



[\[TOP\]](#)

Best Practice 8 Release JDBC resources when done

Failing to close and release JDBC connections can cause other users to experience long waits for connections. Although a JDBC connection that is left unclosed will be reaped and returned by WebSphere Application Server after a timeout period, others may have to wait for this to occur.

Close JDBC statements when you are through with them. JDBC ResultSets can be explicitly closed as well. If not explicitly closed, ResultsSets are released when their associated statements are closed.

Ensure that your code is structured to close and release JDBC resources in all cases, even in exception and error conditions.

Figure 8 - The proper way to close JDBC Connections and PreparedStatements

```
Connection conn = null;
ResultSet rs = null;
PreparedStatement pss = null;
try
{
    conn = dataSource.getConnection(USERID,PASSWORD);

    pss = conn.prepareStatement("SELECT SAVESERIALIZEDDATA FROM SESSION.PINGSESSION3DATA WHERE SESSIONKEY = ?");

    pss.setString(1,sessionKey);

    rs = pss.executeQuery();
}
catch (Throwable t){ // Insert Appropriate Error Handling Here }
finally
{
    //The finally Clause is always executed - even in error conditions PreparedStatements and Connections will always be closed
    try
    {
        if(pss != null)
            pss.close();
    }
    catch(Exception e){}

    try
    {
        if(conn != null)
            conn.close();
    }
    catch(Exception e){}
}
```

[\[TOP\]](#)

Best Practice 9 Use the HttpServlet Init method to perform expensive operations that need only be done once

Because the servlet init() method is invoked when servlet instance is loaded, it is the perfect location to carry out expensive operations that need only be performed during initialization. By definition, the init() method is thread-safe. The results of operations in the HttpServlet.init() method can be cached safely in servlet instance variables, which become read-only in the servlet service method.

There is an example of this in Best Practice 7, [Reuse datasources for JDBC connections](#). Recall in Figure 7b the JDBC DataSource was acquired in the HttpServlet.init() method.

[\[TOP\]](#)

Best Practice 10 Minimize use of System.out.println

Because it seems harmless, this commonly used application development legacy is overlooked for the performance problem it really is. Because System.out.println statements and similar constructs synchronize processing for the duration of disk I/O, they can significantly slow throughput. Figure 10c shows the adverse performance impact of inserting System.out.println statements into an application.

Avoid using indiscriminate System.out.println statements. State of the art enterprise application development facilities, such as IBM VisualAge® Java™, provide developers with excellent debugging tools for unit testing. Moreover, the WebSphere Application Server Distributed Debugger can be used to diagnose code on a running system.

However, even with these tools, there remains a legitimate need for application tracing both in test and production environments for error and debugging situations. Such application level tracing like most system level traces should be configurable to be activated in error and debugging situations only. One good design implementation is to tie tracing to a “final boolean” value, which when configured to false will optimize out both the check and execution of the tracing at compile time. See Figure 10a for more detail.

Consider also that the WebSphere Application Server product allows for the complete deactivation of System.out and System.err for any given application server at runtime. See Figure 10b for more detail.

Figure 10a - Activate Application Level Tracing Only When Absolutely Needed

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class BpSaveATreeServlet extends HttpServlet {
    // False Turns Off Check and Trace. True Turns Both On
    private final static boolean TRACING_ON = false;

    public void service(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        if (TRACING_ON) // IF False Both Check and System.out.println are optimized away
        {
            System.out.println("AN ERROR HAS OCCURRED PLEASE FIX AND RE-RUN!!!!");
        }
    }
}
```

**WebSphere Application Server
White Paper
Best Practices for Developing High Performance Web and Enterprise Applications**

Figure 10b STDOUT and STDERR deactivated on Windows/NT by "" and on UNIX by dev/null

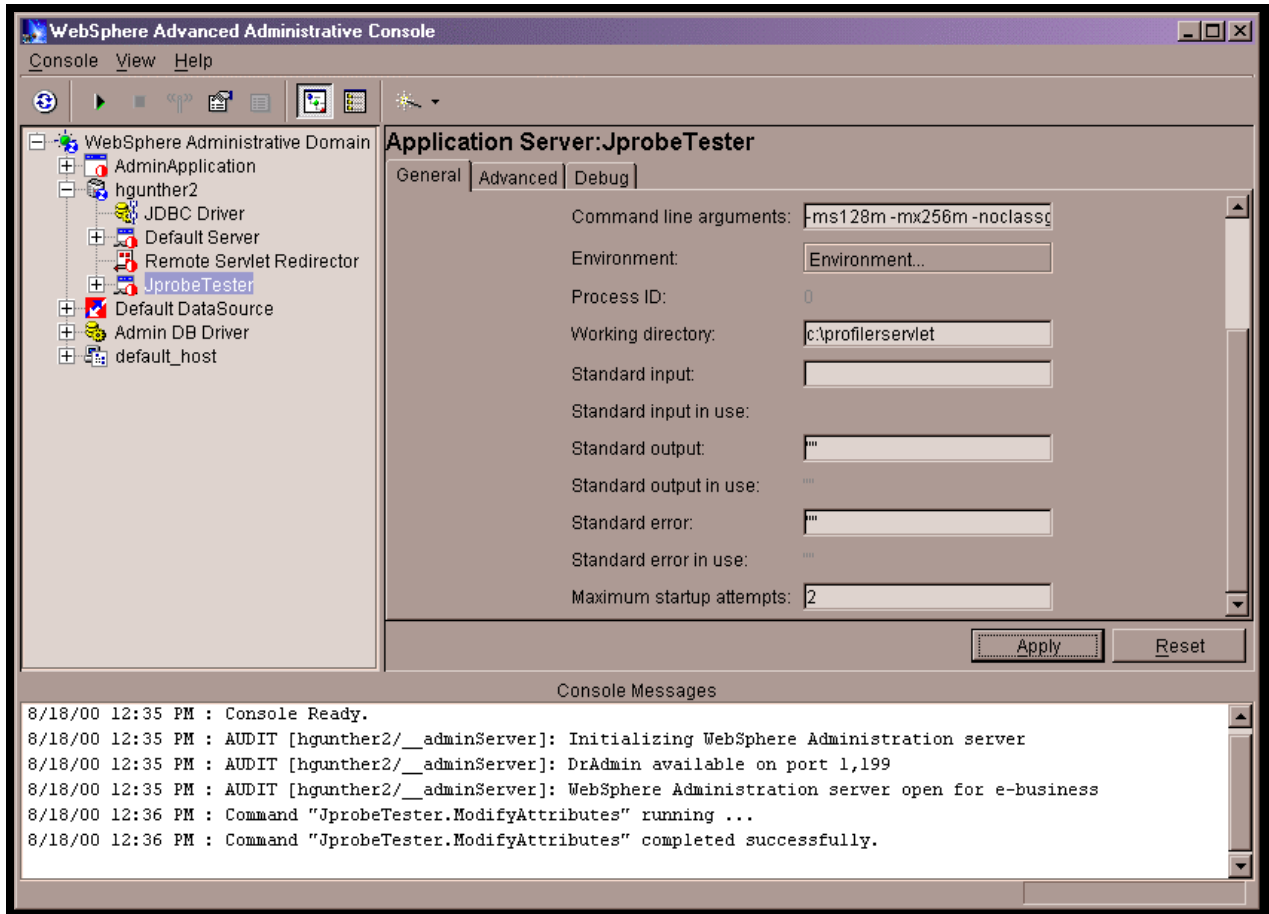
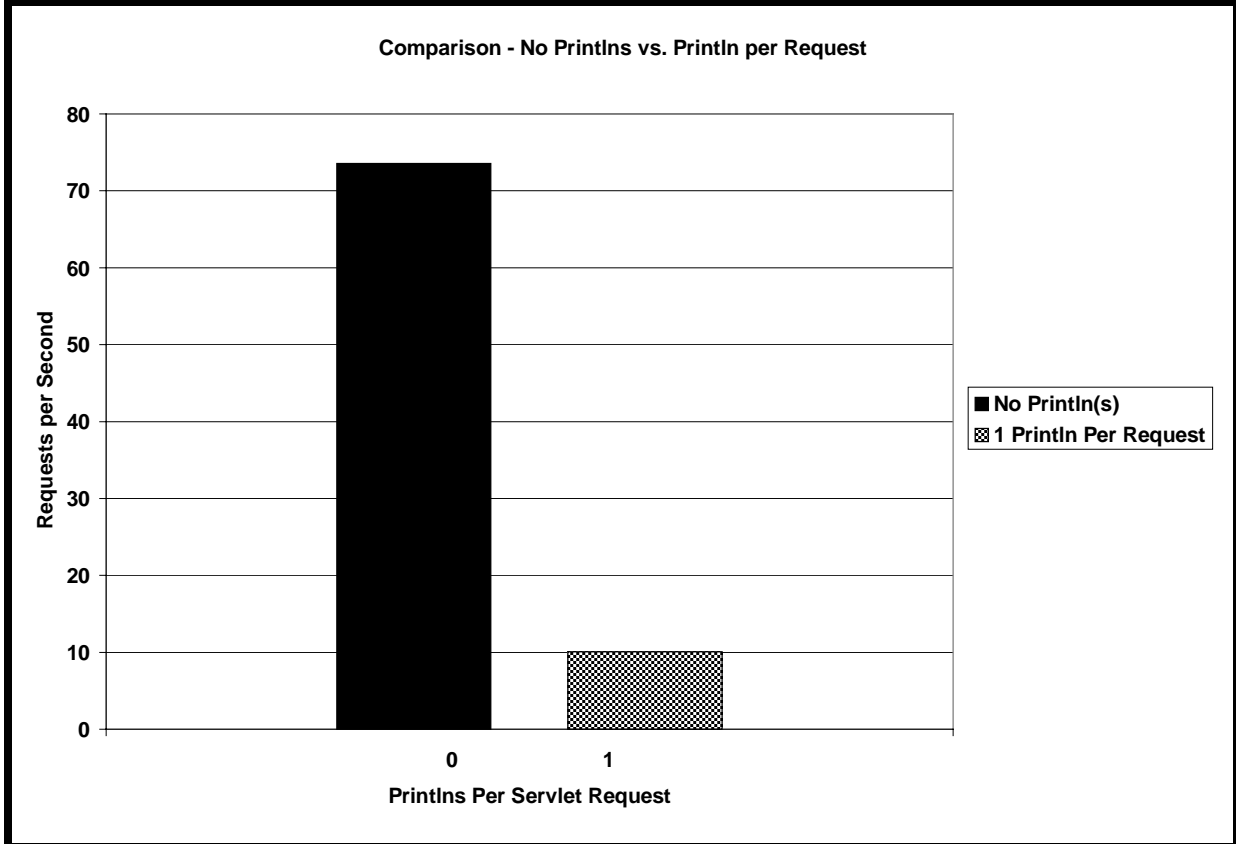


Figure 10c – Performance Impact "System.out.println" in Application Code



[\[TOP\]](#)

Best Practice 11 Avoid String concatenation “+=”

String concatenation is the textbook bad practice that illustrates the adverse performance impact of creating large numbers of temporary Java objects. Because Strings are immutable objects, String concatenation results in temporary object creation that increases Java garbage collection and consequently CPU utilization as well.

The textbook solution is to use `java.lang.StringBuffer` instead of string concatenation:

- **Figure 11a** shows the wrong way to concatenate Strings.
- **Figure 11b** shows the right way to concatenate Strings.
- **Figure 11c** shows the impact on performance.

Figure 11a - Wrong Way (String+=) to Build Strings

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class BpWorstCaseStringServlet extends HttpServlet
{
    final static private String TYPICAL_STRING = "AAAAAAAAAAAA";
    final static int NUMBER_OF_REPS = 100;

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        long startTime = System.currentTimeMillis();

        String workString = new String();

        for(int i = 0; i < 100; i++) {
            workString += TYPICAL_STRING;
        }

        long endTime = System.currentTimeMillis() - startTime;
    }
}
```

Figure 11b - Correct Way to Build Strings – StringBuffer

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class BpBestCaseStringServlet extends HttpServlet
{
    final static private String TYPICAL_STRING = "AAAAAAAAAAAA";
    final static int NUMBER_OF_REPS = 100;

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        long startTime = System.currentTimeMillis();

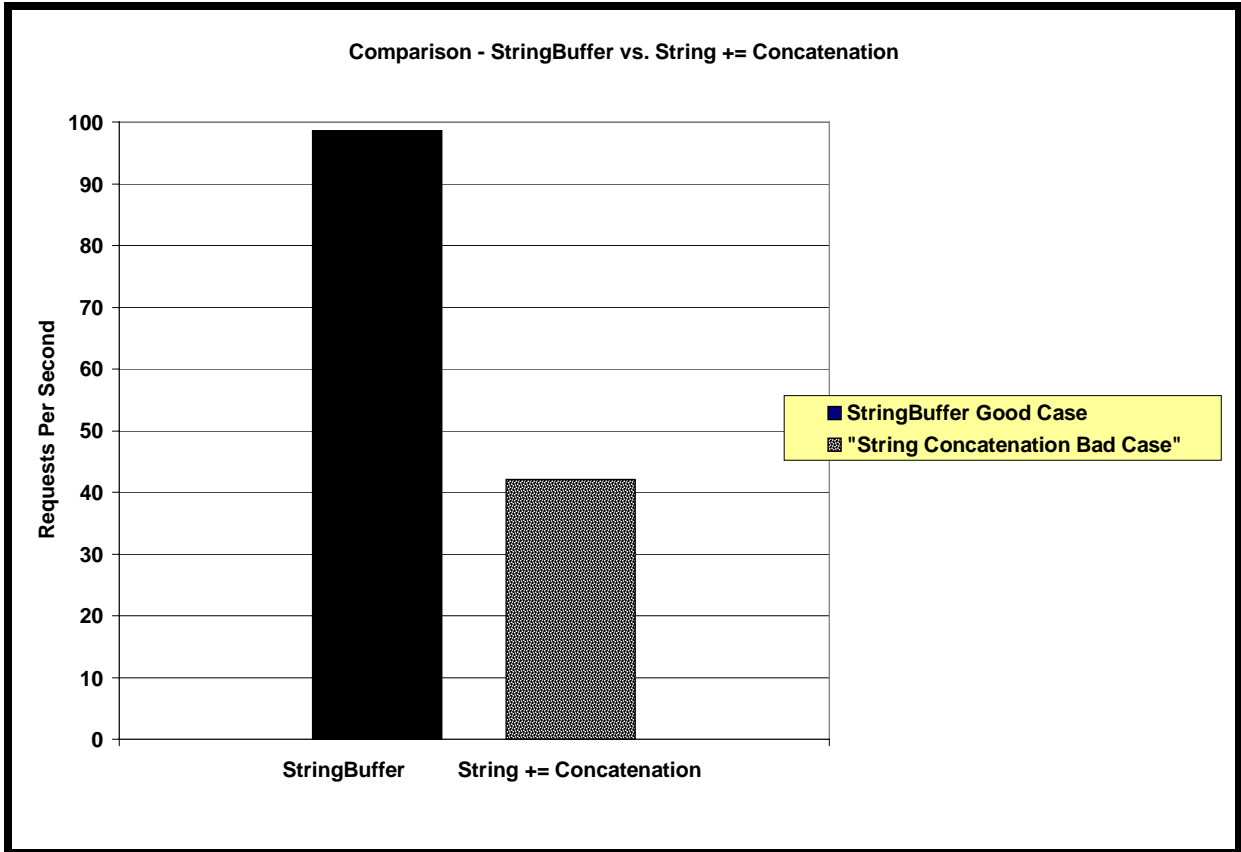
        StringBuffer workBuffer = new StringBuffer(2000);
        String workString;

        for(int i = 0; i < NUMBER_OF_REPS; i++)
        {
            workBuffer.append(TYPICAL_STRING);
        }
        workString = workBuffer.toString();

        long endTime = System.currentTimeMillis() - startTime;
    }
}
```

WebSphere Application Server
White Paper
Best Practices for Developing High Performance Web and Enterprise Applications

Figure 11c - Performance Impact of using StringBuffer instead of String Concatenation



[\[TOP\]](#)

Best Practice 12 Access entity beans from session beans

Avoid accessing EJB entity beans from client or servlet code. Instead wrap and access EJB entity beans in EJB session beans. This best practice satisfies two performance concerns:

- Reducing the number of remote method calls. When the client application accesses the entity bean directly, each getter method is a remote call. A wrapping session bean can access the entity bean locally, and collect the data in a structure, which it returns by value.
- Providing an outer transaction context for the EJB entity bean. An entity bean synchronizes its state with its underlying data store at the completion of each transaction. When the client application accesses the entity bean directly, each getter method becomes a complete transaction. A store and a load follow each method. When the session bean wraps the entity bean to provide an outer transaction context, the entity bean synchronizes its state when outer session bean reaches a transaction boundary.

Figure 12a illustrates accessing EJB entity beans from EJB session beans. Figure 12b demonstrates the impact of this Best Practice.

Figure 12a - Use EJB Session Beans to wrap EJB Entity Beans

```
import java.rmi.RemoteException;
import java.security.Identity;
import java.util.Properties;
import javax.ejb.*;
import com.ibm.uxo.bestpractices.datamodels.*;

public class EmployeeRosterBean implements SessionBean {
    private EmployeeHome employeeHome;
    private javax.ejb.SessionContext mySessionCtx = null;
    final static long serialVersionUID = 3206093459760846163L;

    public void ejbCreate() throws javax.ejb.CreateException, java.rmi.RemoteException {
        employeeHome = EmployeeEjbHomeCacheHelper.getEmployeeHome();
    }

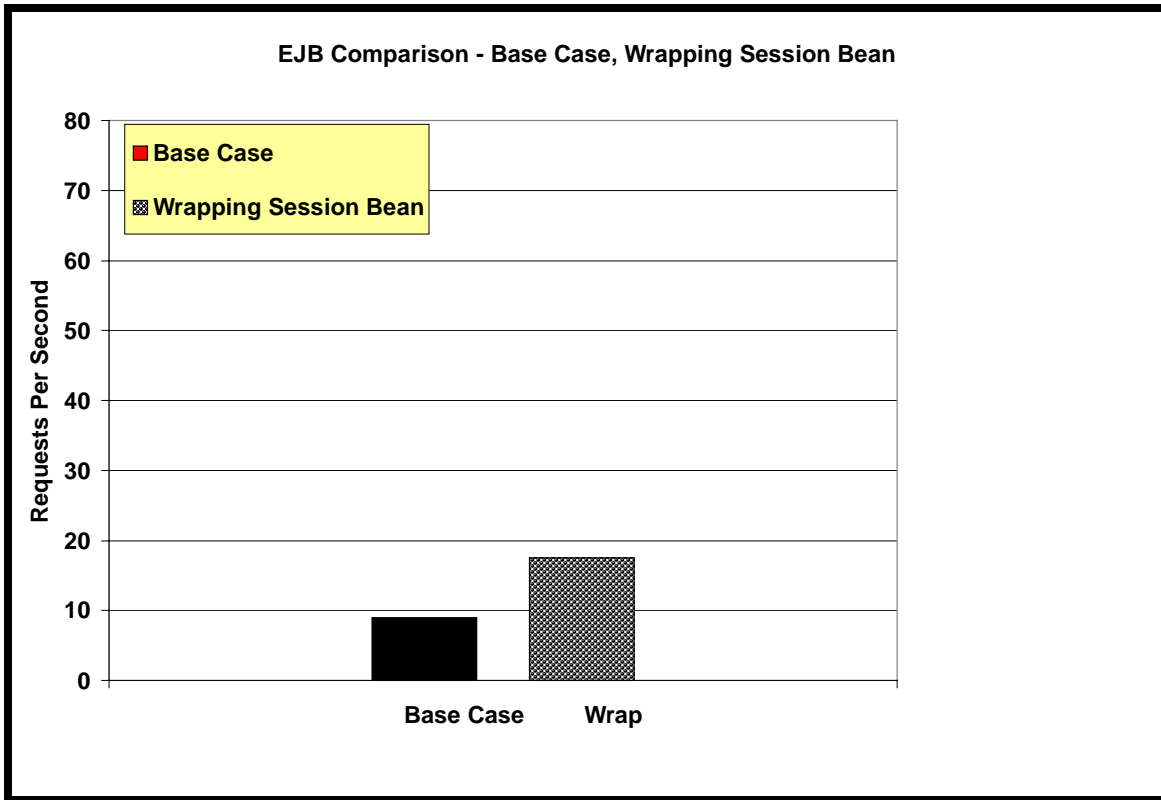
    public EmployeeStruct getEmployeeInfoFor(String empno)
    {
        Employee theEmployee = null;
        EmployeeStructure returnValue = new EmployeeStructure();
        try
        {
            theEmployee = employeeHome.findByPrimaryKey(new EmployeeKey(empno));

            returnValue.setSex(theEmployee.getSex());
            returnValue.setSalary(theEmployee.getSalary());
            returnValue.setPhoneno(theEmployee.getPhoneno());
            returnValue.setMidinit(theEmployee.getMidinit());
            returnValue.setLastname(theEmployee.getLastname());
            returnValue.setJob(theEmployee.getJob());
            returnValue.setHiredate(theEmployee.getHiredate());
            returnValue.setFirstname(theEmployee.getFirstname());
            returnValue.setEmpno(empno);
            returnValue.setEdlevel(theEmployee.getEdlevel());
            returnValue.setComm(theEmployee.getComm());
            returnValue.setBonus(theEmployee.getBonus());
            returnValue.setBirthdate(theEmployee.getBirthdate());
            returnValue.setWorkdept(theEmployee.getWorkdept());
        }
        catch(Exception e){e.printStackTrace();}
        return returnValue;
    }

    public void ejbActivate() throws java.rmi.RemoteException {}
    public void ejbPassivate() throws java.rmi.RemoteException {}
    public void ejbRemove() throws java.rmi.RemoteException {}
}
```

WebSphere Application Server
White Paper
Best Practices for Developing High Performance Web and Enterprise Applications

Figure 12b - Performance Impact - Wrapping EJB Entity Beans Within EJB Session Beans



[\[TOP\]](#)

Best Practice 13 Reuse EJB homes

EJB homes are obtained from WebSphere Application Server through a JNDI naming lookup. This is an expensive operation that can be minimized by caching and reusing EJB Home objects. This is similar to Best Practice 7 [Reuse DataSources for JDBC Connections](#).

For simple applications, it might be enough to acquire the EJB home in the servlet init() method. Figure 13b shows how to acquire an EJB Home in the HttpServlet.init() method. This is consistent with Best Practice 9 [Use the HttpServlet Init Method To Perform Expensive Operations that Need Only Be Done Once](#).

More complicated applications might require cached EJB homes in many servlets and EJBs. One possibility for these applications is to create an EJB Home Locator and Caching class. Figure 13b and Figure 13c show a singleton EJB home locator class that would be appropriate both for client (servlets and others) and inter-EJB access. Finally, Figure 13d shows the performance impact achieved by caching and reusing EJB Homes.

Figure 13a - Example Caching EJB Home in the Servlet Init Method

```
public class BpSessionEJBServlet extends HttpServlet
{
    private BpSessionManagerHome sseHome = null; // Cache the EJB Home Here

    public void init(ServletConfig config) throws ServletException
    {
        super.init(config);
        javax.naming.Context ctx = null;
        try
        {
            java.util.Hashtable env = new java.util.Hashtable();
            env.put(Context.INITIAL_CONTEXT_FACTORY, FACTORY);
            ctx = new InitialContext(env);
            Object homeObject = ctx.lookup("EJB JNDI NAME");
            sseHome = ( BpSessionManagerHome)javax.rmi.PortableRemoteObject.narrow(
                (org.omg.CORBA.Object)homeObject, BpSessionManagerHome.class);
        }
        catch(Exception e)
        {
            throw new ServletException("INIT Error: "+e.getMessage(),e);
        }
        finally
        {
            try
            {
                if(ctx != null)
                    ctx.close();
            }
            catch(Exception e) { // Handle this Exception Appropriately }
        }
    }
}
```

WebSphere Application Server
White Paper
Best Practices for Developing High Performance Web and Enterprise Applications

Figure 13b - EJB Home Caching Singleton

```
public class EmployeeEjbHomeCacheHelper {
    // One Possible Implementation of an EjbHomeCache
    public static final String BESTPRACTICES_EMPLOYEE = "Employee3";
    public static final String BESTPRACTICES_EMPLOYEE_ROSTER_HOME = "EmployeeRoster3";

    private static EmployeeEjbHomeCacheHelper ejbHomeCache;
    private static final String FACTORY = "com.ibm.ejs.ns.jndi.CNInitialContextFactory";
    private java.util.Hashtable fieldHomeTable = new java.util.Hashtable();

    public static EmployeeHome getEmployeeHome() {
        return (EmployeeHome)getInstance().getHomeTable().get(BESTPRACTICES_EMPLOYEE);
    }

    public static EmployeeRosterHome getEmployeeRosterHome() {
        return (EmployeeRosterHome)getInstance().getHomeTable().get(BESTPRACTICES_EMPLOYEE_ROSTER_HOME);
    }

    // Maybe only God can create a tree but only I can create an instance of me!!!!
    private EmployeeEjbHomeCacheHelper();

    private synchronized static EmployeeEjbHomeCacheHelper getInstance() {
        if(ejbHomeCache == null) {
            ejbHomeCache = new EmployeeEjbHomeCacheHelper();
            ejbHomeCache.buildHomeAndDataSourceCache();
        }
        return ejbHomeCache;
    }

    private void buildHomeAndDataSourceCache() { //See Figure 13c For the Code }

    private java.util.Hashtable getHomeTable() {
        return fieldHomeTable;
    }

    private void setHomeTable(java.util.Hashtable homeTable) {
        fieldHomeTable = homeTable;
    }
}
```

Figure 13c - EJB Home Cache Class - One Time build Cache Method

```
// See Figure 13b For the Rest of the Code
private void buildHomeAndDataSourceCache(){
    Context ctx = null;
    Object homeObject = null;

    EmployeeRosterHome mbh = null;
    EmployeeHome mdl = null;

    java.util.Hashtable env = new java.util.Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY, FACTORY);

    try {
        ctx = new InitialContext(env);

        homeObject = ctx.lookup(BESTPRACTICES_EMPLOYEE);
        mdl = (EmployeeHome)javax.rmi.PortableRemoteObject.narrow(
            (org.omg.CORBA.Object)homeObject, EmployeeHome.class);

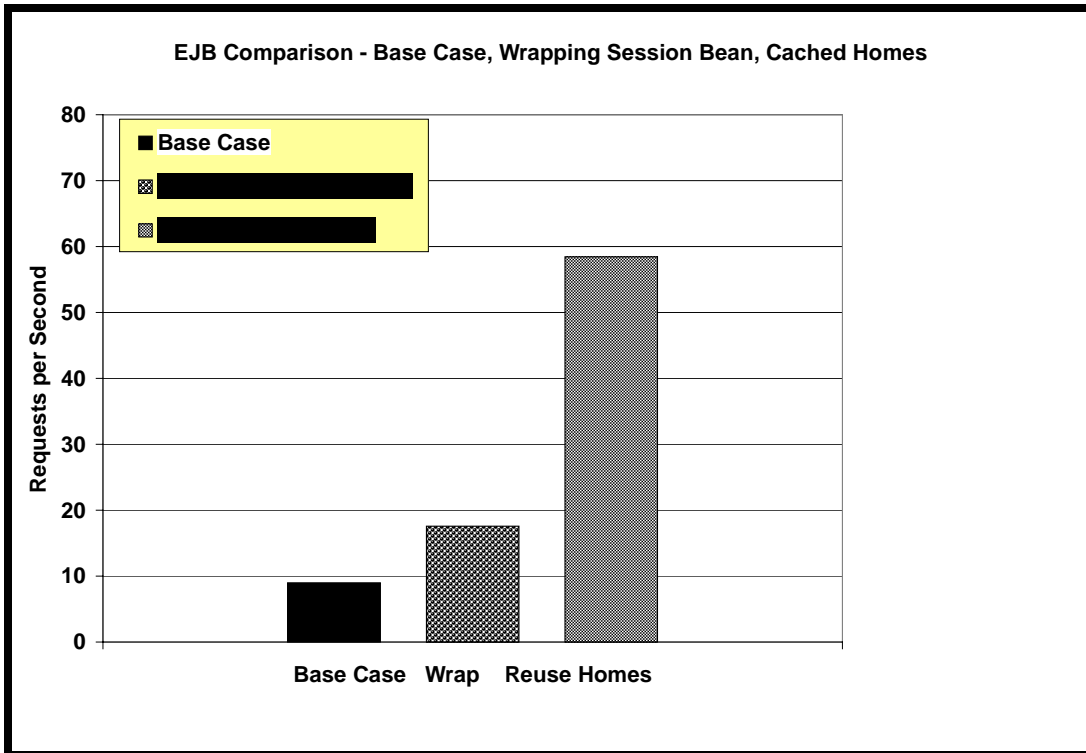
        getHomeTable().put(BESTPRACTICES_EMPLOYEE, mdl);

        homeObject = ctx.lookup(BESTPRACTICES_EMPLOYEE_ROSTER_HOME);
        mbh = (EmployeeRosterHome)javax.rmi.PortableRemoteObject.narrow(
            (org.omg.CORBA.Object)homeObject, EmployeeRosterHome.class);
        getHomeTable().put(BESTPRACTICES_EMPLOYEE_ROSTER_HOME, mbh);
    }
    catch(Throwable th){th.printStackTrace();}

    finally
    {
        if(ctx != null){
            try {
                ctx.close();
            }
            catch(Throwable th){th.printStackTrace();}
        }
    }
}
```

WebSphere Application Server
White Paper
Best Practices for Developing High Performance Web and Enterprise Applications

Figure 13d - Performance Impact - Caching EJB Homes



**WebSphere Application Server
White Paper
Best Practices for Developing High Performance Web and Enterprise Applications**

Caution: – Cached Stale Homes

One problem with caching and reusing EJB homes involves handling stale cached homes. If a client caches an EJB home and then the container shuts down and then returns, the client's cached value is stale and will not work. Accessing a cached stale home will result in the exception stack show in Figure 13e.

Please note that the code samples in Figures 13a through 13c do not handle stale EJB homes.

Figure 13e - Stack Trace from Accessing Stale EJB Home

```
java.rmi.MarshalException: CORBA COMM_FAILURE 3 No; nested exception is:
  org.omg.CORBA.COMM_FAILURE:  minor code: 3  completed: No
  java.lang.Throwable(java.lang.String)
  java.lang.Exception(java.lang.String)
  java.io.IOException(java.lang.String)
  java.rmi.RemoteException(java.lang.String, java.lang.Throwable)
  java.rmi.MarshalException(java.lang.String, java.lang.Exception)
  java.rmi.RemoteException javax.rmi.CORBA.Util.mapSystemException(org.omg.CORBA.SystemException)
  com.ibm.uxo.bestpractices.ejbs4.EmployeeRoster com.ibm.uxo.bestpractices.ejbs4._EmployeeRosterHome_BaseStub.create()
  com.ibm.uxo.bestpractices.ejbs4.EmployeeRoster com.ibm.uxo.bestpractices.ejbs4._EmployeeRosterHome_Stub.create()
  void com.ibm.uxo.bestpractices.employment.servlets.BpEmploymentServletV3.service(javax.servlet.http.HttpServletRequest, javax
  void javax.servlet.http.HttpServlet.service(javax.servlet.ServletRequest, javax.servlet.ServletResponse)
```

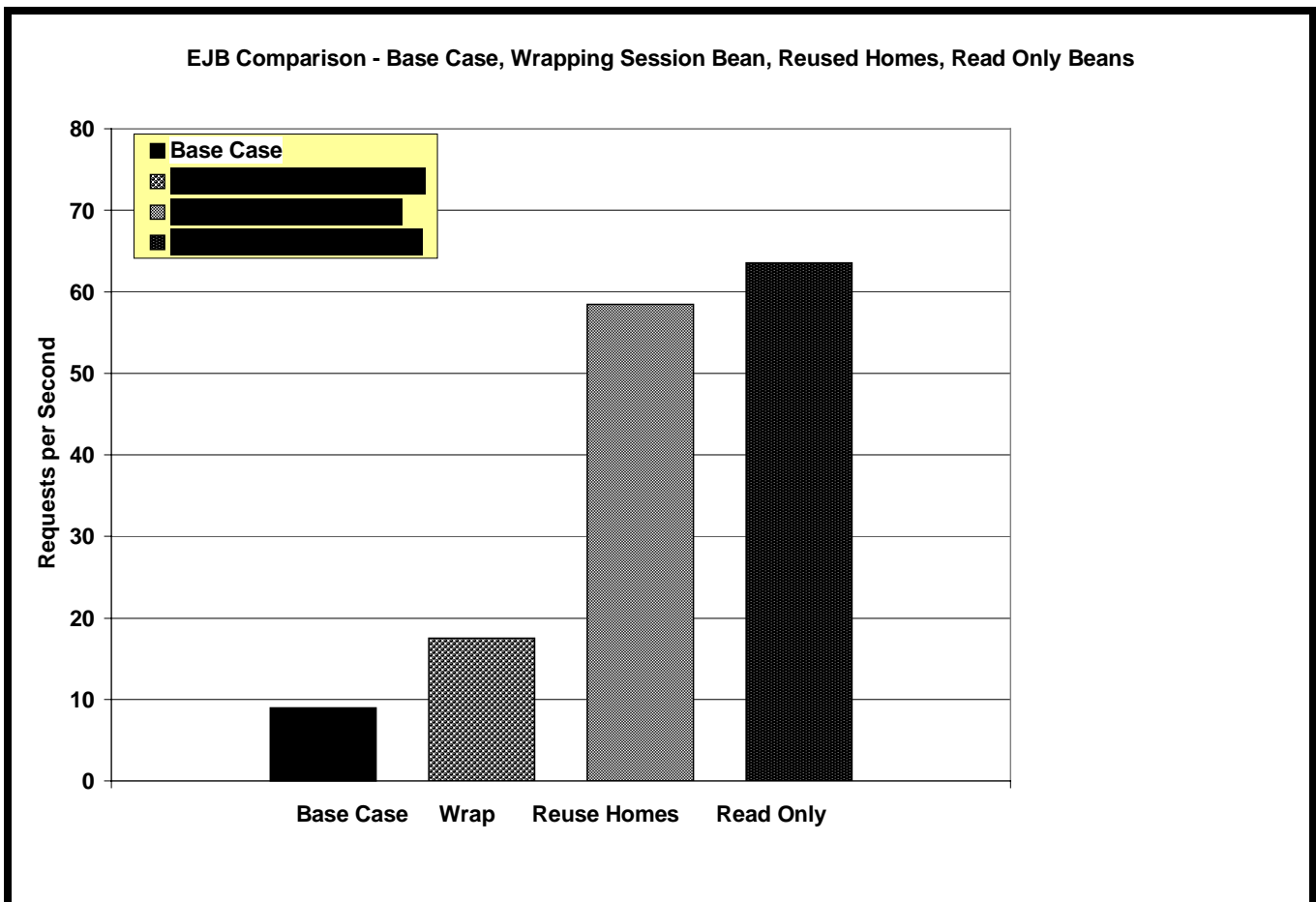
[\[TOP\]](#)

Best Practice 14 Use “Read-Only” methods where appropriate

When setting the deployment descriptor for an EJB Entity Bean, you can mark “getter” methods as “Read-Only.” If a transaction unit of work includes no methods other than “Read-Only” designated methods, then the Entity Bean state synchronization will not invoke store.

Figure 14a shows the performance impact achieved by setting “Read Only” methods in the EJB Entity Bean deployment descriptor.

Figure 14a - Performance Impact – Using "Read-Only" EJB Entity Bean Methods



[\[TOP\]](#)

Best Practice 15 Reduce the transaction isolation level where appropriate

By default, most developers deploy EJBs with the transaction isolation level set to TRANSACTION_SERIALIZABLE. This is the default in IBM VisualAge Java - Enterprise Edition and other EJB deployment tools. It is also the most restrictive and protected transaction isolation level incurring the most overhead.

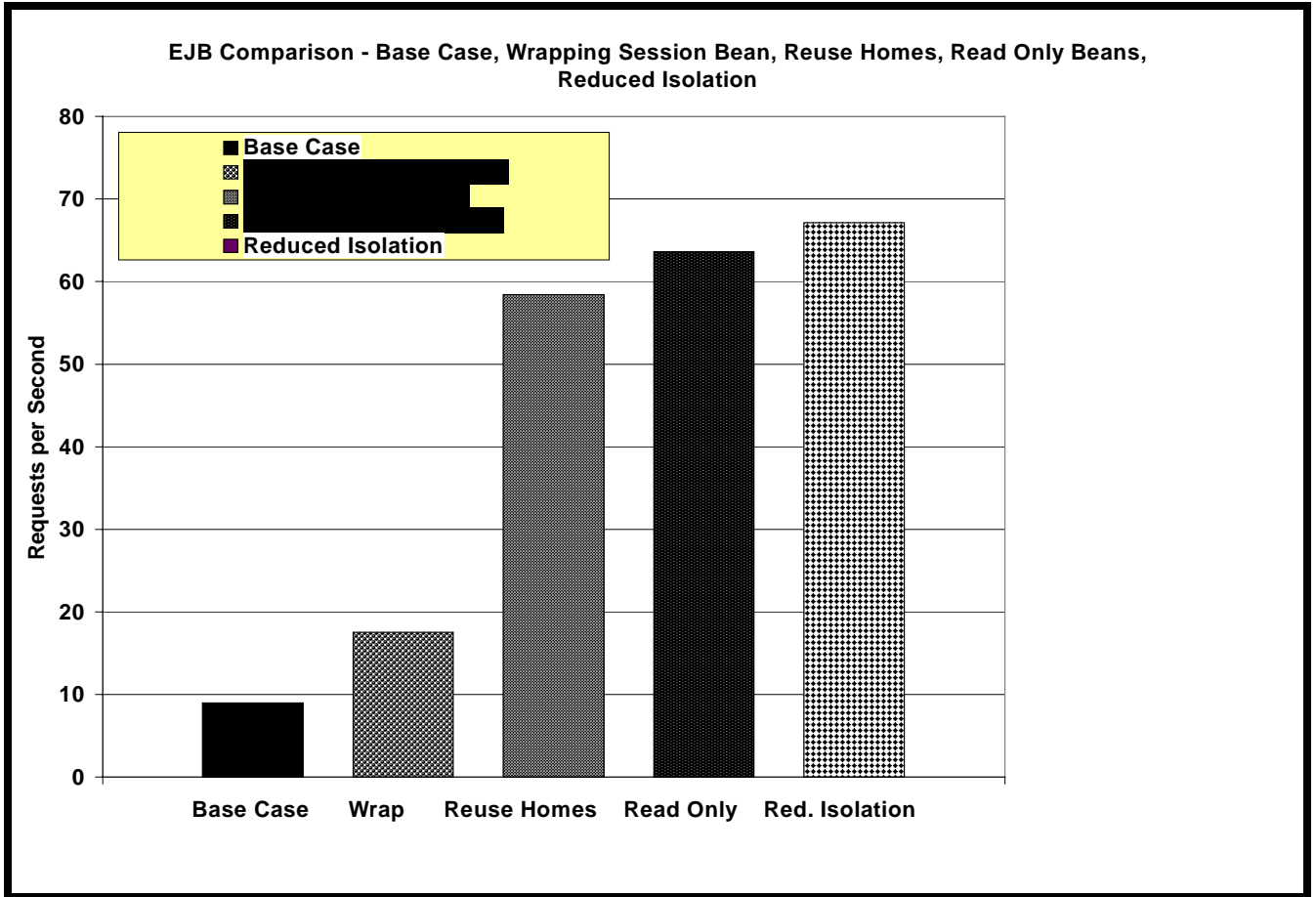
Some workloads do not require the isolation level and protection afforded by TRANSACTION_SERIALIZABLE. A given application might never update the underlying data or be run with other concurrent updaters. In that case, the application would not have to be concerned with dirty, non-repeatable, or phantom reads. TRANSACTION_READ_UNCOMMITTED would probably be sufficient.

Because the EJB's transaction isolation level is set in its deployment descriptor, the same EJB could be reused in different applications with different transaction isolation levels. Review your isolation level requirements and adjust them appropriately to increase performance.

Figure 15a shows the performance impact achieved by reducing the transaction isolation level in the EJB Entity Bean deployment descriptor.

WebSphere Application Server
White Paper
Best Practices for Developing High Performance Web and Enterprise Applications

Figure 15a – Performance Impact - Reducing Transaction Isolation Levels When Appropriate



[\[TOP\]](#)

Best Practice 16 EJBs and Servlets - same JVM - “No local copies”

Because EJBs are inherently location independent, they use a remote programming model. Method parameters and return values are serialized over RMI-IIOP and returned by value. This is the intrinsic RMI “Call By Value” model.

WebSphere provides the “No Local Copies” performance optimization for running EJBs and clients (typically servlets) in the same application server JVM. The “No Local Copies” option uses “Call By Reference” and does not create local proxies for called objects when both the client and the remote object are in the same process. Depending on your workload, this can result in a significant overhead savings.

Configure “No Local Copies” by adding the following two command line parameters to the application server JVM:

- **-Djavax.rmi.CORBA.UtilClass=com.ibm.CORBA.iiop.Util**
- **-Dcom.ibm.CORBA.iiop.noLocalCopies=true**

CAUTION: The “No Local Copies” configuration option improves performance by changing “Call By Value” to “Call By Reference” for clients and EJBs in the same JVM. One side effect of this is that the Java object derived (non-primitive) method parameters can actually be changed by the called enterprise bean. Consider Figure 16a:

Figure 16a - Pass By Reference Side Effects of "No Local Copies"

```
class SomeClass{
    RemoteObjectHome myRemoteHome;
    RemoteObject myRemoteObject;
    ParameterObject myArgument;

    void someMethod() {
        myRemoteObject = myRemoteHome.create();
        myArgument.setProperty("Before");
        String newProperty = myRemoteObject.someRemoteMethod(myArgument);
        // Both String newProperty and ParameterObject myArgument
        // Have Been Changed - "Call By Reference"
    }
}

class RemoteObjectBean implements SessionBean {
    public String someRemoteMethod(ParameterObject aParameter) {
        if(aParameter.getProperty().equals("Before")) {
            aParameter.setProperty("After");
        }
        return aParameter.getProperty();
    }
}
```

[\[TOP\]](#)

Best Practice 17 Remove stateful session beans when finished

Instances of stateful session beans have affinity to specific clients. They will remain in the container until they are explicitly removed by the client, or removed by the container when they timeout. Meanwhile, the container might need to passivate inactive stateful session beans to disk. This requires overhead for the container and constitutes a performance hit to the application. If the passivated session bean is subsequently required by the application, the container activates it by restoring it from disk.

By explicitly removing stateful session beans when finished with them, applications will decrease the need for passivation and minimize container overhead.

Figure 17a - Removing Stateful Session Beans When Finished

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.naming.*;
import com.ibm.uxo.ejbs.*;

public class BestPracticesServlet extends HttpServlet
{
    BestPracticesHome sseHome = null;
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        BestPractices ssmgr = null;
        try {
            ssmgr = sseHome.create(1);
            ssmgr.someBunchOfMethods();
            ssmgr.remove(); // Explicitly Remove When Done !!!!
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
    public void init(ServletConfig config)
        throws ServletException {
        super.init(config);
        try {
            sseHome = EJBhomeCache.getInstance().getMbhHome();
        }
        catch (Exception e) {
            throw new ServletException("INIT Error: "+e.getMessage(),e);
        }
    }
}
```

[\[TOP\]](#)

Best Practice 18 Don't use Beans.instantiate() to create new bean instances

The method, `java.beans.Beans.instantiate()`, will create a new bean instance either by retrieving a serialized version of the bean from disk or creating a new bean if the serialized form does not exist. The problem, from a performance perspective, is that each time `java.beans.Beans.instantiate` is called the file system is checked for a serialized version of the bean. As usual, such disk activity in the critical path of your web request can be costly. To avoid this overhead, simply use “new” to create the instance.

Figure 18a shows the incorrect and worse performing way to create an instance of a class. Figure 18b shows the correct and better performing way to create an instance of a class. Figure 18c shows the performance impact.

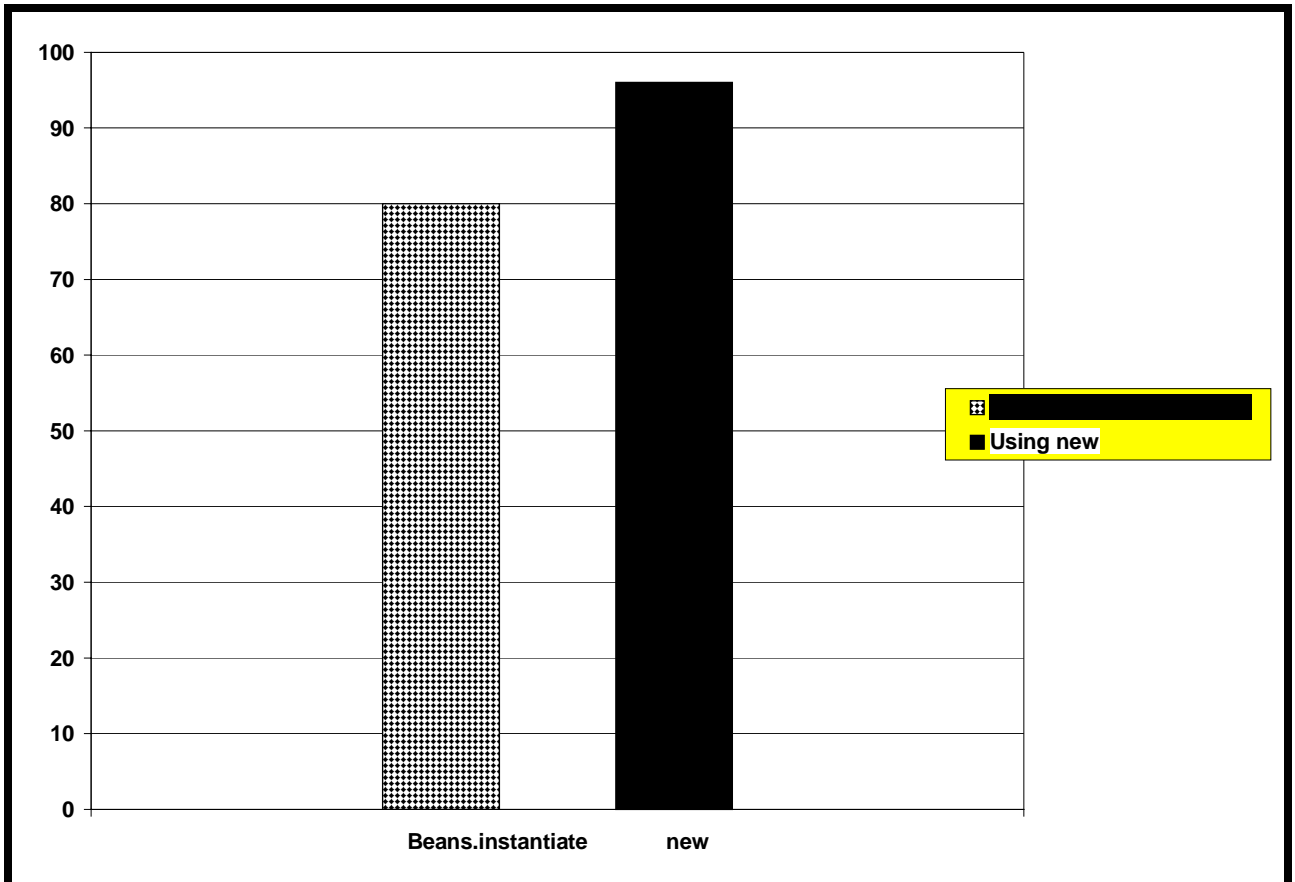
Figure 18a - AVOID Using `java.beans.Beans.instantiate()`

```
public class BadNewsServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        // using Beans.instantiate() Don't do it this way
        PingBean ab = (PingBean) Beans.instantiate(this.getClass().getClassLoader(), "web_prmtv.PingBean");
        ab.setMsg("Hit Count: " + hitCount++);
        req.setAttribute("ab", ab);
        getServletContext().getRequestDispatcher("/servlet/PingServlet2ServletRcv").forward(req, res);
    }
}
```

Figure 18b Use new someClass() to create a new object instance

```
public class WayToGoServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        int hitCount = 0;
        // using new to create the instance
        PingBean ab = new PingBean();
        ab.setMsg("Hit Count: " + hitCount++);
        req.setAttribute("ab", ab);
        getServletContext().getRequestDispatcher("/servlet/PingServlet2ServletRcv").forward(req, res);
    }
}
```

Figure 18c - Performance impact - using new instead of Beans.Instantiate



[\[TOP\]](#)

Bibliography – Additional References

Dov Bulka, Java Performance and Scalability Volume 1 Server-Side Programming Techniques, First Printing May, 2000, Addison Wesley

Peter Haggar, Practical Java Programming Language Guide, First Printing January, 2000, Addison Wesley

Steve Wilson, Jeff Kesselman, Java Platform Performance Strategies and Tactics, First Printing June, 2000, Addison Wesley

Steven L. Halter, Steven J. Munroe, Enterprise Java Performance, First Printing July, 2000, Prentice Hall PTR

IBM International Technical Support Organization, IBM San Francisco Performance Tips and Techniques, SG24-5368-0, See especially Chapter 9, First Printing February, 1999

IBM International Technical Support Organization, WebSphere V3 Performance Tuning Guide, SG24-5657-0, First Printing March, 2000

**WebSphere Application Server
White Paper
Best Practices for Developing High Performance Web and Enterprise Applications**

[\[TOP\]](#)

Trademarks

IBM, WebSphere and VisualAge are trademarks of International Business Machines Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

© Copyright International Business Machines Corporation 2000. All rights reserved.