

IBM Systems - iSeries

e-business and Web serving

WebSphere Application Server - Express Version 5.1
Programming

Version 5 Release 4





IBM Systems - iSeries

e-business and Web serving

WebSphere Application Server - Express Version 5.1
Programming

Version 5 Release 4

Note

Before using this information and the product it supports, be sure to read the information in "Notices," on page 181.

Third Edition (February 2006)

This edition applies to version 5.1 of WebSphere Application Server - Express (5722-E51) and to all subsequent releases and modifications until otherwise indicated in new editions. This version does not run on all reduced instruction set computer (RISC) models nor does it run on CISC models.

© Copyright International Business Machines Corporation 2004, 2006. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Application development 1

| | |
|--|----|
| Step 1: Plan. | 2 |
| Application development tools | 3 |
| Step 2: Design your application | 4 |
| Step 3: Develop your application | 5 |
| Classloaders | 6 |
| ClassLoader hierarchy | 6 |
| Java Virtual Machine classloaders | 7 |
| Java cache for user classloaders | 7 |
| WebSphere extensions classloader | 9 |
| Java execution modes | 10 |
| Classloaders in applications | 10 |
| Administer classloaders using the WebSphere administrative console. | 10 |
| ClassLoader policies. | 12 |
| Servlets. | 14 |
| Servlet lifecycle | 15 |
| Create a servlet | 17 |
| ServletSample.java | 17 |
| Write a servlet | 18 |
| Compile the servlet. | 21 |
| Testing a servlet | 22 |
| Application lifecycle listeners and events | 22 |
| Servlet filtering | 23 |
| Page lists | 25 |
| client_types.xml | 25 |
| Example: Extending PageListServlet | 26 |
| Automatic request and response encoding | 27 |
| Enhanced error reporting. | 28 |
| Internal servlets | 29 |
| Servlet resources. | 29 |
| JavaServer Pages (JSP). | 30 |
| What are JavaServer Pages (JSP) files? | 31 |
| JSP processor | 32 |
| JSP tag extensions support | 33 |
| IBM extensions to JSP tags | 33 |
| <tsx:dbconnect> | 34 |
| <tsx:userid> and <tsx:passwd>. | 35 |
| <tsx:dbquery> | 36 |
| <tsx:dbmodify> | 39 |
| <tsx:repeat> | 40 |
| <tsx:getProperty> | 43 |
| Pre-touch tool for compiling and loading JSP files | 43 |
| JSP batch compiler | 44 |
| Disable JSP run-time compilation | 45 |
| Reduce JSP compile time | 47 |
| Data access | 49 |
| Data access overview | 50 |
| Connection management architecture. | 50 |
| Connection pooling. | 59 |
| Develop data access applications | 61 |
| Data access development model | 62 |
| IBM extensions to the data access API | 63 |
| Access data with J2EE Connector Architecture connectors | 64 |

| | |
|---|-----|
| Access connection pools from your application components | 68 |
| IBM data access JavaBeans | 70 |
| Data access exceptions. | 73 |
| Assemble data access applications. | 78 |
| Configuring the isolation level on a resource reference | 78 |
| Configure WebSphere Application Server - Express to access databases | 79 |
| Configure JDBC data access | 79 |
| Configure JCA data access | 98 |
| Deploy data access applications | 99 |
| Security of lookups. | 99 |
| Java Naming and Directory Interface (JNDI) | 100 |
| JNDI basic concepts | 100 |
| Naming | 101 |
| Name space logical view | 101 |
| Initial context support | 103 |
| Differences between JNDI and CORBA | 104 |
| JNDI implementation. | 105 |
| JNDI caching | 106 |
| JNDI helpers and utilities | 109 |
| Use JNDI. | 115 |
| Obtain the initial JNDI context for the component | 115 |
| Use JNDI to look up Java components | 117 |
| JavaMail | 117 |
| Overview of JavaMail APIs. | 118 |
| Configure JavaMail | 119 |
| Set up and configure e-mail provider services | 119 |
| Configure a JavaMail session using the administrative console | 120 |
| Write JavaMail applications. | 121 |
| Example: JavaMail code | 122 |
| Debug JavaMail | 123 |
| Sessions | 125 |
| Choose a session tracking method | 125 |
| Sessions security | 128 |
| Best practices for session programming. | 129 |
| Session programming model and environment. | 130 |
| Example: SessionSample.java | 131 |
| Configure session management | 131 |
| Configure session tracking for Wireless Application Protocol devices | 132 |
| Assemble applications to share session data | 133 |
| Tune session management | 133 |
| Maximum in-memory session count. | 134 |
| Configure scheduled invalidation. | 134 |
| Bean Scripting Framework | 135 |
| Example: Convert JavaScript source to the Bean Scripting Framework | 136 |
| Scenario: Create a Bean Scripting Framework application | 137 |
| Internationalization of applications | 142 |

| | |
|--|-----|
| Overview of internationalization | 142 |
| Internationalize your application | 143 |
| Identify localizable text | 143 |
| Create message catalogs | 144 |
| Assemble your application code | 145 |
| Internationalization resources | 145 |
| Add logging and tracing to your application | 145 |
| Programming model summary | 146 |
| Overview of JRas | 147 |
| Program with the JRas framework | 149 |
| JRas extensions | 149 |
| Create JRas resource bundles and message files | 151 |
| Create JRas manager and logger instances | 153 |

| | |
|--|-----|
| Extending the JRas framework | 157 |
| Writing User Extensions | 159 |
| Example: user written handler. | 161 |
| Example: user written formatter | 173 |
| JRas messages and trace event types. | 177 |
| Step 4: Assemble your application | 180 |
| Step 5: Deploy your application | 180 |

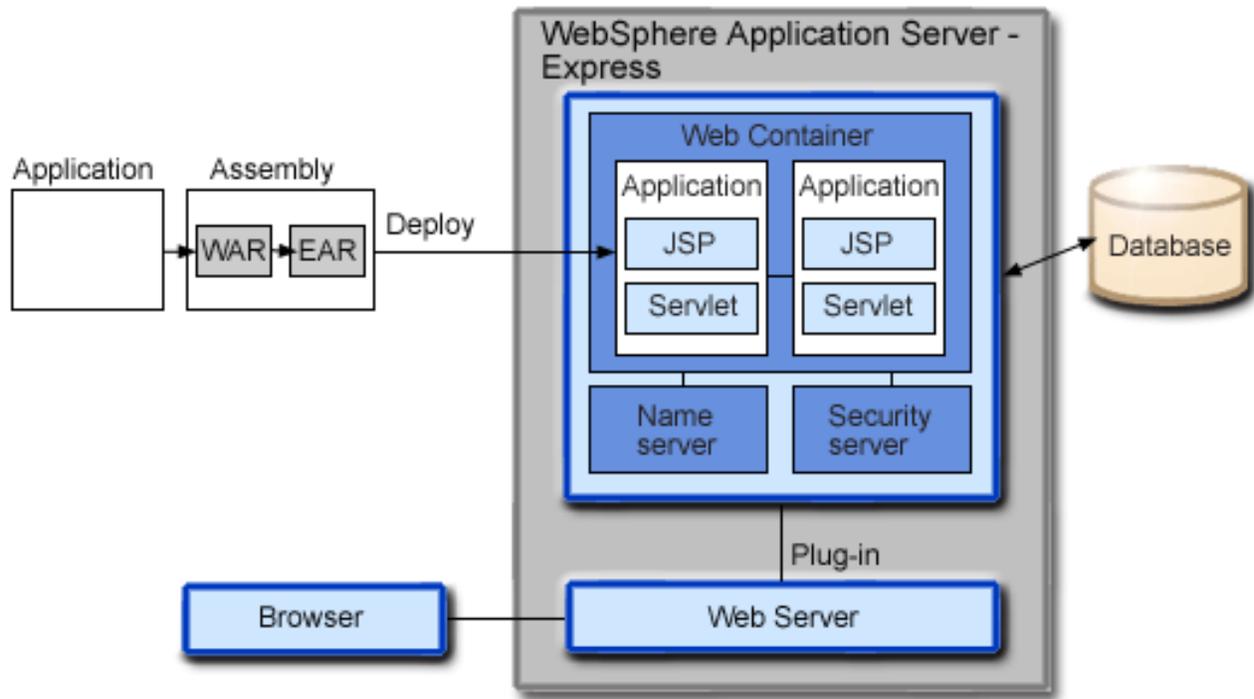
| | |
|---|------------|
| Appendix. Notices | 181 |
| Programming Interface Information | 183 |
| Trademarks | 183 |
| Terms and conditions. | 183 |
| Code license and disclaimer information | 184 |

Application development

WebSphere^(R) Application Server - Express supports these application modules that work together to perform a business logic function:

- Servlets, JavaServer Pages (JSP) files and other Web components, such as HTML and image files
- Resource adapter (connector) implementations
- Other supporting classes and files

This diagram illustrates the overall view comprising these technologies in the WebSphere Application Server - Express environment and the application development process:



Perform these steps to develop and deploy your WebSphere Application Server - Express for iSeries^(TM) applications.

1. PLANNING

“Step 1: Plan” on page 2

“Step 1: Plan” on page 2

This step describes how to prepare your development environment and what information you should gather before you begin designing and developing your application, including information on development tools.



2. DESIGNING YOUR APPLICATION

“Step 2: Design your application” on page 4

“Step 2: Design your application” on page 4
This step provides tips and resources for designing your Java^(TM) components.



3. DEVELOPING YOUR APPLICATION

“Step 3: Develop your application” on page 5

“Step 3: Develop your application” on page 5
This topic describes how to develop applications for WebSphere Application Server.



4. ASSEMBLING YOUR APPLICATION

“Step 4: Assemble your application” on page 180

“Step 4: Assemble your application” on page 180
Application assembly is the process of creating archive files that bundle all of the components belonging to an application and configuring the run-time behavior of these applications. See this topic for more information.



5. DEPLOYING YOUR APPLICATION

“Step 5: Deploy your application” on page 180

“Step 5: Deploy your application” on page 180
After you develop and assemble your application, you can deploy it in your application server. See this topic for more information.

Step 1: Plan

Before you begin designing and developing your application, you should take the time to prepare your development environment and gather information.

- Install development tools on your workstation. WebSphere Application Server - Express for iSeries ships with WebSphere Development Studio Client for iSeries. This tool consolidates development tools for traditional and e-business application development to the Eclipse-based Integrated Development Environment (IDE) WebSphere Studio Workbench. For more information on the WebSphere Development Studio Client, see “Application development tools” on page 3.
- Collect the necessary documentation and resources.
 - WebSphere Application Server - Express for iSeries documentation and the WebSphere Application Server - Express APIs documentation from the WebSphere Application Server - Express for iSeries website



- Sun Microsystems Java Servlet 2.3 API Specification and Javadoc



- Sun Microsystems JavaServer Pages 1.2 Specification



- The JNDI Tutorial: Building Directory-Enabled Java Applications



- Fundamentals of the JavaMail



- JavaServer Pages Technology



- Custom Tags in JSP Pages



- JavaBeans™ 101, Part I



Application development tools

WebSphere Application Server - Express for iSeries ships with WebSphere Development Studio Client for iSeries. The client application is installed on your Windows workstation and interfaces with your iSeries server. With Development Studio Client, you can:

- Develop and maintain iSeries applications using the Remote Systems Explorer
- Develop Web GUIs for iSeries applications using the IBM WebFacing Tool and other Web tools
- Develop client applications for iSeries using the Java tools
- Work with other integrated Site Developer Advanced tools

The following workstation components are included in WebSphere Development Studio Client:

- **IBM WebFacing Tool**
The IBM WebFacing Tool can convert your DDS display source files into an application that runs in the WebSphere Application Server - Express server that you can then access from your browser.
- **Remote System Explorer**
The Remote System Explorer is a programming environment that supports multiple views, editors, tools, and tool extensions.
- **Java development tools**
These tools encompass all the writing, editing, compiling, and debugging functions needed in a complete Java development cycle. Java development tools also contain several iSeries-specific enhancements.
- **Web development tools**
Web development tools encompass all the designing, editing, testing, debugging and publishing functions needed in a Web development cycle. These tools also contain several iSeries-specific enhancements.
- **Database development tools**
Database development tools are included in this product.
- **Web services development tools**
Web services development tools allow you to create modular applications that can be invoked on the World Wide Web.
- **Server development tools**
Server development tools are used to test applications in a local or remotely installed run-time environments.
- **XML development tools**
XML development tools support any XML-based development.
- **CODE**
CODE is the classic set of Windows tools for iSeries development. Its functionality will be subsumed by the Remote Systems Explorer.

- **VisualAge RPG**

VisualAge RPG is a visual development environment that allows you to create and maintain client/server applications on the workstation.

Use these resources for more information on WebSphere Development Studio Client for iSeries:

- **WebSphere Development Studio Client Help**

The WebSphere Development Studio includes detailed Help text.

- **Getting Started with IBM WebSphere Development Studio Client for iSeries**



This document provides overview and getting started information for users new to the WebSphere Development Studio environment.

- **WebSphere Development Studio Client for iSeries product website**



The product website provides resources for learning more about WebSphere Development Studio.

IBM iSeries System Debugger provides a graphical user debugging environment on the iSeries server. Use iSeries System Debugger to debug and test programs that run on your iSeries server. For information on using the iSeries System Debugger to debug your applications, see the iSeries System Debugger topic.

Step 2: Design your application

When you begin to develop actual applications, use the steps below to take advantage of the strengths of each Java component in your applications.

1. Design object-oriented applications to reuse and maintain code.

- Break different types of logic into separate Java classes. Some logical types are:
 - Business processes
 - Data structures and modeling
 - Application flow
 - Database access
 - Web page presentation
- When possible, design complex logical or data structures as collections of simpler Java objects.
- When several processes or data structures share “common denominator” variables or functionality, do the following:
 - a. Group common variables and functions into a base class.
 - b. Extend the base class to create more specialized classes.

2. Design your code to take advantage of the strengths of the Java components.

- Use servlets to perform application logic, call other Java components, and work with Extensible Markup Language (XML).
- Use JavaServer Pages (JSP) files to create dynamic Web pages.

Use these resources for more information on designing web applications:

- Designing Enterprise Applications with the J2EE Platform, Second Edition



This article provides architectural hints and best practices for designing J2EE applications. Note that this article contains information pertaining to enterprise beans, which are not supported by WebSphere Application Server - Express.

- WebSphere Application Server - Express V5.0 Handbook



Chapter 2: *Application design* of this IBM Redbook provides modeling and design consideration for WebSphere Application Server - Express.

Step 3: Develop your application

See these topics for information about developing applications for WebSphere Application Server - Express:

“Classloaders” on page 6

Classloaders are responsible for finding and loading Java class files. Classloaders affect the packaging of applications and the run-time behavior of packaged applications deployed on application servers. See this topic for more information on classloaders for your application.

“Servlets” on page 14

Servlets are Java programs that build dynamic client responses, such as Web pages. Servlets receive and respond to requests from Web clients, usually across HyperText Transfer Protocol (HTTP). See this topic for more information on applications development with servlets.

“JavaServer Pages (JSP)” on page 30

JavaServer Pages technology makes it easier for you to create dynamic Web content while separating business logic from presentation logic. JSP files are comprised of tags (such as HTML tags and special JSP tags) and Java code. WebSphere Application Server - Express generates Java source code for the entire JSP file, compiles the code, and runs the JSP file as if it were a servlet. See this topic for more information on applications development with JSPs.

“Data access” on page 49

JDBC provides uniform access to a wide range of relational databases such as DB2 Universal Database for iSeries. JDBC enables Java programmers to represent database connections, SQL statements and retrieving results in a portable way. JDBC 2.0 has built in support for database connection pooling. See this topic for more information on applications development with data access.

“Java Naming and Directory Interface (JNDI)” on page 100

The Java Naming and Directory Interface, or JNDI, is used to provide access to Java components within a distributed computing environment. JNDI maps names to Java objects (such as data sources) and services (such as mail services). See this topic for more information on JNDI.

“JavaMail” on page 117

The JavaMail APIs model an electronic mail (e-mail) system. WebSphere Application Server - Express supports JavaMail in all Web application components, including servlets and JavaServer Pages (JSP) files. See this topic for more information on JavaMail.

“Sessions” on page 125

WebSphere Application Server - Express for iSeries provides support for HTTP sessions as described by the Java Servlet Specification v2.3. HTTP is by design an stateless protocol. Session tracking attempts to associate HTTP requests emanating from a particular client as belonging to a single HTTP session. See this topic for more information on sessions.

“Bean Scripting Framework” on page 135

The Bean Scripting Framework (BSF) enables you to use scripting language functions in your Java server-side applications. See this topic for more information on the Bean Scripting Framework.

“Internationalization of applications” on page 142

Internationalization is the presentation of information to users in an application according to regional cultural conventions. See this topic for information about internationalizing your applications.

“Add logging and tracing to your application” on page 145

Designers and developers of applications that run with or under WebSphere Application Server - Express, such as servlets and JSP files, may find it useful to use the same facility for generating messages that WebSphere Application Server - Express itself uses, JRas. See this topic for more information on JRas.

Classloaders

Classloaders are responsible for finding and loading Java class files. Classloaders affect the packaging of applications and the run-time behavior of packaged applications deployed on application servers.

For information on configuring classloaders, see these topics:

“Classloader hierarchy”

This topic provides an overview of classloaders and describes how the application server applies different levels of classloaders.

Class preloading

In versions 5.1.1 and later, class preloading enables application servers to start more quickly. This topic provides information about class preloading.

“Classloaders in applications” on page 10

This topic describes how to configure classloader settings when you package your application using the WebSphere administrative console after the application has been deployed.

“Administer classloaders using the WebSphere administrative console” on page 10

This topic describes how to configure classloader settings in the WebSphere administrative console.

Migrate the classloader Module Visibility Mode setting

This topic describes how to migrate from the WebSphere Application Server Version 4 Module Visibility Mode settings to the equivalent Version 5.1 configuration.

“Classloader policies” on page 12

Classloader policies define the number of classloaders and the isolation between application components. This topic describes the classloader policies available, and how they affect the classloaders in which your application runs.

Classloader hierarchy

The run time environment of WebSphere Application Server - Express uses these classloaders to find and load new classes for an application in this order:

1. “Java Virtual Machine classloaders” on page 7

The Java Virtual Machine classloaders are provided by the Java runtime. JDK 1.3 provides 3 classloaders: boot classloader, extensions classloader and user classloader.

2. “WebSphere extensions classloader” on page 9

The WebSphere Application Server - Express run time is loaded by the WebSphere extensions classloader.

3. Application classloaders

To find and load classes for an application, WebSphere Application Server - Express for iSeries uses one or more application classloaders that load elements of enterprise applications running in the server.

The application elements can be Web modules, resource adapters, and dependency JAR files. Application classloaders follow J2EE class loading rules to load classes and JAR files from an enterprise application.

Each classloader is a child of the classloader above it. That is, the application module classloaders are children of the WebSphere-specific extensions classloader, which is a child of the CLASSPATH Java classloader. Whenever a class needs to be loaded, delegation depends on the delegation mode setting, which always defaults to PARENT_FIRST, but can be changed to PARENT_LAST. If none of the parent classloaders can find the class, the original classloader attempts to load the class. Requests can only go to a parent classloader; they cannot go to a child classloader. For example, if a class in the WebSphere extension classloader tried to reference a class in an application classloader, it would not be found. After a class is loaded by a classloader, any new classes that it tries to load reuse the same classloader or go up the precedence list until the class is found.

The i5/OS Java Virtual Machine can run in two modes — JIT (just in time compiler) and DE (direct execution). For more information on how these modes can be used by the different classloaders, see “**Java execution modes**” on page 10.

Java Virtual Machine classloaders: The Java Virtual Machine classloaders, also called the system classloader, are provided by the Java Virtual Machine run time. In JDK 1.3, there are three classloaders, which have the following hierarchy:

1. **boot classloader**
Loads the core JDK 1.3 runtime classes.
2. **extension classloader**
Loads Java Virtual Machine extensions. The classes loaded are defined by the value of the java.ext.dirs Java system property.
3. **user classloader**
Loads user classes. The classes loaded are defined by the value of the CLASSPATH environment variable or the -classpath Java option.

The user classloader contains the J2EE APIs of WebSphere Application Server - Express (inside j2ee.jar). Because the J2EE APIs are in this classloader, you can add libraries that depend on J2EE APIs to the classpath system property to extend a server’s classpath. However, this is not recommended. The normal and recommended configuration is to package J2EE artifacts in an enterprise application (EAR) and have these classes loaded by an application module classloader.

It is normally not recommended to change any of the Java Virtual Machine classloaders in a WebSphere Application Server - Express environment. User code should be added to either application server classloaders or application module classloaders.

The i5/OS Java Virtual Machine user classloader has a cache feature that allows the Java Virtual Machine to “remember” classes that have been loaded with user classloaders. For more information, see “**Java cache for user classloaders**”.

Java cache for user classloaders: The i5/OS Java Virtual Machine user classloader cache is a feature that allows the Java Virtual Machine to “remember” classes that have been loaded with user classloaders. This feature improves the startup performance of classes loaded by user class loaders by allowing the JVAPGMs created by user classloaders to be cached for reuse, avoiding JVAPGM creation and bytecode verification during the initial class load. WebSphere components (servlets and JSPs) are loaded by user classloaders and can take advantage of this feature.

For most applications, the default WebSphere setup provides the best solution. You should only consider the techniques described here if you are experiencing performance problems in the following areas:

- **Application server startup**
Servlets configured to load at startup are loaded when the application server is started.

Having a large number of these components, or complex components that access many classes in your application, makes application server startup time longer.

- **Component runtime during first-touch**

These components are loaded the first time they are accessed after the application server is started:

- servlets that are not configured to load at startup
- JSPs

If these components are complex or access many classes in your application, your first-touch times of these components are longer.

The user classloader cache improves performance in two ways:

- Avoiding bytecode verification
If the class is already in the cache, bytecode verification isn't performed again.
- Avoiding creation of JVAPGMs
If the class is already in the cache, the existing JVAPGM is used. Since any optimization level can be stored in the cache, it is more practical to consider higher optimization levels than previously.

Note in both cases the first time the class is loaded (for example before the cache is primed or after a class is changed), these functions are performed and the load is slower. Once the class is already in the cache, these functions are not performed, so subsequent loads are much quicker. There is no way to prime the cache with classes other than running your application.

Using the User Classloader Cache

To enable the user classloader cache, the Java system property **os400.define.class.cache.file** must be specified with a valid value. Other Java system properties may be optionally specified. Use the following Java system properties to enable and customize the user classloader cache:

- **os400.define.class.cache.file**

This property must be specified to enable the Java user classloader cache function. The value specifies the full path name of a valid JAR file that holds the Java Program objects (JVAPGMs). This JAR file must contain a valid JAR entry, but need have no other content beyond the single member required to make the JAR command function. Here are two ways to create a cache JAR file.

- V5R1 PTF 5722JV1 SI02683 installs a suitable JAR file, /QIBM/ProdData/Java400/QDefineClassCache.jar. You may use this JAR file as is, or copy and rename as desired.
- Create your own cache JAR file as follows:
 1. Start Qshell by entering the command STRQSH.
 2. Switch to the directory where you want the JAR file located. Make the directory first, if necessary.

```
mkdir /cache
cd /cache
```
 3. Create a dummy file to place in the JAR. The name can be anything, this example uses example.

```
touch example
```
 4. Build the JAR file. This example names the JAR file MyAppCache.jar

```
jar -cf MyAppCache.jar example
```
 5. Cleanup the dummy file

```
rm example
```

The value of the **os400.define.class.cache.file** property would be `/cache/MyAppCache.jar`.

This JAR file must not be on any classpath.

DSPJVAPGM can be used on this JAR file to determine how many JVAPGMs are cached. The Java programs field of the DSPJVAPGM display indicates how many JVAPGMs are cached, and the Java

program size field indicates how much storage is consumed by cached JVAPGMs. Other fields of the display are meaningless when DSPJVAPGM is applied to a JAR file used for caching.

CHGJVAPGM can be used on this JAR file to change the optimization of the classes in the cache.

CHGJVAPGM only affects programs currently in the cache. Classes added to the cache are optimized according to the other properties below.

- **os400.define.class.cache.hours**

Specifies how long (in hours) an unused JVAPGM persists in the cache. When a JVAPGM has not been used and this timeout is reached, the JVAPGM is removed from the cache. The default value is 168 (one week). The maximum value is 9999 (about 59 weeks).

- **os400.define.class.cache.maxpgms**

Specifies the maximum number of JVAPGMs the cache can hold. If this value is reached, the least recently used JVAPGMs is replaced first. The default value is 5000. The maximum value is 40000.

Note that other Java system properties, such as `os400.defineClass.optLevel`, can be used to customize how JVAPGMs are created in the cache.

To add Java system properties to an application server, perform these steps:

1. In the WebSphere administrative console click **Servers** —> **Application Servers** —> *server_name*.
2. In the **Additional properties** section, click **Process Definition**.
3. In the **Additional properties** section, click **Java Virtual machine**.
4. In the **Additional properties** section, click **Custom Properties**.
5. Click **New** to add a property.

For example, to use the shipped cache JAR with a maximum of 10,000 JVAPGMs that have a maximum life of 1 year, you would add the following:

| Property | Value |
|---|---|
| <code>os400.define.class.cache.file</code> | <code>/QIBM/ProdData/Java400/QDefineClassCache.jar</code> |
| <code>os400.define.class.cache.hours</code> | 8760 |
| <code>os400.define.class.cache.maxpgms</code> | 10000 |

WebSphere extensions classloader: The WebSphere extensions classloader loads the WebSphere run time. The WebSphere extensions classloader also loads resource provider classes into a server if an application module installed on the server refers to a resource that is associated with the provider and if the provider specifies the directory name of the resource drivers.

The WebSphere run time classloader loads the WebSphere run time (infrastructure). The classloader contents are defined by the `ws.ext.dirs` property, which should not be changed. Classes are loaded from the following locations, in the order listed:

1. `/QIBM/ProdData/WebASE51/ASE/classes`
This directory is used to provide iSeries changes to the core WebSphere Application Server - Express run time.
2. `/QIBM/ProdData/WebASE51/ASE/lib` and `/QIBM/ProdData/WebASE51/ASE/lib/server`
These directories contain the core WebSphere Application Server - Express run time.
3. `/QIBM/ProdData/WebASE51/ASE/lib/ext`
This directory contains extensions to the core WebSphere Application Server - Express run time.
4. `/QIBM/UserData/WebASE51/ASE/instance/lib/ext`, where *instance* is the name of your WebSphere Application Server - Express instance
This directory is maintained for compatibility with WebSphere Application Server Version 4. Utility libraries placed here can make use of the i5/OS Java Virtual Machine's direct execution mode. For more information on the direct execution mode, see "Java execution modes" on page 10.

Java execution modes: Because all WebSphere components (servlets and JSPs) are loaded by an application server classloader, application classloader, and application module classloader, WebSphere components do not use the direct execution (DE) capabilities of the i5/OS Java Virtual Machine. Java program (JVAPGM) objects are not used when running WebSphere component code. Classes loaded by a Java Virtual Machine classloader or the WebSphere extension classloader use DE capabilities and JVAPGM objects. Java user classloader caching can be used to allow WebSphere components to use permanent JVAPGM objects and DE capabilities, though this is not necessary for the vast majority of applications.

By default, application servers run with the Java system property `java.compiler=jitc_de`. The `jitc_de` value means that just-in-time compilation is done for any Java object for which a JVAPGM object does not exist (or cannot be used, in the case of WebSphere application classloaders). This setting provides the best overall performance.

It is possible to change application servers to use direct execution instead of JIT mode. Doing so results in longer startup times, because creation of the JVAPGM takes longer than creation of the JIT stubs. Due to performance improvements in the JIT run time, performance of direct execution even after JVAPGM creation is probably slower also. To make an application server use direct execution for all classes, perform these steps:

1. In the WebSphere administrative console click **Servers** —> **Application Servers** —> *server_name*.
2. In the **Additional properties** section, click **Process Definition**.
3. In the **Additional properties** section, click **Java Virtual machine**.
4. In the **Additional properties** section, click **Custom Properties**.
5. Click **New** to add a property. Add these properties:
 - Property `java.compiler` with value `NONE`
 - Property `os400.defineClass.optLevel` with a value that indicates the optimization level desired (values are 0 for interpret, and 10, 20, 30, or 40 for direct execution optimization levels)

It is also possible to have an application server use the interpreter only (no JIT). To make an application server use full interpretation for all WebSphere components, add Java system property `java.compiler` with value `NONE`.

Classloaders in applications

The classloaders used to load your application classes are affected by these application-level settings:

- When assembling an enterprise application resource (EAR) file, use the **Classpath** field for classes used by the application modules that are packaged in the EAR but not in the WAR files which reference them. Items in the **Classpath** field are considered dependencies and are loaded by the application classloader.
- After installing an enterprise application (EAR), set the **Classloader Mode** and **WAR classloader** policy using the WebSphere administrative console.
- After installing an enterprise application (EAR) that has web modules, set the **Classloader Mode** using the WebSphere administrative console.

Administer classloaders using the WebSphere administrative console

Classloader modes

There are two possible values for a classloader mode. These values can be changed using the WebSphere administrative console.

- **PARENT_FIRST**

The `PARENT_FIRST` classloader mode causes the classloader to first delegate the loading of classes to its parent classloader (the classloader up one level in the classloader hierarchy) before attempting to

load the class from its local classpath. This is the default classloader policy for all classloaders. This default can be changed for the classloaders supplied by the WebSphere run time. It can not be changed for the Java Virtual Machine classloaders.

- **PARENT_LAST**

The PARENT_LAST classloader mode causes the classloader (the classloader up one level in the classloader hierarchy) to first attempt to load classes from its local classpath before delegating the classloading to its parent. This policy allows an application classloader to override and provide its own version of a class that exists in the parent classloader.

There are three Classloader mode settings:

- on an application server
- on an application
- on a web module

There are several classloader policy and classloader mode settings in the WebSphere administrative console. See the topics below for more information:

Setting classloaders in application servers

Click **Servers** → **Application Servers** → *server_name* and, on the settings page for an application server, set the application classloader policy and application class loading mode.

- The **Application classloader policy** controls the isolation of applications running in the system. When set to SINGLE, applications are not isolated; a single application classloader is used to contain all dependency JAR files in the system. When set to MULTIPLE, applications are isolated from each other; each application receives its own classloader to load that application's dependency JAR files.
- The **Application classloader mode** specifies the classloader mode when the application classloader policy is SINGLE. This field is not used if the application classloader policy is MULTIPLE. PARENT_FIRST causes the classloader to first delegate the loading of classes to its parent classloader before attempting to load the class from its local classpath. PARENT_LAST causes the classloader to first attempt to load classes from its local classpath before delegating the classloading to its parent. This allows an application classloader to override and provide its own version of a class that exists in the parent classloader.

Setting classloaders in an application

Click **Applications** → **Enterprise Applications** → *application_name* and, on the settings page for an application server, set the WAR classloader policy and application classloading mode.

- The **WAR classloader policy** specifies whether to use the application classloader to load all WAR files of this application or to use a separate classloader for each WAR file. By default, Web module classloaders load the contents of the WEB-INF/classes and WEB-INF/lib directories. The application classloader is the parent of the Web module classloader. You can change the default behavior by changing the application's WAR classloader policy.

The WAR classloader policy controls the isolation of Web modules. If this policy is set to APPLICATION, then the Web module contents also are loaded by the application classloader (in addition to the RAR files and dependency JAR files). If the policy is set to MODULE, then each web module receives its own classloader whose parent is the application classloader.

- The **application classloading mode** specifies whether the classloader searches in the parent classloader or in the application classloader first to load a class. The standard for JDK classloaders and WebSphere Application Server classloaders is PARENT_FIRST. By specifying PARENT_LAST, your application can override classes contained in the parent classloader, but this action can potentially result in ClassCastException or LinkageErrors if you have mixed use of overridden classes and non-overridden classes.

The options are PARENT_FIRST and PARENT_LAST. The default is to search in the parent classloader before searching in the application classloader to load a class.

Setting classloaders in a Web module

Click **Application** — > **Enterprise Application** —> *application_instance* —> **Web Module** —> *Web Module_instance* and, on the settings page for an Web module, set the web module classloader mode.

- The **Web module classloader mode** Specifies whether the classloader should search in the parent classloader or in the application classloader first to load a class. The standard for JDK classloaders and WebSphere classloaders is PARENT_FIRST. By specifying PARENT_LAST, your application can override classes contained in the parent classloader, but this action can potentially result in ClassCastException or LinkageErrors if you have mixed use of overridden classes and non-overridden classes.

The options are PARENT_FIRST and PARENT_LAST. The default is to search in the parent classloader before searching in the application classloader to load a class.

ClassLoader policies

The number and function of the application and application module classloaders depends on the classloader policies specified in the server and application configuration. Classloaders provide multiple options for isolating applications and modules to enable different application packaging schemes to run on an application server.

Two classloader policies control the isolation of applications and modules:

- **Application classloader policy**

Application classloaders consist of dependency JAR files and resource adapters. Depending on the application classloader policy, an application classloader can be shared by multiple applications (SINGLE) or unique for each application (MULTIPLE). The application classloader policy controls the isolation of applications running in the application server. When set to SINGLE, applications are not isolated. When set to MULTIPLE, applications are isolated from each other.

To change the application classloader policy, perform these steps in the WebSphere administrative console:

1. Click **Servers** — > **Application servers** —> *server_name*.
2. Select the policy you want to use from the **Application classloader policy** drop-down box.
3. Click **Apply** and **Save**.

- **WAR classloader policy**

By default, Web module classloaders load the contents of the WEB-INF/classes and WEB-INF/lib directories. The application classloader is the parent of the Web module classloader. You can change the default behavior by changing the application's WAR classloader policy.

The WAR classloader policy controls the isolation of Web modules. If this policy is set to APPLICATION, then the Web module contents also are loaded by the application classloader (in addition to the RAR files and dependency JAR files). If the policy is set to MODULE, then each web module receives its own classloader whose parent is the application classloader.

To change the application's WAR classloader policy, perform these steps in the WebSphere administrative console:

1. Click **Applications** — > **Enterprise applications** —> *application_name*.
2. Select the policy you want to use from the **WAR classloader policy** drop-down box.
3. Click **Apply** and **Save**.

Note: Application client modules are not loaded by application classloaders.

Example scenarios

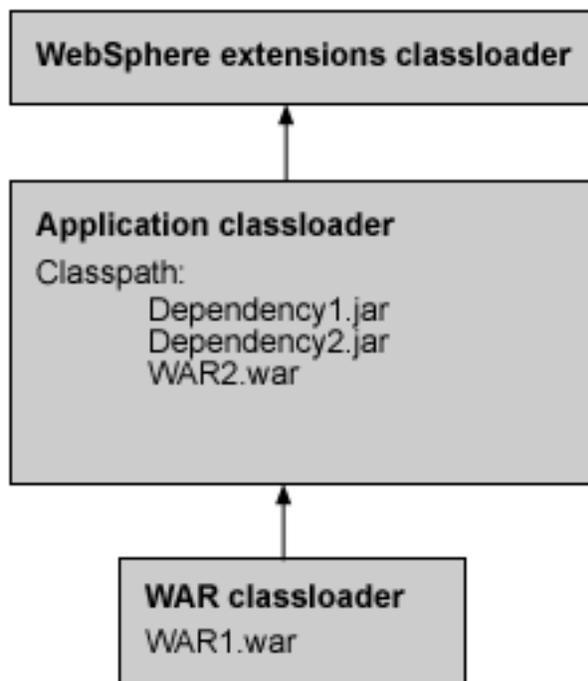
For each application server in the system, you can set the application classloader policy to SINGLE or MULTIPLE. When the application classloader policy is set to SINGLE, then a single application classloader loads all dependency JAR files in the system. When the application classloader policy is set to MULTIPLE, then each application receives its own classloader used for loading that application's dependency JAR files.

This application classloader can load each application's Web modules if that WAR module's classloader policy is also set to APPLICATION. If the WAR module's classloader policy is set to APPLICATION, then the application's loader loads the WAR module's classes. If the WAR classloader policy is set to MODULE, then each WAR module receives its own classloader.

This example shows that when the application classloader policy is set to SINGLE, a single application classloader loads all dependency JAR files of all applications on the server. The single application classloader can also load Web modules if an application has its WAR classloader policy set to APPLICATION. Applications having a WAR classloader policy set to MODULE use a separate classloader for Web modules.

Application classloader policy: SINGLE

```
Application 1
Module: WAR1.war
  MANIFEST Class-Path: Dependency1.jar
  WAR Classloader Policy = MODULE
Application 2
Module: WAR2.war
  WAR Classloader Policy = APPLICATION
  MANIFEST Class-Path: Dependency2.jar
```



This example shows that when the application classloader policy of an application server is set to MULTIPLE, each application on the server has its own classloader. An application classloader also loads its Web modules if the application's WAR classloader policy is set to APPLICATION. If the policy is set to MODULE, then a Web module uses its own classloader.

Application classloader policy: MULTIPLE

Application 1

Module: WAR1.war

MANIFEST Class-Path: Dependency1.jar

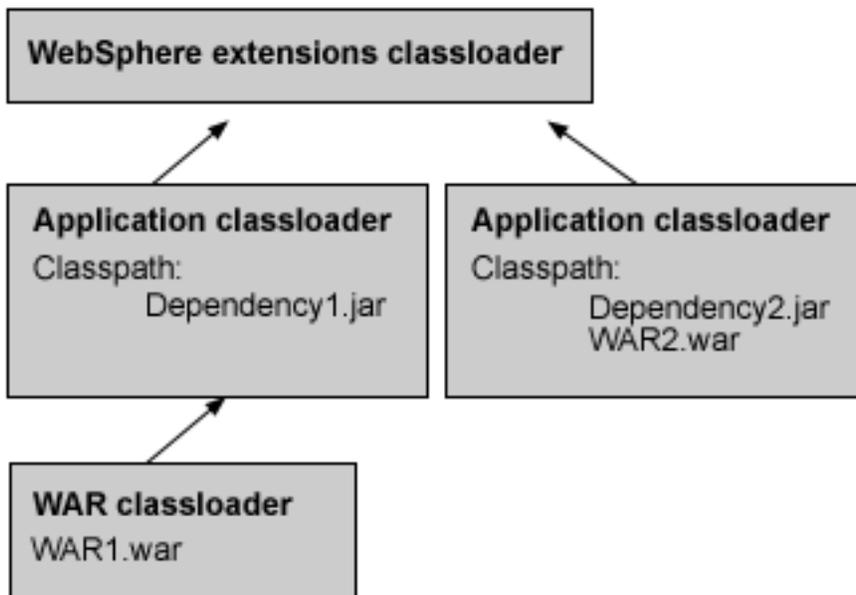
WAR Classloader Policy = MODULE

Application 2

Module: WAR2.war

WAR Classloader Policy = APPLICATION

MANIFEST Class-Path: Dependency2.jar



Servlets

WebSphere Application Server - Express for iSeries allows you to deploy Java servlets. Servlets are Java programs that build dynamic client responses, such as Web pages. Servlets receive and respond to requests from Web clients, usually across HyperText Transfer Protocol (HTTP).

As long as servlets adhere to the Java Servlet API, they can be ported without modification to different operating systems or application servers. Servlets are more efficient than CGI programs because, unlike CGI programs, servlets are loaded into memory once, and each request is handled by a Java virtual machine thread, not an operating system process. Moreover, servlets are scalable, providing support for a multi-application server configuration. Servlets also allow you to cache data, access database information, and share data with other servlets, JSP files, and (in some environments) enterprise beans.

WebSphere Application Server - Express supports the Java Servlet API 2.3. WebSphere Application Server - Express also includes IBM extensions to the Java Servlet API.

See these topics for more information about developing servlets for WebSphere Application Server-Express:

“Servlet lifecycle” on page 15

This topic describes the lifecycle of a typical servlet.

“Create a servlet” on page 17

See this topic for step-by-step instructions on how to write, assemble, and test your own servlets.

“Application lifecycle listeners and events” on page 22

This topic provides an overview of classes and interfaces you can use to monitor session contexts and session change.

“Servlet filtering” on page 23

This topic describes how to filter HTTP responses by MIME-type and how to chain a series of servlets together.

“Page lists” on page 25

The PageListServlet allows you to call a JSP file by name from within your servlet code. See this topic for more information.

“Automatic request and response encoding” on page 27

WebSphere Application Server - Express provides extensions that enable the application server to set encoding values and content type. See this topic for information about the `autoRequestEncoding` and `autoResponseEncoding` extensions.

“Enhanced error reporting” on page 28

You can enhance error reporting in your Web application to provide more detailed and tailored messages to the client. See this topic for more information.

“Internal servlets” on page 29

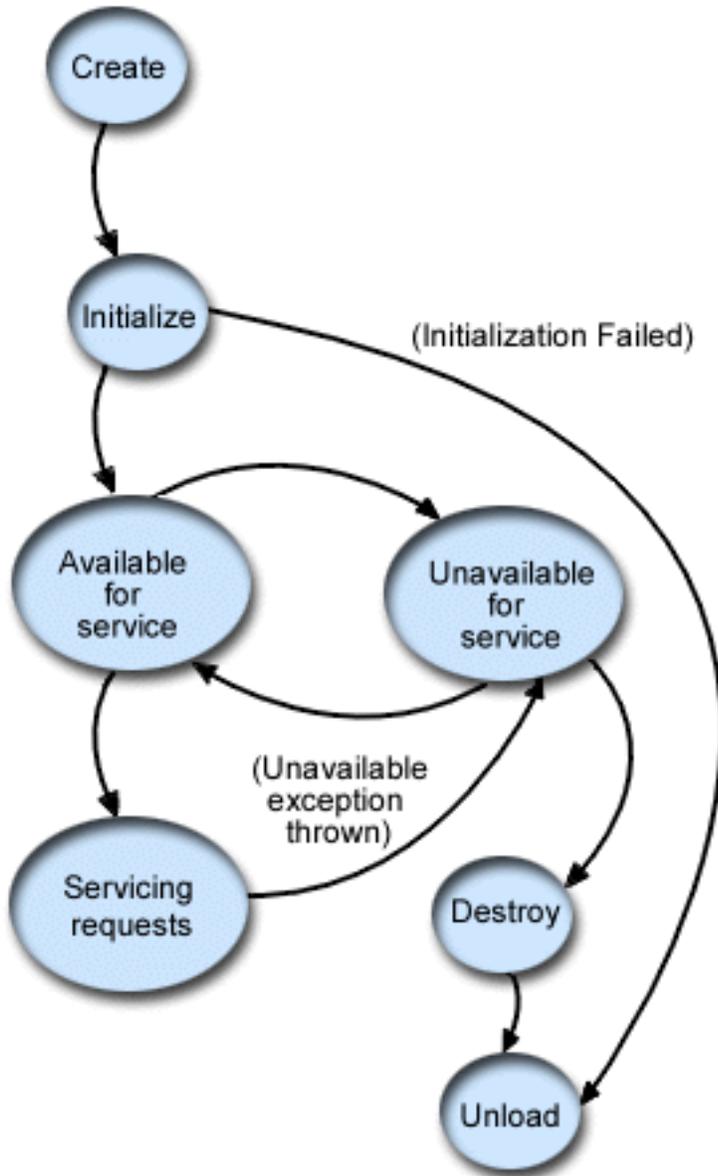
WebSphere Application Server - Express ships with certain internal servlets that you can use in your Web modules. See this topic for a list of the servlets and their functions.

“Servlet resources” on page 29

This topic contains links to other servlet resources. If you are new to servlet programming or are looking for more information about servlets, refer to these links for details about the Java Servlet APIs.

Servlet lifecycle

The lifecycle of a servlet begins when it is loaded into Application Server memory and ends when the servlet is terminated or reloaded.



Instantiation and initialization

The servlet engine (the WebSphere Application Server - Express function that processes servlets and JSP files) creates an instance of the servlet. The servlet engine creates the servlet configuration object and uses it to pass the servlet initialization parameters to the `init()` method. The initialization parameters persist until the servlet is destroyed and are applied to all invocations of that servlet.

If the initialization is successful, the servlet is available for service. If the initialization fails, the servlet engine unloads the servlet. The WebSphere Application Server - Express administrator can set a Web application and its servlets to be unavailable for service. In such cases, the Web application and servlet remain unavailable until the administrator changes them to be available.

Servicing requests

WebSphere Application Server - Express receives a client request. The servlet engine creates a request object and a response object. The servlet engine invokes the servlet service() method, passing the request and response objects.

The service() method gets information about the request from the request object, processes the request, and uses methods of the response object to create the client response. The service method can invoke other methods to process the request, such as doGet(), doPost(), or methods you write.

Termination

The servlet engine stops a servlet by invoking the servlet's destroy() method. Typically, a servlet's destroy() method is invoked when the servlet engine is stopping a Web application which contains the servlet. The destroy() method runs only one time during the lifetime of the servlet and signals the end of the servlet.

After a servlet's destroy() method is invoked, the servlet engine unloads the servlet, and the Java virtual machine eventually performs garbage collection on the memory resources associated with the servlet.

Create a servlet

In this topic, we feature how to create a sample servlet ("ServletSample.java"), from start to finish. We explain the sample code and give you pointers on how to develop your own servlets.

While the details of the steps below are specific to the ServletSample servlet code, you can use the information in this section as an example of how to develop your own servlets.

To create the ServletSample servlet, perform these steps:

"Write a servlet" on page 18

Follow along as we write the ServletSample servlet and give you tips for writing your own servlets.

"Compile the servlet" on page 21

Use Qshell to compile the sample servlet and your own servlets.

Step 3: Package and deploy an application

Use the WebSphere Development Studio Client to package compiled code into a Web module before you install it on the server and to create a deployment descriptor (web.xml) file. For more information, see the WebSphere Development Studio Client.

"Testing a servlet" on page 22

Run the ServletSample to make sure it works.

ServletSample.java: // ServletSample.java

```
// Step 1: Add the necessary import statements.
```

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

```
// Step 2: Extend HttpServlet.
```

```
public class ServletSample extends HttpServlet
{
```

```
// Step 3: Specify the required methods.
```

```
public void doGet (HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
```

```

{
    // Step 4: Get the HTTP request information, if any.

    Enumeration keys;
    String key;
    String myName = "";
    keys = request.getParameterNames();
    while (keys.hasMoreElements())
    {
        key = (String) keys.nextElement();
        if (key.equalsIgnoreCase("myName")) myName = request.getParameter(key);
    }
    System.out.println("Name = ");
    if (myName == "") myName = "Hello";

    // Step 5: Create the HTTP response.

    response.setContentType("text/html");
    response.setHeader("Pragma", "No-cache");
    response.setDateHeader("Expires", 0);
    response.setHeader("Cache-Control", "no-cache");

    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("<head><title>Just a basic servlet</title></head>");
    out.println("<body>");
    out.println("<h1>Just a basic servlet</h1>");
    out.println ("<p>" + myName + ", this is a very basic servlet.");
    out.println("</body></html>");
    out.flush();
}
}

```

Write a servlet: This section shows you step-by-step how to write a sample servlet called “ServletSample.java” on page 17. Each step is included with reference information about servlet methods and syntax and a description of how each piece of code works.

To write the ServletSample servlet, perform these steps:

1. Open your programming editor to edit a new file to be called ServletSample.java. For more information, see the WebSphere Development Studio Client Help.
2. “Enter the servlet import statements.”
3. “Extend the HttpServlet class” on page 19.
4. “Write the required servlet methods” on page 19.
5. “Get the HTTP request information” on page 20.
6. “Create the HTTP response” on page 20.

Enter the servlet import statements: Type these import statements:

```

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

```

Import statements provide a way to shorten Java[™] class names. For example, the example servlet code used elsewhere in this section uses the java.io.PrintWriter class. When you import the java.io package, you can use the short name of the class, PrintWriter, in your code.

The java.io and java.util package are standard packages in the core Java platform. However, the two other packages, javax.servlet and javax.servlet.http, are specific to the Servlet API and the Java 2 Enterprise

Edition platform. The `javax.servlet` package provides the general classes and interfaces for servlets. The `javax.servlet.http` package provides HTTP-specific classes and interfaces for servlets.

Extend the `HttpServlet` class: After the import statements, type the class declaration (in black text below):

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

public class ServletSample extends HttpServlet
{
}
}
```

The example, `ServletSample`, extends the `javax.servlet.http.HttpServlet` class of the `javax.servlet.http` package. Java requires the declared classname to match the name of the java file `ServletSample.java`. The `HttpServlet` class, which is a subclass of `GenericServlet`, provides specialized methods for handling HTML forms. HTTP servlets enable you to send and receive data using an HTML form.

HTML forms are defined by the `<FORM>` and `</FORM>` tags. The forms typically include input fields (such as text entry fields, check boxes, radio buttons, and selection lists) and a button to submit the data. They also specify which program or servlet the server should run when the information is submitted.

Write the required servlet methods: After the class declaration, add the method declaration (in black text below):

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

public class ServletSample extends HttpServlet
{
    public void doGet (HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
    }
}
}
```

The important parts of the above code snippet are:

- **The `doGet()` method**

Classes that extend the `HttpServlet` class should override at least one method of the `HttpServlet` class. `SimpleServlet` overrides the `doGet()` method.

Note: Since `doGet` and `doPost` throw two exceptions (`javax.servlet.ServletException` and `java.io.IOException`), you must include them in the declaration.

- **`HttpServletRequest` and `HttpServletResponse`**

When the server calls an HTTP servlet's `service()` method, it passes the following two objects as parameters:

- **`HttpServletRequest`: the request object**

`HttpServletRequest` represents a client's request. This object gives a servlet access to incoming information such as HTML form data and HTTP request headers.

- **`HttpServletResponse`: the response object**

`HttpServletResponse` represents the servlet's response. The servlet uses this object to return data to the client such as HTTP errors (200, 404, and others), response headers (`Content-Type`, `Set-Cookie`, and others), and output data by writing to the response's output stream or output writer.

The servlet communicates with the HTTP server and ultimately with the client through these objects. The servlet can invoke the `HttpServletRequest` object's (request) methods to get information about the client environment, the HTTP server environment, and any information provided by the client (for example, HTML form information set by GET or POST).

The servlet invokes the `HttpServletResponse` object's (response) methods to send the response that it has prepared back to the client. These same objects are also passed to the `service()` method.

Get the HTTP request information: After the method declaration, add the request code (in black text below):

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

public class ServletSample extends HttpServlet
{

    public void doGet (HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {

        Enumeration keys;
        String key;
        String myName = "";
        keys = request.getParameterNames();
        while (keys.hasMoreElements())
        {
            key = (String) keys.nextElement();
            if (key.equalsIgnoreCase("myName")) myName = request.getParameter(key);
        }
        System.out.println("Name = ");
        if (myName == "") myName = "Hello";
    }
}
```

This new section of code handles the client request parameters. The `HttpServletRequest` class has the following specific methods to retrieve information provided by the client:

- **getParameterNames()**
This method returns the names of the parameters in an enumeration of strings.
- **getParameterValues()**
This method returns the values of the parameters in an array of strings.
- **getParameter()**
This method returns a single parameter value as a string. Use this method when you know the client request returns only one parameter.

In the case of `ServletSample`, the `getParameterNames()` method gets the name of the parameters, and the `getParameter()` method gets the value of the `myName` parameter.

Create the HTTP response: After the servlet request code, add the response code (in black text below):

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

public class ServletSample extends HttpServlet
{

    public void doGet (HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
```

```

{
    Enumeration keys;
    String key;
    String myName = "";
    keys = request.getParameterNames();
    while (keys.hasMoreElements())
    {
        key = (String) keys.nextElement();
        if (key.equalsIgnoreCase("myName")) myName = request.getParameter(key);
    }
    System.out.println("Name = ");
    if (myName == "") myName = "Hello";

    response.setContentType("text/html");
    response.setHeader("Pragma", "No-cache");
    response.setDateHeader("Expires", 0);
    response.setHeader("Cache-Control", "no-cache");

    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("<head><title>Just a basic servlet</title></head>");
    out.println("<body>");
    out.println("<h1>Just a basic servlet</h1>");
    out.println("<p>" + myName + ", this is a very basic servlet.");
    out.println("</body></html>");
    out.flush();
}
}

```

The `HttpServletResponse` object generates a response to return to the requesting client. Its methods allow you to set the response header and the response body.

The first line of the Response header (`response.setContentType("text/html");`) identifies the MIME type of the response. The following three lines are often placed in servlet code to prevent Web browsers and proxy servers from caching dynamically-generated Web pages. If you want your dynamic Web page to be cached, remove these three lines of code.

The response object also has the `getWriter()` method to return a `PrintWriter` object. The `print()` and `println()` methods of the `PrintWriter` object write the servlet response back to the client.

Compile the servlet: After you have written the source code (.java file) for your servlet, you must compile it into a .class file before you can package it for and deploy it on your iSeries server.

Compiling code on the iSeries server

Perform these steps to compile the `ServletSample` code:

1. Start Qshell. On the iSeries command line, type `STRQSH` and press the **Enter** key. The **QSH Command Entry** screen displays. When the `$` prompt displays, Qshell is ready for commands.
2. Use the `cd` command to change the current directory to the directory that holds your servlet source code. On the Qshell command line, type:

```
cd /path_to_mydir
```

where `/path_to_mydir` is the fully qualified path name of your directory, and press **Enter**.

3. Compile your servlet.

On the QSH command line, type:

```
javac my_servlet_name.java
```

where `my_servlet_name` is the name of your servlet, and press the **Enter** key.

Alternatively, you can compile your servlet code using WebSphere Development Studio Client. For more information, see the WebSphere Development Studio Client Help.

Testing a servlet: A quick and easy way to test the ServletSample is to invoke it by URL in your browser. The ServletSample includes a parameter, *myname*, that should be included in the URL.

Perform these steps to invoke ServletSample:

1. Make sure your HTTP Server for i5/OS instance is running. As another option, you can use the HTTP Server internal to WebSphere Application Server - Express.
2. If your server instance is not running, use the HTTP Server Administration interface to start it. For more information, see Start and test a new WebSphere Application Server - Express application server in the *Administration* topic.
3. Go to: `http://your.server.name/servlet/ServletSample?myname=yourname`, where *your.server.name* is your Web server domain name, and where *yourname* is your first name. Press **Enter**.

Note: If your HTTP server instance does not use the default port, specify the port number. For example, *your.server.name:8000*

In your browser, a page that is titled Just a basic servlet displays. Under the heading is a line that reads *Your name*, this is a very basic servlet., where *Your name* is the value you typed at the end of the URL in the previous step.

Application lifecycle listeners and events

Application lifecycle listeners and events, now part of the Servlet API, enable you to notify interested listeners when servlet contexts and sessions change. For example, you can notify users when attributes change and if sessions or servlet contexts are created or destroyed.

The lifecycle listeners give the application developer greater control over interactions with ServletContext and HttpSession objects. Servlet context listeners manage resources at an application level. Session listeners manage resources associated with a series of requests from a single client. Listeners are available for lifecycle events and for attribute modification events. The listener developer creates a class that implements the javax listener interface, corresponding to the desired listener functionality.

At application startup time, the container uses introspection to create an instance of your listener class and registers it with the appropriate event generator.

When a servlet context is created, the contextInitialized method of your listener class is invoked, which creates the database connection for the servlets in your application to use, if this context is for your application.

When the servlet context is destroyed, your contextDestroyed method is invoked, which releases the database connection, if this context is for your application.

Listener classes for servlet context and session changes

The Servlet API provides these interfaces and classes:

- Interface javax.servlet.ServletContextListener
- Interface javax.servlet.ServletContextAttributeListener
- Interface javax.servlet.FilterChain
- Class javax.servlet.ServletContextEvent
- Class javax.servlet.ServletContextAttributeEvent
- Interface javax.servlet.http.HttpSessionListener
- Interface javax.servlet.http.HttpSessionAttributeListener
- Interface javax.servlet.http.HttpSessionActivationListener
- Class javax.servlet.http.HttpSessionEvent

The following methods are defined as part of the `javax.servlet.ServletContextListener` interface:

- `void contextInitialized(ServletContextEvent)`
This method provides notification that the Web application is ready to process requests. Place code in this method to see if the created context is for your Web application, and if it is, allocate a database connection and store the connection in the servlet context.
- `void contextDestroyed(ServletContextEvent)`
This method provides notification that the servlet context is about to shut down. Place code in this method to see if the created context is for your Web application, and if it is, close the database connection stored in the servlet context.

Example: `com.ibm.websphere.DBConnectionListener.java`

The following example shows how to create a servlet context listener:

```
package com.ibm.websphere;

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

public class DBConnectionListener implements ServletContextListener {
    // implement the required context init method
    void contextInitialized(ServletContextEvent sce) {
        ...
    }

    // implement the required context destroy method
    void contextDestroyed(ServletContextEvent sce) {
        ...
    }
}
```

Servlet filtering

Servlet filtering is an integral part of the Servlet API. Servlet filtering provides a new type of object called a filter that can transform a request or modify a response.

You can chain filters together so that a group of filters can act on the input and output of a specified resource or group of resources.

Filters typically include logging filters, image conversion filters, encryption filters, and Multipurpose Internet Mail Extensions (MIME) type filters, which are functionally equivalent to the servlet chaining. Although filters are not servlets, their lifecycle is very similar.

Filters are handled in the following manner:

- The Web container determines whether it needs to construct a `FilterChain` that contains the `LoggingFilter` for the requested resource. The `FilterChain` begins with the invocation of the `LoggingFilter` and ends with the invocation of the requested resource.
- If other filters need to go in the chain, the Web container places them after the `LoggingFilter` and before the requested resource.
- The Web container then instantiates and initializes the `LoggingFilter` (if it was not done previously) and invokes its `doFilter(FilterConfig)` method to start the chain.
- The `LoggingFilter` preprocesses the request and response objects and then invokes the filter chain `doFilter(ServletRequest, ServletResponse)` method. This method passes the processing to the next resource in the chain (in this case, the requested resource).
- Upon return from the filter chain `doFilter(ServletRequest, ServletResponse)` method, the `LoggingFilter` performs post-processing on the request and response object before sending the response back to the client.

Filter, `FilterChain`, `FilterConfig` classes for servlet filtering

The following interfaces are defined as part of the javax.servlet package:

- Filter interface: doFilter(), getFilterConfig(), and setFilterConfig() methods
- FilterChain interface: doFilter() method
- FilterConfig interface: getFilterName(), getInitParameter(), getInitParameterNames(), and getServletContext() methods

The following classes are defined as part of the javax.servlet.http package (see the J2EE 1.3 documentation for information about methods):

- HttpServletRequestWrapper



- HttpServletResponseWrapper



Example: com.ibm.websphere.LoggingFilter.java

The following example shows how to implement a filter:

```
package com.ibm.websphere;

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

public class LoggingFilter implements Filter {

    File _loggingFile = null;

    // implement the required init method
    public void init(FilterConfig fc) {

        // create the logging file
        ...
    }

    // implement the required doFilter method.
    // this is where most of the work is done
    public void doFilter(ServletRequest request,
                        ServletResponse response,
                        FilterChain chain) {

        try {
            // add request info to the log file
            synchronized(_loggingFile) {
                ...
            }

            // pass the request on to the next resource in the chain
            chain.doFilter(request, response);
        }
        catch (Throwable t) {
            // handle problem...
        }
    }

    // implement the required destroy method
    public void destroy() {
        // make sure logging file is closed
        _loggingFile.close();
    }
}
```

Page lists

Page lists allow you to avoid hardcoding Uniform Resource Locators (URLs) in servlets and JavaServer Pages (JSP) files. A page list specifies the location where a request is to be forwarded, but automatically tailors that location depending on the MIME type of the servlet. These properties allow you to specify a markup language and an associated MIME type. For the given MIME type, you also specify a set of pages to invoke.

WebSphere Application Server - Express supplies the `PageListServlet`, which you can use to call a JSP file by name based on the configuration data in the `client_types.xml` file. This file maps a JSP file to a Uniform Resource Identifier (URI). When the URI is invoked, it specifies another JSP file in a Web module. This support allows you to access multiple URLs without hard-coding them in your servlets.

You can also logically group page lists according to the markup language type, as for example, HTML or Wireless Markup Language (WML). This allows applications, using servlets that extend the `PageListServlet`, to call JSP files that return the proper markup-language type for the client request. For example, if a request originates from a PDA device that requires WML data and is sent to a servlet that extends the `PageListServlet`, the servlet can call a JSP file that returns a WML response.

You can define `PageListServlet` configuration information in the IBM Web Extensions file. The IBM Web Extensions file is created and stored in the Web Applications archive (WAR) file.

In addition to providing the page list mapping capability, the `PageListServlet` also provides Client Type Detection support. A servlet determines the markup language type that a calling client needs in the response, using the configuration information in the `client_types.xml` file. For more information, see “`client_types.xml`.”

Client type detection support allows a servlet, extending the `PageListServlet`, to call an appropriate JSP file. The servlet invokes the `callPage()` method, which calls a JSP file based on the markup-language type of the request.

“Example: Extending `PageListServlet`” on page 26

See this topic for an example of how to extend the `PageListServlet` class.

client_types.xml: The `client_types.xml` file provides client type detection support for servlets extending `PageListServlet`. Using the configuration data in the `client_types.xml` file, servlets can determine the language type that calling clients require for the response.

The client type detection support allows servlets to call appropriate JavaServer Pages (JSP) files with the `callPage()` method. Servlets select JSP files based on the markup-language type of the request.

Servlets must use the following version of the `callPage()` method to determine the markup language type required by the client:

```
callPage(String mlName,  
         String pageName,  
         HttpServletRequest request,  
         HttpServletResponse response)
```

where the arguments are:

- `mlName` is a markup language type
- `pageName` is a page name that is defined in the `PageListServlet` configuration
- `request` is the `HttpServletRequest` object
- `response` is the `HttpServletResponse` object

To see how the `callPage()` method is invoked by a servlet, see “Example: Extending `PageListServlet`” on page 26.

In the example, the client type detection method that is provided by the PageListServlet, `getMLTypeFromRequest(HttpServletRequest request)`, inspects the `HttpServletRequest` object `request` headers and searches for a match in the `client_types.xml` file.

The client type detection method performs these functions:

- Uses the input `HttpServletRequest` and the `client_types.xml` file to check for a matching HTTP request name and value.
- Returns the markup-language value configured for the `<client-type>` element, if a match is found.
- If multiple matches are found, this method returns the markup language for the first `<client-type>` element for which a match is found.
- If no match is found, returns the value of the markup language for the default page defined in the PageListServlet configuration.

The `client_types.xml` file is located in the `/QIBM/UserData/WebASE51/ASE/instance/properties` directory, where *instance* is the name of your instance.

Sample file entry

```
<?xml version="1.0"?>

<!DOCTYPE clients [
<!ELEMENT client-type (description,markup-language,request-header+)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT markup-language (#PCDATA)>
<!ELEMENT request-header (name,value)>
<!ELEMENT clients (client-type+)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT value (#PCDATA)>]>

<clients>
  <client-type>
    <description>IBM Speech Client</description>
    <markup-language>VXML</markup-language>
    <request-header>
      <name>user-agent</name>
      <value>IBM VoiceXML pre-release version 000303</value>
    </request-header>
    <request-header>
      <name>accept</name>
      <value>text/vxml</value>
    </request-header>
  </client-type>
  <client-type>
    <description>WML Browser</description>
    <markup-language>WML</markup-language>
    <request-header>
      <name>accept</name>
      <value>text/x-wap.wml</value>
    </request-header>
  <request-header>
    <name>accept</name>
    <value>text/vnd.wap.wml</value>
  </request-header>
</client-type>
</clients>
```

Example: Extending PageListServlet: See the “Code license and disclaimer information” on page 184 for legal information about this code example.

This example shows how a servlet extends the `PageListServlet` class and determines the markup-language type required by the client. The servlet then uses the `callPage()` method to call an appropriate JavaServer Pages (JSP) file. In this example, the JSP file that provides the correct markup language for the response is `Hello.page`.

```
public class HelloPervasiveServlet extends PageListServlet implements Serializable {

    // doGet -- Process incoming HTTP GET requests
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {

        // This is the name of the page to be called:
        String pageName = "Hello.page";

        // First check if the servlet was invoked with a queryString that contains a
        // markup-language value. For example, if the servlet is invoked with the URL
        // http://localhost/servlets/HelloPervasive?mlname=VXML, use this method:
        String mlname= getMLNameFromRequest(request);

        // If no markup language type is provided in the queryString, then try to
        // determine the client type from the request, and use the markup-language name
        // configured in the client_types.xml file.
        if (mlName == null) {
            mlName = getMLTypeFromRequest(request);
        }

        try {
            // Serve the request page.
            callPage(mlName, pageName, request, response);
        }
        catch (Exception e) {
            handleError(mlName, request, response, e);
        }
    }
}
```

Automatic request and response encoding

Two WebSphere Application Server - Express extensions are available, `autoRequestEncoding` and `autoResponseEncoding`.

In WebSphere Application Server - Express, the Web container no longer automatically sets request and response encodings, and response content types. Programmers are expected to set these values using available methods in the Servlet 2.3 Specification. If programmers choose not to use the character encoding methods, they can specify the `autoRequestEncoding` and `autoResponseEncoding` extensions, which enable the application server to set the encoding values and content type.

The Web container tries to determine the correct character encoding for the request parameters and data as follows:

1. It attempts to use the `client.encoding.override` system property, if one is set.
2. It looks at the character set (charset) in the Content-Type header.
3. If the `autoRequestEncoding` value is set to true, it extracts the "Accept-Language" header value. If found, this value is used in conjunction with the `properties/encoding.properties` and `properties/converter.properties` files to derive the character encoding.
4. It attempts to use the `default.client.encoding` system property, if one is set.
5. It uses the ISO-8859-1 character encoding as the default.

The Web container derives the character encoding used for the response as follows:

1. The servlet code uses the `setCharacterEncoding(String encoding)` method.

2. If the `autoResponseEncoding` value is set to `true`, it extracts the "Accept-Language" header value. If found, this value is used in conjunction with the `properties/encoding.properties` and `properties/converter.properties` files to derive the character encoding.
3. It uses the ISO-8859-1 character encoding as the default.

Use the WebSphere Development Studio for iSeries tools to change the default values for the `autoRequestEncoding` and `autoResponseEncoding` extensions. For more information, see the WebSphere Development Studio for iSeries Help.

Enhanced error reporting

A servlet can report errors by performing these actions:

- Calling the `HttpServletResponse.sendError()` method.
- Throwing an uncaught exception within its `service()` method.

The enhanced servlet error reporting function in WebSphere Application Server - Express provides a different way to implement error reporting. The error page (a JSP file or servlet) is configured for the application and used by all of the servlets in that application. The new mechanism handles caught and uncaught errors.

To return the error JSP file to the client, the Web container performs the following functions:

- Gets the `ServletContext.RequestDispatcher` for the URI configured for the Web module error path.
- Creates an instance of the error bean (type `ServletErrorReport`). The bean scope is requested, so that the target servlet (the servlet that encountered the error) can access the detailed error information.

For WebSphere Application Server - Express, the `HttpServletResponse.sendError()` method has been overridden to provide the following function:

```
public void sendError(int statusCode, String message) {
    ServletException e = new ServletException(statusCode, message);
    request.setAttribute(ServletErrorReport.ATTRIBUTE_NAME, e);
    servletContext.getRequestDispatcher(getErrorPath()).forward(request, response);
}
```

To enable this function for your Web module, perform these steps:

1. Create the error JSP file.
2. Place the file in the Web module document root.
3. Use the WebSphere Development Studio Client to configure an error path for the Web module. For more information, see the WebSphere Development Studio Client Help.

To create an error JSP file, you need to know the public methods of the `ServletErrorReport` class (the error bean), which are:

```
public class ServletErrorReport extends ServletException {

    // Get the stack trace of the error as a string
    public String getStackTrace()

    // Get the message associated with the error.
    // The same message is sent to the sendError() method.
    public String getMessage()

    // Get the error code associated with the error.
    // The same error code is sent to the sendError() method.
    // This will also be the same as the status code of the response.
    public int getErrorCode()

    // Get the name of the servlet that reported the error
    public String getTargetServletName()
}
```

The following is an example of an error JSP file:

```
<BEAN name="ErrorReport" type="com.ibm.websphere.servlet.error.ServletErrorReport"
  scope="request"></BEAN>
<html>
<head><title>ERROR: <%= ErrorReport.getErrorCode() %></title></head>
<body>
<H1>An error has occurred while processing the servlet named:
<%= ErrorReport.getTargetServletName() %></H1>

<B>Message: </B><%= ErrorReport.getMessage() %><BR>
<B>StackTrace: </B><%= ErrorReport.getStackTrace() %><BR>
</body>
</html>
```

WebSphere Application Server - Express provides an optional internal servlet, `com.ibm.ws.webcontainer.servlet.DefaultErrorReporter`, that makes it easier to use the enhanced error reporting capability. You simply add the `DefaultErrorReporter` servlet and the error JSP (which you develop) to your Web module. Use the WebSphere Development Studio for iSeries tools to configure the default error page. For more information, see the WebSphere Development Studio Client Help.

Internal servlets

WebSphere Application Server - Express provides internal (built-in) servlets that you can add to your Web module to enable optional functions for that Web module.

Here are the available internal servlets:

- **`com.ibm.ws.webcontainer.servlet.InvokerServlet`**
Invokes a servlet by class name. Add this servlet to a Web module with the WebSphere Development Studio for iSeries tools. For more information on how to add a servlet to your web module, see the WebSphere Development Studio Client Help.
- **`org.apache.jasper.runtime.JspServlet`**
Enables the JSP page compiler for processing a Web module's JSP files.
- **`com.ibm.ws.webcontainer.servlet.DefaultErrorReporter`**
Uses the extended error reporting function. See "Enhanced error reporting" on page 28 for more information.
- **`com.ibm.servlet.PageListServlet`**
This servlet allows you to call a JSP file by name from within your servlet code. See "Page lists" on page 25 for more information.
- **`com.ibm.ws.webcontainer.servlet.SimpleFileServlet`**
Enables file serving for static files such as HTML and GIF files.

Servlet resources

If you are new to servlets, use these resources to get started with servlet programming. If you have more advanced servlet programming skills, you may find these resources to be valuable aids to your programming.

The J2EE Tutorial: Java Servlet Technology

http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Servlets.html



This tutorial from JavaSoft is a good introduction to servlet programming.

Sun Microsystems Java[™] Servlets Web site

<http://java.sun.com/products/servlet/index.html>



The home for Java servlets, this Web site offers product information, technical resources, downloads and specifications, and news and articles that pertain to servlets.

JavaTM Servlet API 2.3

<http://java.sun.com/products/servlet/download.html>



WebSphere Application Server - Express implements JavaSoft's Java Servlet API 2.3 Application Programming Interface (API). To view the servlets Javadoc, download and install the Java Servlet Development Kit (JSDK) 2.3 on your workstation (see the link above).

WebSphere Development Studio for iSeries

<http://www.ibm.com/software/ad/wds400/>



This site provides information and resources for WebSphere Development Studio for iSeries.

JavaServer Pages (JSP)

WebSphere Application Server - Express for iSeries, supports the Sun Microsystems JavaServer Pages (JSP) Specification 1.2. JSPs written to the JSP Specification 1.1 are upward compatible with JSP Specification 1.2.

See the topics below for more information about developing JSP files for your Web applications:

"What are JavaServer Pages (JSP) files?" on page 31

See this topic for what JSP files are and how you can use them, including a description of the JSP lifecycle and JSP access models.

"JSP processor" on page 32

See this topic for an overview of the WebSphere Application Server - Express JSP processor.

"JSP tag extensions support" on page 33

The JSP specification allows you to create your own custom tags. This topic describes how to develop your own JSP tag set and add it to your Web application.

"IBM extensions to JSP tags" on page 33

WebSphere Application Server - Express provides extensions to the JSP tag set for accessing databases and working with database beans. See this topic for syntax and examples.

"Pre-touch tool for compiling and loading JSP files" on page 43

See this topic for information about how to compile, classload, and JIT-compile JSP files at application server startup time.

"JSP batch compiler" on page 44

See this topic for information about the JspBatchCompiler script, which allows you to manually compile JSP files in batch.

"Disable JSP run-time compilation" on page 45

See this topic for information about how to disable JSP run-time compilation for all Web applications or for an individual Web application.

"Reduce JSP compile time" on page 47

See this topic for information about a new initialization parameter that is available in WebSphere Application Server - Express for iSeries V5.1, which helps reduce JSP compile times.

What are JavaServer Pages (JSP) files?

You can use JavaServer Pages technology to create dynamic Web content while separating business logic from presentation logic. JSP files are comprised of tags (such as HTML tags and special JSP tags) and Java code. WebSphere Application Server - Express generates Java source code for the entire JSP file, compiles the code, and runs the JSP file as if it were a servlet.

HTML authors can develop JSP files that access databases and reusable Java components, such as servlets and JavaBeans. Programmers create the reusable Java components and provide the HTML authors with the component names and attributes. Database administrators or application programmers provide the HTML authors with the name of the database access and table information.

JSP lifecycle

JSP files are compiled into servlets. Thus, the JSP run-time lifecycle is similar to the “Servlet lifecycle” on page 15. See the information below for stages of the lifecycle that are particular to JSP files.

Java source generation and compilation

When IBM HTTP Server receives a request for a JSP file, it passes the request to WebSphere Application Server - Express’s servlet engine, which calls the JSP processor. The JSP processor is an internal servlet which converts a JSP file into Java source code and compiles it. The servlet that implements the JSP processor is `org.apache.jasper.runtime.JspServlet`.

If a certain request is the first time the JSP file has been requested or if the compiled copy of the JSP is not found, the JSP compiler generates and compiles a Java source file for the JSP file. The location of the generated files depends on which processor is being used.

The JSP syntax in a JSP file is converted to Java code that is added to the `service()` method of the generated class file.

Request processing

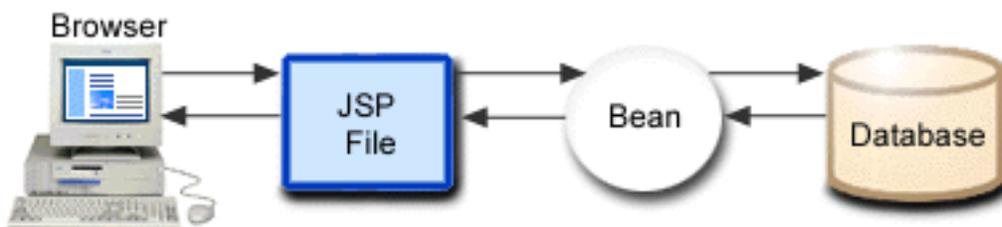
After the JSP processor has created the class file, the servlet engine creates an instance of the servlet and calls the servlet’s `service()` method in response to the request. All subsequent requests for the JSP are handled by that instance of the servlet.

When WebSphere Application Server - Express receives a request for a JSP file, it checks to determine whether the JSP file has changed since the file was loaded. If the JSP file has changed, WebSphere Application Server - Express reloads the updated JSP (that is, the JSP processor generates an updated Java source and class file for the JSP file). The newly-loaded servlet instance receives the client request.

Accessing JSP files

JSP files can be accessed in two ways:

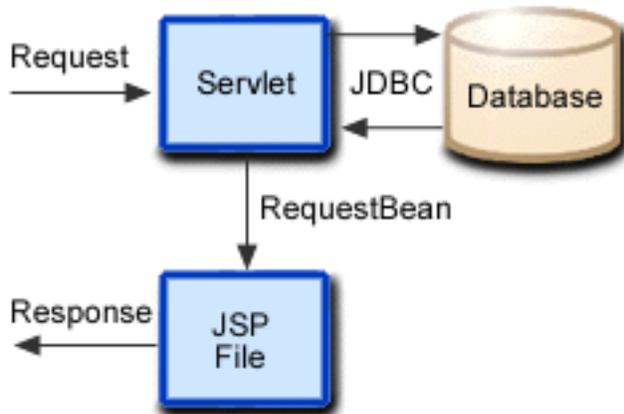
- The browser sends a request for a JSP file.



The JSP file accesses Beans or other components that generate dynamic content that is sent to the browser.

When the IBM HTTP Server receives a request for a JSP file, the server sends the request to WebSphere Application Server - Express. WebSphere Application Server - Express parses the JSP file and generates Java source, which is compiled and run as a servlet. The generation and compilation of the Java source occurs only on the first invocation of the servlet, unless the original JSP file has been updated. In such a case, WebSphere Application Server - Express detects the change, and regenerates and compiles the servlet before executing it.

- A servlet calls the JSP file.



The request is sent to a servlet that generates dynamic content and calls a JSP file to send the content to the browser. This access model facilitates separating content generation from content display.

JSP processor

When you configure your Web server instance to work with the WebSphere Application Server - Express for iSeries product, the Web server configuration is set to pass HTTP requests for JSP files (files with the extension .jsp) to WebSphere Application Server - Express. The JSP processor creates and compiles a servlet for each JSP file.

These following points apply to the JSP processor that is implemented in WebSphere Application Server - Express:

- The servlet that implements the JSP processor is `org.apache.jasper.runtime.JspServlet`.
- WebSphere Application Server - Express places the generated files in a temp directory under the WebSphere instance. For example, the WebSphere instance *myInstance* stores the generated files in `/QIBM/UserData/WebASE51/ASE/myInstance/temp/node_name/application_server/enterprise_app/web_module` where:
 - *node_name* is the name of the iSeries server or partition on which your WebSphere Application Server - Express instance is running
 - *application_server* is the name of your WebSphere Application Server - Express server
 - *enterprise_app* is the name of the enterprise application to which the JSP file belongs
 - *web_module* is the Web module that contains your JSP file.

Note: This path has been wrapped for display purposes. Enter it as one path.

- When the processor generates the servlet, two or three files are created, depending on the JSP processor settings:
 - **.class file**
This is the compiled servlet. This file is always created and kept.

- **.java file**
This is the generated source (.java) file. This file is created during the compilation of a JSP page, but the processor only keeps the file if it is told to do so. You can retain the .java file by setting an initialization parameter (for the JSP attributes) called **keepgenerated** to a value of true. This setting leaves the generated .java file in the `/QIBM/UserData/WebASE51/ASE/instance/temp/node_name/application_server/enterprise_app/web_module` directory. By default this value is not specified as an init parameter, and thus the default behavior is to **not** keep the generated .java file.
- **.dat file**
This contains the static (HTML code and text content) part of the original JSP file.
- **workingDir** is an `init()` parameter that is specified by default. However, this parameter is ignored by the JSP processor.

Keeping generated Java source files

The JSP processor generates a Java source file for each JSP file. By default, this file is deleted immediately. It is recommended that you keep the generated .java file for debugging purposes only. It is safer and more efficient to configure the JSP processor to delete generated .java files in a production environment.

JSP tag extensions support

You can use custom JSP tags to assist in replacing functionality currently implemented with a scriptlet (inline Java code). Encapsulating this functionality with a custom JSP tag allows you to more cleanly separate business logic from the presentation of the HTML output that displays in the user's browser.

The basic steps for using JSP tag extensions are:

1. Create a tag library definition file (*.tld) that contains the definition, or grammar, of the new tag or tags.
2. Write classes to contain the functionality of the tag.
3. Write or change the JSP code to use the new tag and specify the tag library.

For more information about Tag Extensions and Tag Libraries see the Custom Tags in JSP Pages



1. Create the tag library definition. This XML file defines the list of tags, the Java class that implements the tags, and the names that the JSP file uses to refer to those tags.
2. Write Java classes that extend the TagSupport class to implement the function for the tag.
3. Compile the Java files.
4. Write one or more JSP files that use the new tags.
5. Assemble the application that includes your JSP files. Add the TLD file as a resource file using WebSphere Development Studio Client. For more information, see the WebSphere Development Studio Client Help.
6. Place the Java classes that you wrote in the servlet classpath of the Web module in which the JSP files are deployed. This is done so that the JSP processor can find the code to implement the new tags. The class files must be located in the WEB-INF/classes directory.

IBM extensions to JSP tags

WebSphere Application Server - Express supports IBM extensions and additions to the JSP specification.

1. Review the supported specifications and create Java^(TM) components.
Note: Use of these tags are not recommended unless you are migrating your JSPs from WebSphere Application Server Version 3.5. Use of the IBM extension tags in your JSP files diminishes their potential to be ported to a non-WebSphere application server.
Extensions can be categorized as either:

- **Syntax for variable data**
Put variable fields in JSP files and have servlets and beans dynamically replace the variables with values from a database when the JSP output is returned to the browser.
 - **Syntax for database access**
Add a database connection to a Web page and then use that connection to query or update the database. You can provide the user ID and password for the database connection at request time, or you can hard code the user ID and password within the JSP file.
2. Use an integrated development environment (IDE) or text editor to develop or migrate code artifacts that meet the JSP specifications.
 3. Test the code artifacts.
 4. **(Optional)** Run a batch compilation on your JSP files if necessary. See “JSP batch compiler” on page 44 for more information.

See the following topics for more information about the IBM extensions and their syntax:

“<tsx:dbconnect>”

This tag specifies information that is needed to connect to a database.

“<tsx:userid> and <tsx:passwd>” on page 35

Use these tags as variables for user ID and password input.

“<tsx:dbquery>” on page 36

You can use this tag to query a database and return the results.

“<tsx:dbmodify>” on page 39

This tag allows you to add records to a database.

“<tsx:repeat>” on page 40

You can use this tag to iterate through a the results set of a database query.

“<tsx:getProperty>” on page 43

This tag gets the value of a bean to display in the JSP file.

Keep in mind that using the IBM extension tags in your JSP files diminishes their potential to be ported to a non-WebSphere application server.

<tsx:dbconnect>: Use the <tsx:dbconnect> syntax to specify information needed to make a connection to a JDBC-accessible database. The <tsx:dbconnect> syntax does not establish the connection. Instead, the <tsx:dbquery> and <tsx:dbmodify> syntax are used to reference a <tsx:dbconnect> in the same JSP file and establish the connection.

When the JSP file is compiled into a servlet, the Java processor adds the Java coding for the <tsx:dbconnect> syntax to the servlet’s service() method, which means a new database connection is created for each request for the JSP file.

The <tsx:dbconnect> syntax is:

```
<tsx:dbconnect id="connection_id" userid="db_user" passwd="user_password"
  url="jdbc:subprotocol:database" driver="database_driver_name"
  jndiname="JNDI_context/logical name">
</tsx:dbconnect>
```

This describes the attributes and their values:

- **id**
A required identifier. The scope is the JSP file. This identifier is referenced by the connection attribute of a <tsx:dbquery> tag.

- **userid**
An optional attribute that specifies a valid user ID for the database to be accessed. If specified, this attribute and its value are added to the request object.
Although the userid attribute is an optional tag attribute, you must provide a user ID for this tag, either with the attribute or by using the nested tag `<tsx:userid>`. See “`<tsx:userid>` and `<tsx:passwd>`” for an alternative to hardcoding this information in the JSP file.
- **passwd**
An optional attribute that specifies the user password for the userid attribute. (This attribute is not optional if the userid attribute is specified.) If specified, this attribute and its value are added to the request object.
Although the passwd attribute is optional, you must provide a password for this tag, either with the attribute or by using the `<tsx:passwd>` tag. See “`<tsx:userid>` and `<tsx:passwd>`” for an alternative to hardcoding this attribute in the JSP file.
- **url and driver**
To establish a database connection, the URL and driver must be provided.
The url attribute specifies the location of the database. The driver attribute specifies the name of the driver to be used to establish the database connection.
For a connection to a JDBC-accessible database, the URL consists of these colon-separated elements: jdbc, the subprotocol name, and the name of the database to be accessed. For example:

```
url="jdbc:db2:*local"
driver="com.ibm.db2.jdbc.app.DB2Driver"
```
- **jndiname**
This optional attribute identifies a valid context in the WebSphere Application Server - Express JNDI naming context and the logical name of the data source in that context. The context is configured by the Web administrator using an administrative client such as the WebSphere administrative console.

All of the elements shown in the example need to be specified. However, an empty element (such as `<url></url>`) is valid.

When the JSP file is compiled into a servlet, the Java processor adds the Java coding for the `<tsx:dbconnect>` syntax to the servlet’s `service()` method, which means a new database connection is created for each request for the JSP file.

<tsx:userid> and <tsx:passwd>: Instead of hardcoding the user ID and password in the `<tsx:dbconnect>` tag, you can use the `<tsx:userid>` and `<tsx:passwd>` tags to accept user input for the values and then add that data to the request object where it can be accessed by a JSP that requests the database connection.

The `<tsx:userid>` and `<tsx:passwd>` must be used within a `<tsx:dbconnect>` tag.

The `<tsx:userid>` and `<tsx:passwd>` must be used within a `<tsx:dbconnect>` tag. The `<tsx:userid>` and `<tsx:passwd>` syntax is:

```
<% String userID = request.getParameter("USERID"); %>
<% String passWord = request.getParameter("PASSWD"); %>

<tsx:dbconnect id="conn" url="jdbc:db2:*local" driver="com.ibm.db2.jdbc.app.DB2Driver">
  <tsx:userid><%=userID%></tsx:userid>
  <tsx:passwd><%=passWord%></tsx:passwd>
</tsx:dbconnect>
```

This list describes the attributes and their values:

- **<tsx:getProperty>**
This syntax is a mechanism for embedding variable data. For more information, see “`<tsx:getProperty>`” on page 43
- **USERID**
This is a reference to the request parameter that contains the userid. The parameter must have already

been added to the request object that was passed to this JSP file. The attribute and its value can be set in the request object using an HTML form or a URL query string to pass the user-specified request parameters.

- **PASSWD**

This is a reference to the request parameter that contains the password. The parameter must have already been added to the request object that was passed to this JSP. The attribute and its value can be set in the request object using an HTML form or a URL query string to pass user-specified values.

<tsx:dbquery>: Use the <tsx:dbquery> syntax to establish a connection to a database, submit database queries, and return the results set.

The <tsx:dbquery> tag:

- Refers to a <tsx:dbconnect> in the same JSP file and uses the information it provides to determine the database URL and driver. The user ID and password are also obtained from the <tsx:dbconnect> if those values are provided in the <tsx:dbconnect>.
- Establishes a new connection.
- Retrieves and caches data in the results object.
- Closes the connection (releases the connection resource).

The <tsx:dbquery> syntax is:

```
<tsx:dbquery id="query_id" connection="connection_id" limit="value" >  
  
  <%-- SELECT commands and (optional) JSP syntax can be --%>  
  <%-- placed within the tsx:dbquery tag. Any other      --%>  
  <%-- syntax, including HTML comments, are not valid.  --%>  
  
</tsx:dbquery>
```

This list describes the attributes and their values:

- **id**

The identifier of this query. The scope is the JSP file. This identifier is used to reference the query, for example, from the <tsx:getProperty> tag to display query results.

The id is a tsx reference to the bean and can be used to retrieve the bean from the page context. For example, if id is named mySingleDBBean, replace this:

```
if (mySingleDBBean.getValue("UISEAM",0).startsWith("N"))
```

with this:

```
com.ibm.ws.webcontainer.jsp.tsx.db.QueryResults bean =  
  (com.ibm.ws.webcontainer.jsp.tsx.db.QueryResults)pageContext. findAttribute("mySingleDBBean");  
  if (bean.getValue("UISEAM",0).startsWith("N")). . .
```

The bean properties are dynamic and the property names are the names of the columns in the results set. If you want different column names, use the SQL keyword for specifying an alias on the SELECT command. In the following example, the database table contains columns named FNAME and LNAME, but the SELECT statement uses the AS keyword to map those column names to FirstName and LastName in the results set:

```
Select FNAME, LNAME AS FirstName, LastName from Employee where FNAME='Jim'
```

- **connection**

The identifier of a <tsx:dbconnect> in this JSP file. That <tsx:dbconnect> provides the database URL, driver name, and (optionally) the user ID and password for the connection.

- **limit**

An optional attribute that constrains the maximum number of records returned by a query. If the attribute is not specified, no limit is used and the effective limit is determined by the number of records and the system caching capability.

- SELECT command and JSP syntax
Because the <tsx:dbquery> must return a results set, the only valid SQL command is SELECT. For more information about the SELECT command, see this resource:
 - DB2 Universal Database for iSeries SQL Reference (V5R2)
 - DB2 Universal Database for iSeries SQL Reference (V5R3)

In the following example, a database is queried for data about employees in a specified department. The department is specified using the <tsx:getProperty> to embed a variable data field. The value of that field is based on user input.

```
<% String workdept = request.getParameter("WORKDEPT"); %>

<tsx:dbquery id="qs2" connection="conn" >
  select * from WSDemo.EMPLOYEE where WORKDEPT= '<%=workdept%>'
</tsx:dbquery>
```

Displaying query results

To display the query results, use the <tsx:repeat> and <tsx:getProperty> syntax. The <tsx:repeat> loops through each of the rows in the query results. The <tsx:getProperty> uses the query results object (for the <tsx:dbquery> syntax whose identifier is specified by the <tsx:getProperty> bean attribute) and the appropriate column name (specified by the <tsx:getProperty> property attribute) to retrieve the value. For example:

```
<tsx:repeat>
<tr>
<td>
  <tsx:getProperty name="empqs" property="EMPNO" />
  <tsx:getProperty name="empqs" property="FIRSTNAME" />
  <tsx:getProperty name="empqs" property="WORKDEPT" />
  <tsx:getProperty name="empqs" property="EDLEVEL" />
</td>
</tr>
</tsx:repeat>
```

The “JSP10employeeRepeatResults.jsp example” example illustrates the syntax of the <tsx:getProperty> tag.

JSP10employeeRepeatResults.jsp example: See the “Code license and disclaimer information” on page 184 for legal information about this code examples.

```
<!-- JSP10employeeRepeatResults.jsp -->

<HTML>
<HEAD>
<TITLE>JSP 1.0 Employee Results</TITLE>
</HEAD>
<H1><CENTER>EMPLOYEE RESULTS</CENTER></H1>
<BODY>

<% String userID = request.getParameter("USERID"); %>
<% String passWord = request.getParameter("PASSWD"); %>

<% String empno = request.getParameter("EMPNO"); %>
<% String firstname = request.getParameter("FIRSTNAME"); %>
<% String midinit = request.getParameter("MIDINIT"); %>
<% String lastname = request.getParameter("LASTNAME"); %>
<% String workdept = request.getParameter("WORKDEPT"); %>
<% String edlevel = request.getParameter("EDLEVEL"); %>

<!-- Get a connection to the local DB2 database using parameters from Login.jsp -->
<tsx:dbconnect id="conn" url="jdbc:db2:*local" driver="com.ibm.db2.jdbc.app.DB2Driver">
<tsx:userid><%=userID%></tsx:userid>
<tsx:passwd><%=passWord%></tsx:passwd>
```

```

</tsx:dbconnect>

<% if ( ( request.getParameter("Submit")).equals("Update") ) { %>
<tsx:dbmodify connection="conn" >
  INSERT INTO WSDemo.EMPLOYEE
    (EMPNO, FIRSTNAME, MIDINIT, LASTNAME, WORKDEPT, EDLEVEL)
  VALUES
    ( '<%=empno%>',
      '<%=firstname%>',
      '<%=midinit%>',
      '<%=lastname%>',
      '<%=workdept%>',
      '<%=edlevel%>')
</tsx:dbmodify>
<B><UL>UPDATE SUCCESSFUL</UL></B>
<BR><BR>
<tsx:dbquery id="qs" connection="conn" >
  select * from WSDemo.EMPLOYEE where WORKDEPT= '<%=workdept%>'
</tsx:dbquery>

<B><CENTER><U>EMPLOYEE LIST</U></CENTER></B><BR><BR>
<HR>
<TABLE>
<TR VALIGN=BOTTOM>
<TD><B>EMPLOYEE
<BR>
<U>NUMBER</U></B></TD>
<TD><B><U>NAME</U></B></TD>
<TD><B><U>DEPARTMENT</U></B></TD>
<TD><B><U>EDUCATION</U></B></TD>
</TR>

<tsx:repeat>
<TR>
<TD><B><I><tsx:getProperty name="qs" property="EMPNO" /></I></B></TD>
<TD><B><I><tsx:getProperty name="qs" property="FIRSTNAME" /></I></B></TD>
<TD><B><I><tsx:getProperty name="qs" property="WORKDEPT" /></I></B></TD>
<TD><B><I><tsx:getProperty name="qs" property="EDLEVEL" /></I></B></TD>
</TR>
</tsx:repeat>

</TABLE>

<HR>
<BR>
<% } %>

<% if ( ( request.getParameter("Submit")).equals("Query") ) { %>

<tsx:dbquery id="qs2" connection="conn" >
  select * from WSDemo.EMPLOYEE where WORKDEPT= '<%=workdept%>'
</tsx:dbquery>

<B><CENTER><U>EMPLOYEE LIST</U></CENTER></B><BR><BR>
<HR>
<TABLE>
<TR>
<TR VALIGN=BOTTOM>
<TD><B>EMPLOYEE
<BR>
<U>NUMBER</U></B></TD>
<TD><B><U>NAME</U></B></TD>
<TD><B><U>DEPARTMENT</U></B></TD>
<TD><B><U>EDUCATION</U></B></TD>

```

```

</TR>

<tsx:repeat>
<TR>
<TD><B><I><tsx:getProperty name="qs2" property="EMPNO" /></I></B></TD>
<TD><B><I><tsx:getProperty name="qs2" property="FIRSTNAME" /></I></B></TD>
<TD><B><I><tsx:getProperty name="qs2" property="WORKDEPT" /></I></B></TD>
<TD><B><I><tsx:getProperty name="qs2" property="EDLEVEL" /></I></B></TD>
</TR>
</tsx:repeat>
</TABLE>
<HR>
<BR>
<% } %>

</BODY>
</HTML>

```

<tsx:dbmodify>: Use the <tsx:dbmodify> syntax to establish a connection to a database and then add records to a database table.

The <tsx:dbmodify> tag:

- Refers to a <tsx:dbconnect> in the same JSP file and uses the information provided by that database connection to determine the database URL and driver. The user ID and password are also obtained from the <tsx:dbconnect> if those values are provided in the <tsx:dbconnect>.
- Establishes a new connection.
- Updates a table in the database.
- Closes the connection (releases the connection resource).

The <tsx:dbmodify> syntax is:

```

<tsx:dbmodify connection="connection_id" >

    <!-- Any valid database update commands can be      ->
    <!-- placed within the tsx:dbmodify tag. Any other  ->
    <!-- syntax, including HTML comments, are not valid. ->

</tsx:dbmodify>

```

This list describes the attributes and their values:

- **connection**
The identifier of a <tsx:dbconnect> in this JSP file. That <tsx:dbconnect> provides the database URL, driver name, and (optionally) the user ID and password for the connection.
- *Database commands*
For more information about database commands, see this resource:
 - DB2 Universal Database for iSeries SQL Reference (V5R2)
 - DB2 Universal Database for iSeries SQL Reference (V5R3)

In the following example, a new employee record is added to a database. The values of the fields are based on user input from this JSP and referenced in the database commands using the <tsx:getProperty> tag.

```

<% String empno = request.getParameter("EMPNO"); %>
<% String firstnme = request.getParameter("FIRSTNAME"); %>
<% String midinit = request.getParameter("MIDINIT"); %>
<% String lastname = request.getParameter("LASTNAME"); %>
<% String workdept = request.getParameter("WORKDEPT"); %>
<% String edlevel = request.getParameter("EDLEVEL"); %>

<tsx:dbmodify connection="conn" >
    INSERT INTO WSDemo.EMPLOYEE

```

```

    (EMPNO,FIRSTNME,MIDINIT,LASTNAME,WORKDEPT,EDLEVEL)
VALUES
    ( '<%=empno%>',
      '<%=firstnme%>',
      '<%=midinit%>',
      '<%=lastname%>',
      '<%=workdept%>',
      '<%=edlevel%>' )
</tsx:dbmodify>

```

<tsx:repeat>: Use the <tsx:repeat> syntax to iterate over a database query results set. The <tsx:repeat> syntax iterates from the start value to the end value until one of these conditions is met:

- The end value is reached.
- An `ArrayIndexOutOfBoundsException` is thrown.

The output of a <tsx:repeat> block is buffered until the block completes. If an exception is thrown before a block completes, no output is written for that block.

The <tsx:repeat> syntax is:

```

<tsx:repeat index="name" start="starting_index" end="ending_index">
</tsx:repeat>

```

This list describes the attributes and their values:

- **index**
This is an optional name used to identify the index of this repeat block. The value is case-sensitive and its scope is the JSP file.
- **start**
This is an optional starting index value for this repeat block. The default is "0."
- **end**
This is an optional ending index value for this repeat block. The maximum value is "2,147,483,647." If the value of the end attribute is less than the value of the start attribute, the end attribute is ignored.

The results set and the associated bean

The <tsx:repeat> iterates over a results set. The results set is contained within a bean. The bean can be a static bean or a dynamically generated bean (for example, a bean generated by the <tsx:dbquery> syntax).

This table is a graphic representation of the contents of a bean, named "myBean":

| | col1 | col2 | col3 |
|------|---------|---------|------------|
| row0 | friends | Romans | countrymen |
| row1 | bacon | lettuce | tomato |
| row2 | May | June | July |

Some observations about the bean:

- The column names in the database table become the property names of the bean. The "<tsx:dbquery>" on page 36 topic describes a technique for mapping the column names to different property names.
- The bean properties are indexed. For example, `myBean.get(Col1(row2))` returns May.
- The query results are in the rows. The <tsx:repeat> iterates over the rows (beginning at the starting row).

The following table compares using the <tsx:repeat> tag to iterate a static bean to using the <tsx:repeat> tag with a dynamically generated bean:

| Static bean example | Dynamic bean example (<tsx:repeat>) |
|---|--|
| myBean.class <pre>// Code to get // a connection // Code to get the data Select * from myTable; // Code to close // the connection</pre> | JSP file <pre><tsx:dbconnect id="conn" userid="alice" passwd="test" url="jdbc:db2:*local" driver="com.ibm.db2.jdbc.app.DB2Driver" </tsx:dbconnect > <tsx:dbquery id="dynamic" connection="conn" > Select * from myTable; </tsx:dbquery> <tsx:repeat index=abc> <tsx:getProperty name="myBean" property="coll(abc)" /> </tsx:repeat></pre> |
| JSP file <pre><tsx:repeat index=abc> <tsx:getProperty name="myBean" property="coll(abc)" /> </tsx:repeat></pre> | <pre><tsx:repeat index=abc> <tsx:getProperty name="dynamic" property="coll(abc)" /> </tsx:repeat></pre> |
| Notes: <ul style="list-style-type: none"> • The bean (myBean.class) is a static bean. • The method to access the bean properties is myBean.get(<i>property(index)</i>). • You can omit the property index, in which case the index of the enclosing <tsx:repeat> is used. You can also omit the index on the <tsx:repeat>. • The <tsx:repeat> iterates over the bean properties row by row, beginning with the starting row. | Notes: <ul style="list-style-type: none"> • The bean (dynamic) is generated by the <tsx:dbquery> and does not exist until the syntax is processed. • The method to access the bean properties is dynamic.getValue(<i>"property", index</i>). • You can omit the property index, in which case the index of the enclosing <tsx:repeat> is used. You can also omit the index on the <tsx:repeat>. • The <tsx:repeat> syntax iterates over the bean properties row by row, beginning with the start row. |

Implicit and explicit indexing

Examples 1, 2, and 3 show how to use the <tsx:repeat> tag. The examples produce the same output if all indexed properties have 300 or fewer elements. If there are more than 300 elements, Examples 1 and 2 display all elements, while Example 3 shows only the first 300 elements.

Example 1 shows implicit indexing with the default start and default end index. The bean with the smallest number of indexed properties restricts the number of times that the loop repeats.

```
<table>

<tsx:repeat>
<tr>
<td>
<tsx:getProperty name="serviceLocationsQuery" property="city" />
</td>
</tr>

<tr>
<td>
<tsx:getProperty name="serviceLocationsQuery" property="address" />
</td>
</tr>

<tr>
<td>
<tsx:getProperty name="serviceLocationsQuery" property="telephone" />
</td>
</tr>
</tsx:repeat>

</table>
```

Example 2 shows indexing, starting index, and ending index:

```
<table>

<tsx:repeat index=myIndex start=0 end=2147483647>
<tr>
<td>
<tsx:getProperty name="serviceLocationsQuery" property=city(myIndex) />
</td>
</tr>

<tr>
<td>
<tsx:getProperty name="serviceLocationsQuery" property=address(myIndex) />
</td>
</tr>

<tr>
<td>
<tsx:getProperty name="serviceLocationsQuery" property=telephone(myIndex) />
</td>
</tr>
</tsx:repeat>

</table>
```

Example 3 shows explicit indexing and ending index with implicit starting index. Although the index attribute is specified, the indexed property city can still be implicitly indexed because the (myIndex) is not required.

```
<table>

<tsx:repeat index=myIndex end=299>
<tr>
<td>
<tsx:getProperty name="serviceLocationsQuery" property="city" />
</td>
</tr>

<tr>
<td>
<tsx:getProperty name="serviceLocationsQuery" property="address(myIndex)" />
</td>
</tr>

<tr>
<td>
<tsx:getProperty name="serviceLocationsQuery" property="telephone(myIndex)" />
</td>
</tr>
</tsx:repeat>

</table>
```

Nesting <tsx:repeat> blocks

You can nest <tsx:repeat> blocks. Each block is separately indexed. This capability is useful for interleaving properties on two beans, or properties that have subproperties. In the example, two <tsx:repeat> blocks are nested to display the list of songs on each compact disc in the user's shopping cart.

```
<tsx:repeat index=cdindex>

<h1><tsx:getProperty name="shoppingCart" property=cds.title /></h1>

<table>
<tsx:repeat>
```

```

<tr>
<td>
<tsx:getProperty name="shoppingCart" property=cds(cdindex).playlist />
</td>
</tr>
</tsx:repeat>
</table>

</tsx:repeat>

```

<tsx:getProperty>: The <tsx:getProperty> tag gets the value of a bean to display in a JavaServer Pages (JSP) file.

The <tsx:getProperty> is an IBM extension of the Sun Microsystems <jsp:getProperty> tag. The IBM extension implements all of the <jsp:getProperty> function and adds the ability to introspect a database bean that was created using the IBM extension <tsx:dbquery> or <tsx:dbmodify> tag.

The <tsx:getProperty> uses the query results object (for the <tsx:dbquery> syntax whose identifier is specified by the <tsx:getProperty> bean attribute) and the appropriate column name (specified by the <tsx:getProperty> property attribute) to retrieve the value.

Note: You cannot assign the value from the <tsx:getProperty> tag to a variable. The value, generated as output from this tag, is displayed in the browser window.

The <tsx:getProperty> syntax is:

```
<tsx:getProperty name="bean_name" property="property_name" />
```

This list describes the attributes and their values:

- **name**
The name of the bean declared by the id attribute of a <tsx:dbquery> syntax within the JSP file. See “<tsx:dbquery>” on page 36 for more information. The value of this attribute is case-sensitive.
- **property**
The property of the bean to access for substitution. The value of the attribute is case-sensitive and is the locale-independent name of the property.

>Examples

```

<tsx:getProperty name="userProfile" property="username" />
<tsx:getProperty name="request" property=request.getParameter("corporation") />

```

In most cases, the value of the property attribute will be just the property name. However, to access the request bean or access a property of a property (subproperty), you specify the full form of the property attribute. The full form also gives you the option to specify an index for indexed properties. The optional index can be a constant (such as 2) or an index like the one described in “<tsx:repeat>” on page 40.

Some examples of using the full form of the property attribute:

```

<tsx:getProperty name="staffQuery" property=address(currentAddressIndex) />
<tsx:getProperty name="shoppingCart" property=items(4).price />
<tsx:getProperty name="fooBean" property=foo(2).bat(3).boo.far />

```

Pre-touch tool for compiling and loading JSP files

When enabled, the pre-touch mechanism causes all JSPs to be compiled within the Web module for which they are configured. You can also configure some or all JSPs to be classloaded and JIT-compiled.

You have to manually add the JSP attributes to the ibm-web-ext.xmi file in the Web module of your application. Use WebSphere Development Studio Client for iSeries to import your application and configure JSP attributes. (WebSphere Application Server - Express for iSeries ships with WebSphere Development Studio Client for iSeries.) For example:

```
<jspAttributes xmi:id="JSPAttribute_1" name="prepareJSPs" value="0"/>
<jspAttributes xmi:id="JSPAttribute_2" name="prepareJSPAttribute" value="1"/>
```

Export the modified application out of WebSphere Development Studio Client for iSeries as an enterprise archive (EAR) file, and install the EAR file into WebSphere Application Server - Express.

To enable the pre-touch mechanism, specify the following JSP attributes, which are Assembly Property Extensions for your Web module:

- **prepareJSPs** (Required)

When this attribute is present, all JSPs are compiled at application server startup. This activity runs in a separate thread, allowing the application server to finish other startup actions in parallel. The numeric attribute value represents the minimum size (in kilobytes) that a JSP must be in order to also be classloaded and JIT-compiled. The default is 0, which causes all JSPs to be classloaded and JIT-compiled.

Note: JSP compilation is different from JIT compilation. JSP compilation generates bytecodes, whereas JIT translates the bytecodes into machine code at run time.

- **prepareJSPAttribute** (Optional)

The pre-touch mechanism compiles and JIT-compiles JSPs by directly invoking the JSP service method, thus making the JSP susceptible to incurring exceptions because it is being called out of context. Such exceptions are avoided by immediately checking the value of this attribute, causing a quick exit from the service method when the JSP was prepared by this tool. This attribute value is added as a request parameter and is composed of alphanumeric characters that your JSPs do not expect to use during normal execution.

- **prepareJSPThreadCount** (Optional)

Set this numeric attribute to the number of threads that you would like this mechanism to start up to compile your JSPs. Since a thread makes use of just one processor, multi-processor systems may better utilize this pre-touch mechanism by specifying a value greater than 1. The default setting for this attribute is 1, representing the number of threads that are created to perform pre-touch processing for this Web module.

- **prepareJSPClassload** (Optional)

Set this attribute to either a whole number or the word *changed*. By entering *changed*, only those JSPs that have been updated or not previously touched (for example, those JSPs that need to be converted from *.jsp* to *.java*) are classloaded. By entering a numerical value (for example, 1000), the pretouch tool starts classloading at the 1000th JSP that it processes and all subsequent JSPs. This is convenient in the event that the application server is stopped during pretouch execution. A customer can then check the server logs to see how many JSPs were already processed, and update the *prepareJSPClassload* value accordingly to avoid duplicating work. If a JSP is not classloaded, it cannot be JIT compiled. As a result, if a JSP does not satisfy the requirements of the *prepareJSPClassload* attribute, but satisfies the requirements of *prepareJSPs*, the JSP is compiled (if it has been updated), but is not classloaded or JIT compiled.

JSP batch compiler

WebSphere Application Server - Express allows you to compile JavaServer Pages files written to the JSP specification as a batch. This improves performance by reducing the response time on the first request. For iSeries, another option to consider is to use the Pre-touch tool which can compile and load JSP class files into the Application server JVM for improved performance over the JSP batch compiler.

To use the batch compiler for JSP files, follow these steps:

1. On an iSeries command line, run the Start Qshell (STRQSH) command.
2. The Qshell command prompt **\$** appears.
3. To change your directory, enter
`cd /QIBM/ProdData/WebASE51/ASE/bin`
4. Run the `JspBatchCompiler` command.

The syntax of the JspBatchCompiler command:

```
JspBatchCompiler -enterpriseapp.name name
                  [ -instance name ]
                  [ -webmodule.name name ]
                  [ -cell.name name ]
                  [ -node.name name ]
                  [ -server.name name ]
                  [ -filename jsp name ]
                  [ -keepgenerated <true|false> ]
                  [ -verbose <true|false> ]
                  [ -deprecation <true|false> ]
```

where the parameters are:

- **enterpriseapp.name**
The name of the Enterprise Application you want to compile.
- **instance**
The name of the WebSphere instance. This parameter is required.
- **webmodule.name**
The name of the specific Web module that you want to compile. If this argument is not set, all Web modules in the enterprise application are compiled.
- **cell.name**
The name of the cell in which the application is deployed.
- **node.name**
The name of the node in which the application is deployed.
- **server.name**
The name of the server in which the application is deployed. The default is server1.
- **filename**
The name of a single JSP file that you want to compile. If this argument is not set, all files in the Web module are compiled. Alternatively, if filename is set to the name of a directory, only the JSP files in that directory are compiled.
- **keepgenerated**
If set to true, WebSphere Application Server - Express saves the generated .java files used for compilation on your server. By default, this is set to false and the .java files are erased after the class files have been compiled.
- **verbose**
Indicates the compiler should generate verbose output while compiling the generated sources.
- **deprecated**
Indicates the compiler should generate deprecation warnings while compiling the generated sources.

Note: If the names that are specified for these arguments are composed of two or more words separated by spaces, you must add quotation marks (") around the names.

Following compilation, compiled .class files for the JSPs in the examples Web module are found within the temp subdirectory of the instance. For example, /QIBM/UserData/WebASE51/ASE/*instance*/temp where *instance* is the name of your WebSphere instance.

Disable JSP run-time compilation

The JSP engine translates a requested JSP file, compiles the .java file, and loads the compiled servlet into the run-time environment. In previous versions of WebSphere Application Server - Express, if a .class file did not exist, the JSP engine always translated and compiled the JSP file. You had to turn off the reload capability of Web applications to prevent additional translations and recompiles of the file.

In WebSphere Application Server - Express V5.1, you can change the default behavior of the JSP engine by indicating that a JSP file should never be translated or compiled at run time, even when a .class file does not exist.

If run-time compilation is disabled, you must precompile the JSP files, which provides the following advantages:

- Reduces compilation related disk operations.
- Minimizes disk storage requirements necessary for handling temporary .java and .class files generated during a run-time compilation.
- Forces the verification that a JSP file compiled successfully before deploying and installing the application in WebSphere Application Server - Express.

You can disable JSP file run-time compilation for all Web applications or for a specific Web application:

- **Disable compilation for all Web applications**

To disable the translation and compilation of JSP files for all Web applications, set the Web container Custom property `disableJspRuntimeCompilation` to true.

To set this property, perform the following steps:

1. Start the WebSphere administrative console.
2. In the topology tree, expand **Servers**.
3. Expand **Application Servers**, and click **Application Servers**.
4. Click your application server.
5. Click **Web Container**.
6. Click **Custom Properties**, and select **New**.
7. In the **Name** field, specify `disableJspRuntimeCompilation`. In the **Value** field, specify `true`.
8. Save the configuration.
9. Stop and start the application server for the changes to take effect.

Valid values for this setting are true or false. If this property is set to true, then translation and compilation of the JSP files is disabled at run time for all Web applications.

- **Disable compilation for a specific Web application**

To disable the translation and compilation of JSP files for a specific Web application, set the JSP engine initialization parameter `disableJspRuntimeCompilation` to true. When this setting is enabled, it determines the run-time behavior of the JSP engine and overrides the Web container custom property setting.

To set this parameter, you have to manually add the JSP attributes to the `ibm-web-ext.xml` file in the Web module of your application. Use WebSphere Development Studio Client for iSeries to import your application and configure JSP attributes. (WebSphere Application Server - Express for iSeries ships with WebSphere Development Studio Client for iSeries.) For example:

```
<jspAttributes xmi:id="JSPAttribute_1" name="disableJspRuntimeCompilation" value="true"/>
```

Export the modified application out of WebSphere Development Studio Client for iSeries as an enterprise archive (EAR) file, and install the EAR file into WebSphere Application Server - Express.

Valid values for this setting are true or false. If this parameter is set to true, translation and compilation of the JSP files is disabled at run time, and the JSP engine only loads precompiled files for the specific Web application.

Implications of disabling JSP run-time compilation

If the Web container custom property or the JSP attribute assembly parameter is not set, the first request for a JSP file results in the translation and compilation of the JSP file when the .class file does not exist. Subsequent requests for the file also result in compilations and translations, but only if the following conditions are met:

- Compilations and translations are required.
- Reloading is enabled for the Web module.
- Reload interval is exceeded.

If you disable run-time compilation and a request arrives for a JSP file that does not have a matching .class file, the JSP engine returns HTTP error 501 (Not implemented) to the browser. If the JSP file does not exist, the JSP engine returns HTTP error 404 (File not found) to the browser. If a JSP file has a matching .class file but that file is out of date, the JSP engine still loads the .class file into memory.

Perform the following steps to determine whether the `disableJspRuntimeCompilation` option is enabled in WebSphere Application Server - Express:

1. Enable the Diagnostic Trace Service and set the trace specification to `com.ibm.ws.webcontainer.jsp.servlet.*=all=enabled`.
2. Request a JSP file.
3. Ensure the `jspUri`: entry matches the requested JSP file.

If both the `disableJspRuntimeCompilation:true` string and the matching `jspUri`: entry appear in the trace, the `disableJspRuntimeCompilation` setting is enabled for the Web application.

Reduce JSP compile time

When WebSphere Application Server compiles a JSP, it creates a large classpath that includes every WebSphere Application Server .jar file to ensure that any class referenced by a given JSP is found. The greater the number of .jar files and classes on the classpath, the longer it takes for a JSP to compile.

However, most JSPs only invoke a few WebSphere Application Server APIs, which makes the large classpath unnecessary. In WebSphere Application Server for iSeries V5.1, there is a new feature that helps reduce JSP compile times. You can now edit the classpath that is used for compiling JSPs in your application.

Note: This feature only controls the classpath for compiling your JSP. The environment used to load and run your JSP remains unchanged.

The feature to reduce compile times is controlled by a JSP initialization parameter. This gives you control over the JSP compile classpath on each web module.

Name:

`jsp.compile.classpath`

Value:

A space-separated list of paths. If a path contains a space, put quotation marks around the path. Paths that are not fully qualified are resolved to the web module directory during run time.

Configuration

There are several ways to configure the `jsp.compile.classpath` feature:

1. To configure the `jsp.compile.classpath` feature using the WebSphere Development Studio Client for iSeries (recommended), perform the following steps:
 - a. Start the WebSphere Development Studio Client for iSeries.
 - b. In the J2EE perspective, expand **Web Modules**, and right click the Web module you want to modify.
 - c. Select **Open With** —> **Deployment Descriptor Editor**.
 - d. In the Web Deployment Descriptor, click the **Extensions** tab.
 - e. On the **WebSphere Extensions** page, click **Add** under the **JSP Attributes** heading.
 - f. Set the value of **Name** to `jsp.compile.classpath`.
 - g. Set the value of **Value** to the desired classpath. See Determine the classpath to use (page 48) for more information on setting the correct classpath.
 - h. Click **File** —> **Save**.
 - i. Deploy and start your application.

- j. Verify that the modified classpath is being used. To do this, look at the `/QIBM/UserData/WebASE51/ASE/instance/logs/server/SystemOut.log` file. Once you start your application, a message similar to the following appears in the log:

```
JSP Processor in webapp [Default Web Application] Using compile classpath
[/QIBM/ProdData/WebASE51/ASE/lib/webcontainer.jar:/QIBM/ProdData/WebASE51/
ASE/lib/j2ee.jar:/QIBM/UserData/WebASE51/ASE/myInstance/installedApps/server/
DefaultApplication.ear/DefaultWebApplication.war/WEB-INF/classes:]
```

Note: This message is given because the `jsp.compile.classpath` is set to an empty-string value.

2. To configure the `jsp.compile.classpath` feature on an installed application using a text editor, perform the following steps:
 - a. Use a text editor to open the `/QIBM/UserData/WebASE51/ASE/instance/config/cells/cell-name/applications/MyEnterpriseApplication.ear/deployments/MyEnterpriseApplication/MyWebApplication.war/WEB-INF/ibm-web-ext.xmi` file (where *instance* is the name of your application server instance and *cell-name* is the name of your cell).
 - b. Add the following `jspAttribute` to the file:

```
name="jsp.compile.classpath" value="classpath"
```

where *classpath* is the value of the desired classpath. See Determine the classpath to use (page 48) for more information on setting the correct classpath. For example, your file should look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
.
.
.
  <jspAttributes xmi:id="JSPAttribute_1" name="jsp.compile.classpath" value=""/>
</webappext:WebAppExtension>
```

- c. Save the updated file.
- d. Copy the updated file to `/QIBM/UserData/WebASE51/ASE/installedApps/MyEnterpriseApplication.ear/MyWebApplication.war/WEB-INF`. Replace the old version of the file with the modified version.
- e. Restart your enterprise application in the administrative console.
- f. Verify that the modified classpath is being used. To do this, look at the `/QIBM/UserData/WebASE51/ASE/instance/logs/server/SystemOut.log` file. Once you start your application, a message similar to the following appears in the log:

```
JSP Processor in webapp [Default Web Application] Using compile classpath
[/QIBM/ProdData/WebASE51/ASE/lib/webcontainer.jar:/QIBM/ProdData/WebASE51/
ASE/lib/j2ee.jar:/QIBM/UserData/WebASE51/ASE/myInstance/installedApps/server/
DefaultApplication.ear/DefaultWebApplication.war/WEB-INF/classes:]
```

Note: This message is given because the `jsp.compile.classpath` is set to an empty-string value.

Determine the classpath to use:

Determining the classpath for compiling a JSP is similar to determining a classpath for a servlet or other type of Java file. Look at the classes that are referenced in the JSP source, and make sure that the class or the `.jar` file where the class is found is on the specified classpath. However, you must give special consideration to JSPs that use custom tags, because custom tags contain references to classes.

JSPs always require the `webcontainer.jar` and `j2ee.jar` files. JSPs are also likely to reference classes in a web application for custom tags and other application-specific criteria. For these reasons, the following paths are automatically prepended to the paths you specify for `jsp.compile.classpath` when your application is started:

- `/QIBM/ProdData/WebASE51/ASE/lib/webcontainer.jar`
- `/QIBM/ProdData/WebASE51/ASE/lib/j2ee.jar`

- *your_web_module_directory*/WEB-INF/classes
- Any .jar or .zip files found in the *your_web_module_directory*/WEB-INF/lib directory

It is likely that the above files are sufficient for your JSPs to compile, in which case you only need to specify an empty string for the `jsp.compile.classpath` value.

You can use WebSphere Application Server variables for your classpath entries. For example, if you reference a WebSphere Application Server .jar file, such as `/QIBM/ProdData/WebASE51/ASE/lib/admin.jar`, you can use the `WAS_LIBS_DIR` variable, which changes your classpath entry to `${WAS_LIBS_DIR}/admin.jar`

Example: Determine classpath and configure `jsp.compile.classpath`

The following JSP references `com.ibm.websphere.cache.CacheEntry`, `com.mycompany.somepackage.MyClass`, and `com.mycompany.earlevel.EarClass`:

```
<% com.ibm.websphere.cache.CacheEntry entry = null; %>
<% com.mycompany.somepackage.MyClass object = null; %>
<% com.mycompany.earlevel.EarClass object2 = null; %>
```

Three classes are referenced:

- `CacheEntry.class`, which is found in `/QIBM/ProdData/WebASE51/ASE/lib/dynacache.jar`
- `MyClass.class`, which is assumed to be in `/QIBM/UserData/WebASE51/ASE/default/installedApps/node_name/MyEnterpriseApplication.ear/MyWebApplication.war/WEB-INF/lib/mywebjar.jar` (where *node_name* is the name of your application server node, *MyEnterpriseApplication.ear* is the name of your enterprise application, *MyWebApplication.war* is the name of your Web archive file, and *mywebjar.jar* is the name of your Java archive file).
- `EarClass.class`, which is assumed to be in `/QIBM/UserData/WebASE51/ASE/default/installedApps/node_name/MyEnterpriseApplication.ear/myearjar.jar` (where *node_name* is the name of your application server node, *MyEnterpriseApplication.ear* is the name of your enterprise application, and *myejbjar.jar* is the name of your Java archive file).

The file `mywebjar.jar` is automatically added to the classpath because it exists in the `WEB-INF/lib` directory. However, `dynacache.jar` and `myearjar.jar` must be explicitly added to the classpath. The JSP initialization parameters are now as follows:

Name:

`jsp.compile.classpath`

Value:

`${WAS_LIBS_DIR}/dynacache.jar`

`${APP_INSTALL_ROOT}/node_name/MyEnterpriseApplication.ear/myearjar.jar` (where *node_name* is the name of your application server node, *MyEnterpriseApplication.ear* is the name of your enterprise application, and *myejbjar.jar* is the name of your Java archive file).

Note: The space separator is used between paths (not a colon).

Data access

Whether you use databases to persist application data or allow users to query and update records, database access is an important part of Web application programming. WebSphere Application Server - Express for iSeries provides a variety of ways to access databases with your application components.

See these topics for information about accessing databases through WebSphere Application Server - Express:

“Data access overview”

This topic provides an overview of the WebSphere Application Server - Express data access architecture and iSeries-specific database considerations.

“Develop data access applications” on page 61

See this topic for information about data access APIs and programming.

“Assemble data access applications” on page 78

This topic describes the process of assembling your data access application for deployment into WebSphere Application Server - Express.

“Configure WebSphere Application Server - Express to access databases” on page 79

This topic provides configuration information about WebSphere Application Server - Express and databases.

“Deploy data access applications” on page 99

See this topic for considerations when you deploy a data access application.

Data access overview

See these topics for an overview of WebSphere Application Server - Express and data access.

“Connection management architecture”

This topic describes the architecture of connection management in WebSphere Application Server - Express.

“Connection pooling” on page 59

WebSphere Application Server - Express provides a pool of database connections, which reduces the performance overhead of creating new connections. See this topic for a description of how connection pooling works.

Connection management architecture: The connection management architecture for both relational and procedural access to enterprise information systems is based on the J2EE Connector Architecture (JCA) specification. The connection manager, which pools and manages connections within an application server, is capable of managing connections obtained through both resource adapters that are defined by the JCA specification, and DataSources that are defined by the JDBC 2.0 Extensions Specification.

To make DataSource connections manageable by this connection manager that works only with resource adapters, WebSphere Application Server - Express provides its own resource adapter. From the connection manager point of view, JDBC DataSources and JCA connection factories look the same. Users of DataSources do not experience any programmatic or behavioral differences in their applications because of the underlying JCA architecture. JDBC users still configure and use DataSources according to the JDBC programming model.

“Connection pooling” on page 59

See this topic for information about database connection pools.

“Connection life cycle” on page 51

This topic describes the life cycle of a database connection.

“Unshareable and shareable connections” on page 54

Database connections can be shared by more than one component of an application. See this topic for considerations about whether to share database connections or not.

“Connection handles” on page 56

This topic compares the characteristics of shared and unshared database connection references.

“Connections and transactions” on page 58

See this topic for information about how connections behave in various transaction scopes.

Connection life cycle: A ManagedConnection object is always in one of three states: DoesNotExist, InFreePool, or InUse.

Before a connection is created, it must be in the DoesNotExist state. After a connection is created, it can be in either the InUse or the InFreePool state, depending on whether it is allocated to an application.

Between these three states are transitions. These transitions are controlled by guarding conditions. A guarding condition is one in which true indicates when you can take the transition into another legal state. For example, you can make the transition from the InFreePool to InUse state only if:

- The application has called the data source or connection factory getConnection() method.
- A free connection is available in the pool with matching properties (freeConnectionAvailable).
- One of these conditions is true:
 - The getConnection request is on behalf of a resource reference that is marked unshareable.
 - The getConnection request is on behalf of a resource reference that is marked shareable but no shareable connection in use has the same properties.

This transition description follows:

```
InFreePool > InUse:  
getConnection AND  
freeConnectionAvailable AND  
NOT(shareableConnectionAvailable)
```

Here is a list of guarding conditions and descriptions.

| Condition | Description |
|------------------------------|--|
| ageTimeoutExpired | Connection is older than its ageTimeout value. |
| close | Application calls close method on the Connection object. |
| fatalErrorNotification | A connection has just experienced a fatal error. |
| freeConnectionAvailable | A connection with matching properties is available in the free pool. |
| getConnection | Application calls getConnection method on DataSource or ConnectionFactory object. |
| markedStale | Connection is marked as stale, typically in response to a FatalErrorNotification. |
| noOtherReferences | There is only one connection handle to the ManagedConnection, and the Transaction Service is not holding a reference to the ManagedConnection. |
| noTx | No transaction is in force. |
| poolSizeGTMin | Connection pool size is greater than the minimum pool size (minimum number of connections) |
| poolSizeLTMax | Pool size is less than the maximum pool size (maximum number of connections) |
| shareableConnectionAvailable | The getConnection request was for a shareable connection and one with matching properties is in use and available to share. |
| TxEnds | The transaction has ended. |
| unshareableConnectionRequest | The getConnection request is for an unshareable connection. |

| Condition | Description |
|----------------------|--|
| unusedTimeoutExpired | Connection is in the free pool and not in use past its unused timeout value. |

Getting connections

The first set of transitions covered are those in which the application requests a connection from either a data source or a connection factory. In some of these scenarios, a new connection to the database results. In others, the connection might be retrieved from the connection pool or shared with another request for a connection.

DoesNotExist

Every connection begins its life cycle in the DoesNotExist state. When an application server starts, the connection pool does not exist. Therefore, there are no connections. The first connection is not created until an application requests its first connection. Additional connections are created as needed, according to the guarding condition.

```
getConnection AND
NOT(freeConnectionAvailable) AND
poolSizeLTMax AND
(NOT(shareableConnectionAvailable) OR
(unshareableConnectionRequest))
```

This transition specifies that a Connection object is not created unless the following conditions occur:

- The application calls the getConnection() method on the data source or connection factory.
- No connections are available in the free pool (NOT(freeConnectionAvailable)).
- The pool size is less than the maximum pool size (poolSizeLTMax).
- If the request is for a sharable connection and there is no sharable connection already in use with the same sharing properties (NOT(shareableConnectionAvailable)) OR the request is for an unshareable connection (unshareableConnectionRequest).

All connections begin in the DoesNotExist state and are only created when the application requests a connection. The pool grows from 0 to the maximum number of connections as applications request new connections. The pool is not created with the minimum number of connections when the server starts.

If the request is for a sharable connection and a connection with the same sharing properties is already in use by the application, the connection is shared by two or more requests for a connection. In this case, a new connection is not created. For users of the JDBC API these sharing properties are most often userid/password and transaction context; for users of the Resource Adapter Common Client Interface (CCI) they are typically ConnectionSpec, Subject, and transaction context.

InFreePool

The transition from the InFreePool state to the InUse state is the most common transition when the application requests a connection from the pool.

```
InFreePool > InUse:
getConnection AND
freeConnectionAvailable AND
(unshareableConnectionRequest OR
NOT(shareableConnectionAvailable))
```

This transition states that a connection is placed in use from the free pool if:

- The application has issued a getConnection() call.

- A connection is available for use in the connection pool (`freeConnectionAvailable`), and one of the following is true:
 - The request is for an unshareable connection (`unsharableConnectionRequest`).
 - No connection with the same sharing properties is already in use in the transaction. (`NOT(sharableConnectionAvailable)`).

Any connection request that a connection from the free pool can fulfill does not result in a new connection to the database. Therefore, if there is never more than one connection used at a time from the pool by any number of applications, the pool never grows beyond a size of one. This number can be less than the minimum number of connections specified for the pool. One way that a pool grows to the minimum number of connections is if the application has multiple concurrent requests for connections that must result in a newly created connection.

InUse

The idea of connection sharing is seen in the transition on the InUse state:

```
InUse > InUse:
getConnection AND
ShareableConnectionAvailable
```

This transition states that if an application requests a shareable connection (`getConnection`) with the same sharing properties as a connection that is already in use (`ShareableConnectionAvailable`), the existing connection is shared.

The same user (user name and password, or subject, depending on authentication choice) can share connections but only within the same transaction and only when all of the sharing properties match. For JDBC connections, these properties include the `isolationLevel` which is configurable on the resource-reference (IBM WebSphere extension) to data source default. For a resource adapter factory connection, these properties include those specified on the `ConnectionSpec`. Because a transaction is normally associated with a single thread, you should never share connections across threads.

Note: It is possible to see the same connection on multiple threads at the same time, but this situation is an error state usually caused by an application programming error.

Returning connections

All of the transitions so far have covered getting a connection for application use. From this point, the transitions result in a connection closing and either returning to the free pool or being destroyed. Applications should explicitly close connections (note: the connection that the user gets back is really a connection handle) by calling `close()` on the `Connection` object. In most cases, this action results in the following transition:

```
InUse > InFreePool:
(close AND
noOtherReferences AND
NoTx AND
UnshareableConnection)
OR
(ShareableConnection AND
TxEnds)
```

Conditions that cause the transition from the InUse state are:

- If the application (or the container) calls `close()` (`close`) and there are no references (`noOtherReferences`) either by the application (application sharing) or by the transaction manager (`NoTx` - who holds a reference when the connection is enlisted in a transaction), the `Connection` object returns to the free pool.

- If the connection was enlisted in a transaction but the transaction manager ends the transaction (txEnds), and the connection was a shareable connection (ShareableConnection), the connection closes and returns to the pool.

When the application calls close() on a connection, it is returning the connection to the pool of free connections; it is not closing the connection to the data store. When the application calls close() on a currently shared connection, the connection is not returned to the free pool. Only after the application drops the last reference to the connection, and the transaction is over, is the connection returned to the pool. Applications using unshareable connections must take care to close connections in a timely manner. Failure to do so can starve out the connection pool making it impossible for any application running on the server to get a connection.

When the application calls close() on a connection enlisted in a transaction, the connection is not returned to the free pool. Because the transaction manager must also hold a reference to the connection object, the connection cannot return to the free pool until the transaction ends. Once a connection is enlisted in a transaction, you cannot use it in any other transaction by any other application until after the transaction is complete.

There is a case where an application calling close() can result in the connection to the data store closing and bypassing the connection being returned to the pool. This situation happens if one of the connections in the pool is considered stale. A connection is considered stale if you can no longer use it to contact the data store. For example, a connection is marked stale if the data store server is shut down. When a connection is marked as stale, the entire pool is cleaned out by default because it is very likely that all of the connections are stale for the same reason (or you can set your configuration to clean just the failing connection). This cleansing includes marking all of the currently InUse connections as stale so they are destroyed upon closing. The following transition states the behavior on a call to close() when the connection is marked as stale:

```
InUse > DoesNotExist:
close AND
markedStale AND
NoTx AND
noOtherReferences
```

This transition states that if the application calls close() on the connection and the connection is marked as stale during the pool cleansing step (markedStale), the connection object closes to the data store and is not returned to the pool.

Finally, you can close connections to the data store and remove them from the pool.

This transition states that there are three cases in which a connection is removed from the free pool and destroyed.

- If a fatal error notification is received from the resource adapter (or data source). A fatal error notification (FatalErrorNotification) is received from the resource adapter when something happens to the connection to make it unusable. All connections currently in the free pool are destroyed.
- If the connection is in the free pool for longer than the unused timeout period (UnusedTimeoutExpired) and the pool size is greater than the minimum number of connections (poolSizeGTMin), the connection is removed from the free pool and destroyed. This mechanism enables the pool to shrink back to its minimum size when the demand for connections decreases.
- If an age timeout is configured and a given connection is older than the timeout. The number of connections in the pool can shrink below the minimum size if connections are removed based on the age timeout value. This mechanism provides a way to recycle connections based on age.

Unshareable and shareable connections: WebSphere Application Server - Express supports both unshareable and shareable connections. An unshareable connection is not shared with other components in the application. The component using this connection has full control of this connection.

You can share a shareable connection with other components within the same transaction as long as each `getConnection` request has the same connection properties. To enable connection sharing for data sources, the following connection properties must be the same:

- Java Naming and Directory Interface (JNDI) name. While not actually a connection property, this requirement simply means that you can only share connections from the same `DataSource` in the same server.
- Resource authentication
- In relational databases:
 - Isolation level
 - Readonly
 - Catalog
 - TypeMap
- In addition, the `ConnectionSpec` used to get the connection must also be the same.

Access to a resource marked as `Unshareable` means that there is a one-to-one relationship between the connection handle a component is using and the physical connection the handle is associated with. This access implies that every call to `getConnection` returns a connection handle solely for the requesting user. Typically, you must choose `unshareable` if you might do things to the connection that could result in unexpected behavior occurring to another application that is sharing the connection (for example, changing the isolation level).

Marking a resource as `Shareable` allows for greater scalability. Instead of creating new physical connections on every `getConnection` invocation, the physical connection (that is, managed connection) is shared through multiple connection handles, as long as each `getConnection` request has the same connection properties. But, sharing a connection means that each user must not do anything to the connection that could change its behavior and disrupt a sharing partner (for example, changing the isolation level). The user also cannot code an application expecting sharing to take place because it is up to the run time to decide whether or not to share a particular connection.

For WebSphere Application Server - Express, all sharing of connections is relative to the current Unit of Work (UOW) boundary. Anyone within a specific transaction, when getting a connection from a specific connection pool, gets a handle to the same physical connection (if the sharing properties are the same).

Factors that determine sharing

The listing here is not an exhaustive one. The product might or might not share connections under different circumstances.

- Only connections acquired with the same resource reference (`resource-ref`), which specifies the `res-sharing-scope` as `Shareable`, are candidates for sharing. The `resource-ref` properties of `res-sharing-scope`, `res-auth`, and `res-isolation-level` help determine if it is possible to share a connection. The `res-isolation-level` is a WebSphere extension.
- You can only share connections that are requested with the same properties.
- Connection Sharing only occurs between different component instances if they are within a transaction (container- or user-initiated transaction).
- Connection Sharing only occurs within a sharing boundary. Current sharing boundaries include Transactions and `LocalTransactionContainment` (LTC) boundaries.
- Within an LTC Scope for shareable connections, only Connection Reuse is allowed within a single component instance. Connection reuse occurs when the following actions are taken with a connection: `get, use, commit/rollback, close; get, use, commit/rollback, close`. Note that if you use the LTC `resolution-control` of `ContainerAtBoundary` then no `start/commit` is needed because that action is handled by the container.

The connection returned on the second get is the same connection as that returned on the first get (if the same properties are used). Because the connection use is serial, only one connection handle to the underlying physical connection is used at a time, so true connection sharing does not take place. The term “reuse” is more accurate. This reuse feature enables an application to issue multiple getConnection() requests from a single dataSource (connection pool) within a single phase transaction.

- You cannot set the IsolationLevel when using a shareable connection for the JDBC API using a relational resource adapter in a global transaction. The product provides an extension to the resource reference to enable you to specify the isolation level. If your application requires the use of multiple isolation levels, create multiple resource references and map them to the same data source or connection factory.

Connection handles: A connection handle is a representation of a physical connection.

To use a back end resource (such as a relational database) in the WebSphere Application Server - Express you must get a connection to that resource. When you call the getConnection() method, you get a connection handle returned. The handle is not the physical connection. The physical connection is managed by the connection manager.

There are two significant configurations or usage patterns that affect how connection handles are used and how they behave. The first is the res-sharing-scope, which is defined by the resource-reference used to look up the DataSource or Connection Factory. This property tells the connection manager whether or not you can share this connection.

The second factor that affects connection handle behavior is the usage pattern. There are essentially two usage patterns. The first is called the **get/use/close** pattern. This usage pattern is where an application, within a single method and without calling another method that might get a connection from the same data source or connection factory:

1. Get a connection.
2. Do the work.
3. Commit (if appropriate).
4. Close the connection.

The second usage pattern is called the **cached handle** pattern. This is where an application performs these steps:

1. Get a connection.
2. Begin a global transaction.
3. Do work on the connection.
4. Commit a global transaction.
5. Do work on the connection again.

A cached handle is a connection handle that is held across transaction and method boundaries by an application. Cached handle support requires some additional connection handle management across these boundaries, which can impact performance. For example, in a JDBC application, Statements, PreparedStatements, and ResultSets are closed implicitly after a transaction ends, but the connection remains valid.

Note: There is limited connection caching available for servlets. Connection handles can be cached only in single-threaded servlets. Caching of connection handles across servlet methods is limited to JDBC resources. Other non-relational resources, such as Customer Information Control System (CICS) or Information Management System (IMS), currently cannot have their connection handles cached in a servlet. When connection caching is not available, you must get, use, and close the connection handle within each method invocation.

You are encouraged not to cache the connection across the transaction boundary for shareable connections, the get/use/close pattern is preferred. This code segment shows the cached connection pattern:

```
Connection conn = ds.getConnection();
ut.begin();
conn.prepareStatement("...."); // Connection runs in global transaction mode
...
ut.commit();
conn.prepareStatement("...."); // Connection still valid but runs in autoCommit(True);
...
```

Unshareable connections

Some characteristics of connection handles retrieved with a res-sharing-scope of unshareable are described in the following topics.

Here are the possible benefits of unshared connections:

- Your application always maintains a direct link with a physical connection (managed connection).
- The connection always has a one-to-one relationship between the connection handle and the managed connection.
- In most cases, the connection does not close until the application closes it.
- You can use a cached unshared connection handle across multiple transactions.
- The connection can have a performance advantage in some cached handle situations. Because unshared connections do not have the overhead of moving connection handles off managed connections at the end of the transaction, there is less overhead in using a cached unshared connection.

Here are the possible drawbacks of unshared connections:

- Inefficient use of your connection resources. For example, if within a single transaction you get more than one connection (with the same properties) using the same data source or connection factory (same resource-ref) then you use multiple physical connections when you use unshareable connections. This situation also precludes the use of a single phase transaction.
- Wasted connections. It is important not to keep the connection handle open (that is, you have not called the close() method) any longer than it is needed. As long as you keep an unshareable connection open you tie up the physical connection, even if you currently are not using it.
- Deadlock considerations. Depending on how your components interact with the database within a transaction, using unshared connections can lead to deadlocks in the database. For example, within a transaction, component A gets a connection to data source X and updates table 1, and then calls component B. Component B gets another connection to data source X, and updates/reads table 1 (or even worse the same row as component A). In some circumstances, depending on the particular database, its locking scheme, and the transaction isolation level, a deadlock can occur.

In the same scenario, but with a shared connection, a deadlock does not occur because all the work was done on the same connection. It is worth noting that when writing code which uses shared connections, it is important that the code be written in such a way that it expects other work to be done on the same connection, possibly within the same transaction. If you decide to use an unshareable connection, you must set the maximum connections property on the connection factory or data source correctly. An exception occurs if you try to exceed the maximum connections value.

Shareable connections

Some characteristics of connection handles retrieved with a res-sharing-scope of shareable are described in the following topics.

Here are the possible benefits of shared connections:

- They can share a managed connection with one or more connection handles within a sharing, boundary depending upon how the handle is retrieved and which connection properties are used.
- They can more efficiently use resources. Shareable connections are not valid outside of their sharing boundary. For this reason, when using the cached handle pattern, a connection handle is no longer associated with a managed connection when a sharing boundary (such as a transaction) ends. The managed connection is returned to the free connection pool for reuse. Connection resources are not held longer than the end of the current sharing scope.

If the cached handle pattern is used, then the next time the handle is used within a new sharing scope, the application server run time assures that the handle is reassociated with a managed connection appropriate for the current sharing scope and with the same properties with which the handle was originally retrieved. Remember that it is not appropriate to change properties on a shareable connection. If properties are changed, other components that share the same connection might experience unexpected behavior. Furthermore, when using cached handles, the value of the changed property might not be remembered across sharing scopes.

Here are the possible drawbacks of shared connections:

- Sharing within a single component (such as an enterprise bean and its related Java objects) is not always supported. The current specification allows resource adapters the choice of only allowing one active connection handle at a time.

If a resource adapter chooses to implement this option then the following scenario results in an invalid handle exception: A component using shareable connections gets a connection and uses it. Without closing the connection, the component calls a utility class (Java object) which gets a connection (handle) to the same managed connection and uses it. Because the resource adapter only supports one active handle, the first connection handle is no longer valid. If the utility object returns without closing its handle, the first handle remains invalid and use of it causes an exception.

Note: This exception occurs only when calling a utility object (a Java object).

Not all resource adapters have this limitation, it depends on their implementation. The WebSphere Relational Resource Adapter (RRA) does not have this limitation. Any DataSource used through the RRA does not have this limitation. If you encounter a resource adapter with this limitation you can work around it by serializing your access to the managed connection. If you always close your connection handle before getting another, or close your handle before calling code which gets another handle, and you always close your handle before you return from the method, you can allow two pieces of code to share the same managed connection. You just cannot use the connection for both events at the same time.

- Trying to change the isolation level on a shareable JDBC based connection in a global transaction (those supported by the RRA) causes an exception. The correct way to get connections with different transaction isolation levels is by configuring the IBM extended resource-reference.
- Closing connection handles for shareable connections by an application is not supported and causes errors. However, you can avoid this limitation by using the Relational Resource Adapter.

Connections and transactions: All connection usage occurs within the scope of either a global transaction or a local transaction containment (LTC).

Connection behavior depends on your current operating scope. This topic discusses some of the common characteristics you see when using connections in one of the transaction scopes of WebSphere Application Server - Express.

You can only share connections within a global transaction scope (assuming other sharing rules are met). However, you can serially reuse connections within an LTC scope. A get/use/close connection pattern followed by another get/use/close (to the same data source or connection factory) enables you to reuse the same connection. See “Unshareable and shareable connections” on page 54 for more details.

JDBC AutoCommit behavior

All JDBC connections, when first obtained through a `getConnection` call, have `AutoCommit` set to “true” by default.

- If you operate within an LTC and have its resolution-control set to `Application`, then `AutoCommit` remains `TRUE` unless changed by the application.
- If you operate within an LTC and have its resolution-control set to `ContainerAtBoundary`, then the application should not touch the `AutoCommit` setting. The WebSphere Application Server - Express run time sets the `AutoCommit` value to `FALSE` before work begins, then commits or rolls back the work as appropriate at the end of the LTC scope.
- If you use a connection within a global transaction, then regardless of the user changing the `AutoCommit` setting, upon first use of the connection to do work the database ignores the `AutoCommit` setting so that the transaction service that controls the commit and rollback processing can manage the transaction. After the transaction completes, the `AutoCommit` value returns to the value it had before the first use of the connection. So even if the `AutoCommit` value is set to `TRUE` before the connection is used in a global transaction, you need not set the value to `FALSE` since the value is ignored by the database. In this example, after the transaction completes, the `AutoCommit` value of the connection returns to `TRUE`.
- If you use multiple distinct connections within a global transaction, all work is guaranteed to commit or roll back together. This is not the case for a local transaction containment (LTC scope). Within an LTC, work done on one connection commits or rolls back independently from work done on any other connection within the LTC.

One phase commit and two phase commit resources

One phase commit resources are such that work being done on a one phase connection cannot mix with other connections and ensure that the work done on all of the connections completes or fails atomically . The product does not allow more than one one-phase commit connection in a global transaction. Additionally, the product does not allow a one phase commit connection in a global transaction with one or more two phase commit connections. You can coordinate only multiple two phase commit connections within a global transaction.

Note that any time you do multiple `getConnection()` calls using a resource reference that specifies `res-sharing-scope=Unshareable`, then you get multiple physical connections. This situation also occurs when `res-sharing-scope=Shareable` but the sharing rules are broken. In either case, if you run in a global transaction, ensure the resources involved are enabled for two phase commit (also sometimes referred to as JTA Enabled). Failure to do so results in an `XAException` that logs the following message:
WTRN0063E: An illegal attempt to enlist a one phase capable resource with existing two phase capable resources has occurred.

Connection pooling: When accessing any database, the initial database connection is an expensive operation. Connection pooling enables administrators to establish a pool of database connections that applications can share on an application server. When connection pooling capabilities are used, performance improvements up to 20 times the normal results are realized.

Each time a resource attempts to access a back-end store (such as a database), the resource must connect to that data store. A connection requires resources to create, maintain, and then release the connection when it is no longer required.

The total data store overhead for an application is particularly high for Web-based applications because Web users connect and disconnect more frequently. In addition, user interactions are typically shorter. Often, more effort is spent connecting and disconnecting than is spent during the interactions. Also, because Internet requests can arrive from virtually anywhere, you can find usage volumes large and difficult to predict.

To help lessen these overhead problems, WebSphere Application Server - Express enables administrators to establish a pool of back end connections that applications can share on an application server. Connection pooling spreads the connection overhead across several user requests, thereby conserving resources for future requests.

WebSphere Application Server - Express supports JDBC 2.0 Standard Extension APIs to provide support for connection pooling and connection reuse. The connection pool is used to direct JDBC calls within the application.

If clones are used, one data pool exists for each clone. This is important when configuring the database maximum connections.

Benefits of connection pooling

Connection pooling can improve the response time of any application that requires connections, especially Web-based applications. When a user makes a request over the Web to a resource, the resource accesses a data source. With connection pooling, most user requests do not incur the overhead of creating a new connection because the data source can locate and use an existing connection from the pool of connections. When the request is satisfied and the response is returned to the user, the resource returns the connection to the connection pool for reuse. The overhead of a disconnect is avoided. Each user request incurs a fraction of the cost for connecting or disconnecting. After the initial resources are used to produce the connections in the pool, additional overhead is insignificant because the existing connections are reused.

When to use connection pooling

Use WebSphere connection pooling in an application that meets any of the following criteria:

- It cannot tolerate the overhead of obtaining and releasing connections whenever a connection is used.
- It requires Java Transaction API (JTA) transactions within WebSphere Application Server - Express.
- It needs to share connections among multiple users within the same transaction.
- It needs to take advantage of product features for managing local transactions within the application server.
- It does not manage the pooling of its own connections.
- It does not manage the specifics of creating a connection, such as the database name, user name, or password.

How connections are pooled together

Whenever you configure a unique data source or connection factory you are required to give it a unique Java Naming and Directory Interface (JNDI) name. Use this name, along with its configuration information, to create a connection pool. A separate connection pool exists for each configured data source or connection factory.

It is also important to note that when using connection sharing, it is only possible to share connections obtained from the same connection pool.

Avoiding a deadlock

Deadlock can occur if the application requires more than one concurrent connection per thread, and the database connection pool is not large enough for the number of threads. For example, each application thread requires two concurrent database connections, and the number of threads is equal to the maximum connection pool size. Deadlock can occur when both of the following are true:

- Each thread has its first database connection, and all connections are in use.

- Each thread is waiting for a second database connection, and none become available, because all threads are blocked.

To prevent deadlock in this example, the value set for the database connection pool must be at least one higher, allowing one of the waiting threads to complete its second database connection and free up to allow other database connections.

To avoid deadlock, code the application to use, at most, one connection per thread. If the application is coded to require C concurrent database connections per thread, the connection pool must support at least the following number of connections, where T is the maximum number of threads.

$$T * (C - 1) + 1$$

The connection pool settings are directly related to the number of connections that the database server is configured to support. If the maximum number of connections in the pool is raised, and the corresponding settings in the database are not raised, the application fails and SQL exception errors are displayed in the stderr.log file.

Develop data access applications

Various enterprise information systems use different methods for storing data. These back end data stores may be relational databases, procedural transaction programs, or object-oriented databases. WebSphere Application Server - Express provides several options for accessing an information system's back end data store:

- Program directly to the database through the JDBC 2.0 Optional Package API.
- Program to the procedural back end transaction through various J2EE Connector Architecture (JCA) 1.0 compliant connectors.
- Program servlets to indirectly access the back end store through either the JDBC API or JCA compliant connectors.
- Use IBM data access beans, which also use the JDBC API, but give you additional ability to manipulate result sets.

See these topics for information about developing data access applications, including code examples:

"Data access development model" on page 62

This topic discusses the general steps your code performs to access databases.

"IBM extensions to the data access API" on page 63

WebSphere Application Server - Express provides extended data access APIs that you can use to access data.

"Access data with J2EE Connector Architecture connectors" on page 64

See this topic for information about programming your data access applications to the JCA specification.

"Access connection pools from your application components" on page 68

This topic shows an example of how to write code that utilizes the WebSphere Application Server - Express connection manager.

"IBM data access JavaBeans" on page 70

Another option you can use to access data is by using IBM's extension to the JavaBeans specification.

"Data access exceptions" on page 73

See this topic for information about handling exceptions in your data access applications with IBM extensions to data access exceptions.

Data access development model: Follow these general steps to develop a data access application:

1. Decide how to implement data access.

You can access data in various ways:

- By using standard or extended APIs
- With Web components

2. Create a JDBC provider and data source.

An application component uses a connection factory to access a connection instance, which the component then uses to connect to the underlying enterprise information system (EIS). A data source is associated with a JDBC provider that supplies the specific JDBC driver implementation class. The data source represents the JCA connection factory for the relational resource adapter. For more information, see “Configure WebSphere Application Server - Express to access databases” on page 79 or “Configuring Java 2 Connector connection factories” on page 98.

3. Look up a data source or connection factory using a resource reference.

Using a resource reference to access your data source or connection factory is required when running in WebSphere Application Server - Express. For more information, see “Looking up data sources with resource references for relational access.”

4. Get a connection to a data source.

The connection management architecture for both relational and procedural access to enterprise information systems (EIS) is based on the J2EE Connector Architecture (JCA) specification. The Connection Manager (CM), which pools and manages connections within an application server, is capable of managing connections obtained through both resource adapters (RAs) defined by the JCA specification, and DataSources defined by the JDBC 2.0 Extensions Specification.

Looking up data sources with resource references for relational access: Using a resource reference to access your data source or connection factory is required when running in WebSphere Application Server - Express. Some of the reasons follow:

- If a data source is looked up directly, the connection gets all default properties for the missing resource reference. For example, the sharing-scope is a shareable connection resulting in the possibility that the physical connection is the same each time the connection is requested from the data source. This situation can cause a multitude of problems if you expect unshareable connections.
- It relieves the programmer from having to know the name of the actual data source at the target application server.
- You can set the default isolation level for the data source through resource references. With no resource reference you get the default for the JDBC driver you use. For more information, see “Isolation level and resource reference” on page 63.

Use a resource reference (resource-ref) for looking up a data source through the standard Java Naming and Directory Interface (JNDI) naming interface. The JNDI name defined in the resource-ref is a logical name of the data source. Have your application use this JNDI name to look up a data source instead of using the JNDI name that is defined on the data source.

Later, you can substitute the real name during installation of the application EAR file onto the server.

For example, assume that you use a DataSource jdbc/Section as illustrated in the code below.

```
import javax.sql.*;
import javax.rmi.*;
...
DataSource specificDataSource = (DataSource)
    (new InitialContext()).lookup("java:comp/env/jdbc/Section");
```

Using the WebSphere Development Studio Client, specify the name (jdbc/Section) as the resource reference. If you know the name of the DataSource, specify it also. For more information, see the WebSphere Development Studio Client Help.

Isolation level and resource reference: For all JDBC connections (excluding those used by CMP beans), you can specify an isolation level default on the resource reference. For shareable connections that run in global transactions, this default is the only way to set the `isolationLevel` for connections. Trying to directly set the isolation level through the `setTransactionIsolation()` method on a shareable connection that runs in a global transaction is not allowed. To use a different isolation level on connections, you must provide a different resource reference.

Each resource reference associates with one isolation level. When your application uses this resource reference Java Naming and Directory Interface (JNDI) name to look up a data source, every connection returned from this data source using this resource reference has the same isolation level.

Components needing to use shareable connections with multiple isolation levels can create multiple resource references, giving them different JNDI names, and have their code look up the appropriate data source for the isolation level they need. In this way, you use separate connections with the different isolation levels enabled on them.

It is possible to map these multiple resource references to the same configured data source. The connections still come from the same underlying pool, however, the connection manager does not allow sharing of connections requested by resource references with different isolation levels.

For example, a data source is bound to two resource references: `jdbc/RRResRef` and `jdbc/RResRef`. `RRResRef` has the `RepeatableRead` isolation level defined. `RResRef` has the `ReadCommitted` isolation level defined. If your application wants to update the tables or a BMP bean updates some attributes, it can use the `jdbc/RRResRef` JNDI name to look up the data source instance. All connections returned from the data source instance have a `RepeatableRead` isolation level. If the application wants to perform a query for read only, then it is better to use the `jdbc/RResRef` JNDI name to look up the data source.

IBM extensions to the data access API: Applications can access the back end data through the standard J2EE 1.3 defined application programming interfaces (APIs). However, the standard APIs do not always provide a complete solution for an application that runs in an application server. For example, the JDBC programming model sometimes does not completely work with the J2EE Connector Architecture (JCA) Specification (even though the JCA architecture has explicitly specified that it integrates with the JDBC programming model). These gaps cause some incompatibility between the JDBC and JCA programming models.

When getting and using shareable connections in a global transaction, it is not valid to change a property on the connection after you obtain it. Changes can unknowingly affect other users who share the same connection.

The J2EE Connector Architecture (JCA) Specification supports telling the resource adapter the specific properties settings at the time you request the connection (using the `getConnection` method) by passing in a `ConnectionSpec`. The `ConnectionSpec` contains the necessary connection properties used to get a connection. After you obtain a connection from this environment, your application does not need to alter the properties.

The JDBC programming model does not have the same interface to specify the connection properties. Instead, it gets the connection first, then sets the properties on the connection. In the case of a shareable connection, changing the connection properties impacts all the connections shared with the same physical connection.

WebSphere Application Server - Express provides these extensions to fill in the gaps between these two specifications.

- **WSDataSource interface**

This interface extends `javax.sql.DataSource`, and enables a component or an application to specify the connection properties through the WebSphere Application Server - Express `JDBCConnectionSpec` to get a specific connection. The `getConnection(JDBCConnectionSpec)` method returns a specific connection

which has the JCA compliant connection behavior. For more information, see Interface WSDataSource



in the Javadoc.

- **JDBCConnectionSpec interface**

This interface extends the `com.ibm.websphere.rsadapter.WSConnectionSpec`, which extends `javax.resource.cci.ConnectionSpec`. The standard `ConnectionSpec` interface provides only the interface marker without any get and set methods. The `WSConnectionSpec` and `JDBCConnectionSpec` define a set of get and set methods used by the WebSphere Application Server - Express run time. This interface enables the application to specify all the essential connection properties in order to get an appropriate connection. You can create this class from the WebSphere WSRRAFactory. For more information, see Interface `JDBCConnection`



in the Javadoc.

- **WSRRAFactory**

This is the Relational Resource Adapter Factory which allows the user to create a `JDBCConnectionSpec` or other resource adapter related object. For more information, see Class `WSRRAFactory`



in the Javadoc.

Access data with J2EE Connector Architecture connectors: As indicated in the J2EE Connector Architecture (JCA) Specification, each enterprise information system (EIS) needs a resource adapter and a connection factory. This connection factory is then accessed through the following programming model. If you use WebSphere Studio Application Development (WSAD) tools, most of the following deployment descriptors and code are generated for you. This example shows the manual method of accessing an EIS resource.

For each EIS connection, do the following:

1. Declare a connection factory resource reference in your application component's deployment descriptors, as shown in this example:

```
<resource-ref>
  <description>description</description>
  <res-ref-name>eis/myConnection</res-ref-name>
  <res-type>javax.resource.cci.ConnectionFactory</res-type>
  <res-auth>Application</res-auth>
</resource-ref>
```

2. Configure, during deployment, each resource adapter and associated connection factory through the console. See "Installing Java 2 Connector resource adapters" on page 98 and "Configuring Java 2 Connector connection factories" on page 98 for more information.
3. Locate the corresponding connection factory for the EIS resource adapter using Java Naming and Directory Interface (JNDI) lookup in your application component, during run time. For more information, see "Example: Connection factory lookup" on page 65.
4. Get the connection to the EIS from the connection factory.
5. Create an interaction from the Connection object.
6. Create an InteractionSpec object. Set the function to execute in the InteractionSpec object.
7. Create a Record instance for the input and output data used by function.
8. Run the function through the Interaction object.
9. Process the record data from the function.
10. Close the connection.

This code segment shows how an application component might create an interaction and execute it on the EIS:

```
...
javax.resource.cci.ConnectionFactory connectionFactory = null;
javax.resource.cci.Connection connection = null;
javax.resource.cci.Interaction interaction = null;
javax.resource.cci.InteractionSpec interactionSpec = null;
javax.resource.cci.Record inRec = null;
javax.resource.cci.Record outRec = null;

try {
    // Locate the application component and perform a JNDI lookup
    javax.naming.InitialContext ctx = new javax.naming.InitialContext();
    connectionFactory = (javax.resource.cci.ConnectionFactory)
        ctx.lookup("java:comp/env/eis/myConnection");

    // create a connection
    connection = connectionFactory.getConnection();

    // Create Interaction and an InteractionSpec
    interaction = connection.createInteraction();
    interactionSpec = new InteractionSpec();
    interactionSpec.setFunctionName("GET");

    // Create input record
    inRec = new javax.resource.cci.Record();

    // Execute an interaction
    interaction.execute(interactionSpec, inRec, outRec);

    // Process the output...
}
catch (Exception e) {
    // Exception Handling
}

finally {
    if (interaction != null) {
        try {
            interaction.close();
        }
        catch (Exception e) { /* ignore the exception*/}
    }

    if (connection != null) {
        try {
            connection.close();
        }
        catch (Exception e) { /* ignore the exception */}
    }
}
}
```

See the “Code license and disclaimer information” on page 184 for legal information about this code example.

Example: Connection factory lookup:

```
import javax.resource.cci.*;
import javax.resource.ResourceException;
import javax.naming.*;
import java.util.*;

/**
 * This class is used to look up a connection factory.
 */
public class ConnectionFactoryLookup {
```

```

String jndiName = "java:comp/env/eis/SampleConnection";
boolean verbose = false;

/**
 * main method
 */
public static void main(String[] args) {
    ConnectionFactoryLookup cfl = new ConnectionFactoryLookup();
    cfl.checkParam(args);

    try {
        cfl.lookupConnectionFactory();
    }
    catch(javax.naming.NamingException ne) {
        System.out.println("Caught this " + ne);
        ne.printStackTrace(System.out);
    }
    catch(javax.resource.ResourceException re) {
        System.out.println("Caught this " + re);
        re.printStackTrace(System.out);
    }
}

/**
 * This method does a simple Connection Factory lookup.
 *
 * After the Connection Factory is looked up, a connection is got from
 * the Connection Factory. Then the Connection MetaData is retrieved
 * to verify the connection is workable.
 */
public void lookupConnectionFactory()
    throws javax.naming.NamingException, javax.resource.ResourceException {

    javax.resource.cci.ConnectionFactory factory = null;
    javax.resource.cci.Connection conn = null;
    javax.resource.cci.ConnectionMetaData metaData = null;

    try {
        // lookup the connection factory
        if (verbose) {
            System.out.println("Look up the connection factory...");
        }

        InitialContext ic = new InitialContext();
        factory = (ConnectionFactory) ic.lookup(jndiName);

        // Get connection
        if (verbose) System.out.println("Get the connection...");
        conn = factory.getConnection();

        // Get ConnectionMetaData
        metaData = conn.getMetaData();

        // Print out the metadata Information.
        if (verbose) System.out.println(" ** EISProductName   :"+
            + metaData.getEISProductName());
        if (verbose) System.out.println("      EISProductVersion:"
            + metaData.getEISProductVersion());
        if (verbose) System.out.println("      UserName       :"+
            + metaData.getUserName());

        System.out.println("Connection factory " + jndiName +
            " is successfully looked up");
    }
    catch (javax.naming.NamingException ne) {
        // Connection factory cannot be looked up.
    }
}

```

```

        throw re;
    }
    catch (javax.resource.ResourceException re) {
        // Something wrong with connections.
        throw re;
    }
}

finally {
    if (conn != null) {
        try {
            conn.close();
        }
        catch (javax.resource.ResourceException re) {
        }
    }
}
}

/**
 * Check and gather all the parameters.
 */
private void checkParam(String args[]) {
    int i = 0, j;
    String arg;
    char flag;
    boolean help = false;

    // parse out the options
    while (i < args.length && args[i].startsWith("-")) {
        arg = args[i++];

        // get the database name
        if (arg.equalsIgnoreCase("-jndiName")) {
            if (i < args.length) {
                jndiName = args[i++];
            }
            else {
                System.err.println("-jndiName requires a "
                    + "J2C Connection Factory JNDI name");
                break;
            }
        }
        else { // check for verbose, cmp , bmp
            for (j = 1; j < arg.length(); j++) {
                flag = arg.charAt(j);
                switch (flag) {
                    case 'v' :
                    case 'V' :
                        verbose = true;
                        break;
                    case 'h' :
                    case 'H' :
                        help = true;
                        break;
                    default :
                        System.err.println("illegal option " + flag);
                        break;
                }
            }
        }
    }
}

if ((i != args.length) || help) {
    System.err.println("Usage: java ConnectionFactoryLookup [-v] [-h]");
    System.err.println("    [-jndiName the J2C Connection Factory JNDI name]");
    System.err.println("-v=verbose");
    System.err.println("-h=this information");
}

```

```

        System.exit(1);
    }
}

```

Access connection pools from your application components: This topic describes how to access connection pools from application components such as servlets and JavaServer Pages (JSP) files. This includes instructions for obtaining a connection from a connection pool and returning that connection back to the pool once the application component is finished using it.

Connection pooling is defined as part of the JDBC 2.0 Optional Package. Another part of the Optional Package provides for the use of the Java Naming and Directory Interface (JNDI) and DataSource objects to access relational databases. IBM has implemented these extensions in WebSphere Application Server - Express. These extensions are explained using code fragments that show how relational databases are accessed by application components using the JDBC 2.0 Optional Package.

JDBC 2.0



defines the Java APIs for access to relational databases. With the introduction of JDBC 2.0, the APIs have been split into two parts:

- **The JDBC 2.0 Core API**



The Core API contains evolutionary improvements, but has been kept small and focused in order to promote ease of use. The 2.0 API classes and interfaces are located in the `java.sql` package that is shipped with each JDBC driver. You can use these classes and interfaces in your application components by using this import statement: `import java.sql.*;`

- **The JDBC 2.0 Optional Package**



The Optional Package defines specific kinds of additional functionality when vendors are ready to provide the functionality and programmers are ready to use the functionality. WebSphere Application Server - Express supports the JDBC 2.0 Optional Package and implements connection pooling. Therefore, application components can be coded to make efficient use of database connection resources. The JDBC 2.0 Optional Package classes and interfaces are located in the `javax.sql` package that is shipped with each JDBC driver. You can use these classes and interfaces in your application components by using this import statement: `import javax.sql.*;`

WebSphere Application Server - Express uses JNDI caching. This function has implications on the results received from JNDI lookups. Because data sources are generally static, this should not affect most application components using connection pooling.

Connection pooling can be used by application components such as servlets and JSP files.

To obtain a connection in your application components, you must obtain a connection from a connection pool using a data source object, use that connection, and then return the connection back to the connection pool. These coding concepts are shown in the code fragments in these steps:

1. Make sure that `j2ee.jar` is specified in your classpath. These files are necessary to import the packages mentioned in the next step and to successfully compile your application component. These files are in the `/QIBM/ProdData/WebASE51/ASE/lib` directory.
2. Import JDBC packages and IBM implemented extensions. Import the JDBC 2.0 Core classes, the IBM implementation of the JDBC 2.0 Optional Package classes, and the JNDI naming classes. These are the classes needed for database access:

```
import java.sql.*;
import javax.sql.*;
import javax.naming.*;
```

Additionally, import the application component specific packages required for the application components that you are writing. For example, import the following packages if you are writing a servlet application component:

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

- Using JNDI, obtain the initial JNDI context for the naming service. (For more information, see “Obtain the initial JNDI context for the component” on page 115.) JNDI provides a way to store objects and files (such as DataSource objects) as well as a way to retrieve them. The initial JNDI context is used to access the JNDI naming server to retrieve a reference to a DataSource object.
- Using the initial JNDI context, look up a data source object. (For more information, see “Use JNDI to look up Java components” on page 117.) The DataSource object is defined in the JDBC 2.0 Optional Package. The actual name to look up is defined by the DataSource object. See “Configure WebSphere Application Server - Express to access databases” on page 79 for information on configuring a data source and its JDBC provider using the WebSphere Application Server administrative console.

DataSource objects are stored in the jdbc context. For example, if you have a DataSource named AccountDataSource, the JNDI name for that DataSource would be jdbc/AccountDataSource.

Note: The JNDI name can be read from an external property file or application environment variable, making the application component more portable.

- The DataSource object is a **factory** that you use to get an actual JDBC Connection object. For each client request made to the application component, a connection is retrieved from the DataSource object. For example:

```
conn = ds.getConnection();
```

Note: Obtaining a connection from the DataSource object retrieves an available connection from the connection pool that is associated with the DataSource object.

- Use the Connection object to create one or more Statement objects. Queries and updates use Statement objects to perform SQL interactions with the database. The JDBC 2.0 Core API provides classes and interfaces for accessing relational database. For example, you could use a Statement object to run an SQL SELECT statement against a relational database file that returns its results in the form of a ResultSet object.
- Close all result set and statement objects that were used by the connection:

```
myResultSet.close()
myStatement.close()
```

Note: Implicitly, these objects are closed when the connection is closed, but explicitly closing them can improve performance and can eliminate JDBC timing issues that can cause memory leaks in your application components.

- At the end of each client request, free the connection resource:

```
conn.close()
```

Note: The Connection object is not actually closed when the close() method is invoked on it, so there is no close-related overhead. Instead, the Connection object is returned to the connection pool for reuse. In a similar fashion, the earlier getConnection() method on the DataSource object did not incur the overhead of creating a new Connection object, but instead returned an existing Connection object from the connection pool. (A new Connection object is created only if the connection pool does not have an available Connection object.)

In some application environments or with some data sources, there might be no automatic connection pooling mechanism associated with a DataSource object. Thus, every getConnection() method call on a DataSource object and the related close() operation can in fact incur the overhead of creating a new connection and closing that connection. Within WebSphere Application Server - Express, however, using a DataSource object to get a Connection object automatically provides the efficiencies of connection pooling.

IBM data access JavaBeans: Data access beans provide a rich set of features and function, while hiding much of the complexity associated with accessing relational databases. They are Java classes that are written to the JavaBeans Specification. You can use the data access beans in JavaBeans-compliant tools, such as the IBM WebSphere Studio Application Developer (WSAD). Because the data access beans are also Java classes, you can use them like ordinary classes.

“Data access JavaBeans overview”

See this topic for benefits and features of data access JavaBeans.

“Example: Using WebSphere Application Server Version 4.0 data access beans” on page 71

This code example shows how to develop a data access JavaBean application with the WebSphere Application Server 4.0 APIs, which are still supported in WebSphere Application Server - Express.

“Example: Using WebSphere Application Server - Express data access beans” on page 72

This code example shows how to develop a data access JavaBean application with the WebSphere Application Server - Express APIs.

Data access JavaBeans overview: The data access beans are located in the com.ibm.db package, and they offer these capabilities:

- **Caching query results**
You can retrieve SQL query results all at once and place them in a cache. Programs that use the result set can move forward and backward through the cache or jump directly to any result row in the cache. For large result sets, the data access beans provide ways to retrieve and manage packets, which are subsets of the complete result set.
- **Updating through result cache**
Programs can use standard Java statements (rather than SQL statements) to change, add, or delete rows in the result cache. You can propagate changes to the cache in the underlying relational table.
- **Querying parameter support**
The base SQL query is defined as a Java String, with parameters that replace some of the actual values. When the query runs, the data access beans provide a way to replace the parameters with values that are made available at run time. Default mappings for common data types are provided, but you can specify whatever your Java program and database require.
- **Supporting metadata**
A StatementMetaData object contains the base SQL query. Information about the query (metadata) enables the object to pass parameters into the query as Java data types. Metadata in the object maps Java data types to SQL data types (as well as the reverse). When the query runs, the Java-datatype parameters are automatically converted to SQL data types as specified in the metadata mapping. When results are returned, the metadata object automatically converts SQL data types back into the Java data types that are specified in the metadata mapping.

Data access beans are essentially a class library that makes it easier to access a database. The library contains a set of beans with methods that access the database through the JDBC API. There are several sets of classes referred to as data access beans. To make things clearer, you can refer to the classes by the name of the JAR file that contains them:

- **databeans.jar**
This JAR file ships with the WebSphere Application Server - Express. This file contains classes that enable you to access the database using the JDBC API.
- **ivjdab.jar**
This JAR file ships with IBM Visual Age for Java. This file contains all of the classes in the databeans.jar file and classes that support easy use of the data access beans from the VisualAge for Java Visual Composition Editor.
- **dbbeans.jar**
This JAR file ships with WebSphere Studio Site Developer (WSSD) and WebSphere Studio Application

Developer (WSAD). This file contains a set of data access beans to more closely conform to the JDBC 2.0 RowSet standard. It is recommended that you develop any new data access JavaBeans applications with this package.

The com.ibm.db package is provided to support existing applications that use data access beans. If you want to continue using applications that use the com.ibm.db package, see the Version 4.0 API documentation for com.ibm.db. For an example, see "Example: Using WebSphere Application Server Version 4.0 data access beans."

If you want to create new applications that use the com.ibm.db.beans package, see the WSAD documentation concerning data access beans. For an example, see "Example: Using WebSphere Application Server - Express data access beans" on page 72.

Example: Using WebSphere Application Server Version 4.0 data access beans: This example shows the code for a WebSphere Application Server Version 4.0 data access JavaBean. The example was created with VisualAge for Java.

See the "Code license and disclaimer information" on page 184 for legal information about this code example.

```
package examples;
```

```
import com.ibm.db.uibbeans.*;
import com.ibm.db.*;
/**
 * This type was created in VisualAge.
 */

public class SelectStatementExample {
/**
 * GenericTest constructor comment.
 */

    public SelectStatementExample() {
        super();
    }

/**
 * Starts the application.
 * @param args an array of command-line arguments
 */
    public static void main(java.lang.String[] args) {

        // Objects
        SelectStatement stmt = new SelectStatement();
        DatabaseConnection conn = new DatabaseConnection();
        StatementMetaData metaData = new StatementMetaData();
        SelectResult result;

        // Set properties for connection
        conn.setDriverName("com.ibm.db2.jdbc.app.DB2Driver");
        conn.setDataSourceName("jdbc:db2:Sample");
        conn.setUserID("userid");
        conn.setPassword("password");

        // Set SQL statement
        metaData.setSQL("SELECT * FROM DEPARTMENT");

        // Associate connection and metadata with stmt
        stmt.setConnection(conn);
        stmt.setMetaData(metaData);

        try {
            // Execute SQL statement
```

```

stmt.execute();

// Process results
result = stmt.getResult();
for (int i = 1; i <= result.getNumRows(); i++) {
    System.out.println(result.getColumnValueToString(1));
    System.out.println(result.getColumnValueToString(2));
    result.nextRow();
}

// Release JDBC resources
result.close();

// Close the database connection
conn.disconnect();
}
catch (DataException ex) {
    ex.printStackTrace();
}
}
}

```

Example: Using WebSphere Application Server - Express data access beans: This example shows the code for a WebSphere Application Server - Express data access JavaBean.

See the “Code license and disclaimer information” on page 184 for legal information about this code example.

```

package example;
import com.ibm.db.beans.*;
import java.sql.SQLException;

public class DBSelectExample {

    public static void main(String[] args) {

        DBSelect select = null;
        select = new DBSelect();

        try {

            // Set database connection information
            select.setDriverName("com.ibm.db2.jdbc.app.DB2Driver");
            select.setUrl("jdbc:db2:SAMPLE");
            select.setUsername("userid");
            select.setPassword("password");

            // Specify the SQL statement to be executed
            select.setCommand("SELECT * FROM DEPARTMENT");

            // Execute the statement and retrieve the result set into the cache
            select.execute();

            // If result set is not empty
            if (select.onRow()) {
                do {
                    // display first column of result set
                    System.out.println(select.getColumnAsString(1));
                    System.out.println(select.getColumnAsString(2));
                } while (select.next());
            }

            // Release the JDBC resources and close the connection
            select.close();

        }
    }
}

```

```

        catch (SQLException ex) {
            ex.printStackTrace();
        }
    }
}

```

Data access exceptions: JDBC applications receive a standard `SQLException` if any JDBC operation fails.

The product provides special exceptions for its relational resource adapter (RRA), to indicate that the connection currently held is no longer valid:

- **“ConnectionWaitTimeout exception” on page 74**
This exception indicates that the application timed out trying to get a connection.
- **“StaleConnectionException” on page 76**
This exception indicates that the connection is no longer valid.

Error mapping

Error mapping is necessary because various database vendors can provide differing SQL errors and codes that might mean the same things. For example, the `StaleConnectionException` has different codes in different databases. The DB2 SQLCODEs of 1015, 1034, 1036 and so on, indicate that the connection is no longer available because of a temporary database problem. Other database products use other SQLCODEs to indicate the same situation.

To provide portability for applications, WebSphere Application Server - Express provides a `DataStoreHelper` interface to enable mapping of these codes to the WebSphere Application Server - Express exceptions. In addition, WebSphere Application Server - Express enables you to plug in a data source that is not supported by WebSphere persistence. However, the data source must be implemented as either the `XADataSource` or the `ConnectionPoolDataSource`, and it must be in compliance with the JDBC 2.0 specification.

You can achieve application portability through the following:

- **DataStoreHelper interface**
With this interface, each data store platform can plug in its own private data store specific functions that the relational resource adapter run time uses. WebSphere Application Server provides an implementation for each supported JDBC provider.
In addition, the interface also provides a `GenericDataStoreHelper` class for unsupported data sources to use. You can subclass the `GenericDataStoreHelper` or other WebSphere provided helpers to support any new data source.

For more information, see Interface `DataStoreHelper`



in the Javadoc.

The following code segment illustrates how to add two error codes into the error map:

```

public class NewDSHelper extends GenericDataStoreHelper
{
    public NewDSHelper()
    {
        super(null);
        java.util.Hashtable myErrorMap = null;
        myErrorMap = new java.util.Hashtable(2);
        myErrorMap.put(new Integer(-803), myDuplicateKeyException.class);
        myErrorMap.put(new Integer(-1015), myStaleConnectionException.class);
        myErrorMap.put("S1000", MyTableNotFoundException.class);
        setUserDefinedMap(myErrorMap);
        ...
    }
}

```

See the “Code license and disclaimer information” on page 184 for legal information about this code example.

- **WSCallHelper class**

With this class, applications can invoke any JDBC object proprietary methods that are not defined through the administrative console or standard APIs. This helper also enables applications to invoke many non-JDBC object methods.

All methods are static. For more information, see Class WSCallHelper



in the Javadoc.

The following code segment illustrates using this helper class (with a DB2 data source):

```
Connection conn = ds.getConnection();
// get connection attribute
String connectionAttribute =(String) WSCallHelper.jdbcCall(DataSource.class, ds,
    "getConnectionAttribute", null, null);
// setAutoClose to false
WSCallHelper.jdbcCall(java.sql.Connection.class,
    conn, "setAutoClose",
    new Object[] { new Boolean(false)},
    new Class[] { boolean.class });
// get data store helper
DataStoreHelper dsHelper = WSCallHelper.getDataStoreHelper(ds);
```

See the “Code license and disclaimer information” on page 184 for legal information about this code example.

ConnectionWaitTimeout exception: The ConnectionWaitTimeout exception indicates that the application has waited for the number of seconds specified by the connection timeout setting and has not received a connection. This situation can occur when the pool is at maximum size and all of the connections are in use by other applications for the duration of the wait. In addition, there are no connections currently in use that the application can share because either the connection properties do not match, or the connection is in a different transaction.

- When using a Version 4.0 data source, the ConnectionWaitTimeout throws an exception whose class is `com.ibm.ejs.cm.pool.ConnectionWaitTimeoutException`.
- For connection factories, the ConnectionWaitTimeout throws a `ResourceException` whose class is `com.ibm.websphere.ce.j2c.ConnectionWaitTimeoutException`.
- Finally, WebSphere Application Server - Express data sources throw an `SQLException` subclass called `com.ibm.websphere.ce.cm.ConnectionWaitTimeoutException`.

These code examples show how to use the ConnectionWaitTimeoutException with the JDBC API and the J2EE Connector Architecture (JCA) API.

See the “Code license and disclaimer information” on page 184 for legal information about these code examples.

Example: Using ConnectionWaitTimeoutException for the JDBC API

In all cases in which the ConnectionWaitTimeoutException is caught, there is very little to do for recovery. The following code fragment shows how to use this exception in the JDBC API:

```
import java.sql.*;
import java.naming.*;
import javax.rmi.*;

public void test1() {
    Connection conn = null;
    Statement stmt = null;
    ResultSet rs = null;
```

```

try {
    // Look for datasource
    java.util.Properties props = new java.util.Properties();
    props.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.ibm.websphere.naming.WsnInitialContextFactory");
    ic = new InitialContext(props);
    DataSource dsl =
        (DataSource) PortableRemoteObject.narrow(ic.lookup(jndiString), DataSource.class);

    // Get Connection.
    conn = dsl.getConnection();
    stmt = conn.createStatement();
    rs = stmt.executeQuery("select * from mytable where this = 54");
}
catch (com.ibm.websphere.ce.cm.ConnectionWaitTimeoutException cwte) {
    // Notify the user that the system could not provide a connection
    // to the database. This usually happens when the connection pool
    // is full and there is no connection available to share.
}
catch (SQLException sqle) {
    // Handle other database problems.
}
finally {
    if (rs != null) {
        try {
            rs.close();
        }
        catch (SQLException sqle1) {
        }
    }

    if (stmt != null) {
        try {
            stmt.close();
        }
        catch (SQLException sqle1) {
        }
    }

    if (conn != null)
        try {
            conn.close();
        }
        catch (SQLException sqle1) {
        }
    }
}
}

```

Example: Using ConnectionWaitTimeoutException for the JCA API

In all cases in which the ConnectionWaitTimeoutException is caught, there is very little to do for recovery. The following code fragment shows how to use this exception in JCA:

```

import javax.naming.*;
import javax.resource.*;
import javax.resource.cci.*;
import javax.rmi.*;

/**
 * This method does a simple Connection test.
 */
public void testConnection() throws NamingException, ResourceException,
    com.ibm.websphere.ce.j2c.ConnectionWaitTimeoutException {

    ConnectionFactory factory = null;
    Connection conn = null;

```

```

ConnectionMetaData metaData = null;

try {
    // lookup the connection factory
    if (verbose) System.out.println("Look up the connection factory...");
    try {
        factory = (ConnectionFactory) PortableRemoteObject.narrow(
            (new InitialContext()).lookup("java:comp/env/eis/Sample"),
            ConnectionFactory.class);
    }
    catch (NamingException ne) {
        // Connection factory cannot be looked up.
        throw ne;
    }

    // Get connection
    if (verbose) System.out.println("Get the connection...");
    conn = factory.getConnection();

    // Get ConnectionMetaData
    metaData = conn.getMetaData();

    // Print out the metadata Information.
    System.out.println("EISProductName is " + metaData.getEISProductName());
}
catch (com.ibm.websphere.ce.j2c.ConnectionWaitTimeoutException cwtoe) {
    // Connection Wait Timeout
    throw cwtoe;
}
catch (ResourceException re) {
    // Something wrong with connections.
    throw re;
}

finally {
    if (conn != null) {
        try {
            conn.close();
        }
        catch (ResourceException re) {
        }
    }
}
}

```

StaleConnectionException: WebSphere Application Server - Express provides a special subclass of `java.sql.SQLException` when using connection pooling to access a relational database. This `com.ibm.websphere.ce.cm.StaleConnectionException` subclass exists in both a Version 4.0 data source and in the new data source using the relational resource adapter, and is used to indicate that the connection currently held is no longer valid. This situation can occur for many reasons, including the following:

- The application tries to get a connection and fails, as when the database is not started.
- A connection is no longer usable because of a database failure. When an application tries to use a previously obtained connection, the connection is no longer valid. In this case, all connections currently in use by the application can get this error when they try to use the connection.
- The connection is orphaned (because the application had not used it in at most two times the value of the unused timeout setting) and the application tries to use the orphaned connection. This case applies only to Version 4.0 data sources.
- The application tries to use a JDBC resource, such as a statement, obtained on a stale connection. A connection is closed by the Version 4.0 data source auto connection cleanup and is no longer usable. Auto connection cleanup is the standard mode in which connection management operates. This mode indicates that at the end of a transaction, the transaction manager closes all connections enlisted in that

transaction. This enables the transaction manager to ensure that connections are not held for excessive periods of time and that the pool does not reach its maximum number of connections prematurely.

One ramification of having the transaction manager close the connections and return the connection to the free pool after a transaction ends, is that an application cannot obtain a connection in one transaction and try to use it in another transaction. If the application tries this, a `StaleConnectionException` is thrown because the connection is already closed.

In the case of trying to use an orphaned connection or a connection cleaned up by automatic connection cleanup, a `StaleConnectionException` indicates that the application has attempted to use a connection that already returned to the connection pool. It does not indicate an actual problem with the connection. However, other cases of a `StaleConnectionException` indicate that the connection to the database has gone bad, or stale. Once a connection has gone stale, you cannot recover it.

Detecting stale connections

When a connection to the database becomes stale, operations on that connection result in an `SQLException` from the JDBC driver. Because an `SQLException` is a rather generic exception, it contains state and error code values that you can use to determine the meaning of the exception. However, the meanings of these states and error codes vary depending on the database vendor. The connection pooling run time maintains a mapping of which SQL state and error codes indicate a `StaleConnectionException` for each database vendor supported. When the connection pooling run time catches any `SQLException`, it checks to see if this `SQLException` is considered a `StaleConnectionException` for the database server in use.

Recovering from stale connections

Recovering from stale connections is a joint effort between the application server run time and the application developer. From an application server perspective, the connection pool is purged based on its `PurgePolicy` setting.

Explicitly catching a `StaleConnectionException` is not required in an application. Because applications are already required to catch `java.sql.SQLException`, and `StaleConnectionException` extends `SQLException`, `StaleConnectionException` can be thrown from any method that is declared to throw `SQLException`, and is caught automatically in the general catch block. However, explicitly catching `StaleConnectionException` makes it possible for an application to recover from bad connections. When application code catches `StaleConnectionException`, it should take explicit steps to handle the exception.

Handling the `StaleConnectionException` data access exception

When an application receives a `StaleConnectionException` on a database operation, it indicates that the connection currently held is no longer valid. While it is possible to get a `StaleConnectionException` on any database operation, the most common time to see a `StaleConnectionException` thrown is the first time that a connection is used, just after it is retrieved. Because connections are pooled, a database failure is not detected until the operation immediately following its retrieval from the pool, which is the first time communication to the database is attempted. It is only when a failure is detected that the connection is marked stale. `StaleConnectionException` occurs less often if each method that accesses the database gets a new connection from the pool.

Many `StaleConnectionExceptions` are caused by intermittent problems with the network of the database server. Obtaining a new connection and retrying the operation can result in successful completion without exceptions to the end user. In some cases it is advantageous to add a small wait time between the retries to give the database server more time to recover. However, applications should not retry operations indefinitely, in case the database is down for an extended period of time.

Before the application can obtain a new connection for a retry of the operation, roll back the transaction in which the original connection was involved and begin a new transaction. You can break down details on this action into these categories:

- **Objects operating in a global transaction context and transaction not begun in the same method as the database access.**

When the object which receives the `StaleConnectionException` does not have direct control over the transaction, the object must mark the transaction for rollback, and then indicate to its caller to retry the transaction. In most cases, you can do this by throwing an application exception which indicates to retry that operation. However this action is not always allowed, and often a method is defined only to throw a particular exception.

- **Objects operating in a local transaction context.**

When a database operation occurs outside of a global transaction context, a local transaction is implicitly begun by the container. This includes servlets or JSPs which do not begin transactions with the `UserTransaction` interface. As with global transactions, you must roll back the local transaction before the operation is retried. In these cases, the local transaction containment usually ends when the business method ends. The one exception is if you are using activity sessions. In this case the activity session must end before attempting to get a new connection.

When the local transaction containment takes place as part of a servlet or JSP file, there is no client object available to retry the operation. For this reason, it is recommended to avoid database operations in servlets and JSP files unless they are a part of a user transaction.

Assemble data access applications

1. Define the resource reference attributes through the WebSphere Development Studio Client. For more information, see the WebSphere Development Studio Client Help.
2. Bind this resource reference to a resource such as a J2EE Connector Architecture (JCA) connection factory or a data source. During either application assembly or deployment, you must bind the resource reference to the actual name of the resource in the run time environment. You can do this as one of the steps during installation of the application EAR file.
3. Configure isolation level and access intent assembly settings. See “Configuring the isolation level on a resource reference” for more information.

Configuring the isolation level on a resource reference: Specify the appropriate isolation level for each of your resource references. The isolation level is used only by JDBC users.

If a JDBC application or a servlet runs in a global transaction, and you are using shareable connections, you cannot set the isolation level on a connection.

For all JDBC connections, you can specify a default isolation level on the resource reference. For shareable connections that run in global transactions, this default is the only way to set the isolation level for connections. You cannot directly set the isolation level through the `setTransactionIsolation()` method on a shareable connection that runs in a global transaction. To use a different isolation level on connections, you must provide a different resource reference.

Each resource reference associates with one isolation level. When your application uses this resource reference Java Naming and Directory Interface (JNDI) name to look up a data source, every connection returned from this data source using this resource reference has the same isolation level.

Components that need to use shareable connections with multiple isolation levels can create multiple resource references, give them different JNDI names, and have their code look up the appropriate data source for the isolation level they need. In this way, you use separate connections with the different isolation levels enabled on them.

It is possible to map these multiple resource references to the same configured data source. The connections still come from the same underlying pool, however, the connection manager does not allow sharing of connections requested by resource references with different isolation levels.

For example, a data source is bound to two resource references: jdbc/RRResRef and jdbc/RResRef. RRResRef has the RepeatableRead isolation level defined. RResRef has the ReadCommitted isolation level defined. If your application wants to update the tables, it can use the jdbc/RRResRef JNDI name to look up the data source instance. All connections returned from the data source instance have a RepeatableRead isolation level. If the application wants to perform a query for read only, then it is better to use the jdbc/RResRef JNDI name to look up the data source.

You can define multiple resource references for the same servlet. Each resource reference has its own Java Naming and Directory Interface (JNDI) logical name. You can also bind multiple resource references to the same data source. Each JNDI resource reference naming lookup returns the connections from the data source that has the same isolation level.

Configure WebSphere Application Server - Express to access databases

WebSphere Application Server can be configured to access databases through data access applications. An application can be implemented using either the JDBC specification or the J2EE Connection Architecture (JCA) specification.

Select one of the following topics to configure WebSphere Application Server - Express to access databases:

“Configure JDBC data access”

If your data access application implements the JDBC specification, you configure a JDBC provider and a data source to access your database.

“Configure JCA data access” on page 98

If your data access application implements the J2EE Connector Architecture (JCA) specification, you must configure a connection factory and resource adapter to provide connections to your database.

Configure JDBC data access: You must create a JDBC provider and data source for your application to use to access a database. Perform these steps:

1. **(Optional) Configure a J2C authentication data entry** in the *Security* topic.

If your application is configured using WebSphere Application Server security, and you require a user ID and password to access the database, you can configure a J2C authentication data entry to define authentication data, which includes user identity and passwords. Authentication data entries can be referenced by resource adapters, data sources, and other configurations that require authentication data using an alias.

During data source creation, the J2C authentication data entry is used as a part of the Component-managed Authentication Alias and Container-managed Authentication Alias fields.

2. **“Create a JDBC provider and a data source” on page 80.**

A JDBC provider represents a JDBC driver. To access a database, your applications use data sources, which use a JDBC driver to access the database.

3. Bind the resource reference.

4. **“Test the connection” on page 82** (for non-container-managed persistence usage).

The test connection service enables developers to test a connection to a data source.

Note: You can use a JACL script to automate the above steps. For an example of how to create a JDBC provider and data source using Java Management Extensions (JMX) APIs, see “Example: Creating a JDBC provider and data source using Java Management Extensions API and the scripting tool” on page 96.

If your connection is not successful after you complete these steps and review the applicable information, check the SystemOut.log for warning or exception messages. See View the Java virtual machine log files in the *Troubleshooting* topic for more information. You can also use the Technical support search



to find known problems.

Create a JDBC provider and a data source: A JDBC provider represents a JDBC driver. To access a database, your applications use data sources, which use a JDBC driver to access the database.

For information on how to create a JDBC provider and data source, see Manage JDBC providers for your application server in the *Administration* topic.

Consider the following before you create a JDBC provider and data source:

- Decide which data source to use: a Version 4.0 or a Version 5.0 data source.
A Version 5.0 data source is used by a J2EE 1.3 application to access the data from the database. A data source is created under a JDBC provider, which provides the specific JDBC driver implementation class. If your application uses EJB 2.0 modules, only Version 5.0 data sources can be used.
A Version 4.0 data source is used by a J2EE 1.2 application to access the data from the database. A data source is created under a JDBC provider, which provides the specific JDBC driver implementation class. If your application uses EJB 1.1 modules, only Version 4.0 data source can be used.
Although WebSphere Application Server - Express supports WebSphere Application Server Version 4.0 data sources, it is recommended that you use the WebSphere Application Server - Express specifications for all new data sources.
- Decide which JDBC provider to implement.
For more information, see "Available JDBC providers."
- (Optional) If your application is configured using WebSphere Application Server - Express security, and you have created a J2C authentication data entry, the **Component-managed Authentication Alias** field is used for database authentication in run time. If you do not set this field, and your database requires a user ID and password to get a connection, you receive an exception during run time. If your resource authentication (res-auth) is set to Application, set the alias in the Component-managed Authentication Alias.
- (Optional) If your application is configured using WebSphere Application Server - Express security, and you have created a J2C authentication data entry, the **Container-managed Authentication Alias** field is used for database authentication in run time. If you do not set this field, and your database requires a user ID and password to get a connection, then you receive an exception during run time. If your resource authentication (res-auth) is set to Container, set the alias in the Container-managed Authentication Alias.

Available JDBC providers: The following list contains descriptions for JDBC providers supported in WebSphere Application Server - Express. It also shows the supported data source classes and required properties.

Specific fields are designated for the user and password properties. Inclusion of a property in the list does not imply that you should add them to the data source properties list. Rather, inclusion in the list means that a value is typically required for that field.

1. **DB2 UDB for iSeries (Native - Version 5 Release 2 and later)**

The iSeries Developer Kit for Java contains this Type 2 JDBC driver that is built on top of the iSeries DB2 Call Level Interface (CLI) native libraries. Use this driver for local DB2 connections on the iSeries server. It is not recommended for remote access. Use this driver for OS/400 V5R2 or later releases.

- DB2 UDB for iSeries (OS/400 V5R2 and later) supports one phase data source:
`com.ibm.db2.jdbc.app.UDBConnectionPoolDataSource`
- This driver requires the `db2_classes.jar` JDBC driver file.
- This driver requires the following `DataStoreHelper` class:
`com.ibm.websphere.rsadapter.DB2AS400DataStoreHelper`

- This driver requires the following property:
databaseName: The name of the relational database to which the data source connections are established. This name must appear in the iSeries Relational Database Directory. The default is *LOCAL.

2. DB2 UDB for iSeries (Native XA - Version 5 Release 2 and later)

The iSeries Developer Kit for Java contains this XA-compliant Type 2 JDBC driver built on top of the iSeries DB2 Call Level Interface (CLI) native libraries. Only use this driver for local DB2 connections on iSeries. It is not recommended for remote access. Use this driver for iSeries V5R2 or later releases.

- DB2 UDB for iSeries (Native XA - V5R2 and later) supports two phase data source:
`com.ibm.db2.jdbc.app.UDBXADataSource`
- This driver requires the `db2_classes.jar` * JDBC driver files.
- This driver requires the following `DataStoreHelper` class:
`com.ibm.websphere.rsadapter.DB2AS400DataStoreHelper`
- This driver requires the following property:
databaseName: The name of the relational database to which the data source connections are established. This name must appear in the iSeries Relational Database Directory. The default is *LOCAL.

3. DB2 UDB for iSeries (Native - Version 5 Release 1 and earlier)

The iSeries Developer Kit for Java contains this Type 2 JDBC driver that is built on top of the iSeries DB2 Call Level Interface (CLI) native libraries. Only use this driver for local DB2 connections on iSeries. It is not recommended for remote access. Use this driver for iSeries V5R1, or earlier releases.

- DB2 UDB for iSeries (Native V5R1 and earlier) supports one phase data source:
`com.ibm.db2.jdbc.app.DB2StdConnectionPoolDataSource`
- This driver requires the `db2_classes.jar` * JDBC driver files.
- This driver requires the following `DataStoreHelper` class:
`com.ibm.websphere.rsadapter.DB2AS400DataStoreHelper`
- This driver requires the following property:
databaseName: The name of the relational database to which the data source connections are established. This name must appear in the iSeries Relational Database Directory. The default is *LOCAL.

The V5R1 and earlier JDBC drivers are supported, but will not receive any future enhancements. It is recommended that you replace them with the V5R2 JDBC driver providers whenever possible.

4. DB2 UDB for iSeries (Native XA - Version 5 Release 1 and earlier)

The iSeries Developer Kit for Java contains this XA-compliant Type 2 JDBC driver built on top of the iSeries DB2 Call Level Interface (CLI) native libraries. Only use this driver for local DB2 connections on iSeries. It is not recommended for remote access. Use this driver for iSeries V5R1, or earlier releases.

- DB2 UDB for iSeries (Native XA - V5R1 and earlier) supports two phase data source:
`com.ibm.db2.jdbc.app.DB2StdXADataSource`
- This driver requires the `db2_classes.jar` JDBC driver files.
- This driver requires the following `DataStoreHelper` class:
`com.ibm.websphere.rsadapter.DB2AS400DataStoreHelper`
- This driver requires the following properties:
databaseName: The name of the relational database to which the data source connections are established. This name must appear in the iSeries Relational Database Directory. The default is *LOCAL.

The V5R1 and earlier JDBC drivers are supported, but will not receive any future enhancements. It is recommended that you replace them with the V5R2 JDBC driver providers whenever possible.

5. DB2 UDB for iSeries (Toolbox)

This JDBC driver, also known as iSeries Toolbox driver for Java, is provided in the DB2 for iSeries

database server. Use this driver for remote DB2 connections on iSeries. We recommend you use this driver instead of the IBM Developer Kit for Java JDBC Driver to access remote DB2 UDB for iSeries servers.

- DB2 UDB for iSeries (Toolbox) supports one phase data source:
`com.ibm.as400.access.AS400JDBCConnectionPoolDataSource`
- This driver requires the `jt400.jar` * JDBC driver files.
- This driver requires the following `DataStoreHelper` class:
`com.ibm.websphere.rsadapter.DB2AS400DataStoreHelper`
- This driver requires the following property:
serverName: The name of the server from which the data source obtains connections. Example: `myserver.mydomain.com`.

6. DB2 UDB for iSeries (Toolbox XA)

This XA compliant JDBC driver, also known as iSeries Toolbox XA compliant driver for Java, is provided in the DB2 for iSeries database server. Use this driver for remote DB2 connections on iSeries. We recommend you use this driver instead of the IBM Developer Kit for Java JDBC Driver to access remote DB2 UDB for iSeries systems.

- DB2 UDB for iSeries (Toolbox XA) supports two phase data source:
`com.ibm.as400.access.AS400JDBCXADataSource`
- This driver requires the `jt400.jar` JDBC driver files.
- This driver requires the following `DataStoreHelper` class:
`com.ibm.websphere.rsadapter.DB2AS400DataStoreHelper`
- This driver requires the following property:
serverName: The name of the server from which the data source obtains connections. Example: `myserver.mydomain.com`.

For more information on DB2 UDB for iSeries, see the following Web site:

DB2 Universal Database for iSeries Web site

<http://www.ibm.com/servers/eserver/series/db2/>



Test the connection: The test connection service enables developers to test a connection to a data source.

There are several ways to test a connection to a database that uses the parameters defined in a data source in WebSphere Application Server - Express. You can use the administrative console, the `wsadmin` tool, or a Java stand alone program. All three processes invoke the same methods on the same MBean.

- **Administrative console**

WebSphere Application Server - Express enables you to test a connection from the administrative console by simply pushing a button. The Data Source Collection and Data Source Details pages have Test Connection buttons. After you have defined and saved a data source, you can click this button to ensure that the parameters in the data source definition are correct. On the collection page, you can select several data sources and test them all at once. Note that there are certain conditions that must be met first.

See "Testing a connection with the administrative console" on page 83 for more information.

- **Wsadmin tool**

The `wsadmin` tool provides a scripting interface to a full range of WebSphere Application Server - Express administration activities. Because the Test Connection functionality is implemented as a method on an MBean, and `wsadmin` can invoke MBean methods, `wsadmin` can be used to test connections to DataSources. There are three ways to test data source connections from `wsadmin`:

1. Using the testConnection operation in the AdminControl object of wsadmin. This operation tests the configuration properties of a data source object. See “Testing a connection using wsadmin” on page 84 for more information.
2. Using the testConnection facility in wsadmin. See Example: Migrating - Testing the DataSource object connection for more information.
3. Invoking the MBean operation. See Example: Testing data source connection using wsadmin for more information.

- **Java stand alone program**

The test connection commands can also be invoked from a Java program by using Java Management Extensions (JMX) to connect directly to the MBean.

The preferred way of testing connections in WebSphere Application Server - Express is invoking the test connection method through a Java program. The advantage to this method is that you can pass the configuration ID of a configured data source, rather than the properties of the data source. The Java program uses JMX to connect to a running server and invoke the testConnection method on the DataSourceCfgHelper MBean. You connect to the running server, usually on port 8880.

The return value from the invocation is zero (0), a positive number, or an exception. Zero (0) indicates that the operation successfully completed with no warnings. A positive number indicates that the operation successfully completed with the number of warnings. An exception indicates that the test of the connection failed.

See “Example: Test a connection using testConnection(ConfigID)” on page 84 for more information about how to invoke the test connection method by passing in the configuration ID of a configured data source.

See “Example: Test a connection using country and language (properties)” on page 87 for more information about how to invoke the test connection operation on the DataSourceCfgHelper MBean from a Java program that wsadmin uses by passing in the properties you want to test.

See “Example: Test a connection to a data source” on page 91 for more information about how to invoke the test connection operations in previous releases of WebSphere Application Server - Express.

Testing a connection with the administrative console: After you have defined and saved a data source, you can use the Test Connection functionality to make sure that the parameters in the data source definition are correct. On the Collection panel, you can select multiple data sources and test them all at once. Make sure that the following conditions are met before using Test Connection:

To test a connection with the administrative console, perform the following steps:

1. Make sure that a valid Authorization Data alias exists and is used on the data source panels.
2. If you are testing a connection using a WebSphere Application Server Version 4.0 data source, make sure that the user and password information is filled in.
3. If you are using a WebSphere Environment entry for the classpath or other fields, such as \${DB2_JDBC_DRIVER_PATH}/db2java.zip, make sure that you assign it a value in the Manage WebSphere Variables page. The values of \${OS400_NATIVE_JDBC_DRIVER_PATH} and \${OS400_TOOLBOX_JDBC_DRIVER_PATH} are set automatically on an iSeries server.

Note: If you add or modify a new WebSphere Environment Variable, the process (nodeagents and Deployment manager) must be restarted.

4. Verify that the environment variables used exist in the scope of the test.
For example, if the node scoped data source you defined uses \${DB2_JDBC_DRIVER_PATH}, check that a node level definition exists for DB2_JDBC_DRIVER_PATH = c:\sqllib\java (or your system dependent value).
5. Make sure that the Deployment manager and nodeagent are up and running.
6. Click **Test Connection**.

A Test Connection operation can have three different outcomes, each resulting in a different message displayed in the messages panel of the page on which you click Test Connection.

- The test can successfully complete, meaning that a connection is successfully obtained to the database using the configured data source parameters. The resulting message states: Test Connection for DataSource DataSourceName on process ProcessName at node NodeName was successful.
- The test can successfully complete with warnings. This means that while a successful connection is obtained to the database, warnings are issued. The resulting message states: Test Connection for DataSource DataSourceName on process ProcessName at node NodeName was successful with warning(s).View the JVM Logs for more details.
- The test can fail. A connection to the database with the configured parameters is not obtained. The resulting message states: Test Connection failed for DataSource DataSourceName on process ProcessName at node NodeName with the following exception: ExceptionText. View the JVM Logs for more details.

Testing a connection using wsadmin: The AdminControl object of wsadmin has a testConnection operation that tests the configuration properties of a data source object. It takes a configuration ID as an argument.

Note: This invocation style is currently supported only for databases that do not require a user ID and password to make a connection.

To test a connection using wsadmin, perform the following steps:

1. Invoke the getid method for your data source.
2. Set the value of the configuration id to a variable.

```
set myds [$AdminConfig getid /JDBCProvider:mydriver/DataSource:mydatasrc/]
```

where `/JDBCProvider:mydriver/DataSource:mydatasrc/` is the data source you want to test. After you have the configuration ID of the data source, you can test the connection to the database.

3. Test the connection to the database.

```
$AdminControl testConnection $myds
```

Example: Test a connection using testConnection(ConfigID): You can pass the configuration ID of a configured data source, rather than the properties of the data source. This example program uses Java Management Extensions (JMX) to connect to a running server and invoke the testConnection method on the DataSourceCfgHelper MBean.

See the “Code license and disclaimer information” on page 184 for legal information about these code examples.

```
/**
 * Description
 * Resource adapter test program to make sure that the MBean interfaces work.
 * Following interfaces are tested
 *
 * --- testConnection()
 *
 *
 * We need following to run
 *
 * From an iSeries command line:
 *
 * -> QSHELL (Start the QSHELL environment)
 *
 * -> cd (yourDirectory)
 * (change current directory to the directory that holds the java source)
 *
 * -> javac -extdirs /QIBM/ProdData/WebASE51/ASE/lib -d . testDSGUI.java
 * (compile the testDSGUI class)
 *
 * -> ./QIBM/ProdData/WebASE51/ASE/bin/setupClient
 * (produce the $JAVA_FLAGS_EXT environment variable)
 *
 */
```

```

* -> java $JAVA_FLAGS_EXT testDSGUI
* (call the program)
*
*/

import java.util.Iterator;
import java.util.Locale;
import java.util.Properties;
import java.util.Set;

import javax.management.InstanceNotFoundException;
import javax.management.MBeanException;
import javax.management.MalformedObjectNameException;
import javax.management.ObjectName;
import javax.management.RuntimeMBeanException;
import javax.management.RuntimeOperationsException;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.ws.rsadapter.exceptions.DataStoreAdapterException;

public class testDSGUI {

    //Use port 8880 for base installation or port 8879 for ND installation
    String port = "8880";
    // String port = "8879";
    String host = "localhost";
    final static boolean verbose = true;

    // eg a configuration ID for 5.0 DataSource declared at the node level for base
    private static final String resURI = "cells/cat/nodes/cat:resources.xml#DataSource_1";

    // eg a 4.0 DataSource declared at the node level for base
    // private static final String resURI = "cells/cat/nodes/cat:resources.xml#WAS40DataSource_1";

    // eg Cloudscape DataSource declared at the server level for base
    //private static final String resURI = "cells/cat/nodes/cat/servers/server1/resources.xml#DataSource_6";

    // eg node level DataSource for ND
    //private static final String resURI = "cells/catNetwork/nodes/cat:resources.xml#DataSource_1";

    // eg server level DataSource for ND
    //private static final String resURI =
    //     "cells/catNetwork/nodes/cat/servers/server1:resources.xml#DataSource_4";

    // eg cell level DataSource for ND
    //private static final String resURI = "cells/catNetwork:resources.xml#DataSource_1";

    public static void main(String[] args) {
        testDSGUI cds = new testDSGUI();
        cds.run(args);
    }

    /**
     * This method tests the ResourceMbean.
     *
     * @param args
     * @exception Exception
     */
    public void run(String[] args) {

        try {

            System.out.println("Connecting to the application server.....");

            /*****/

```

```

        /** Initialize the AdminClient */
        /*******
Properties adminProps = new Properties();
adminProps.setProperty(AdminClient.CONNECTOR_TYPE, AdminClient.CONNECTOR_TYPE_SOAP);
adminProps.setProperty(AdminClient.CONNECTOR_HOST, host);
adminProps.setProperty(AdminClient.CONNECTOR_PORT, port);
AdminClient adminClient = null;
try {
    /* Port 8880 must be listening when this program is called or the SOAP connection will fail */
adminClient = AdminClientFactory.createAdminClient(adminProps);
} catch (com.ibm.websphere.management.exception.ConnectorException ce) {
    System.out.println("NLS: Cannot make a connection to the application server\n");
    ce.printStackTrace();
    System.exit(1);
}

        /*******
        /** Locate the Mbean */
        /*******
ObjectName handle = null;
try {
    // Send in a locator string
    // eg for a Base installation this is enough
    ObjectName queryName = new ObjectName("WebSphere:type=DataSourceCfgHelper,*");

    // for ND you need to specify which node/process you would like to test from
    // eg run in the server
    //ObjectName queryName = new ObjectName("WebSphere:cell=catNetwork,node=cat,
process=server1,type=DataSourceCfgHelper,*");
    // eg run in the node agent
    //ObjectName queryName = new ObjectName("WebSphere:cell=catNetwork,node=cat,
process=nodeagent,type=DataSourceCfgHelper,*");
    // eg run in the Deployment Manager
    //ObjectName queryName = new ObjectName("WebSphere:cell=catNetwork,node=catManager,
process=dmgr,type=DataSourceCfgHelper,*");
    Set s = adminClient.queryNames(queryName, null);
    Iterator iter = s.iterator();
    while (iter.hasNext()) {
        // use the first MBean that is found
        handle = (ObjectName) iter.next();
        System.out.println("Found this ->" + handle);
    }
    if (handle == null) {
        System.out.println("NLS: Did not find this MBean>>" + queryName);
        System.exit(1);
    }
} catch (MalformedObjectNameException mone) {
    System.out.println("Check the program variable queryName" + mone);
} catch (com.ibm.websphere.management.exception.ConnectorException ce) {
    System.out.println("Cannot connect to the application server" + ce);
}

        /*******
        /** Build parameters to pass to Mbean */
        /*******
String[] signature = { "java.lang.String" };
Object[] params = { resURI };
Object result = null;

if (verbose) {
    System.out.println("\nTesting connection to the database using " + handle);
}

try {
    /*******
    /** Start to test the connection to the database */
    /**

```

```

        /** Note that the datasource must exist prior to calling this program */
        /*******
result = adminClient.invoke(handle, "testConnection", params, signature);
} catch (MBeanException mbe) {
// ***** all user exceptions come in here
if (verbose) {
    Exception ex = mbe.getTargetException(); // this is the real exception from the Mbean
    System.out.println("\nNLS:Mbean Exception was received contains " + ex);
    ex.printStackTrace();
    System.exit(1);
}
} catch (InstanceNotFoundException infe) {
System.out.println("Cannot find " + infe);
} catch (RuntimeMBeanException rme) {
    Exception ex = rme.getTargetException();
    ex.printStackTrace(System.out);
    throw ex;
} catch (Exception ex) {
System.out.println("\nUnexpected Exception occurred: " + ex);
ex.printStackTrace();
}

        /*******
        /** Process the result. The result will be the number of warnings */
        /** issued. A result of 0 indicates a successful connection with */
        /** no warnings. */
        /*******

//A result of 0 indicates a successful connection with no warnings.
System.out.println("Result= " + result);

    } catch (RuntimeOperationsException roe) {
Exception ex = roe.getTargetException();
ex.printStackTrace(System.out);
    } catch (Exception ex) {
System.out.println("General exception occurred");
ex.printStackTrace(System.out);
    }
}
}
}

```

Note: Example may be wrapped for display purposes.

Example: Test a connection using country and language (properties): It is possible to invoke the same test connection operation on the DataSourceCfgHelper MBean from a Java program that wsadmin uses by passing in the properties you wish to test. The following interfaces are tested:

- getPropertiesForDataSource()
- reload()
- testConnectionToDataSource()

See the “Code license and disclaimer information” on page 184 for legal information about these code examples.

```

//
// This program may be used, executed, copied, modified and distributed without royalty for the
// purpose of developing, using, marketing, or distributing.
//
// Product 5630-A36, (C) COPYRIGHT International Business Machines Corp., 2001, 2002
// All Rights Reserved * Licensed Materials - Property of IBM
//
import java.util.*;
import javax.sql.DataSource;
import javax.transaction.*;
import javax.management.*;

```

```

import com.ibm.websphere.management.*;
import com.ibm.websphere.management.configservice.*;
import com.ibm.ws.exception.WsException;
import com.ibm.websphere.rsadapter.DSPropertyEntry;

/**
 * Resource adapter test program to make sure that the MBean interfaces work.
 * Following interfaces are tested
 *
 * -getPropertiesForDataSource()
 * -reload()
 * -testConnectionToDataSource()
 *
 * We need following to run .....
 *
 * From an iSeries command line:
 *
 * -> QSHELL
 * (Start the QSHELL environment)
 *
 * -> cd (yourDirectory)
 * (change current directory to the directory that holds the java source)
 *
 * -> . /QIBM/ProdData/WebASE51/ASE/base/bin/setupClient
 * (produce the $JAVA_FLAGS_EXT environment variable)
 *
 * -> javac -extdirs /QIBM/ProdData/WebASE51/ASE/base/lib -d . testDS.java
 * (compile the testDS class)
 *
 * -> java $JAVA_FLAGS_EXT testDS
 * (call the program)
 */
public class testDS {

    String port = "8880";
    String host = "localhost";
    final static boolean verbose = true;

/**
 * Main method.
 *
 * @param args - Not used
 */
public static void main(String[] args) {
    testDS cds = new testDS();
    try {
        cds.run(args);
    } catch (com.ibm.ws.exception.WsException ex) {
        System.out.println("Caught this " + ex );
        ex.printStackTrace();
        //ex.getCause().printStackTrace();
    } catch (Exception ex) {
        System.out.println("Caught this " + ex );
        ex.printStackTrace();
    }
}

/**
 * This method tests the DataSourceCfgHelper Mbean.
 *
 * @param args - Not used
 * @exception Exception
 */
public void run(String[] args) throws Exception {

```

```

try {
System.out.println("Connecting to the application server.....");

/*****
/** Initialize the AdminClient */
*****/
Properties adminProps = new Properties();
adminProps.setProperty(AdminClient.CONNECTOR_TYPE, AdminClient.CONNECTOR_TYPE_SOAP);
adminProps.setProperty(AdminClient.CONNECTOR_HOST, host);
adminProps.setProperty(AdminClient.CONNECTOR_PORT, port);
AdminClient adminClient = null;
try {
/* Port 8880 must be listening when this program is called or the SOAP connection will fail */
adminClient = AdminClientFactory.createAdminClient(adminProps);
} catch (com.ibm.websphere.management.exception.ConnectorException ce) {
System.out.println("Cannot make a connection to the application server\n"+ce);
System.exit(1);
}

/*****
/** Locate the Mbean */
*****/
ObjectName handle = null;
try {
ObjectName queryName = new ObjectName("WebSphere:type=DataSourceCfgHelper,*");
Set s = adminClient.queryNames(queryName, null);
Iterator iter = s.iterator();
if (iter.hasNext()) handle = (ObjectName)iter.next();
} catch (MalformedObjectNameException mone) {
System.out.println("Check the program variable queryName" + mone);
} catch (com.ibm.websphere.management.exception.ConnectorException ce) {
System.out.println("Cannot connect to the application server" + ce);
}

//System.out.println("Connected to the application server" + handle);

/*****
/** Call the Mbean to get the data source properties */
*****/
String dsClassName = "com.ibm.db2.jdbc.app.DB2StdXADDataSource";
String providerLibPath = "/QIBM/ProdData/Java400/ext/db2_classes.jar";
String[] signature = {"java.lang.String", "java.lang.String"};
Object[] params = { dsClassName, providerLibPath};
Object result = null;

if (verbose) {
System.out.println("Calling getPropertiesForDataSource() for " + dsClassName + "\n");
}
try {
// get the properties
result = adminClient.invoke(handle, "getPropertiesForDataSource", params, signature);
} catch (MBeanException mbe) {
if (verbose) {
System.out.println("\tMbean Exception " + dsClassName);
}
} catch (InstanceNotFoundException infe) {
System.out.println("Cannot find " + dsClassName);
} catch (Exception ex) {
System.out.println("Exception occurred calling getPropertiesForDataSource() for " + dsClassName + ex);
}

// Pretty print what we found
Iterator propIterator = ((List)result).iterator();
System.out.println(format("Name",21)+ "|" + format("Default Value",34) + "|" +
format("Type",17) + "|Reqd");
String line = "_____";
System.out.println(line);

```

```

while (propIterator.hasNext()) {
    DSPPropertyEntry dspe = (DSPPropertyEntry)propIterator.next();
    System.out.print(format(dspe.getPropertyName(),21)+"|" + format(dspe.getDefaultValue(),34) + "|");
System.out.println(format(dspe.getPropertyType(),17) + "|" + ((dspe.isRequired())? " Y" : " N"));
}
System.out.println(line);

/*****
/** Invoke the reload function from the AdminClient to pickup the      */
/* data source from the naming space.                                */
*****/

if (verbose) {
    System.out.println("Calling reload()");
}
try {
    result = adminClient.invoke(handle, "reload", new Object[] {}, new String[] {});
} catch (MBeanException mbe) {
    if (verbose) {
        System.out.println("\tMbean Exception calling reload" + mbe);
    }
} catch (InstanceNotFoundException infe) {
    System.out.println("Cannot find reload ");
} catch (Exception ex) {
    System.out.println("Exception occurred calling reload()" + ex);
}
if (result==null && verbose) {
    System.out.println("OK reload()");
}

/*****
/** Start to test the connection to the database                      */
*****/

if (verbose) {
    System.out.println("\nTesting connection to the database using " + dsClassName);
}

String user = "db2admin";
String password = "db2admin";
Properties props = new Properties();
props.setProperty("databaseName", "section");

// There are two ways to pass the locale: In WS 5.0, you can only pass in the
// language and the country in a String format. In WS 5.0.1 release, you can also pass
// in a Locale object.
// String[] signature2 = { "java.lang.String", "java.lang.String", "java.lang.String",
// "java.util.Properties", "java.lang.String", "java.util.Locale"};
// Object[] params2 = { dsClassName, user, password,props ,providerLibPath, Locale.US};

Object result2 = null;

String[] signature2 = { "java.lang.String", "java.lang.String", "java.lang.String",
"java.util.Properties",
"java.lang.String", "java.lang.String", "java.lang.String");
Object[] params2 = { dsClassName, user, password,props ,providerLibPath, "EN", "US"};

try {
    result2 = adminClient.invoke(handle, "testConnectionToDataSource", params2, signature2);
} catch (MBeanException mbe) {
    if (verbose) {
        System.out.println("\tMbean Exception " + dsClassName);
    }
}

```

```

} catch (InstanceNotFoundException infe) {
    System.out.println("Cannot find " + dsClassName);
} catch (RuntimeMBeanException rme) {
    Exception ex = rme.getTargetException();
    ex.printStackTrace(System.out);
    throw ex;
} catch (Exception ex) {
    System.out.println("Exception occurred calling testConnectionToDataSource() for " + dsClassName + ex);
    ex.printStackTrace();
}

if (result2 != null) {
    System.out.println("ERROR Result= " + result2);
} else if (verbose) {
    System.out.println("OK testConnectionToDataSource()");
}

} catch (RuntimeOperationsException roe) {
    Exception ex = roe.getTargetException();
    ex.printStackTrace(System.out);
    throw ex;
    } catch (Exception ex) {
        ex.printStackTrace(System.out);
        throw ex;
    }
}

/**
 * Format the string right justified in the space provided,
 * or truncate the string.
 *
 * @param in
 * @param length
 * @return
 */
public String format(Object in, int length) {
    if (in == null) {
        in = "-null-";
    }

    String ins = in.toString();
    int insLength = ins.length();
    if (insLength > length) {
        return ins.substring(0, length);
    } else {
        StringBuffer sb = new StringBuffer(length);
        while (length - insLength > 0) {
            sb.append(" ");
            length--;
        }
        sb.append(ins);
        return sb.toString();
    }
}
}

```

Note: This example may be wrapped for display purposes.

Example: Test a connection to a data source: This resource adapter test program ensures that the MBean interfaces work. The following interfaces are tested:

- `getPropertiesForDataSource()`
- `reload()`
- `testConnectionToDataSource()`

See the "Code license and disclaimer information" on page 184 for legal information about these code examples.

```
//
// This program may be used, executed, copied, modified and distributed without royalty for the
// purpose of developing, using, marketing, or distributing.
//
// Product 5630-A36, (C) COPYRIGHT International Business Machines Corp., 2001, 2002
// All Rights Reserved * Licensed Materials - Property of IBM
//

import java.util.*;
import javax.sql.DataSource;
import javax.transaction.*;
import javax.management.*;

import com.ibm.websphere.management.*;
import com.ibm.websphere.management.configservice.*;
import com.ibm.ws.exception.WsException;
import com.ibm.websphere.rsadapter.DSPropertyEntry;

/**
 * Resource adapter test program to make sure that the MBean interfaces work.
 * Following interfaces are tested
 *
 * -getPropertiesForDataSource()
 * -reload()
 * -testConnectionToDataSource()
 *
 * We need following to run
 *
 * From an iSeries command line:
 *
 * -> QSHELL
 * (Start the QSHELL environment)
 *
 * -> cd (yourDirectory)
 * (change current directory to the directory that holds the java source)
 *
 * -> . /QIBM/ProdData/WebASE51/ASE/base/bin/setupClient
 * (produce the $JAVA_FLAGS_EXT environment variable)
 *
 * -> javac -extdirs /QIBM/ProdData/WebASE51/ASE/base/lib -d . testDS.java
 * (compile the testDS class)
 *
 * -> java $JAVA_FLAGS_EXT testDS
 * (call the program)
 */
public class testDS {

    String port = "8880";
    String host = "localhost";
    final static boolean verbose = true;

    /**
     * Main method.
     *
     * @param args - Not used
     */
    public static void main(String[] args) {
        testDS cds = new testDS();
        try {
            cds.run(args);
        } catch (com.ibm.ws.exception.WsException ex) {
            System.out.println("Caught this " + ex );
        }
    }
}
```

```

    ex.printStackTrace();
    //ex.getCause().printStackTrace();
  } catch (Exception ex) {
    System.out.println("Caught this " + ex );
    ex.printStackTrace();
  }
}

/**
 * This method tests the DataSourceCfgHelper Mbean.
 *
 * @param args - Not used
 * @exception Exception
 */
public void run(String[] args) throws Exception {

  try {
    System.out.println("Connecting to the application server.....");

    /**
     * Initialize the AdminClient
     */
    Properties adminProps = new Properties();
    adminProps.setProperty(AdminClient.CONNECTOR_TYPE, AdminClient.CONNECTOR_TYPE_SOAP);
    adminProps.setProperty(AdminClient.CONNECTOR_HOST, host);
    adminProps.setProperty(AdminClient.CONNECTOR_PORT, port);
    AdminClient adminClient = null;
    try {
      /* Port 8880 must be listening when this program is called or the SOAP connection will fail */
      adminClient = AdminClientFactory.createAdminClient(adminProps);
    } catch (com.ibm.websphere.management.exception.ConnectorException ce) {
      System.out.println("Cannot make a connection to the application server\n"+ce);
      System.exit(1);
    }

    /**
     * Locate the Mbean
     */
    ObjectName handle = null;
    try {
      ObjectName queryName = new ObjectName("WebSphere:type=DataSourceCfgHelper,*");
      Set s = adminClient.queryNames(queryName, null);
      Iterator iter = s.iterator();
      if (iter.hasNext()) handle = (ObjectName)iter.next();
    } catch (MalformedObjectNameException mone) {
      System.out.println("Check the program variable queryName" + mone);
    } catch (com.ibm.websphere.management.exception.ConnectorException ce) {
      System.out.println("Cannot connect to the application server" + ce);
    }

    //System.out.println("Connected to the application server" + handle);

    /**
     * Call the Mbean to get the data source properties
     */
    String dsClassName = "com.ibm.db2.jdbc.app.DB2StdXADataSource";
    String providerLibPath = "/QIBM/ProdData/Java400/ext/db2_classes.jar";
    String[] signature = { "java.lang.String", "java.lang.String" };
    Object[] params = { dsClassName, providerLibPath };
    Object result = null;

    if (verbose) {
      System.out.println("Calling getPropertiesForDataSource() for " + dsClassName + "\n");
    }
    try {
      // get the properties
      result = adminClient.invoke(handle, "getPropertiesForDataSource", params, signature);

```

```

} catch (MBeanException mbe) {
    if (verbose) {
        System.out.println("\tMbean Exception " + dsClassName);
    }
} catch (InstanceNotFoundException infe) {
    System.out.println("Cannot find " + dsClassName);
} catch (Exception ex) {
    System.out.println("Exception occurred calling getPropertiesForDataSource() for " + dsClassName + ex);
}

// Pretty print what we found
Iterator propIterator = ((List)result).iterator();
System.out.println(format("Name",21)+ "|" + format("Default Value",34) + "|" +
    format("Type",17) +"|Reqd");
String line = "_____";
System.out.println(line);
    while (propIterator.hasNext()) {
        DSPropertyEntry dspe = (DSPropertyEntry)propIterator.next();
        System.out.print(format(dspe.getPropertyName(),21)+"|" + format(dspe.getDefaultValue(),34) + "|"");
System.out.println(format(dspe.getPropertyType(),17) +"|" + ((dspe.isRequired())? " Y" : " N"));
    }
System.out.println(line);

/*****
/** Invoke the reload function from the AdminClient to pickup the      */
/* data source from the naming space.                                  */
*****/

if (verbose) {
    System.out.println("Calling reload()");
}
try {
    result = adminClient.invoke(handle, "reload", new Object[] {}, new String[] {});
} catch (MBeanException mbe) {
    if (verbose) {
        System.out.println("\tMbean Exception calling reload" + mbe);
    }
} catch (InstanceNotFoundException infe) {
    System.out.println("Cannot find reload ");
} catch (Exception ex) {
    System.out.println("Exception occurred calling reload()" + ex);
}
if (result==null && verbose) {
    System.out.println("OK reload()");
}

/*****
/** Start to test the connection to the database                        */
*****/

if (verbose) {
    System.out.println("\nTesting connection to the database using " + dsClassName);
}

String user = "db2admin";
String password = "db2admin";
Properties props = new Properties();
props.setProperty("databaseName", "section");

// There are two ways to pass the locale: In WS 5.0, you can only pass in the
// language and the country in a String format. In WS 5.0.1 release, you can also pass
// in a Locale object.
// String[] signature2 = { "java.lang.String", "java.lang.String", "java.lang.String",
// "java.util.Properties", "java.lang.String", "java.util.Locale");
// Object[] params2 = { dsClassName, user, password,props ,providerLibPath, Locale.US};

```

```

Object result2 = null;

String[] signature2 = { "java.lang.String", "java.lang.String",
"java.lang.String", "java.util.Properties", "java.lang.String",
"java.lang.String", "java.lang.String"};
Object[] params2 = { dsClassName, user, password, props ,providerLibPath, "EN", "US"};

try {

    result2 = adminClient.invoke(handle, "testConnectionToDataSource", params2, signature2);
} catch (MBeanException mbe) {
    if (verbose) {
        System.out.println("\tMbean Exception " + dsClassName);
    }
} catch (InstanceNotFoundException infe) {
    System.out.println("Cannot find " + dsClassName);
} catch (RuntimeMBeanException rme) {
    Exception ex = rme.getTargetException();
    ex.printStackTrace(System.out);
    throw ex;
} catch (Exception ex) {
    System.out.println("Exception occurred calling testConnectionToDataSource()
for " + dsClassName + ex);
    ex.printStackTrace();
}

if (result2 != null) {
    System.out.println("ERROR Result= " + result2);
} else if (verbose) {
    System.out.println("OK testConnectionToDataSource()");
}

} catch (RuntimeOperationsException roe) {
    Exception ex = roe.getTargetException();
    ex.printStackTrace(System.out);
    throw ex;
    } catch (Exception ex) {
        ex.printStackTrace(System.out);
        throw ex;
    }
}

/**
 * Format the string right justified in the space provided,
 * or truncate the string.
 *
 * @param in
 * @param length
 * @return
 */
public String format(Object in, int length) {
    if (in ==null) {
        in = "-null-";
    }

    String ins = in.toString();
    int insLength = ins.length();
    if ( insLength > length) {
        return ins.substring(0,length);
    } else {
        StringBuffer sb = new StringBuffer(length);
        while (length - insLength > 0) {
            sb.append(" ");
            length--;
        }
    }
}

```

```

        sb.append(ins);
        return sb.toString();
    }
}
}

```

Note: Example may be wrapped for display purposes.

Example: Creating a JDBC provider and data source using Java Management Extensions API and the scripting tool: Following is a JACL (wsadmin - scripting tool) script used to create a data source and test the connection. This script:

- Creates a data source fvtDS_1
- Creates a 4.0 data source fvtDS_3
- Creates a container-managed persistence (CMP) data source linked to fvtDS_1
- Tests the connection

```

#AWE -- Set up XA DB2 data sources, both 5.0 and 4.0

#UPDATE THESE VALUES:
#The classpath that will be used by your database driver
set driverClassPath "/QIBM/UserData/Java400/ext/db2_classes.jar"

set server "server1"

#Users and passwords..
set defaultUser1 "dbuser1"
set defaultPassword1 "dbpwd1"
set aliasName "alias1"

set databaseName1 "localhost"
set databaseName2 "localhost"
#END OF UPDATES

puts "Add an alias alias1"
set cell [${AdminControl getCell}]
set sec [${AdminConfig getid /Cell:$cell/Security:/}]

#-----
# Create a JAASAuthData object for component-managed authentication
#-----
puts "create JAASAuthData object for alias1"

set alias_attr [list alias $aliasName]
set desc_attr [list description "Alias 1"]
set userid_attr [list userId $defaultUser1]
set password_attr [list password $defaultPassword1]
set attrs [list $alias_attr $desc_attr $userid_attr $password_attr]

set authdata [${AdminConfig create JAASAuthData $sec $attrs}]
${AdminConfig save}

puts "Installing DB2 datasource for XA"

puts "Finding the old JDBCProvider.."
#Remove the old jdbc provider...
set jps [${AdminConfig list JDBCProvider}]
foreach jp $jps {
    set jpname [lindex [lindex [${AdminConfig show $jp {name}} 0] 1]
    if {($jpname == "FVTProvider")} {
        puts "Removing old JDBC Provider"
        ${AdminConfig remove $jp}
        ${AdminConfig save}
    }
}
}

```

```

#Get the server name...
puts "Finding the server $server"
set servlist [$AdminConfig list Server]
set servsize [llength $servlist]
foreach srvr $servlist {
  set sname [lindex [lindex [$AdminConfig show $srvr {name}] 0] 1]
  if {($sname == $server)} {
    puts "Found server $srvr"
    set serv $srvr
  }
}

puts "Finding the Resource Adapter"
set rsadapter [$AdminConfig list J2CResourceAdapter $serv]

#Now create a JDBC Provider for the 5.0 data sources
puts "Creating the provider for com.ibm.db2.jdbc.app.DB2StdXADataSource"
set attrs1 [subst {{classpath $driverClassPath}
{implementationClassName com.ibm.db2.jdbc.app.DB2StdXADataSource}
{name "FVTProvider2"} {description "DB2 UDB for iSeries JDBC Provider"}}]
set provider1 [$AdminConfig create JDBCProvider $serv $attrs1]

#Create the first data source
puts "Creating the datasource fvtDS_1"
set attrs2 [subst {{name fvtDS_1} {description "FVT DataSource 1"}}]
set ds1 [$AdminConfig create DataSource $provider1 $attrs2]

#Set the properties for the data source.
set propSet1 [$AdminConfig create J2EEResourcePropertySet $ds1 {}]

set attrs3 [subst {{name databaseName} {type java.lang.String} {value $databaseName1}}]
$AdminConfig create J2EEResourceProperty $propSet1 $attrs3

set attrs10 [subst {{jndiName jdbc/fvtDS_1} {statementCacheSize 10}
{datasourceHelperClassname com.ibm.websphere.rsadapter.DB2DataStoreHelper}
{relationalResourceAdapter $rsadapter}
{authMechanismPreference "BASIC_PASSWORD"} {authDataAlias $aliasName}}]
$AdminConfig modify $ds1 $attrs10

#Create the connection pool object...
$AdminConfig create ConnectionPool $ds1 {{connectionTimeout 1000}
{maxConnections 30} {minConnections 1} {agedTimeout 1000} {reapTime 2000} {unusedTimeout 3000} }

#Now lets create the 4.0 data sources..
puts "Creating the 4.0 datasource fvtDS_3"
set ds3 [$AdminConfig create WAS40DataSource $provider1 {{name fvtDS_3}
{description "FVT 4.0 DataSource"}}]

#Set the properties on the data source
set propSet3 [$AdminConfig create J2EEResourcePropertySet $ds3 {}]

#These attributes should be the same as fvtDS_1
set attrs4 [subst {{name user} {type java.lang.String} {value $defaultUser1}}]
set attrs5 [subst {{name password} {type java.lang.String} {value $defaultPassword1}}]
$AdminConfig create J2EEResourceProperty $propSet3 $attrs3
$AdminConfig create J2EEResourceProperty $propSet3 $attrs4
$AdminConfig create J2EEResourceProperty $propSet3 $attrs5
set attrs10 [subst {{jndiName jdbc/fvtDS_3} {databaseName $databaseName1}}]
$AdminConfig modify $ds3 $attrs10

$AdminConfig create WAS40ConnectionPool $ds3 {{orphanTimeout 3000}
{connectionTimeout 1000} {minimumPoolSize 1} {maximumPoolSize 10} {idleTimeout 2000}}

#Now we will add a connection factory for the CMPs..

```

```
puts "Creating the CMP Connector Factory for fvtDS_1"
set attrs12 [subst {{name "FVT DS 1_CF"
{authMechanismPreference BASIC_PASSWORD} {cmpDatasource $ds1} {authDataAlias $aliasName}}}]
set cf1 [$AdminConfig create CMPConnectorFactory $rsadapter $attrs12]

#Set the properties for the data source.
$AdminConfig create MappingModule $cf1 {{mappingConfigAlias "DefaultPrincipalMapping"
{authDataAlias "alias1"}}}

$AdminConfig save
```

Note: Example may be wrapped for display purposes.

Configure JCA data access: To configure WebSphere Application Server - Express to access databases using the J2EE Connector Architecture (JCA) specification, you must configure a connection factory and install a resource adaptor for your application. Perform these steps:

1. **“Configuring Java 2 Connector connection factories”**

If your data access application implements the J2EE Connector Architecture (JCA) specification, you must configure a connection factory to provide connections to your database.

2. **“Installing Java 2 Connector resource adapters”**

After you have configured a connection factory, install a resource adapter, which is used to connect to your database.

Configuring Java 2 Connector connection factories: Perform these steps in the WebSphere Application Server administrative console:

1. Click **Resources**.
2. Click **Resource Adapters**.
3. Select a resource adapter under Resource Adapters.
4. Click **J2C Connection Factories** under Additional Properties.
5. Click **New**.
6. Specify General Properties.
7. Select the authentication preference.
8. Select aliases for component-managed authentication, container-managed authentication, or both. If none are available, or you want to define a different one, click **Apply** → **J2C Authentication Data Entries** under Related Items.
9. Click **J2C Auth Data Entries** under Related Items.
10. Click **New**.
11. Specify General Properties.
12. Click **OK**.
13. Click **OK**.
14. Click the J2C connection factory you just created.
15. Under Additional Properties click **Connection Pool**.
16. Change any values desired by clicking the property name.
17. Click **OK**.
18. Click **Custom Properties** under Additional Properties.
19. Click any property name to change its value. Note that Username and Password if present, are overridden by the component-managed authentication alias you specified in a previous step.
20. Click **Save**.

Installing Java 2 Connector resource adapters: Perform these steps in the Websphere Application Server administrative console:

1. Expand **Resources**.

2. Click **Resource Adapters**.
3. Click **Install RAR**.

The **Install RAR** button opens a dialog that enables you to install a J2EE Connector Architecture (JCA) connector and create a resource adapter for it. You can also use the **New** button, but the **New** button creates only a new resource adapter (the JCA connector must already be installed on the system).

Note: When installing a RAR file using this dialog, the scope you define on the Resource Adapters page has no effect on where the RAR file is installed. You can install RAR files only at the node level, and the node in which the file is installed on is determined by the scope on the Install RAR page. The scope you set on the Resource Adapters page determines the scope of the new resource adapters, which you can install at the server, node, or cell level.

4. Browse to find the appropriate RAR file.
5. Click **Next**.
6. Enter the resource adapter name and any other properties needed under **General Properties**.
7. Click **OK**.

Deploy data access applications

Before you deploy a data access application into the WebSphere Application Server - Express environment, you must first ensure that the appropriate database objects are available. This action includes creating and configuring any databases or tables required, setting necessary configuration parameters to handle expected load, and configuring any necessary JDBC providers and data source objects for JSPs and servlets to use.

If your database configuration does not already exist, create a database and the tables that are required by your application. Use your database server interfaces to create the tables.

In addition, consider “Security of lookups.”

See the WebSphere Development Studio Client Help for more information about deploying your data access application.

Security of lookups: Any client that runs in WebSphere Application Server - Express, such as a servlet or JSP, can look up a Java 2 Connector (J2C) resource (such as a data source) in the Java Naming and Directory Interface (JNDI) namespace and obtain connections without providing authentication data. These clients use a component managed authentication alias defined on the resource, which is the default value used when the user and password are not supplied on the `getConnection` call. However, you can pass the user and password on the `getConnection` call, as well as disable security of lookups using WebSphere Application Server - Express.

See the following topics for detailed information on how to manage lookup security:

“Pass user and password on the `getConnection` call”

This topic explains the necessary prerequisites before the user and password can be passed on the `getConnection` call.

“Disable lookup security” on page 100

Although it is not recommended, it is possible to turn off the secure mode for a particular data source or connection factory.

Pass user and password on the `getConnection` call: If the user and password are supplied on the `getConnection` call by a client that runs outside the WebSphere Application Server - Express process, the connection factory or data source that is looked up to obtain connections must not have a component managed authentication alias defined on it.

Make sure the connection factory or data source does not have a component managed authentication alias defined on it. See “Configuring Java 2 Connector connection factories” on page 98 for more information.

Disable lookup security: By default, all lookups are secure as described in “Security of lookups” on page 99.

Although it is not recommended, it is possible to turn off the secure mode for a particular data source or connection factory.

1. Edit the /QIBM/ProdData/WebASE51/ASE/properties/j2c.properties file.
2. Remove the comments around the following lines of code:

```
<!-- The security-properties are in a comment block. Uncomment to use -->
<!--
<security-properties connectionFactoryJNDIName="myDataSource">
    <secureMode>false</secureMode>
</security-properties>
-->
```

where *myDataSource* is the JNDI name of the data source or connection factory you want to run unsecure.

3. Save and close the file.

Java Naming and Directory Interface (JNDI)

Distributed computing environments often employ naming and directory services to obtain shared components and resources. Naming and directory services associate names with locations, services, information, and resources. Naming services provide name-to-object mappings. Directory services provide information on objects and the search tools required to locate those objects. There are many naming and directory service implementations, and the interfaces to them vary.

Java Naming and Directory Interface (JNDI) provides a common interface that is used to access the various naming and directory services. See <http://java.sun.com/products/jndi/serviceproviders.html>



for a list of naming and directory service providers which support access through the JNDI interface.

JNDI is an integral part of other Java programming models and technologies, such as data access and JavaMail.

See these topics for more detailed information about JNDI.

“JNDI basic concepts”

This topic provides conceptual information about naming, name space logical views, initial context support, and the differences between JNDI and CORBA.

“JNDI implementation” on page 105

This topic describes the WebSphere Application Server - Express implementation of JNDI, including package and interface support. This page also links to JNDI caching and JNDI helpers and utilities information.

“Use JNDI” on page 115

This topic provides an overview of the steps necessary for using JNDI within your Java components.

JNDI basic concepts

Naming is used by WebSphere Application Server - Express applications and components to obtain references to objects they use. These objects are bound into a mostly hierarchical structure, referred to as

a name space. You can access and manipulate the name space through a name server. Users of a name server are referred to as naming clients. Naming clients typically use JNDI to perform naming operations.

“Naming”

This topic describes naming, which is used by WebSphere Application Server - Express applications and components to obtain references to objects they use.

“Name space logical view”

This topic describes a logical view of the name space for the entire cell.

“Initial context support” on page 103

This topic describes the default initial context, which is used as the first step for all naming operations.

“Differences between JNDI and CORBA” on page 104

This topic describes WebSphere Application Server - Express’s support for Common Object Request Broker Architecture (CORBA) object URLs as JNDI provider URLs and lookup names.

Naming: Naming is used by WebSphere Application Server - Express applications and components to obtain references to objects they use.

These objects are bound into a mostly hierarchical structure, referred to as a name space. In this structure, all non-leaf objects are called contexts. Leaf objects can be contexts and other types of objects. Naming operations, such as lookups and binds, are performed on contexts. All naming operations begin with obtaining an initial context. You can view the initial context as a starting point in the name space.

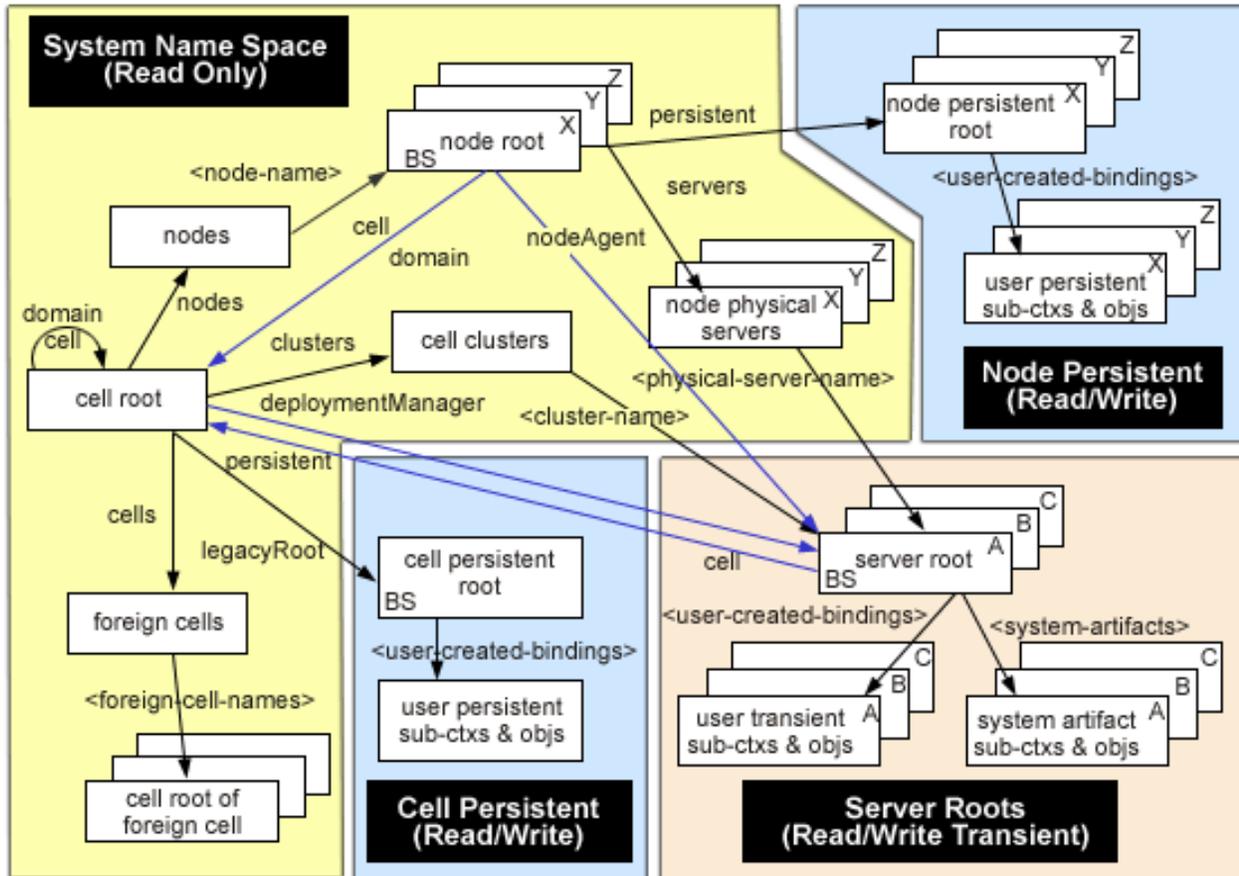
The name space structure consists of a set of name bindings, each consisting of a name relative to a specific context and the object bound with that name. For example, the name `myApp/myDataSource` consists of one non-leaf binding with the name `myApp`, which is a context. The name also includes one leaf binding with the name `myDataSource`, relative to `myApp`. The object bound with the name `myDataSource` in this example happens to be a data source home reference. The whole name `myApp/myDataSource` is relative to the initial context, which you can view as a starting place when performing naming operations.

You can access and manipulate the name space through a name server. Users of a name server are referred to as naming clients. Naming clients typically use the Java Naming and Directory Interface (JNDI) to perform naming operations. Naming clients can also use the Common Object Request Broker Architecture (CORBA) CosNaming interface.

Typically, objects bound to the name space are resources and objects associated with installed applications. These objects are bound by the system, and client applications perform lookup operations to obtain references to them. Occasionally, server and client applications bind objects to the name space. An application can bind objects to transient or persistent partitions, depending on requirements.

In J2EE environments, some JNDI operations are performed with `java:` URL names. Names bound under these names are bound to a completely different name space which is local to the calling process. However, some lookups on the `java:` name space may trigger indirect lookups to the name server.

Name space logical view: The name space for the entire cell is federated among all servers in the cell. Every server process, including the node agents, and application servers, contains a name server. All name servers provide the same logical view of the cell name space. The various server roots and persistent partitions of the name space are interconnected by a system name space. You can use the system name space structure to traverse to any context in a the cell’s name space. A logical view of the name space is shown in this diagram.



Note: Express has only one cell, node, and server.

The bindings in the preceding diagram appear with solid arrows, labeled in bold, and dashed arrows, labeled in gray. Solid arrows represent primary bindings. A primary binding is formed when the associated subcontext is created. Dashed arrows show linked bindings. A linked binding is formed when an existing context is bound under an additional name. Linked bindings are added for convenience or interoperability with previous WebSphere Application Server versions.

A cell name space is composed of contexts which reside in servers throughout the cell. All name servers in the cell provide the same logical view of the cell name space. A name server constructs this view at startup by reading configuration information. Each name server has its own local in-memory copy of the name space and does not require another running server to function. There are, however, a few exceptions. Server roots for other servers are not replicated among all the servers. The respective server for a server root must be running to access that server root context.

Name space partitions

There are four major partitions in a cell name space:

- System name space partition (page 102)
- Server roots partition (page 103)
- Cell persistent partition (page 103)
- Node persistent partition (page 103)

System name space partition

The system name space contains a structure of contexts based on the cell topology. The system structure supports traversal to all parts of a cell name space and to the cell root of other cells, which are configured as foreign cells. The root of this structure is the cell root. In addition to the cell root, the system structure contains a node root for each node in the cell. You can access other contexts of interest specific to a node from the node root, such as the node persistent root and server roots for servers configured in that node.

All contexts in the system name space are read-only. You cannot add, update, or remove any bindings.

Server roots partition

Each server in a cell has a server root context. A server root is specific to a particular server. You can view the server roots for all servers in a cell as being in a transient read/write partition of the cell name space. System artifacts are bound under the server root context of the associated server. A server application can also add bindings under its server root. These bindings are transient. Therefore, the server application creates all required bindings at application startup, so they exist anytime the application is running.

Server-scoped bindings are relative to a server's server root.

Cell persistent partition

The root context of the cell persistent partition is the cell persistent root. A binding created under the cell persistent root is saved as part of the cell configuration and continues to exist until it is explicitly removed. Applications that need to create additional persistent bindings of objects generally associated with the cell can bind these objects under the cell persistent root.

It is important to note that the cell persistent area is not designed for transient, rapidly changing bindings. The bindings are more static in nature, such as part of an application setup or configuration, and are not created at run time.

Node persistent partition

The node persistent partition is similar to the cell partition except that each node has its own node persistent root. A binding created under a node persistent root is saved as part of that node configuration and continues to exist until it is explicitly removed.

Applications that need to create additional persistent bindings of objects associated with a specific node can bind those objects under that particular node's node persistent root. As with the cell persistent area, it is important to note that the node persistent area is not designed for transient, rapidly changing bindings. These bindings are more static in nature, such as part of an application setup or configuration, and are not created at run time.

Initial context support: All naming operations begin with obtaining an initial context. You can view the initial context as a starting point in the name space. Use the initial context to perform naming operations, such as looking up and binding objects in the name space.

The default initial context depends on the type of client. Here are the different categories of clients and the corresponding default initial contexts:

WebSphere Application Server JNDI interface implementation prior to V5.0

WebSphere Application Server clients running in releases prior to WebSphere Application Server V5.0 by default use WebSphere Application Server's V4.0 CosNaming JNDI plug-in implementation. The default initial context for clients of this type is the cell persistent root, also known as the legacy root.

Other JNDI implementation

Some applications can perform name space lookups with a non-WebSphere Application Server - Express CosNaming JNDI plug-in implementation. Assuming the key NamingContext is used to obtain the initial context, the default initial context for clients of this type is the cell root.

Differences between JNDI and CORBA: WebSphere Application Server - Express contains support for Common Object Request Broker Architecture (CORBA) object URLs (corbaloc and corbname) as JNDI provider URLs and lookup names. Issues can exist when mapping JNDI name strings to and from CORBA names.

CORBA names

Each component in a CORBA name consists of an id and kind field.

JNDI names

JNDI name components do not consist of an id and kind field. Each component in a JNDI name is atomic. Typical JNDI clients do not need to make a distinction between the id and kind fields of a name component, or know how JNDI name strings map to CORBA names.

When a name is parsed according to JNDI syntax, each name component is mapped to the id field of the corresponding CORBA name component. The kind field always has an empty value. This basic syntax is the least obtrusive to the JNDI client in that it has the fewest special characters. However, you cannot represent with this syntax a CORBA name with a non-empty kind field.

INS name syntax

Some clients, however must interoperate with CORBA applications that use CORBA names with non-empty kind fields. These JNDI clients **must** make a distinction between id and kind so that JNDI names are correctly mapped to CORBA names, particularly when the CORBA names contain components with non-null kind fields. Such JNDI clients can use the INS name syntax.

The INS syntax allows a JNDI client to make the proper mapping to and from a CORBA name. INS syntax is very similar to the JNDI syntax with the additional special character, dot (.). Dots are used to delimit the id and kind fields in a name component. A dot is interpreted literally when it is escaped. Only one unescaped dot is allowed in a name component. A name component with a non-empty id field and empty kind field is represented with only the id field value and must not end with an unescaped dot. An empty name component (empty id and empty kind field) is represented with a single unescaped dot. An empty string is not a valid name component representation.

Use of this syntax is not recommended unless it is necessary, because this syntax is more restrictive from the JNDI client's perspective in that the JNDI client must be aware that name components with multiple unescaped dots are syntactically invalid. INS name syntax is part of the OMG CosNaming Interoperable Naming Specification.

For an example of the INS name syntax for use with JNDI clients, see "Example: Set the syntax used to parse name strings."

Example: Set the syntax used to parse name strings: JNDI clients which must interoperate with CORBA applications might need to use INS name syntax to represent names in string format. The name syntax property may be passed to the InitialContext constructor through its parameter, in the System properties, or in a jndi.properties file. The initial context and any contexts looked up from that initial context parses name strings based on the specified syntax.

This example shows how to set the name syntax to make the initial context parse name strings according to INS syntax.

```
...
import java.util.Hashtable;
import javax.naming.Context;
```

```

import javax.naming.InitialContext;
import com.ibm.websphere.naming.PROPS; // WebSphere naming constants
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.ibm.websphere.naming.WsnInitialContextFactory");
env.put(Context.PROVIDER_URL, ...);
env.put(PROPS.NAME_SYNTAX, PROPS.NAME_SYNTAX_INS);
Context initialContext = new InitialContext(env);
// The following name maps to a CORBA name component as follows:
//   id = "a.name", kind = "in.INS.format"
// The unescaped dot is used as the delimiter. Escaped dots are interpreted literally.
java.lang.Object o = initialContext.lookup("a\\.name\\.in\\.INS\\.format");
...

```

JNDI implementation

WebSphere Application Server - Express includes a name server to provide shared access to Java components, and an implementation of the `javax.naming`



JNDI package, which allows users to access the WebSphere Application Server - Express name server through the JNDI naming interface.

In order to provide access to LDAP servers, Option 5 of the IBM Developer Kit for Java provides Sun Microsystems Java 2 Software Development Kit (J2SDK), Standard Edition, version 1.3, and contains the following packages:

- `javax.naming.ldap`



- `com.sun.jndi.ldap.LdapCtxFactory`



WebSphere Application Server - Express does not provide implementations for the `javax.naming.directory`



package and the `javax.naming.ldap`



package. WebSphere Application Server - Express does not support interfaces defined in the `javax.naming.event`



package.

WebSphere Application Server - Express's JNDI implementation is based on version 1.2 of the JNDI interface and was tested with version 1.2.1 of the Sun Microsystems JNDI Service Provider Interface (SPI). The default behavior of this implementation should be adequate for most users. However, users with specific requirements can control certain aspects of the JNDI behavior.

See these topics for information on modifying JNDI behavior:

"JNDI caching" on page 106

This topic describes the caching features and properties, including the effects of the different properties on caching behavior.

“JNDI helpers and utilities” on page 109

This topic describes the `com.ibm.websphere.naming.JndiHelper` class and the Name Space Dump utility.

JNDI caching: In WebSphere Application Server - Express, Java Naming and Directory Interface (JNDI) context objects use caching to improve the performance of JNDI lookup operations. Objects that have been bound and looked up are cached to speed up subsequent lookups for those objects. Objects are cached as they are bound or initially looked up. JNDI clients should typically be able to use the default cache behavior.

These sections describe cache behavior and how JNDI clients can override default cache behavior when necessary.

“JNDI cache behavior”

This topic discusses how caches interact with initial contexts, and how this association affects lookup operations performed for cached objects.

“JNDI cache properties” on page 107

This topic discusses the ways in which you can control cache behavior by setting the properties for a cache.

“JNDI coding examples” on page 108

This topic illustrates how cache properties are set by giving Java code examples that control cache behavior.

JNDI cache behavior: A cache is associated with an initial context when a `javax.naming.InitialContext` object is instantiated with the `java.naming.factory.initial` property set to `com.ibm.websphere.naming.WsnInitialContextFactory`.

`WsnInitialContextFactory` searches the environment properties for a cache name, defaulting to the provider URL. If no provider URL is defined, the cache name `iiop:///` is used. All instances of `InitialContext` that use a cache of a given name share the same cache instance.

After an association between an `InitialContext` instance and cache is established, the association does not change. A `javax.naming.Context` object returned from a lookup operation will inherit the cache association of the `Context` object on which the lookup was performed. Changing cache property values with the `Context.addToEnvironment()` or `Context.removeFromEnvironment()` method does not affect cache behavior. Properties affecting a given cache instance, however, may be changed with each `InitialContext` instantiation.

A cache is restricted to a process and does not persist beyond the life of the process. A cached object is returned from lookup operations until either the maximum cache life (page 108) for the cache is reached, or the maximum entry life (page 108) for the object's cache entry is reached.

After the object's cache reaches its maximum life or the object reaches its maximum entry life, a lookup on the object causes the cache entry for the object to be refreshed. If a bind or rebind operation is performed on an object, the change is not reflected in any caches other than the one associated with the context from which the bind or rebind operation was issued. This “stale data” scenario is most likely to happen when multiple processes are involved because different processes do not share the same cache, and `Context` objects in all threads in a process typically share the same cache instance for a given name service provider.

Cached objects are typically relatively static entities, and objects becoming stale should not be a problem. However, you can set timeout values on cache entries or on a cache to periodically refresh cache contents.

JNDI cache properties: JNDI clients can use several properties to control cache behavior. These properties can be set in the Java virtual machine system environment or in the environment Hashtable that is passed to the InitialContext constructor.

You can set properties in these ways:

- **Through the command line:** To set properties through the command line, enter the actual string value as indicated in this example:

```
java -Dcom.ibm.websphere.naming.jndicache.maxentrylife=1440 MyProgram
```

- **In a file:** To set properties in a file, create a text file listing the properties. For example:

```
...
com.ibm.websphere.naming.jndicache.cacheobject=none
...
```

- **Within a program:** To set properties in a Java program, use the following **PROPS.JNDI_CACHE*** Java constants, defined in the `com.ibm.websphere.naming.PROPS` class:

```
import com.ibm.websphere.naming.PROPS;

...

public static final String JNDI_CACHE_OBJECT =
    "com.ibm.websphere.naming.jndicache.cacheobject";
public static final String JNDI_CACHE_OBJECT_NONE = "none";
public static final String JNDI_CACHE_OBJECT_POPULATED = "populated";
public static final String JNDI_CACHE_OBJECT_CLEARED = "cleared";
public static final String JNDI_CACHE_OBJECT_DEFAULT = JNDI_CACHE_OBJECT_POPULATED;

public static final String JNDI_CACHE_NAME =
    "com.ibm.websphere.naming.jndicache.cachename";
public static final String JNDI_CACHE_NAME_DEFAULT = "providerURL";

public static final String JNDI_CACHE_MAX_LIFE =
    "com.ibm.websphere.naming.jndicache.maxcachelife";
public static final int JNDI_CACHE_MAX_LIFE_DEFAULT = 0;

public static final String JNDI_CACHE_MAX_ENTRY_LIFE =
    "com.ibm.websphere.naming.jndicache.maxentrylife";
public static final int JNDI_CACHE_MAX_ENTRY_LIFE_DEFAULT = 0;
```

To set a property in your program, enter the following:

```
env.put(PROPS.JNDI_CACHE_OBJECT, PROPS.JNDI_CACHE_OBJECT_NONE).
// Sets a property to a value
```

Cache properties are evaluated when an InitialContext instance is created. The resulting cache association, including “none” (page 107), cannot be changed. The “max life” cache properties affect the individual cache’s behavior. If the cache already exists, cache behavior is updated according to the new “max life” property settings. If no “max life” properties exist in the environment, the cache assumes default “max life” settings, regardless of the previous settings. The various cache properties are described below. All property values must be String values.

- **com.ibm.websphere.naming.jndicache.cacheobject**

Caching is turned on or off with this property. Additionally, an existing cache can be cleared. The valid values for this property and the resulting cache behavior are listed below.

- **“populated”** (default): Use a cache with the specified name (page 108). If the cache already exists, leave existing cache entries in cache; otherwise, create a new cache.
- **“cleared”**: Use a cache with the specified name (page 108). If the cache already exists, clear all cache entries from cache; otherwise, create a new cache.
- **“none”**: Do not cache. If this option is specified, the cache name (page 108) is irrelevant. Therefore, this option will not disable a cache that is already associated with other InitialContext instances. The InitialContext being instantiated will not be associated with any cache.

Note: You must include the quotation marks when specifying any of the above property values.

- **com.ibm.websphere.naming.jndicache.cachename**

It is possible to create multiple InitialContext instances, each operating on the namespace of a different name service provider. By default, objects from each service provider are cached separately, since they each involve independent namespaces and name collisions could occur if they used the same cache. The provider URL specified when the initial context is created serves as the default cache name. With this property, a JNDI client can specify a cache name other than the provider URL. The valid option for cache names are listed below.

- **“providerURL”** (default): Use the value for java.naming.provider.url property as the cache name. The default provider URL is “iiop:///”. URLs are normalized by stripping off everything after the port. For example, “iiop://server1:2809” and “iiop://server1:2809/com/ibm/initCtx” are normalized to the same cache name.
- **any_string**: Use the specified string as the cache name. Any arbitrary string with a value other than “providerURL” can be used as a cache name.

Note: You must include the quotation marks when specifying any of the above property values.

- **com.ibm.websphere.naming.jndicache.maxcachelife**

By default, cached objects remain in the cache for the life of the process or until cleared with the com.ibm.websphere.naming.jndicache.cacheobject property set to “cleared”. This property enables a JNDI client to set the maximum life of a cache as follows:

- **“0”** (default): Make the cache lifetime unlimited.
- **positive_integer**: Set the maximum lifetime of the cache, in minutes, to the specified value. When the maximum cache lifetime is reached, the cache is cleared before another cache operation is performed. The cache is repopulated as bind, rebind, and lookup operations are performed.

Note: You must include the quotation marks when specifying any of the above property values.

- **com.ibm.websphere.naming.jndicache.maxentrylife**

By default, cached objects remain in the cache for the life of the process or until cleared with the com.ibm.websphere.naming.jndicache.cacheobject property set to “cleared”. This property enables a JNDI client to set the maximum lifetime of individual cache entries as follows:

- **“0”** (default): Lifetime of cache entries is unlimited.
- **positive_integer**: Set the maximum lifetime of individual cache entries, in minutes, to the specified value. When the maximum lifetime for an entry is reached, the next attempt to read the entry from the cache will cause the entry to be refreshed.

Note: You must include the quotation marks when specifying any of the above property values.

JNDI coding examples: This Java code snippet demonstrates several techniques for controlling JNDI cache behavior, discussed in “JNDI cache properties” on page 107. Note that the caching discussed in this section pertains only to the WebSphere Application Server - Express implementation of the initial context factory. Assume that the property, java.naming.factory.initial, is set to “com.ibm.ejs.ns.WsnInitialContextFactory” as a java.lang.System property.

```
import java.util.Hashtable;
import javax.naming.InitialContext;
import javax.naming.Context;
import com.ibm.websphere.naming.PROPS;

// ... class declaration, method declaration code here ...

Hashtable env;
Context ctx;

// To clear a cache:
// ...class declaration, method declaration code here...

try {
env = new Hashtable();
env.put(PROPS.JNDI_CACHE_OBJECT, PROPS.JNDI_CACHE_OBJECT_CLEARED);
```

```

ctx = new InitialContext(env);
}
catch(javax.naming.NamingException e) {
    // error-handling code
}

// To set a cache's maximum cache lifetime to 60 minutes:

try {
env = new Hashtable();
env.put(PROPS.JNDI_CACHE_MAX_LIFE, "60");
ctx = new InitialContext(env);
}
catch(javax.naming.NamingException e) {
    // error-handling code
}

// To turn caching off:

try {
env = new Hashtable();
env.put(PROPS.JNDI_CACHE_OBJECT, PROPS.JNDI_CACHE_OBJECT_NONE);
ctx = new InitialContext(env);
}
catch(javax.naming.NamingException e) {
    // error-handling code
}

// To use caching and no caching:

try {
env = new Hashtable();
env.put(PROPS.JNDI_CACHE_OBJECT, PROPS.JNDI_CACHE_OBJECT_POPULATED);
ctx = new InitialContext(env);
env.put(PROPS.JNDI_CACHE_OBJECT, PROPS.JNDI_CACHE_OBJECT_NONE);
Context noCacheCtx = new InitialContext(env);
}
catch(javax.naming.NamingException e) {
    // error-handling code
}

try {
Object o;

// Use caching to look up the datasource, since it should rarely change.
o = ctx.lookup("java:comp/env/jdbc/MyDataSource");
// Narrow, etc. ...

// Do not use cache if data is volatile.
o = noCacheCtx.lookup("com/mycom/VolatileObject");
// ...
}
catch(javax.naming.NamingException e) {
    // error-handling code
}

```

JNDI helpers and utilities: WebSphere Application Server - Express provides extensions to the Sun Microsystems JNDI specification. These utilities are designed to simplify working with JNDI contexts and subcontexts as well as obtain information about specific JNDI contexts set up within your WebSphere Application Server - Express environment.

“JNDI helper class” on page 110

This topic illustrates the additional functionality provided by the `com.ibm.websphere.naming.JndiHelper` class. The `com.ibm.websphere.naming.JndiHelper` class contains static methods to simplify common JNDI programming and administration tasks.

“Namespace dump utility”

This topic describes the namespace dump utility. The namespace stored by a given name server can be dumped with the name space dump utility that is shipped with WebSphere Application Server - Express.

“Example: Output from the namespace dump utility” on page 112

This topic illustrates the output received when you invoke the namespace dump utility.

“Invoke the NameServer MBean to dump java:, local:, and server name spaces” on page 113

This topic describes how to use the NameServer MBean to dump name spaces.

JNDI helper class: The `com.ibm.websphere.naming.JndiHelper`



class contains static methods to simplify common JNDI programming and administration tasks.

These helper class functions include:

- **Recursively creating subcontexts**

```
import com.ibm.websphere.naming.JndiHelper;

...

try {
    Context startingContext = new InitialContext();
    startingContext = startingContext.lookup("com/mycompany");

    // Create each intermediate subcontext, if necessary, as well as leaf context.
    // AlreadyBoundException is not thrown.
    JndiHelper.recursiveCreateSubcontext(startingContext, "apps/
accounting");
}
catch (NamingException e){
    // Handle other errors.
}
```

- **Rebinding objects and creating intermediate contexts that do not already exist**

```
import com.ibm.websphere.naming.JndiHelper;

...

try {
    Context startingContext = new InitialContext();

    // Creates each intermediate subcontext, if necessary, and rebinds object.
    JndiHelper.recursiveRebind(startingContext, "com/mycompany/apps/accounting",
someObject);
}
catch (NamingException e){
    // Handle other errors.
}
```

Namespace dump utility: The namespace stored by a given name server can be dumped with the name space dump utility that is shipped with WebSphere Application Server - Express. You can invoke this utility from the command line or from a Java program. The naming service for the WebSphere Application Server - Express host must be active when this utility is invoked.

You can invoke the namespace dump utility in these ways:

- Invoke the name space dump utility by adding the following code to your Java program:

```
import com.ibm.websphere.naming.DumpNameSpace;
java.io.PrintStream filePrintStream = ...
Context ctx = new InitialContext();
Context ctx = (Context) ctx.lookup("<starting_context>");
// Starting context for dump
DumpNameSpace dumpUtil = new DumpNameSpace(filePrintStream, DumpNameSpace.LONG);
dumpUtil.generateReport(ctx);
```

where *<starting_context>* is the starting context for the dump.

For more information, see “JNDI implementation” on page 105 for the API documentation.

- Invoke the namespace dump utility on your iSeries server:
 - Enter the Start Qshell (STRQSH) command on an CL command line.
 - At the Qshell command prompt, enter the following command:


```
cd /QIBM/ProdData/WebASE51/ASE/bin
```
 - Run the dumpNameSpace script. The syntax is as follows:


```
dumpNameSpace -host myhost.mycompany.com -port <port_number>
```

 where *<port_number>* is the name service port which, if not specified, defaults to 2809. In this case, assume that the port number is 2809.

The keywords and associated values for the dumpNameSpace utility are:

| Keyword | Values | Description |
|----------|---|---|
| -host | Example: <i>myhost.myserver.mycompany.com</i> | Represents the bootstrap host or the WebSphere Application Server - Express host whose name space you want to dump. The value defaults to localhost. |
| -port | Example: <i>nnn</i> | Represents the bootstrap port which, if not specified, defaults to 2809. |
| -factory | Example: <i>com.ibm.websphere.naming.WsnInitialContextFactory</i> | Indicates the initial context factory to be used to get the JNDI initial context. The value defaults to com.ibm.websphere.naming.WsnInitialContextFactory. You typically do not need to change the default value. |
| -root | cell server node legacy host tree default | <ul style="list-style-type: none"> • The cell option dumps the tree starting at the cell root context and is the default value. • The server option dumps the tree starting at the server root context. • The node option dumps the tree starting at the node root context. • In earlier versions of WebSphere Application Server, the legacy option dumps the tree starting at the legacy root context and is the default value. • In earlier versions of WebSphere Application Server, the host option dumps the tree starting at the bootstrap host root context. • In earlier versions of WebSphere Application Server, the tree option dumps the tree starting at the tree root context. • The default option dumps the tree starting at the initial context that JNDI returns by default for that server type. |
| -url | Example: <i>some_url</i> | Represents the value for the java.naming.provider.url property, which is used to get the initial JNDI context. |

| Keyword | Values | Description |
|--------------|--|---|
| -startAt | Example: <i>some/subcontext/in/the/tree</i> | Indicates the path from the bootstrap host's root context to the top level context where the dump should begin. The utility recursively dumps subcontexts below this point. It defaults to an empty string, which represents context of the bootstrap host root. |
| -format | jndi ins | Displays name components as atomic strings. (Note: This is the default value.) Displays name components parsed per INS rules (id.kind). |
| -report | short long | Dumps the binding name and bound object type. This output is also provided by JNDI Context.list(). (Note: This is the default value.) Dumps the binding name, bound object type, local object type, and string representation of the local object (that is, the IORs, string values, and other values that are printed). |
| -traceString | Example: <i>"some.package.name.to.trace.*=all=enabled"</i> | Represents the trace string with the same format as that generated by the servers. The output is sent to the file DumpNameSpaceTrace.out in the current working directory. |
| -help | | Provides a description of Name Space Dump utility and command line usage. |

See "Example: Output from the namespace dump utility" for sample output.

Example: Output from the namespace dump utility: In this example, the output from either the command line or code invocation would be:

```
Getting the initial context
Getting the starting context
```

```
=====
Name Space Dump
Provider URL: corbaloc:iiop:appsvr1.rchland.ibm.com:3500
Context factory: com.ibm.websphere.naming.WsnInitialContextFactory
Requested root context: cell
Starting context: (top)=APPSVR1_myInst
Formatting rules: jndi
Time of dump: Mon Dec 01 15:18:15 UTC 2003
=====
```

```
=====
Beginning of Name Space Dump
=====
```

```

1 (top)
2 (top)/legacyRoot                javax.naming.Context
2   Linked to context: APPSVR1_myInst/persistent
3 (top)/cell                       javax.naming.Context
3   Linked to context: APPSVR1_myInst
4 (top)/domain                     javax.naming.Context
4   Linked to context: APPSVR1_myInst
5 (top)/cellname                   java.lang.String
6 (top)/cells                       javax.naming.Context
7 (top)/persistent                 javax.naming.Context
8 (top)/persistent/cell            javax.naming.Context
```

```

8   Linked to context: APPSVR1_myInst
9 (top)/clusters                               javax.naming.Context
10 (top)/nodes                                 javax.naming.Context
11 (top)/nodes/APPSVR1_myInst                 javax.naming.Context
12 (top)/nodes/APPSVR1_myInst/cell           javax.naming.Context
12   Linked to context: APPSVR1_myInst
13 (top)/nodes/APPSVR1_myInst/nodename       java.lang.String
14 (top)/nodes/APPSVR1_myInst/persistent     javax.naming.Context
15 (top)/nodes/APPSVR1_myInst/servers        javax.naming.Context
16 (top)/nodes/APPSVR1_myInst/servers/myInst javax.naming.Context
17 (top)/nodes/APPSVR1_myInst/servers/myInst/eis javax.naming.Context
18 (top)/nodes/APPSVR1_myInst/servers/myInst/eis/DefaultDatasource_CMP
18                                           Default_CF
19 (top)/nodes/APPSVR1_myInst/servers/myInst/thisNode javax.naming.Context
19   Linked to context: APPSVR1_myInst/nodes/APPSVR1_myInst
20 (top)/nodes/APPSVR1_myInst/servers/myInst/jta javax.naming.Context
21 (top)/nodes/APPSVR1_myInst/servers/myInst/jta/usertransaction
21                                           java.lang.Object
22 (top)/nodes/APPSVR1_myInst/servers/myInst/TransactionFactory
22                                           com.ibm.ws.Transaction.JTS.TransactionFactoryImpl
23 (top)/nodes/APPSVR1_myInst/servers/myInst/DefaultDatasource
23                                           Default Datasource
24 (top)/nodes/APPSVR1_myInst/servers/myInst/servername
24                                           java.lang.String
25 (top)/nodes/APPSVR1_myInst/servers/myInst/cell javax.naming.Context
25   Linked to context: APPSVR1_myInst
26 (top)/nodes/APPSVR1_myInst/domain         javax.naming.Context
26   Linked to context: APPSVR1_myInst
27 (top)/nodes/APPSVR1_myInst/node           javax.naming.Context
27   Linked to context: APPSVR1_myInst/nodes/APPSVR1_myInst

```

```

=====
End of Name Space Dump
=====

```

Invoke the NameServer MBean to dump java:, local:, and server name spaces: It may be helpful to dump the java: name space for a J2EE application. You cannot use the dumpNameSpace command line utility for this purpose because the application's java: name space is accessible only by that J2EE application. From the WebSphere Application Server scripting tool, you can invoke a NameServer MBean to dump the java: name space for any J2EE application running in that same server process.

There is another name space local to server process which you cannot dump with the dumpNameSpace command line utility. This name space has the URL scheme of local: and is used by the container to bind objects locally instead of through the name server. The local: name space contains references to enterprise beans with local interfaces. There is only one local: name space in a server process. You can dump the local: name space by invoking the NameServer MBean associated with that server process.

To invoke the NameServer MBean:

1. Start the wsadmin tool.
2. Select the NameServer MBean instance to invoke.

Execute the following script commands to select the NameServer instance you want to invoke. For example:

```
set mbean [$AdminControl completeObjectName WebSphere:*,type=NameServer,cell=
  cellName,node=nodeName,process=serverName]
```

where *cellName*, *nodeName*, and *serverName* are the names of the cell, node, and server for the MBean you want to invoke. The specified server must be running before you can invoke a method on the MBean.

You can see a list of all NameServer MBeans current running by issuing the following query:

```
$AdminControl queryNames {*:*,type=NameServer}
```

3. Invoke the NameServer MBean.

java: name space

Dump a java: name space by invoking the `dumpJavaNameSpace` method on the `NameServer MBean`. Since each server application has its own java: name space, the application must be specified on the method invocation. An application is identified by the application name, module name, and component name. The method syntax follows:

```
$AdminControl invoke $mbean dumpJavaNameSpace
{{appName}{modName}{compName}{opts}}
```

where *appName* is the application name, *modName* is the module name, and *compName* is the component name of the java: name space you want to dump. The value for *opts* is the list of name space dump options described earlier in this section. The list can be empty.

local: name space

Dump a java: name space by invoking the `dumpLocalNameSpace` method on the `NameServer MBean`. Since there is only one local: name space in a server process, you have to specify the name space dump options only.

```
$AdminControl invoke $mbean dumpLocalNameSpace {{opts}}
```

where *opts* is the list of name space dump options described earlier in this section. The list can be empty.

Server name space

Dump a server name space by invoking the `dumpServerNameSpace` method on an application server's `NameServer MBean`. This provides an alternative way to dump the name space on an application server, much like the `dumpNameSpace` command line utility.

```
$AdminControl invoke $mbean dumpServerNameSpace {{opts}}
```

where *opts* is the list of name space dump options described earlier in this section. The list can be empty.

Examples: Invoking the name space dump utility for java: and local: name spaces

It is often helpful to view the dump of a java: or local: name space to understand why a naming operation is failing. The `NameServer MBean` running in the application's server process can be invoked from the WebSphere Application Server scripting tool to generate a dump of these name spaces. Examples of `NameServer MBean` calls to generate dumps of java: and local: name spaces follow.

Dumping a java: name space

Assume you want to dump the java: name space of an application component running in server `server1` on node `node1` of the cell `MyCell`. The application name is `AcctApp` in module `AcctApp.war`, and the component name is `Acct Servlet`. The following script commands generate a long format dump of the application's java: name space of that application:

```
set mbean [$AdminControl completeObjectName
WebSphere:*,type=NameServer,cell=MyCell,node=node1,process=server1]
$AdminControl invoke $mbean dumpJavaNameSpace {{AcctApp}{AcctApp.war}{Acct Servlet}{-report
long}}
```

Dumping a local: name space

Assume you want to dump the local: name space for the server `server1` on node `node1` of cell `MyCell`. The following script commands will generate a short format dump of that server's local name space:

```
set mbean [$AdminControl completeObjectName
WebSphere:*type=NameServer,cell=MyCell,node=node1,process=server1]
$AdminControl invoke $mbean dumpLocalNameSpace {{-report short}}
```

Use JNDI

You can use the JNDI API to support the Java components you deploy on WebSphere Application Server-Express. Specifically, you use the JNDI API to locate and refer to resources such as data sources and JavaMail sessions within a distributed computing environment.

The process of using JNDI to access enterprise-level Java components involves these steps:

1. "Obtain the initial JNDI context for the component"
This topic describes how to obtain an initial JNDI context that contains the resource (a Java object).
2. "Use JNDI to look up Java components" on page 117
This topic explains how to look up resources such as data sources and JavaMail sessions.

After you have completed these steps, you can use the enterprise resource (such as a data source) in your code exactly as you would if Java object had been available locally.

Obtain the initial JNDI context for the component: To access an enterprise resource such as a data source or JavaMail session in a distributed computing environment, you can use the JNDI API to locate that object so that you can use it in your code. To do this, you must first obtain the initial context that contains the resource (a Java object).

You can obtain an initial JNDI context in one of two ways:

- Get an initial context using JNDI properties found in the current environment (page 115)
- Get an initial context by explicitly setting JNDI properties (page 115)

Get an initial context using JNDI properties found in the current environment

The current environment includes the Java system properties and properties defined in properties files found in the classpath. This code snippet illustrates how to obtain the initial context from these properties:

```
import javax.naming.Context;
import javax.naming.InitialContext;

// ... class declaration, method declaration code here ...

Context initialContext = new InitialContext();
```

Get an initial context by explicitly setting JNDI properties

In general, JNDI clients should assume that the correct environment is already configured. If this is the case, there is no need for you to explicitly set property values and pass them to the constructor of the InitialContext object. However, a JNDI client might need to access a namespace other than the one identified in its environment. In this event, you must explicitly set one or more properties used by the InitialContext constructor. Any property values you pass to the InitialContext constructor take precedence over settings of the equivalent properties found elsewhere in the client's environment.

You can use two different provider URL forms with WebSphere Application Server's initial context factory:

- A CORBA object URL (new for J2EE 1.3)
- An IIOP URL

CORBA object URLs are more flexible than IIOP URLs and are the recommended URL format to use. CORBA object URLs are part of the OMG CosNaming Interoperable Naming Specification. A corbaname URL, for example, can include initial context and lookup name information and can be used as a lookup name without the need to explicitly obtain another initial context. The IIOP URLs are the legacy JNDI format, but are still supported by the WebSphere Application Server initial context factory.

- **Using a CORBA object URL**

```

...
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.ibm.websphere.naming.WsnInitialContextFactory");
env.put(Context.PROVIDER_URL, "corbaloc:iiop:myhost.mycompany.com:2809");
Context initialContext = new InitialContext(env);
...

```

- **Using a CORBA object URL with multiple name server addresses**

CORBA object URLs can contain more than one bootstrap address. You can use this feature when attempting to obtain an initial context from a server cluster. You can specify the bootstrap addresses for all servers in the cluster in the URL. The operation succeeds if at least one of the servers is running, eliminating a single point of failure. There is no guarantee of any particular order in which the address list will be processed. For example, the second bootstrap address may be used to obtain the initial context even though the server at the first bootstrap address in the list is available.

Multiple-address provider URLs should only contain the bootstrap addresses of members of the same cluster. Otherwise, incorrect behavior may occur.

```

...
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.ibm.websphere.naming.WsnInitialContextFactory");
// All of the servers in the provider URL below are members of
// the same cluster.
env.put(Context.PROVIDER_URL,
        "corbaloc::myhost1:9810,:myhost1:9811,:myhost2:9810");
Context initialContext = new InitialContext(env);
...

```

- **Using a CORBA object URL from a non-WebSphere Application Server JNDI implementation**

Initial context factories for CosNaming JNDI plug-in implementations other than the WebSphere Application Server initial context factory most likely obtain an initial context using the object key, `NameService`. When you use such a context factory to obtain an initial context from a WebSphere Application Server name server, the initial context is the cell root context. Since system artifacts such as EJB homes associated with a server are bound under the server's server root context, names used in JNDI operations must be qualified. If you want to use relative names, ensure your initial context is the server root context under which the target object is bound. In order to make the server root context the initial context, specify a corbaloc provider URL with an object key of `NameServiceServerRoot`.

This example shows a CORBA object type URL from a non-WebSphere Application Server JNDI implementation. This example assumes full CORBA object URL support by the non-WebSphere Application Server JNDI implementation. The object key of `NameServiceServerRoot` is specified so that the initial context will be the specified server's server root context.

```

...
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.somecompany.naming.TheirInitialContextFactory");
env.put(Context.PROVIDER_URL,
        "corbaname:iiop:myhost.mycompany.com:9810/NameServiceServerRoot");
Context initialContext = new InitialContext(env);
...

```

If qualified names are used, you can use the default key of `NameService`.

- **Using an IIOP URL**

The IIOP type of URL is a legacy format which is not as flexible as CORBA object URLs. However, URLs of this type are still supported. The following example shows an IIOP type URL as the provider URL.

```
...
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.ibm.websphere.naming.WsnInitialContextFactory");
env.put(Context.PROVIDER_URL, "iiop://myhost.mycompany.com:2809");
Context initialContext = new InitialContext(env);
...
```

Use JNDI to look up Java components: To access some Java objects (such as data sources, or JavaMail sessions) in a distributed computing environment, you can use the JNDI API. These Java objects are sometimes referred to as **enterprise resources**.

Before you can access an enterprise resource in a JNDI environment, you must first “Obtain the initial JNDI context for the component” on page 115. The examples that follow assume that you have already written code to obtain the initial context, which is represented by the object name `initialContext`:

-
- Example: Look up a data source (page 117)
- Example: Look up a JavaMail session (page 117)

Example: Look up a data source

The `DataSource` object is defined in the JDBC 2.0 Optional Package. The actual name to lookup is defined by the JNDI Name property for the `DataSource` object. You can use the Websphere administrative console to view this property. For example, if you have a `DataSource` object named `AccountDataSource`, the JNDI name for that `DataSource` would be `java:comp/env/jdbc/AccountDataSource`.

```
try {
    DataSource ds = (DataSource)initialContext.lookup("java:comp/env/jdbc/AccountDataSource");
}
catch (NamingException e) {
    // Error getting the data source object
    // ... error-handling code ...
}
```

Example: Look up a JavaMail session

The actual name to look up is defined in the deployment descriptor of the Web application that contains your servlets or JSP files that use the JavaMail API.

```
try {
    Session session = (Session)initialContext.lookup("java:comp/env/mail/MailSession")
}
catch (NamingException e) {
    // Error getting the mail session
    // ... error-handling code ...
}
```

JavaMail

The JavaMail APIs model an electronic mail (e-mail) system. The APIs provide a platform-independent and protocol-independent framework to build Java-based e-mail client applications. WebSphere Application Server - Express supports the Sun Microsystems Inc. JavaMail version 1.2



and the JavaBeans Activation Framework (JAF) version 1.0.1



specifications.

WebSphere Application Server - Express supports JavaMail in all Web application components, including servlets and JavaServer Pages (JSP) files.

These topics provide more information on using the JavaMail APIs to write applications for the iSeries server e-mail providers.

“Overview of JavaMail APIs”

This topic provides an overview of the requirements for using JavaMail, including the necessary APIs, service providers, and protocols.

“Configure JavaMail” on page 119

This topic provides instructions for configuring a JavaMail session using the WebSphere administrative console and links to information about setting up e-mail services on your iSeries server.

“Write JavaMail applications” on page 121

This topic discusses how to access e-mail using the JavaMail APIs.

“Debug JavaMail” on page 123

This topic explains how to send debugging information to the Java virtual machine log file for your application server.

For more information about JavaMail, see these resources:

Programming model and decisions

- **JavaMail documentation**



This documentation includes information about the JavaMail programming model.

- **JavaMail 1.2 API documentation**



(<http://www.javasoft.com/products/javamail/1.2/docs/javadocs/overview-summary.html>)

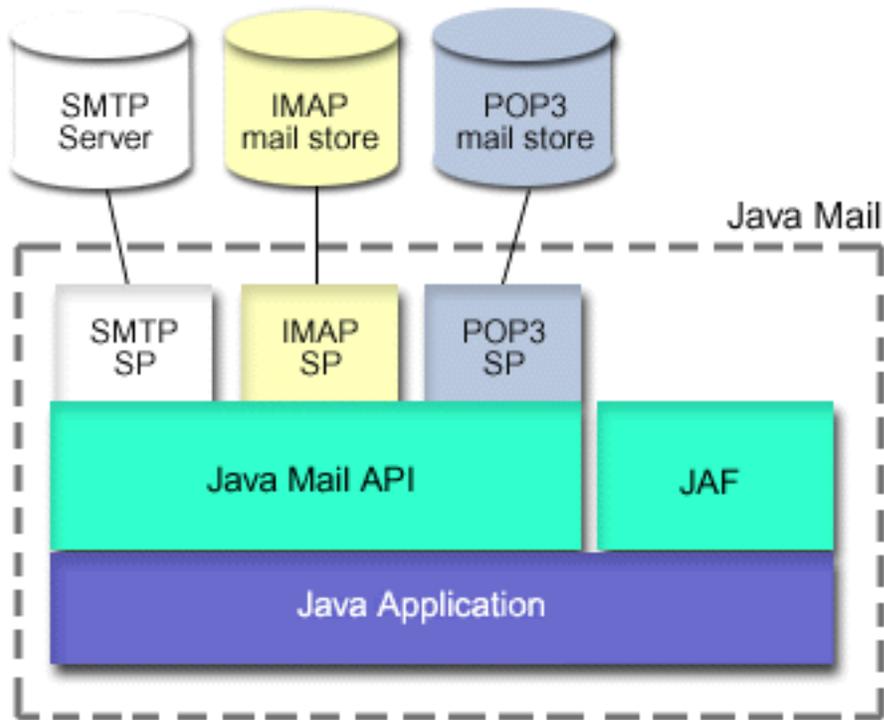
This documentation provides programming specifications for JavaMail.

Overview of JavaMail APIs

The JavaMail APIs only provide general mail facilities for reading and sending mail. These APIs require service providers to implement the protocols.

Service providers implement specific protocols. For example, Simple Mail Transfer Protocol (SMTP) is a transport protocol for sending e-mail. Post Office Protocol 3 (POP3) is the standard protocol for receiving e-mail. Internet Message Access Protocol (IMAP) is an alternative protocol to POP3.

In addition to service providers, JavaMail requires the JavaBeans Activation Framework (JAF) to handle mail content that is not plain text. For example, this includes MIME (Multipurpose Internet Mail Extensions), URL (Uniform Resource Locator) pages, and file attachments.



The JavaMail APIs, the JAF, the service providers and the protocols are shipped as part of WebSphere Application Server - Express using the following Sun licensed packages:

- **mail.jar** : This JAR file contains JavaMail APIs, and the SMTP, IMAP, and POP3 service providers.
- **activation.jar**: This JAR file contains the JavaBeans Activation Framework.

Note: These JAR files are located in the Java extensions directory for WebSphere Application Server - Express (/QIBM/ProdData/WebASE51/ASE/java/ext).

Configure JavaMail

Before you can write and deploy JavaMail applications, you must configure e-mail services on your iSeries server. Enabling JavaMail in the WebSphere Application Server - Express for iSeries environment involves the following steps:

1. "Set up and configure e-mail provider services"

On the iSeries platform, e-mail services are supported by Lotus Domino R5 Mail Server and TCP/IP Connectivity Utilities. This topic helps you set up and configure e-mail services.

2. Configure a JavaMail session for your applications to use

The WebSphere administrative console is used to configure a JavaMail session for servlets or JavaService Pages (JSP). See "Configure a JavaMail session using the administrative console" on page 120 for more information.

Set up and configure e-mail provider services: The JavaMail APIs are platform-independent, meaning that you can write JavaMail applications to access e-mail providers that support the IMAP, SMTP, and POP3 protocols, regardless of the platform on which those services are running. JavaMail applications running on your iSeries server can access e-mail services running on different platforms.

On the iSeries platform, e-mail services are supported by two products:

- Lotus Domino R5 Mail Server. Domino supports the IMAP, POP3, and SMTP protocols.

- TCP/IP Connectivity Utilities (5722TC1), an installable option of i5/OS. It supports the POP3 and SMTP protocols.

Platform-specific considerations

You must perform some extra steps to enable and configure e-mail services on your iSeries server. For instructions on setting up Domino Mail Server, consult the product documentation.

For instructions on setting up e-mail services through the TCP/IP Connectivity Utilities, see these topics:

- Configure e-mail (V5R2)
- Configure e-mail (V5R3)
- Configure e-mail (V5R4)

Additionally, the e-mail services provided by the TCP/IP Connectivity Utilities only support the SMTP and POP3 protocols, not IMAP. See these related topics for more information on the e-mail protocols supported on iSeries:

- SMTP on iSeries (V5R2)
- SMTP on iSeries (V5R3)
- SMTP on iSeries (V5R4)
- POP on iSeries (V5R2)
- POP on iSeries (V5R3)
- POP on iSeries (V5R4)

For more information about related TCP/IP Connectivity Utilities, see these topics:

- Send and receive e-mail on iSeries (V5R2)
- Send and receive e-mail on iSeries (V5R3)
- Send and receive e-mail on iSeries (V5R4)

Configure a JavaMail session using the administrative console: The WebSphere administrative console is used to configure a JavaMail session for a servlet or JavaServer Pages (JSP).

To configure a JavaMail session using the administrative console, perform these steps:

1. Open the WebSphere Application Server administrative console. For more information, see *Start the WebSphere administrative console* in the *Administration* topic.
2. In the topology tree, expand **Resources**, and click **Mail Providers**.
3. Select the **Server** radio button, and click **Apply**.
4. Click **Built-in Mail Provider**.
5. Click **Mail Sessions**, and click **New** to create a new Mail Session.
6. Fill in these fields:
 - **Name**
This is a required field.
 - **JNDI Name**
This is a required field. It specifies the Java Naming and Directory Interface (JNDI) name for the resource, including any naming subcontexts.
 - **Mail Transport Host**
This is a required field. Enter a value such as yourcompany.com.
 - **Mail From**
This is a required field. Enter a value such as userid@yourcompany.com.
7. Click **OK**.
8. Click **Save** on the toolbar to save the configuration.

9. Click **Save** again to update the master repository with your changes.
10. Restart your application server instance. For more information, see *Start and test your application server* in the *Administration* topic.

Write JavaMail applications

After you have configured e-mail services on your iSeries server and created a JavaMail Session object with the WebSphere Application Server administrative console, you can write applications that interface with these services through the Session object. According to the J2EE specification, each instance of the `javax.mail.Session` class is treated as a JNDI resource factory.

J2EE applications can use JavaMail APIs by looking up references to logically named mail connection factories through the `java:comp/env/mail` subcontext declared in the application deployment descriptor and mapped to installation specific mail session resources. As in the case of other J2EE resources, this can be done in order to eliminate the need for the application to hard code references to external resources.

To use the JavaMail APIs in an enterprise application component, such as a servlet, enterprise bean, or application client, perform the following steps:

1. Locate a resource through Java Naming and Directory Interface (JNDI).

The J2EE specification considers a mail session instance as a resource (a factory where mail transport and store connections can be obtained). You should never hard-code mail sessions. Instead, you must follow the J2EE programming model of configuring resources through the system facilities and then locating them through JNDI lookups.

In “Example: JavaMail code” on page 122, the line `javax.mail.Session mail_session = (javax.mail.Session) ctx.lookup("java:comp/env/mail/MailSession3");` is an example of using a resource name located through JNDI rather than hard-coding a mail session. You can consider the lookup name `mail/MailSession3` a soft link to the real resource.

See “Java Naming and Directory Interface (JNDI)” on page 100 for more information.

2. Use the Assembly Toolkit to declare mail resource references in your application.

- a. Start the Assembly Toolkit.
- b. In the J2EE perspective, expand **Web Modules**, and right click the Web module you want to modify.
- c. Select **Open With** → **Deployment Descriptor Editor**.
- d. In the **Web Deployment Descriptor**, click the **References** tab.
- e. On the **References** page, click the **Resource** tab at the top of the page.
- f. Under the **Resource references** heading, click **Add** to declare mail resource references. Fill in the following information:
 - **Name**
This is a required field. The **Name** field specifies the JNDI name of the resource relative to the `java:comp/env` context. Make sure that the name of the reference matches the name used in the code. For example, in “Example: JavaMail code” on page 122, it uses `java:comp/env/mail/MailSession3` in the lookup; therefore, the name of this reference must be `mail/Session3`.
 - **Type**
This is a required field. The **Type** field specifies the type of the resource. Enter `javax.mail.Session` in the field.
 - **Description**
This is an optional field. The **Description** field specifies a description for the resource.
 - **Authentication**
The type of authentication to use for the resource. Select **Container** from the drop-down list.
- g. Click **File** → **Save**.

Note: Use the steps above to edit additional enterprise application components as needed.

The tool generates a deployment descriptor that has XML tags similar to those in the following example:

```
<resource-ref>
  <description>description</description>
  <res-ref-name>mail/MailSession</res-ref-name>
  <res-type>javax.mail.Session</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

3. Administer mail resources

You must configure the mail resource that is referenced by your application component. The mail session you configure must have both the transport and mail access portions defined. Transport settings are required because the code is sending a message; mail access settings are required because they also save a copy to the local mail store. When you configure the mail session, you are asked to specify a JNDI name. This name is important and is required when you install your application (to link up the resource references in your application with the real resources that you have configured). See Example: Look up a JavaMail session (page 117) for more information.

4. Install your application in the *Administration* topic.

You can install your application using either the administrative console or the scripting tool. An important step in the install process is that the system goes through all resource references and expects you to supply a JNDI name for each of them. This is not an arbitrary JNDI name; it is the JNDI name given to a particular, configured resource that is the target of the reference.

5. (Optional) Manage existing mail providers and sessions.

- a. Start the administrative console in the *Administration* topic.
- b. In the topology tree, expand **Resources** and click **Mail Providers** —> **Mail session**.
- c. Click the mail provider or mail session that you want to modify.
- d. To remove a mail provider or mail session, click **Remove**.
- e. Click **Apply** or **OK**.
- f. Save the configuration.

6. (Optional) Enable debugger for a mail session.

- a. Start the administrative console in the *Administration* topic.
- b. In the topology tree, expand **Resources** and click **Mail Providers** —> **Mail session**.
- c. Click **Debug**. Debug is enabled for the session.
- d. Click **Apply** or **OK**.

Example: JavaMail code: The following code illustrates how an application component sends a message and saves it to the mail account's Sent folder:

```
...
// obtain JNDI initial context
javax.naming.InitialContext ctx = new javax.naming.InitialContext();
// use JNDI lookup to obtain Session
mail_session = (javax.mail.Session) ctx.lookup
("java:comp/env/mail/MailSession");

// create new mail message object using Session
MimeMessage msg = new MimeMessage(mail_session);

// set message properties
msg.setRecipients(Message.RecipientType.TO, InternetAddress.parse
("somebody@some_domain.net"));
msg.setFrom(new InternetAddress("me@my_company.com"));
msg.setSubject("Important message from me");
String msg_text = new String("Hello!");
msg.setText(msg_text);

// get Session object's store
```

```

Store store = mail_session.getStore();
// connect to store
store.connect();
// obtain reference to "Sent" folder
Folder f = store.getFolder("Sent");
// create "Sent" folder if it does not exist
if (!f.exists()) f.create(Folder.HOLDS_MESSAGES);
// add message to "Sent" folder
f.appendMessages(new Message[] {msg});

```

- The J2EE specification considers a mail session instance as a resource (a factory where mail transport and store connections can be obtained). You should never hard-code mail sessions. Instead, you must follow the J2EE programming model of configuring resources through the system facilities and then locating them through JNDI lookups.

In the sample code above, the line `javax.mail.Session mail_session = (javax.mail.Session) ctx.lookup("java:comp/env/mail/MailSession3");` is an example of using a resource name located through JNDI rather than hard-coding a mail session. You can consider the lookup name `mail/MailSession3` a soft link to the real resource.

- You must define a resource reference for the mail resource in the deployment descriptor of the component, because a mail session is referenced in the JNDI lookup.

When you create this reference, Make sure that the name of the reference matches the name used in the code. For example, the code above uses `java:comp/env/mail/MailSession3` in the lookup; therefore, the name of this reference must be `mail/Session3` and the type of the resource must be `javax.mail.Session`. After being defined, the deployment descriptor contains the following entry for the mail resource reference:

```

<resource-reference>
<description>description</description>
<res-ref-name>mail/MailSession3</res-ref-name>
<res-type>javax.mail.Session</res-type>
<res-auth>Container</res-auth>

```

- You must configure the mail resource that is referenced by your application component. The mail session you configure must have both the transport and mail access portions defined. Transport settings are required because the code is sending a message; mail access settings are required because they also save a copy to the local mail store. When you configure the mail session, you are asked to specify a JNDI name. This name is important and is required when you install your application (to link up the resource references in your application with the real resources that you have configured).

Note: The preceding example uses the IMAP protocol for receiving e-mail messages. On the iSeries platform, it is supported by the Domino e-mail server, but not the TCP/IP Connectivity Utilities e-mail services, which only provides access to received messages through the POP3 protocol.

Debug JavaMail

At times, you may need to debug a JavaMail application. WebSphere Application Server - Express gives you the option to turn on the JavaMail debugging feature. Using this option, the JavaMail API prints interactions with the mail servers and the properties of the mail session to the Java virtual machine system out log file for your application server.

The mail debug feature is enabled on a per session basis. Perform these steps to enable the JavaMail debugging feature:

1. Open the WebSphere Application Server administrative console. For more information, see *Start the WebSphere administrative console* in the *Administration* topic.
2. In the topology tree, expand **Resources**, and click **Mail Providers**.
3. Click the mail session you want to work with.
4. Click **Mail Session**, and click the mail session you want to work with.
5. Click **Debug**. Debug is enabled only for this session.
6. Click **Apply** or **OK**.

7. Click **Save** in the toolbar to save changes to the configuration.

Once you have enabled debugging for JavaMail, the JavaMail APIs log the interactions between JavaMail applications and the e-mail server to the SystemOut Java virtual machine log file for the application server. See the *JVM log files* topic for more information on configuring and viewing JVM log files.

A sample of the JavaMail debugging output is as follows:

```
DEBUG: not loading system providers in <java.home>/lib
DEBUG: not loading optional custom providers file: /META-INF/javamail.providers
DEBUG: successfully loaded default providers

DEBUG: Tables of loaded providers
DEBUG: Providers listed by Class Name:
{com.sun.mail.smtp.SMTPTransport=javax.mail.Provider
 [TRANSPORT,smtp,com.sun.mail.smtp.SMTPTransport,Sun
 Microsystems, Inc], com.sun.mail.imap.IMAPStore=javax.mail.Provider
 [STORE,imap,com.sun.mail.imap.IMAPStore,Sun
 Microsystems, Inc], com.sun.mail.pop3.POP3Store=javax.mail.Provider
 [STORE,pop3,com.sun.mail.pop3.POP3Store,Sun
 Microsystems, Inc]}
DEBUG: Providers Listed By Protocol:
{imap=javax.mail.Provider[STORE,imap,com.sun.mail.imap.IMAPStore,Sun Microsystems,
 Inc], pop3=javax.mail.Provider[STORE,pop3,com.sun.mail.pop3.POP3Store,Sun
 Microsystems, Inc], smtp=javax.mail.Provider
 [TRANSPORT,smtp,com.sun.mail.smtp.SMTPTransport,Sun
 Microsystems, Inc]}
DEBUG: not loading optional address map file: /META-INF/javamail.address.map
*** In SessionFactory.getObjectInstance,
    The default SessionAuthenticator is based on:
        store_user = john_smith
        store_pw = abcdef
*** In SessionFactory.getObjectInstance, parameters in the new session:
    mail.store.protocol="imap"
    mail.transport.protocol="smtp"
    mail.imap.user="john_smith"
    mail.smtp.host="smtp.coldmail.com"
    mail.debug="true"
    ws.store.password="abcdef"
    mail.from="john_smith@coldmail.com"
    mail.smtp.class="com.sun.mail.smtp.SMTPTransport"
    mail.imap.class="com.sun.mail.imap.IMAPStore"
    mail.imap.host="coldmail.com"
DEBUG: mail.smtp.class property exists and points to com.sun.mail.smtp.SMTPTransport
DEBUG SMTP: useEhlo true, useAuth false
DEBUG: SMTPTransport trying to connect to host "smtp.coldmail.com", port 25

javax.mail.SendFailedException: Sending failed;
    nested exception is:
    javax.mail.MessagingException: Unknown SMTP host: smtp.coldmail.com;
    nested exception is
    java.net.UnknownHostException: smtp.coldmail.com
    at javax.mail.Transport.send0(Transport.java:219)
    at javax.mail.Transport.send(Transport.java:81)
    at ws.mailfvt.SendSaveTestCore.runAll(SendSaveTestCore.java:48)
    at testers.AnyTester.main(AnyTester.java:130)
```

This sample output illustrates a connection failure to a Simple Mail Transfer Protocol (SMTP) server because a fictitious name, `smtp.coldmail.com`, is specified as the server name.

The following are tips on how to read the debugging output:

- The lines headed by `DEBUG` are printed by the JavaMail run-time, while the two lines headed by `***` are printed by the WebSphere Application Server - Express environment run-time.

- The first two lines say that some configuration files are skipped. At run-time the JavaMail API attempts to load a number of configuration files from different locations. All those files are not required. If a required file cannot be accessed, the JavaMail API throws an exception. In this sample, there is no exception and the third line announces that default providers are loaded.
- The next few lines, headed by either Providers listed by Class Name or Providers Listed by Protocols, show the protocol providers loaded. The three providers listed here are the default protocol providers that come under the WebSphere Application Server - Express built-in mail provider, for protocols SMTP, IMAP, and POP3, respectively. If you have installed special protocol providers (or, in JavaMail terminology, service providers) and these providers are used in the current mail session, you should see them listed here with the default providers. If you do not see the special protocol providers, there is a problem.
- The two lines headed by ****** and the few lines below them are printed by the WebSphere Application Server - Express environment to show the properties used to configure the current mail session. Although these properties are listed by their internal name rather than that used in the administrative console interface, it is not difficult to establish the similarities between them. For example, the property mail.store.protocol corresponds to the Protocol Name property in the Store Access section of the mail session configuration page. Make sure to review the listed properties and values to make sure they correspond.
- The few lines above the exception stack show the JavaMail activities when sending a message. First, the JavaMail API recognizes the transport protocol is set to SMTP and the provider com.sun.mail.smtp.SMTPTransport exists. Next, the parameters used by SMTP, useEhlo and useAuth, are shown. Finally, the log shows the SMTP provider trying to connect to the mail server smtp.coldmail.com. Listed next is the exception stack. By now it should occur to us that the specified mail server either does not exist or is not functioning.

Sessions

WebSphere Application Server - Express for iSeries provides support for HTTP sessions as described by the Java Servlet Specification v2.3. HTTP is by design an stateless protocol. Session tracking attempts to associate HTTP requests originating from a particular client as belonging to a single HTTP session.

The following steps for session tracking are summarized:

1. Before you implement session tracking, become familiar with these topics about the sessions programming model:
 - **“Choose a session tracking method”**
This topic describes the different session tracking methods.
 - **“Sessions security” on page 128**
This topic describes security options for sessions, including HTTP authentication.
 - **“Best practices for session programming” on page 129**
This topic describes some of the practices to optimize your session programming.
2. Create or modify your servlets to use session support to maintain sessions on behalf of your Web module. For more information, see “Session programming model and environment” on page 130.
3. “Configure session management” on page 131.
4. “Assemble applications to share session data” on page 133.
5. “Tune session management” on page 133.

Choose a session tracking method

Suppose a servlet that implements HTTP sessions receives HTTP requests from three different clients (browsers). For each client request, the servlet must be able to determine the HTTP session to which the client request pertains. Each client request belongs to just one of the three client sessions being tracked by the servlet. Currently, the product offers three ways to track sessions:

- With cookies (page 126)
- With URL rewriting (page 126)

- With SSL information (page 127)

Cookies

Using cookies is the simplest and most common way to track HTTP sessions, because it requires no special programming to track sessions. When session management is enabled and a client makes a request, the `HttpSession` object is created and a unique session ID is generated for the client and sent to the browser as a cookie. On subsequent requests to the same hostname, the browser continues to send the same session ID back as a cookie and the Session Manager uses the cookie to find the `HttpSession` associated with the client.

URL rewriting

There are situations in which cookies do not work. Some browsers do not support cookies. Other browsers allow the user to disable cookie support. In such cases, the Session Manager must resort to a second method, URL rewriting, to manage the user session.

With URL rewriting, links returned to the browser or redirect have the session ID appended to them. For example, the following link in a Web page:

```
<a href="/store/catalog">
```

is rewritten as:

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

When the user clicks the link, the rewritten form of the URL is sent to the server as part of the client's request. The Web container recognizes `;jsessionid=DA32242SSGE2` as the session ID and saves it for obtaining the proper `HttpSession` object for this user.

Note: Do not make assumptions about the length or exact content of the ID that follows the equals sign (=). In fact, the IDs are often longer than what this example shows.

An application that uses URL rewriting to track sessions must adhere to certain programming guidelines. The application developer needs to:

- Program session servlets to encode URLs
- Supply a servlet or JSP file as an entry point to the application
- Avoid using plain HTML files in the application

Program session servlets to encode URLs

Depending on whether the servlet is returning URLs to the browser or redirecting them, include either the `encodeURL()` or `encodeRedirectURL()` method in the servlet code. Here are examples demonstrating what to replace in your current servlet code.

Rewrite URLs to return to the browser

Suppose you currently have this statement:

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

Change the servlet to call the `encodeURL` method before sending the URL to the output stream:

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

Rewrite URLs to redirect

Suppose you currently have the following statement:

```
response.sendRedirect ("http://myhost/store/catalog");
```

Change the servlet to call the `encodeRedirectURL` method before sending the URL to the output stream:

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

The `encodeURL()` and `encodeRedirectURL()` methods are part of the `HttpServletResponse` object. These calls check to see if URL rewriting is configured before encoding the URL. If it is not configured, they return the original URL.

If both cookies and URL rewriting are enabled and `response.encodeURL()` or `encodeRedirectURL()` is called, the URL is encoded, even if the browser making the HTTP request processed the session cookie.

You can also configure session support to enable protocol switch rewriting. When this option is enabled, the product encodes the URL with the session ID for switching between HTTP and HTTPS protocols.

Supply a servlet or JSP file as an entry point

The entry point to an application (such as the initial screen presented) may not require the use of sessions. However, if the application in general requires session support (meaning some part of it, such as a servlet, requires session support) then after a session is created, all URLs must be encoded in order to perpetuate the session ID for the servlet (or other application component) requiring the session support.

The following example shows how Java code can be embedded within a JSP file:

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

Avoid using plain HTML files in the application

Note: To use URL rewriting to maintain session state, do not link to parts of your applications from plain HTML files (files with `.html` or `.htm` extensions).

The restriction is necessary because URL encoding cannot be used in plain HTML files. To maintain state using URL rewriting, every page that the user requests during the session must have code that can be understood by the Java interpreter.

If you have plain HTML files in your application (or Web module) or in portions of the site that the user might access during the session, convert the files to JSP files.

This impacts the application writer because maintaining sessions with URL rewriting requires that each servlet in the application must use URL encoding for every `HREF` attribute on `<A>` tags, as described previously.

Sessions are lost if one or more servlets in an application do not call the `encodeURL(String url)` or `encodeRedirectURL(String url)` methods.

Session tracking with SSL information

No special programming is required to track sessions with Secure Sockets Layer (SSL) information.

To use SSL information, select **Enable SSL ID tracking** in the Session Management page of the administrative console. Because the SSL session ID is negotiated between the Web browser and HTTP server, this ID cannot survive an HTTP server failure.

SSL tracking is supported by the IBM HTTP Server only. You can control the lifetime of an SSL session ID by configuring options in the Web server. For example, in the IBM HTTP Server, set the configuration

variable `SSLV3TIMEOUT` to provide an adequate lifetime for the SSL session ID. An interval that is too short can cause a premature termination of a session. Also, some Web browsers might have their own timers that affect the lifetime of the SSL session ID. These Web browsers may not leave the SSL session ID active long enough to serve as a useful mechanism for session tracking. The internal HTTP Server of WebSphere Application Server - Express also supports SSL tracking.

Sessions security

HTTP sessions and security can be integrated in WebSphere Application Server - Express for iSeries. When security integration is enabled in Session Manager and a session is accessed in a protected resource, every resource from then on must be secured. You cannot mix secured and unsecured resources, otherwise you may incur an `UnauthorizedSessionRequest` exception when the unsecured resource attempts to access an authenticated session (see chart below). Security integration in Session Manager is supported only through the Lightweight Third-Party Authentication (LTPA) authentication mechanism. If you are using Simple WebSphere Authentication Mechanism (SWAM) as your authentication mechanism, you cannot integrate security with the Session Manager.

Security integration rules for HTTP sessions

- Sessions in unsecured pages are treated as accesses by anonymous users.
- Sessions created in unsecured pages are created under the identity of that anonymous user.
- Sessions in secured pages are treated as accesses by the authenticated user.
- Sessions created in secured pages are created under the identity of the authenticated user. They can only be accessed in other secured pages by the same user. To protect these sessions from use by unauthorized users, they cannot be accessed from an insecure page.

Programmatic details and scenarios

WebSphere Application Server - Express maintains the security of individual sessions.

An identity or user name, readable by the `com.ibm.websphere.servlet.session.IBMSession` interface, is associated with a session. An unauthenticated identity is denoted by the user name `anonymous`.

WebSphere Application Server - Express includes the `com.ibm.webSphere.servlet.session.UnauthorizedSessionRequestException` interface, which is used when a session is requested without the necessary credentials. For more information on these interfaces, see Package `com.ibm.websphere.servlet.session`.



The Session Manager uses the WebSphere Application Server - Express security infrastructure to determine the authenticated identity associated with a client HTTP request that either retrieves or creates a session. WebSphere Application Server - Express security determines identity using certificates, LTPA, and other methods.

After obtaining the identity of the current request, the Session Manager determines whether the session requested using a `getSession()` call should be returned.

The table lists possible scenarios in which security integration is enabled. The security integration outcomes depend on whether the HTTP request was authenticated and whether a valid session ID and user name was passed to the Session Manager.

| Scenario | Unauthenticated HTTP request is used to retrieve a session | HTTP request is authenticated, with an identity of FRED used to retrieve a session |
|--|---|--|
| No session ID was passed in for this request, or the ID is for a session that is no longer valid | A new session is created. The user name is anonymous | A new session is created and the user name is marked as FRED |
| A session ID for a valid session is passed in. The current session user name is anonymous | The session is returned | The session is returned. The Session Manager changes the user name to FRED |
| A session ID for a valid session is passed in. The current session user name is FRED | The session is not returned. UnauthorizedSessionRequestException is thrown | The session is returned. |
| A session ID for a valid session is passed in. The current session user name is BOB | The session is not returned. UnauthorizedSessionRequestException is thrown. | The session is not returned. UnauthorizedSessionRequestException is thrown. |

Best practices for session programming

Before you develop new objects to be stored in the HTTP session, make sure to consider enabling security integration. HTTP sessions are identified by session IDs, which are pseudo-random numbers that are generated at runtime. Session hijacking is a known attack on HTTP sessions and can be prevented if all requests going over the network are over a secure connection (HTTPS). Not every configuration in a customer environment enforces the security constraint because of potential SSL performance impacts, and HTTP sessions can become vulnerable to hijacking. WebSphere Application Server - Express can integrate HTTP sessions and application server security. Enable security in WebSphere Application Server - Express to protect sessions so that only session creators have access to those sessions.

When adding Java objects to a session, make sure they are in the correct class path. If Java objects are added to a session, be sure to place the class files for those objects in the application server class path or in the Web module path. Because the HttpSession object is shared among servlets that the user might access, consider adopting a site-wide naming convention to avoid conflicts.

Note: Do not store large Object graphs in HttpSession.

In most applications, each servlet requires only a fraction of the total session data. However, by storing the data in HttpSession as one large object, an application forces WebSphere Application Server - Express to process all of it each time.

Release HttpSession objects when you are finished. HttpSession objects live inside the Web container until one of the following occurs:

- The application explicitly and programmatically releases it using `javax.servlet.http.HttpSession.invalidate()`. Quite often, programmatic invalidation is part of an application logout function.
- The application server destroys the allocated HttpSession object when it expires (default is 1800 seconds or 30 minutes).

Note: Do not try to save and reuse the HttpSession object outside of each servlet or JSP.

The HttpSession object is a function of the HttpServletRequest: you can get it only through the getSession() method. A copy of the HttpSession object is valid only for the life of the service() method of the servlet or JSP. You cannot cache the HttpSession object and refer to it outside the scope of a servlet or JSP.

Session programming model and environment

The session lifecycle, from creation to completion, is as follows:

1. Get the HttpSession object.
2. Store and retrieve user-defined data in the session.
3. (Optional) Output an HTML response page containing data from the HttpSession object.
4. (Optional) Notify Listeners.
5. End the session.

The steps are described in detail below. This information, combined with the coding example, provides a programming model for implementing session in servlets. For more information, see “Example: SessionSample.java” on page 131.

For more information, see the API documentation: Package com.ibm.websphere.servlet.session.



The lifecycle in detail

1. Get the HttpSession object.

To obtain a session, use the getSession() method of the javax.servlet.http.HttpServletRequest object in the Java Servlet 2.3 API.

When you first obtain the HttpSession object, the Session Manager uses one of these ways to establish tracking of the session:

- cookies
- URL rewriting
- SSL information

See “Choose a session tracking method” on page 125 for more information.

Assume the Session Manager uses cookies. In such a case, the Session Manager creates a unique session ID and typically sends it back to the browser as a cookie. Each subsequent request from this user (at the same browser) passes the cookie containing the session ID, and the Session Manager uses this to find the user’s existing HttpSession object.

In Step 1 of the code sample, the Boolean(create) is set to true so that the HttpSession is created if it does not already exist. (With the Servlet 2.3 API, the javax.servlet.http.HttpServletRequest.getSession() method with no boolean defaults to true and creates a session if one does not already exist for this user.)

2. Store and retrieve user-defined data in the session.

After a session is established, you can add and retrieve user-defined data to the session. The HttpSession object has methods similar to those in java.util.Dictionary for adding, retrieving, and removing arbitrary Java objects.

In Step 2 of the code sample, the servlet reads an integer object from the HttpSession, increments it, and writes it back. You can use any name to identify values in the HttpSession object. The code sample uses the name sessiontest.counter.

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

3. (Optional) Output an HTML response page containing data from the HttpSession object.

To provide feedback to the user that an action has taken place during the session, you may wish to pass HTML code to the client browser that indicates that an action has occurred.

For example, in step 3 of the code sample the servlet generates a Web page that is returned to the user and displays the value of the `sessiontest.counter` each time the user visits that Web page during the session.

4. (Optional) Notify Listeners.

Objects stored in a session that implement the `javax.servlet.http.HttpSessionBindingListener` interface are notified when the session is preparing to end, that is, about to be invalidated. This notice enables you to perform post-session processing.

5. End the session.

You can end a session in one of these ways:

- Automatically with the Session Manager, if a session has been inactive for a specified time. The administrative clients provide a way to specify the amount of time after which to invalidate a session.
- By coding the servlet to call the `invalidate()` method on the session object.

Example: `SessionSample.java`:

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

public class SessionSample extends HttpServlet {

    public void doGet (HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        // Step 1: Get the Session object
        boolean create = true;
        HttpSession session = request.getSession(create);

        // Step 2: Get the session data value
        Integer ival = (Integer) session.getValue ("sessiontest.counter");
        if (ival == null) {
            ival = new Integer (1);
        }
        else {
            ival = new Integer(ival.intValue () + 1);
        }
        session.putValue ("sessiontest.counter", ival);

        // Step 3: Output the page
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head><title>Session Tracking Test</title></head>");
        out.println("<body>");
        out.println("<h1>Session Tracking Test</h1>");
        out.println("You have hit this page " + ival + " times" + "<br>");
        out.println("Your " + request.getHeader("Cookie"));
        out.println("</body></html>");
    }
}
```

Configure session management

When you configure session management at the Web container level, all applications and the respective Web modules in the Web container normally inherit that configuration, setting up a basic default configuration for the applications and Web modules below it.

However, you can set up different configurations individually for specific applications and Web modules that vary from the Web container default. These different configurations override the default for these applications and Web modules only.

Note: When you overwrite the default session management settings on the application level, all the Web modules below that application inherit this new setting unless they too are set to overwrite these settings.

To configure session management, perform these steps:

1. Start the WebSphere administrative console.
2. Select the level to which this configuration applies:
 - For the Web container level:
 - a. Click **Servers** → **Application Servers**.
 - b. Click the name of the server in which the Web container runs.
 - c. Under **Additional Properties**, click **Web Container**.
 - d. Under **Additional Properties**, click **Session Management**.
 - For the enterprise application level:
 - a. Click **Applications** → **Enterprise Applications**.
 - b. Click the name of the application.
 - c. Under **Additional Properties**, click **Session Management**.
 - For the Web module level:
 - a. Click **Applications** → **Enterprise Applications**.
 - b. Click the name of the application.
 - c. Under **Related Items**, click **Web Modules**.
 - d. Under **Additional Properties**, click **Session Management**.
3. If you are working on the Web module or enterprise application level and want these settings to override the inherited Session Management settings, select **Override**.
4. Specify which session tracking mechanism you want to pass the session ID between the browser and the servlet:
 - To track sessions with SSL information, click **Enable SSL ID tracking**.
 - To track sessions with cookies, click **Enable Cookies**. To change the cookie settings, click **Cookies**.
 - To track sessions with URL rewriting, click **Enable URL Rewriting**. If you want to enable protocol switch rewriting, click **Enable protocol switch rewriting**.
5. Configure other settings or accept the default values.
6. (Optional) “Configure session tracking for Wireless Application Protocol devices.”
7. Click **Apply** and **Save**.

Configure session tracking for Wireless Application Protocol devices: Most Wireless Application Protocol (WAP) devices do not support cookies. The preferred way to track sessions for WAP devices is to use URL rewriting. However on most WAP devices, the maximum allowed URL length is 128 characters. With URL rewriting, a session identifier is added to the URL itself, effectively decreasing the space available for the actual URL and the number of parameters that can be sent on a request.

To reduce the length of session identifier, you can configure key (jsessionid), session ID length and clone ID.

Perform these steps to make configuration changes:

1. Start the WebSphere administrative console.
2. Expand **Servers**.
3. Click **Application Servers**.
4. Select a server from the list of application servers.
5. Under **Additional Properties**, click **Web Container**.
6. Under **Additional Properties**, click **Custom Properties**.

7. Add the appropriate properties from the following list:

- HttpSessionIdLength
- SessionRewriteIdentifier
- HttpSessionCloneId
- CloneSeparatorChange
- NoAdditionalSessionInfo
- SessionIdentifierMaxLength

See the help text for additional information on these Web container custom properties.

8. Click **Apply** and **Save**.

Assemble applications to share session data

In accordance with the Servlet 2.3 API specification, by default the Session Management facility supports session scoping by Web module. Only servlets in the same Web module can access the data associated with a particular session. WebSphere Application Server - Express provides an option that you can use to extend the scope of the session attributes to an enterprise application. Therefore, you can share session attributes across all the Web modules in an enterprise application. This option is provided as an IBM extension.

Restriction: To use this option, you must install all the Web modules in the enterprise application on a given server. You cannot split up Web modules in the enterprise application by servers. For example, with an enterprise application containing two Web modules, you cannot use this option when one Web module is installed on one server and second Web module is installed on a different server. In such split installations, applications might share session attributes across Web modules using distributed sessions, but session data integrity is lost when concurrent access to a session is made in different Web modules. It also severely restricts use of some Session Management features, like TIME_BASED_WRITES. For enterprise applications on which this option is enabled, the Session Management configuration on the Web module inside the enterprise application is ignored. Then Session Management configuration defined on enterprise application is used if Session Management is overwritten at the enterprise application level. Otherwise, the Session Management configuration on the Web container is used.

Servlet API Behavior: If shared HttpSession context is turned on in an enterprise application, HttpSession listeners defined in all the Web modules inside the enterprise application are invoked for session events. The order of listener invocation is not guaranteed.

Perform these steps to share session data across Web modules in an enterprise application:

1. Start the WebSphere Development Studio Client for iSeries (WDS*c*).
2. Import the EAR file if it does not already exist in the Workspace.
3. In the J2EE Perspective, expand **Enterprise Applications**, and right click the application you want to modify.
4. Select **Open With** —> **Deployment Descriptor Editor**.
5. In the **WebSphere Extensions** section, select the **Shared session context** check box.
6. Click **File** —> **Save**.
7. Verify that the class definition of attributes in the session are available to all the Web modules in the enterprise application.
8. Export the EAR file and deploy the application.

Tune session management

WebSphere Application Server - Express for iSeries session support has features for tuning session performance and operating characteristics. You configure these options in the WebSphere administrative console. The settings are listed by which page in the console that they appear.

Session management

The Session Management page of the console contains these settings that are related to performance tuning:

- **Overflow and Maximum in-memory session count**
These settings specify the maximum number of sessions that are retained in memory, and whether or not to ever exceed that number. For more information, see “Maximum in-memory session count.”
- **Serialize session access**
Specifies that concurrent session access in a given server is not allowed.

Schedule invalidation

Instead of relying on the periodic invalidation timer that runs on an interval based on the session timeout parameter, you can set specific times for the session management facility to scan for invalidated sessions in a distributed environment. For more information, see “Configure scheduled invalidation.”

Maximum in-memory session count: The maximum in-memory session count number has different meanings, depending on session support configuration:

- When sessions are being stored in memory, session access is optimized for up to this number of sessions. This value also specifies the number of sessions in the base session table.
- When sessions are being stored in another instance, it also specifies the cache size and the number of last access time updates that are saved in manual update mode.

General memory requirements for the hardware system, and the usage characteristics of the e-business site, determines the optimum value.

Overflow in non-persistent sessions

By default, the number of sessions maintained in memory is specified by the maximum in-memory session count. If you do not wish to place a limit on the number of sessions maintained in memory and allow overflow, select **Allow overflow**.

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

When overflow is disallowed, the Session Manager still returns a session with the `HttpServletRequest`'s `getSession(true)` method if the memory limit has currently been reached, but it would be an invalid session that is not saved.

With the WebSphere Application Server - Express extension to `HttpSession`, `com.ibm.websphere.servlet.session.IBMSession`, an `isOverflow()` method returns true if the session is an invalid session. An application could check this and react accordingly.

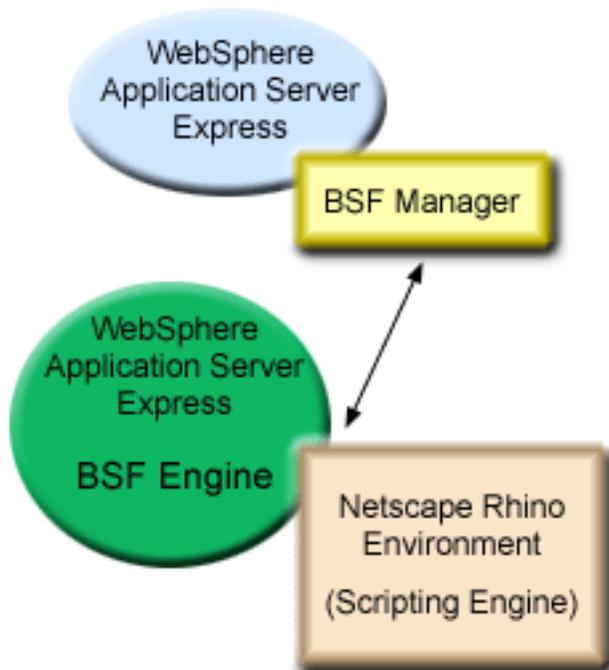
Configure scheduled invalidation: Instead of relying on the periodic invalidation timer that runs on an interval based on the session timeout parameter, you can set specific times for the session management facility to scan for invalidated sessions in a distributed environment. When used with distributed sessions, this feature has the following benefits:

- You can schedule the scan for invalidated sessions for times of low application server activity, avoiding contention between invalidation scans of database or another WebSphere Application Server - Express instance and read and write operations to service HTTP session requests.
- Significantly fewer external write operations can occur when running with the End of Service Method Write mode because the last access time of the session does not need to be written out on each HTTP request. (Manual Update options and Time Based Write options already minimize the writing of the last access time.)

Usage considerations

- With scheduled invalidation configured, HttpSession timeouts are not strictly enforced. Instead, all invalidation processing is handled at the configured invalidation times.
- HttpSessionBindingListener processing is handled at the configured invalidation times unless the HttpSession.invalidate() method is explicitly called.
- The HttpSession.invalidate() method immediately invalidates the session from both the session cache and the external store.
- The periodic invalidation thread still runs with scheduled invalidation. If the current hour of the day does not match one of the configured hours, sessions that have exceeded the invalidation interval are removed from cache, but not from the external store. Another request for that session results in returning that session back into the cache.
- When the periodic invalidation thread runs during one of the configured hours, all sessions that have exceeded the invalidation interval are invalidated by removal from both the cache and the external store.
- The periodic invalidation thread can run more than once during an hour and does not necessarily run exactly at the top of the hour.
- If you specify the interval for the periodic invalidation thread using the HttpSessionReaperPollInterval custom property, do not specify a value of more than 3600 seconds (1 hour) to ensure that invalidation processing happens at least once during each hour.

Bean Scripting Framework



The Bean Scripting Framework (BSF) enables you to use scripting language functions in your Java server-side applications. This framework also extends scripting languages so that you can use existing Java classes and Java beans in the JavaScript language. BSF in WebSphere Application Server - Express for iSeries supports the Rhino ECMAScript.

Note: Support for the Bean Scripting Framework in the JSP Engine is being deprecated in WebSphere Application Server - Express, Version 5.1 and will not be supported in the JSP Engine in future versions of WebSphere Application Server - Express.

BSF provides an access mechanism to Java objects for the scripting languages it supports, so that both the scripting language and the Java code can access code exclusive functions. The access mechanism is implemented through a registry of objects that is maintained by BSF. With BSF, you can write scripts that create, manipulate and access values from Java objects, or you can write Java programs that evaluate and access results from scripts.

WebSphere Application Server - Express provides the Bean Scripting Framework, which consists of a BSF manager, a BSF engine, and a scripting engine. (See the figure.)

See these examples of using BSF with WebSphere Application Server - Express:

“Example: Convert JavaScript source to the Bean Scripting Framework”

This example demonstrates how to convert JavaScript code into BSF code.

“Scenario: Create a Bean Scripting Framework application” on page 137

This scenario demonstrates how to implement a BSF application by using JavaScript code within JSP tags.

See the “Code license and disclaimer information” on page 184 for legal information about these code examples.

For more information about BSF, see these references:

- Using Rhino with BSF and Apache



- Web Standards Project’s Scripting page



Example: Convert JavaScript source to the Bean Scripting Framework

JavaScript code is one of the most popular languages of Web developers. This language supports the following base objects, plus additional objects from the Document Object Model:

- array
- date
- math
- number
- string

Server-side JavaScript code supports the same base objects, and additional objects that support user access to databases, file systems and e-mail systems. Like client-side JavaScript code, server-side JavaScript code is also platform, browser, and language independent.

You can convert server-side JavaScript applications to the Bean Scripting Framework. This topic describes how to perform this conversion.

Server-side JavaScript source code

Suppose you have the following server-side JavaScript application:

```
<html>
<head>
<title>Hello World server-side JavaScript example</title>
</head>
<body>
<br><br>
```

```

</body>
</html>

<server>
function writePage()
  write("<center><font size='6'>Hello World</font></center>");
</server>

```

Convert server-side JavaScript source code to the Bean Scripting Framework (BSF)

Make the following changes to the JavaScript source code to enable BSF:

```

<%@ page language="javascript" %>
<html>
<head>
<title>Hello World server-side BSF/JavaScript example</title>
</head>
<body>
<br><br>
</body>
</html>

<%
  out.println("<center><font size='6'>Hello World</font></center>");
%>

```

Review “Bean Scripting Framework” on page 135 and “Scenario: Create a Bean Scripting Framework application” for deployment information and additional programming examples.

Scenario: Create a Bean Scripting Framework application

Programming skills in JavaScript code are more prevalent than programming skills using JavaServer Pages (JSP) tags. Using the Bean Scripting Framework, JavaScript programmers can gradually introduce JSP tags in their JavaScript applications without completely rewriting the source code. The BSF method not only reduces the potential of programming errors, but also provides a painless way to learn a new technology.

This scenario illustrates how to implement a BSF application using JavaScript within JSP tags.

Develop the BSF application

At ABC elementary school, John Doe teaches third grade mathematics. He wants to help his students memorize their multiplication tables and thinks a small Web-based quiz could help meet his objective. However, John Doe only knows JavaScript.

Using the Bean Scripting Framework to help leverage his JavaScript skills, John Doe creates two JSP files, `multiplication_test.jsp` and `multiplication_scoring.jsp`.

In the `multiplication_test.jsp` file, John Doe uses both client-side and server-side JavaScript code to generate a test of 100 random multiplication questions, displayed using a three minute timer. He then writes the `multiplication_scoring.jsp` file to read the data submitted by the `multiplication_test.jsp` file and to generate the scoring results.

John Doe creates these two files:

`multiplication_test.jsp`

```

<html>
<head>
<title>Multiplication Practice Test</title>
<script language="javascript">
var countMin=3;
var countSec=0;

```

```

function updateDisplay (min, sec) {
  var disp;
  if (min <= 9) disp = " 0";
  else disp = " ";
  disp += (min + ":");
  if (sec <= 9) disp += ("0" + sec);
  else disp += sec;
  return(disp);
}
function countDown() {
  countSec--;
  if (countSec == -1) {
    countSec = 59;
    countMin--;
  }
  document.mulpttest.counter.value = updateDisplay(countMin, countSec);
  if((countMin == 0) &&(countSec == 0)) document.mulpttest.submit();
  else var down = setTimeout("countDown();", 1000);
}
</script>
</head>
<body bgcolor="#ffffff" onLoad="countDown();">
<%@ page language="javascript" %>
<h1>Three Minute Multiplication Drill</h1>
<hr>
<h2>Remember: this is an opportunity to excel!</h2>
<p>
<form method="POST" name="multttest" action="multiplication_scoring.jsp">
<div align="center">
<table>
<tr>
<td>
<h3>Time left:
<input type="text" name="counter" size="9" value="03:00" readonly>
</h3>
</td>
<td>
<input type="submit" value="Submit for scoring!">
</td>
</tr>
</table>
<table border="1">
<%
var newrow = 0;
var q_num = 0;
function addQuestion(num1, num2) {
  if (newrow == 0) out.println("<tr>");
  out.println("<td>");
  out.println(num1 + " x " + num2 + " = ");
  out.println("</td><td>");
  out.print("<input name=\"" + q_num + "\"|\" + num1 + \":\" + num2 + \"\");
  out.println("type=\"text\" size=\"10\">");
  out.println("</td>");
  if (newrow == 3) {
    out.println("</tr>");
    newrow = 0;
  }
  else newrow++;
  q_num++;
}
for (var i = 0; i < 100; i++) {
  var rand1 = Math.ceil(Math.random() * 12);
  var rand2 = Math.ceil(Math.random() * 12);
  addQuestion(rand1, rand2);
}
%>
</table>

```

```

</div>
</form>
</body>
</html>

```

multiplication_scoring.jsp

```

<html>
<head>
<title>Multiplication Practice Test Results</title>
</head>
<body bgcolor="#ffffff">
<%@ page language="javascript" %>
<h1>Multiplication Drill Score</h1>
<hr>
<div align="center">
<table border="1">
<tr><th>Problem</th><th>Correct Answer</th><th>Your Answer</th></tr>
<%
var total_score = 0;
function score (current, pos1, pos2) {
    var multiplier = current.substring(pos1 + 1, pos2);
    var multiplicand = current.substring(pos2 + 1, current.length());
    var your_product = request.getParameterValues(current)[0];
    var true_product = multiplier * multiplicand;
    out.println("<tr>");
    out.println("<td>" + multiplier + " x " + multiplicand + " = </td>");
    out.println("<td>" + true_product + "</td>");
    if (your_product == true_product) {
        total_score++;
        out.print("<td bgcolor=\"\#00ff00\">");
    }
    else {
        out.print("<td bgcolor=\"\#ff0000\">");
    }
    out.println(your_product + "</td>");
    out.println("</tr>");
}
var equations = request.getParameterNames();
while(equations.hasMoreElements()) {
    var currElt = equations.nextElement();
    var splitPos1 = currElt.indexOf("|");
    var splitPos2 = currElt.indexOf(":");
    if (splitPos1 >=0 && splitPos2 >= 0) score(currElt, splitPos1, splitPos2);
}
%>
</table>
<h2>Total Score: <%= total_score %></h2>
<h3><a href="multiplication_test.jsp">Try again?</a></h3>
</div>
</body>
</html>

```

Perform these steps to see how John Doe uses BSF to implement JavaScript in a JSP application:

1. Give your files a .jsp extension.
2. Use server-side JavaScript code in your application.

The multiplication_test.jsp file incorporates both client-side and server-side JavaScript. Server-side JavaScript is similar to client-side JavaScript; the primary difference consists of using a different set of objects. Whereas client-side Javascript programmers invoke document and window objects, server-side JavaScript programmers, using the Bean Scripting Framework, invoke a set of objects provided by the JSP technology. Also, client-side scripts are enclosed in <script> tags, but server-side scripts use JSP scriptlet and expression tags.

Examine the following blocks of code:

```

<script language="javascript">
var countMin=3;
var countSec=0;
function updateDisplay (min, sec) {
  var disp;
  if (min <= 9) disp = " 0";
  else disp = " ";
  disp += (min + ":");
  if (sec <= 9) disp += ("0" + sec);
  else disp += sec;
  return(disp);
}
function countDown() {
  countSec--;
  if (countSec == -1) {
    countSec = 59;
    countMin--;
  }
  document.mulpttest.counter.value = updateDisplay(countMin, countSec);
  if((countMin == 0) && (countSec == 0)) document.mulpttest.submit();
  else var down = setTimeout("countDown();", 1000);
}
</script>
...
<body bgcolor="#ffffff" onLoad="countDown();">
...
<form method="POST" name="multttest" action="multiplication_scoring.jsp">
...
<input type="text" name="counter" size="9" value="03:00" readonly>
...

```

The JavaScript code contained in the <script> block implements a timer set within the <input> field named counter. The onLoad event handler in the <body> tag causes the browser to load and execute the code when the page is loaded.

The document.mulpttest.submit() statement causes the form named multttest to be submitted when the timer expires.

3. Identify the code to the BSF function.

This code example, from the multiplication_test.jsp file, displays the use of a JSP directive. This directive tells the WebSphere Express BSF function that this file is using the JavaScript language, and that the JavaScript code is enclosed by the <% ... %> scriptlet tags. The out implicit JSP object in this code example, creates the body section of a table from 100 randomly generated questions.

```

...
<%@ page language="javascript" %>
...
<%
var newrow = 0;
var q_num = 0;

function addQuestion(num1, num2) {
  if (newrow == 0) out.println("<tr>");

  out.println("<td>");
  out.println(num1 + " x " + num2 + " = ");
  out.println("</td><td>");
  out.print("<input name=\" " + q_num + "\" | " + num1 + ":" + num2 + "\" ");
  out.println("type=\"text\" size=\"10\">");
  out.println("</td>");

  if (newrow == 3) {
    out.println("</tr>");
    newrow = 0;
  }
  else newrow++;

  q_num++;
}

```

```

}

for (var i = 0; i < 100; i++) {
    var rand1 = Math.ceil(Math.random() * 12);
    var rand2 = Math.ceil(Math.random() * 12);

    addQuestion(rand1, rand2);
}

%>
...

```

4. Read the results.

To score the results of the practice drill, John Doe uses the request implicit JSP object in the `multiplication_scoring.jsp` file to obtain the POST data created within the `<form>` tags in the `multiplication_test.jsp` file.

The `multiplication_scoring.jsp` file uses the POST data to build an output file containing the original question, the student's answer, and the correct answer, and then prints the text in a table format using the `out` implicit object.

The following code example from the `multiplication_scoring.jsp` file illustrates the use of the request and out JSP objects:

```

...
<%@ page language="javascript" %>
...
<%
var total_score = 0;
function score (current, pos1, pos2) {
    var multiplier = current.substring(pos1 + 1, pos2);
    var multiplicand = current.substring(pos2 + 1, current.length());
    var your_product = request.getParameterValues(current)[0];
    var true_product = multiplier * multiplicand;
    out.println("<tr>");
    out.println("<td>" + multiplier + " x " + multiplicand + " = </td>");
    out.println("<td>" + true_product + "</td>");
    if (your_product == true_product) {
        total_score++;
        out.print("<td bgcolor=\"\#00ff00\">");
    }
    else {
        out.print("<td bgcolor=\"\#ff0000\">");
    }
    out.println(your_product + "</td>");
    out.println("</tr>");
}
var equations = request.getParameterNames();
while(equations.hasMoreElements()) {
    var currElt = equations.nextElement();
    var splitPos1 = currElt.indexOf("|");
    var splitPos2 = currElt.indexOf(":");
    if (splitPos1 >=0 && splitPos2 >= 0) score(currElt, splitPos1, splitPos2);
}

%>
...
<h2>Total Score: <%= total_score %></h2>
...

```

Note: Although using separate scriptlet blocks of code for different portions of a conditional expression is common in JSP files implemented in Java, it is invalid for JSP files implemented using JavaScript through the Bean Scripting Framework. The JavaScript code must be entirely contained within the scriptlet tags.

The following code example illustrates invalid usage:

```
<% if (pass == 0) %>
  <i>pass is true</i>
<% else %>
  <i>pass is not true</i>
```

Deploy the BSF application

You assemble and deploy BSF applications in the same manner as JSP applications.

Deploy the BSF code examples in WebSphere Express to view this applications processing and output. Use these quick steps to deploy the application.

Note: The intent of these “quick steps” is to provide you with instant application output. However, the supported method for deployment is the same as for standard JSP files.

Perform these steps:

1. Add your BSF files to your application using the WebSphere Development Studio Client. Copy your JSP files to your application directory.
2. Start the application server.
3. Open a browser and request your BSF application. Use the following URL to request your application:
`http://your.server.name:port/jspFileName.jsp`
where *your.server.name* is the host name of your server, *port* is the port number, and *jspFileName* is the name of your JSP file.

Internationalization of applications

For applications that are used in multiple regions around the world, corresponding user interfaces need to support multiple locales and time zones. WebSphere Application Server - Express supports the maintenance and deployment of centralized message catalogs for the output of properly formatted, language-specific (localized) interface strings.

See the following topics for detailed information about internationalization and how to set up your applications to use internationalization:

“Overview of internationalization”

This topic explains what internationalization is and how applications can be configured to interact with users from different localities in culturally appropriate ways.

“Internationalize your application” on page 143

This topic provides instructions on how to internationalize your application, including development, assembly, and deployment.

“Internationalization resources” on page 145

This topic provides a list of resources that are available to learn more about internationalization and internationalization implementation.

Overview of internationalization

Internationalization is defined as the presentation of information to users in an application according to regional cultural conventions. The application can be configured to interact with users from different localities in culturally appropriate ways. In an internationalized application, a user in one region sees error messages, output, and interface elements in the requested language. Date formats, time formats, and currencies are presented appropriately for users in the specified region. A user in another region sees output in the conventional language or format for that region.

Historically, the creation of internationalized applications has been restricted to large corporations that write complex systems. Internationalization techniques have been traditionally expensive and difficult to

implement; they have been applied only to major development efforts. However, increasing usage of distributed computing and the World Wide Web have forced application developers to internationalize a much wider variety of applications. This requires making internationalization techniques more accessible to application developers.

In a non-internationalized application, the interface that is seen by the user is unalterably written into the application code. Alternatively, however, the application developer can localize the displayed strings on the interface and add a layer of abstraction into the design of the application. Instead of printing a simple error message, an internationalized application represents the error message with some language-neutral information. In the simplest example, each error condition corresponds to a key. To print a usable error message, the application looks up the key in the configured message catalog. Each message catalog is a list of keys with associated strings. Different message catalogs provide strings for the different languages supported. The application looks up the key in the appropriate catalog, retrieves the corresponding error message in the requested language, and prints this string for the user.

Localization of text can be used for more than translating error messages. For example, by using keys to represent each element in a graphical user interface (GUI), and by providing the appropriate message catalogs, the GUI (including buttons, menus, etc.) can support multiple languages. Extending support to additional languages requires that the application developer provide message catalogs for those languages. In many cases, the application itself needs no further modification.

Internationalization of an application is driven by two variables: the time zone and the locale. The time zone indicates how to compute the local time as an offset from a standard time (such as Greenwich Mean Time). The locale is a collection of information about language, currency, and the conventions for presenting information (such as dates). In a localized application, the locale also indicates the message catalog where an application should retrieve message strings. A time zone can cover many locales, and a single locale can span time zones. With both time zone and locale, the date, time, currency, and language for users in a specific region can be determined.

Internationalize your application

For applications that are used in multiple regions around the world, corresponding user interfaces need to support multiple locales and time zones. WebSphere Application Server - Express supports the maintenance and deployment of centralized message catalogs for the output of properly formatted, language-specific (localized) interface strings.

Perform the following steps to set up internationalization in your application:

1. **“Identify localizable text”**

This topic explains how to identify the elements of your application that are potential candidates for internationalization.

2. **“Create message catalogs” on page 144**

This topic explains how to create message catalogs. Message catalogs are necessary for locales that will be supported by your application.

3. **“Assemble your application code” on page 145**

Your application code must be assembled as one or more application components before you can prepare for and deploy your application into the WebSphere Application Server - Express environment.

4. **“Step 5: Deploy your application” on page 180** in the *Application development* topic.

After you develop your internationalized application and assemble it for deployment into WebSphere Application Server - Express, you can use the administrative console to install application files on the server and manage the activity of deployed applications.

Identify localizable text: In a non-internationalized application, the interface that is seen by the user is unalterably written into the application code. Alternatively, however, the application developer can localize the displayed strings on the interface and add a layer of abstraction into the design of the application.

For example, suppose you are localizing the GUI for a banking system. The first window contains a pull-down list to be used for selecting a type of account. The labels for the list and the account types in the list are good choices for localization. Three elements require keys: the list itself and two items in the list.

Perform the following steps to identify localizable text in your application:

1. Determine the elements of the application that need to be translated.

Good candidates for localization include:

- Graphical user interfaces: window titles, menus and menu items, buttons, and on screen instructions
- Prompts in command line interfaces
- Application output: messages and logs

2. Assign a unique key to each element for use in message catalogs for the application.

The key provides a language-neutral link between the application and language-specific strings in the message catalogs. Establishing a naming convention for keys before creating the catalogs can make writing code with these keys more intuitive for interface programmers.

Next step: “Create message catalogs.”

Create message catalogs: You can create a catalog several ways:

- As a subclass of `java.util.ResourceBundle`
- As a Java properties file

The properties file approach is more common, because properties files can be prepared by people without programming experience and can be added into the application without modifying the application code.

Perform the following steps to create message catalogs:

1. For each string identified for localization, add a line to the message catalog that lists the string’s key and value in the current language.

In a properties file, each line has the following structure:

```
key = string associated with the key
```

2. Save the catalog, and give it a locale-specific name.

To enable resolution to a specific properties file, the Java API specifies naming conventions for the properties files in a resource bundle as `bundleName_localeID.properties`. Give the set of message catalogs a collective name (for example, `BankingResources`).

For information about locale IDs that are recognized by the Java API, see “Internationalization resources” on page 145.

Example: Message catalogs

The following English catalog (`BankingResources_en.properties`) supports the labels for the list and two list items:

```
accountString = Accounts
savingsString = Savings
checkingString = Checking
```

The corresponding German catalog (`BankingResources_de.properties`) supports the labels as follows:

```
accountString = Konten
savingsString = Sparkonto
checkingString = Girokonto
```

Next step: “Assemble your application code” on page 145.

Assemble your application code: You must assemble application modules including Web application archives (WAR) and resource adapter archives (RAR) for your application components before you can install them. Application assembly is the process of creating archive files that bundle all of the components belonging to an application and configuring the run-time behavior of these applications.

Before assembling, gather all of the code you need to package into your assembled modules including:

- Servlets, JavaServer Pages (JSP) files and other Web components
- Resource adapter (connector) implementations
- Other supporting classes and files

To assemble your application, use the WebSphere Development Studio for iSeries. For more information, see the WebSphere Development Studio Client Help.

Next step: “Step 5: Deploy your application” on page 180 in the *Application development* topic.

Internationalization resources

Use the following resources for supplemental information about internationalization.

Programming instructions and examples

- **Java internationalization tutorial**



(<http://java.sun.com/docs/books/tutorial/i18n/index.html>)

An online tutorial that explains how to use the Java 2 SDK Internationalization API.

Programming specifications

- **Java 2 SDK, Standard Edition Documentation: Internationalization**



(<http://java.sun.com/j2se/1.4.2/docs/guide/intl/>)

The Java internationalization documentation from Sun Microsystems, including a list of supported locales and encodings.

- **Making the WWW truly World Wide**



(<http://www.w3.org/International/>)

The W3C's effort to make World Wide Web technology work with the many writing systems, languages, and cultural conventions of the global community.

- **developerWorks - Unicode**



(<http://www.ibm.com/developerworks/unicode/>)

Articles on various subjects relating to Unicode from IBM's developerWorks Web site.

Add logging and tracing to your application

Designers and developers of applications that run with or under WebSphere Application Server - Express, such as servlets and JSP files, and their supporting classes, may find it useful to use the same facility for generating messages that WebSphere Application Server - Express itself uses, JRas.

This approach has advantages over simply adding `System.out.println()` statements to your code:

- Your messages appear in the WebSphere Application Server - Express standard message format with additional data, such as a date and time stamp, added automatically.

- You can more easily correlate problems and events in your own application to problems and events associated with WebSphere Application Server - Express components.
- You can take advantage of the WebSphere Application Server - Express log file management features.

See the following topics for more information about the JRas facility:

“Programming model summary”

This topic provides a summary of the JRas programming model and formalizes usage requirements and restrictions.

“Overview of JRas” on page 147

This topic describes message logging and diagnostic trace and explains some basic JRas concepts.

“Program with the JRas framework” on page 149

This topic describes how to use the JRas extensions to incorporate message logging and diagnostic trace into WebSphere Application Server - Express applications.

“Extending the JRas framework” on page 157

This topic defines the various extension points have been defined to JRas extension classes.

“JRas messages and trace event types” on page 177

This topic describes JRas message and trace event types and their associated parameters.

Programming model summary

The programming model described in this topic builds upon and summarizes some of the concepts already introduced. This section also formalizes usage requirements and restrictions. Use of the WebSphere Application Server - Express JRas extensions in a manner that does not conform to the following programming guidelines is prohibited.

As described previously, you can use the WebSphere Application Server - Express JRas extensions in three distinct operational modes. The programming models concepts and restrictions apply equally across all modes of operation.

- You must not use implementation classes provided by the standalone JRas logging toolkit directly, unless specifically noted otherwise. Direct usage of those classes is not supported. IBM Support will provide no diagnostic aid or bug fixes relating to direct usage of classes provided by the standalone JRas logging toolkit.
- You must obtain message and trace loggers directly from the Manager class. You cannot directly instantiate loggers.
- There is no provision that allows you to replace the WebSphere Application Server - Express message and trace logger classes.
- You must guarantee that the logger names passed to the Manager are unique, and follow the naming constraints documented below. Once a logger is obtained from the Manager, you must not attempt to change the name of the logger by calling the setName() method.
- For any given name, the first call to the Manager results in the Manager creating a logger that is associated with that name. Subsequent calls to the Manager that specify the same name result in a reference to the existing logger being returned.
- The Manager maintains a hierarchical namespace for loggers. It is recommended but not required that a dot-separated, fully qualified class name be used to identify any given logger. Other than dots or periods, logger names cannot contain any punctuation characters, such as asterisk (*), comma(.), equals sign(=), colon(:), or quotes.
- Group names must comply with the same naming restrictions as logger names.

- The loggers returned from the Manager are subclasses of the RASMessageLogger and RASTraceLogger provided by the standalone JRas logging toolkit. You are allowed to call any public method defined by the RASMessageLogger and RASTraceLogger classes. You are not allowed to call any public method introduced by the provided subclasses.
- If you want to operate in either standalone or combined mode, you must provide your own Handler and Formatter subclasses. You are not allowed to use the Handler and Formatter classes provided by the standalone JRas logging toolkit. User written Handlers and Formatters must conform to the documented guidelines.
- Loggers obtained from the Manager come with a WebSphere Application Server - Express handler installed. This handler will write message and trace records to logs defined by the WebSphere Application Server - Express runtime. Manage these logs using the provided systems management interfaces.
- You can programmatically add and remove user-defined Handlers from a logger at any time. Multiple additions and removals of user defined handlers are allowed. You are responsible for creating an instance of the handler to add, configuring the handler by setting the handler's mask value and formatter appropriately, then adding the handler to the logger using the addHandler() method. You are responsible for programmatically updating the masks of user-defined handlers as appropriate.
- You may get a reference to the handler installed within a logger by calling the getHandlers() method on the logger and processing the results. You must not call any methods on the handler obtained in this fashion. You are allowed to remove the WebSphere Application Server - Express handler from the logger by calling the logger's removeHandler() method, passing in the reference to the WebSphere Application Server - Express handler. Once removed, the WebSphere Application Server - Express handler cannot be re-added to the logger.
- You are allowed to define your own message type. The behavior of user-defined message types and restrictions on their definitions is discussed in "Extending the JRas framework" on page 157.
- You are allowed to define your own message event classes. The usage of user-defined message event classes is discussed in "Extending the JRas framework" on page 157.
- You are allowed to define your own trace types. The behavior of user-defined trace types and restrictions on your definitions is discussed in "Extending the JRas framework" on page 157.
- You are allowed to define your own trace event classes. The usage of user-defined trace event classes is discussed in "Extending the JRas framework" on page 157.
- You must programmatically maintain the bits in the message and trace logger masks that correspond to any user-defined types. If WebSphere Application Server - Express facilities are being used to manage the predefined types, these updates must not modify the state of any of the bits corresponding to those types. If you are assuming ownership responsibility for the predefined types then you can change all bits of the masks.

Overview of JRas

Developing, deploying and maintaining applications are complex tasks. For example, when a running application encounters an unexpected condition it might not be able to complete a requested operation. In such a case you might want the application to inform the administrator that the operation has failed and give information as to why. This enables the administrator to take the proper corrective action.

Application developers or maintainers might need to gather detailed information relating to the execution path of a running application in order to determine the root cause of a failure that is due to a code bug. The facilities that are used for these purposes are typically referred to as message logging and diagnostic trace.

Message logging (messages) and diagnostic trace (trace) are conceptually quite similar, but do have important differences. It is important for application developers to understand these differences in order to use these tools properly. To start with, the following operational definitions of messages and trace are provided.

Message

A message entry is an informational record intended to be viewed by end users, systems administrators and support personnel. The text of the message must be clear, concise and interpretable by an end user. Messages are typically localized, meaning they are displayed in the national language of the end user. Although the destination and lifetime of messages might be configurable, some level of message logging is always enabled in normal system operation. Message logging must be used judiciously due to both performance considerations and the size of the message repository.

Trace

A trace entry is an information record that is intended to be used by service engineers or developers. As such a trace record may be considerably more complex, verbose and detailed than a message entry. Localization support is typically not used for trace entries. Trace entries may be fairly inscrutable, understandable only by the appropriate developer or service personnel. It is assumed that trace entries are not written during normal runtime operation, but may be enabled as needed to gather diagnostic information.

WebSphere Application Server - Express provides a message logging and diagnostic trace API that can be used by applications. This API is based on the standalone JRas logging toolkit which was developed by IBM. The standalone JRas logging toolkit is a collection of interfaces and classes that provide message logging and diagnostic trace primitives. These primitives are not tied to any particular product or platform. The standalone JRas logging toolkit provides a limited amount of support (typically referred to as systems management support), including log file configuration support based on property files.

As designed, the standalone JRas logging toolkit does not contain the support required for integration into the WebSphere Application Server - Express runtime or for usage in a J2EE environment. WebSphere Application Server - Express provides a set of extension classes to address these shortcomings. This collection of extension classes is referred to as the JRas extensions. The JRas extensions do not modify the interfaces introduced by the standalone JRas logging toolkit, but simply provide the appropriate implementation classes. The conceptual structure introduced by the standalone JRas logging toolkit is described below. It is equally applicable to the JRas extensions.

JRas Concepts

The following is a basic overview of important concepts and constructs introduced by the standalone JRas logging toolkit. It is not meant to be an exhaustive overview of the capabilities of this logging toolkit, nor is it intended to be a detailed discussion of usage or programming paradigms. More detailed information, including code examples, is available in JRas extensions and its subtopics, and in the Javadoc for the various interfaces and classes that make up the logging toolkit.

- **Event Types**

The standalone JRas logging toolkit defines a set of event types for messages and a set of event types for trace. Examples of message types include informational, warning and error. Examples of trace types include entry, exit and trace.

- **Event Classes**

The standalone JRas logging toolkit defines both message and trace event classes.

- **Loggers**

A logger is the primary object with which the user code interacts. Two types of loggers are defined. These are message loggers and trace loggers. The set of methods on message loggers and trace loggers are different, since they provide different functionality. Message loggers create only message records and trace loggers create only trace records. Both types of loggers contain masks that indicates which categories of events the logger should process and which it should ignore. Although every JRas logger is defined to contain both a message and trace mask, the message logger only uses the message mask and the trace logger only uses the trace mask. For example, by setting a message logger's message

mask to the appropriate state, it can be configured to process only Error messages and ignore Informational and Warning messages. Changing the state of a message logger's trace mask has no effect.

A logger contains one or more handlers to which it forwards events for further processing. When the user calls a method on the logger, the logger will compare the event type specified by the caller to its current mask value. If the specified type passes the mask check, the logger will create an event object to capture the information relating to the event that was passed to the logger method. This might include information such as the names of the class and method which is logging the event, a message and parameters to log, among others. Once the logger has created the event object, it forwards the event to all handlers currently registered with the logger.

- **Handlers**

A handler provides an abstraction over an output device or event consumer. An example is a file handler, which knows how to write an event to a file. The handler also contains a mask that is used to further restrict the categories of events the handler will process. For example, a message logger may be configured to pass both warning and error events, but a handler attached to the message logger may be configured to only pass error events. Handlers also include formatters, which the handler invokes to format the data in the passed event before it is written to the output device.

- **Formatters**

Handlers are configured with Formatters, which know how to format events of certain types. A handler may contain multiple formatters, each of which know how to format a specific class of event. The event object is passed to the appropriate formatter by the handler. The formatter returns formatted output to the handler, which then writes it to the output device.

Program with the JRas framework

Use the JRas extensions to incorporate message logging and diagnostic trace into WebSphere Application Server - Express applications. The JRas extensions are based on the standalone JRas logging toolkit. To program with the JRas framework, you retrieve a reference to the JRas manager, retrieve message and trace loggers by using methods on the returned manager, and call the appropriate methods on the returned message and trace loggers to create message and trace entries, as appropriate.

To program an application using the WebSphere Application Server - Express JRas extensions, perform the following steps:

1. Determine the mode of the extensions: integrated, standalone or combined. For more information on extension modes, see "JRas extensions."
2. If the extensions are used in either standalone or combined mode, create the necessary handler and formatter classes. For examples of how to create handler and formatter classes, see "Extending the JRas framework" on page 157.
3. If localized messages will be used by the application, create a resource bundle as described in "Create JRas resource bundles and message files" on page 151.
4. In the application code, get a reference to the Manager class and create the manager and logger instances as described in "Create JRas manager and logger instances" on page 153.
5. Insert the appropriate message and trace logging statements in the application as described in "Create JRas manager and logger instances" on page 153.

JRas extensions: The standalone JRas logging toolkit defines interfaces and provides a variety of concrete classes that implement these interfaces. Since the standalone JRas logging toolkit was developed as a general purpose toolkit, the implementation classes do not contain the configuration interfaces and methods necessary for use in the WebSphere Application Server - Express product. In addition, many of the implementation classes are not written appropriately for use in a J2EE environment. To overcome these shortcomings, WebSphere Application Server - Express provides the appropriate implementation classes that allow integration into the WebSphere Application Server - Express environment. The collection of these implementation classes is referred to as the JRas extensions.

JRas extensions

You can use the JRas extensions in three distinct operational modes:

- **Integrated**

In this mode, message and trace records are written only to logs defined and maintained by the WebSphere Application Server - Express runtime. This is the default mode of operation and is equivalent to the WebSphere Application Server V4.0 mode of operation.

- **Standalone**

In this mode, message and trace records are written solely to standalone logs defined and maintained by the user. You control which categories of events are written to which logs, and the format in which entries are written. You are responsible for configuration and maintenance of the logs. Message and trace entries are not written to WebSphere Application Server - Express runtime logs.

- **Combined**

In this mode message and trace records are written to both WebSphere Application Server - Express runtime logs and to standalone logs that you must define, control, and maintain. You can use filtering controls to determine which categories of messages and trace are written to which logs.

The JRas extensions are specifically targeted to an integrated mode of operation. The integrated mode of operation can be appropriate for some usage scenarios, but there are many scenarios that are not adequately addressed by these extensions. Many usage scenarios require a standalone or combined mode of operation instead. A set of user extension points has been defined that allow the JRas extensions to be used in either a standalone or combined mode of operations.

JRas extension classes

WebSphere Application Server - Express provides a base set of implementation classes that collectively are referred to as the JRas extensions. Many of these classes provide the appropriate implementations of loggers, handlers and formatters for use in a WebSphere Application Server - Express environment. As previously noted, this collection of classes is targeted at an Integrated mode of operation. If you choose to use the JRas extensions in either standalone or combined mode, you can reuse the logger and manager class provided by the extensions, but you must provide your own implementations of handlers and formatters.

- **WebSphere Application Server - Express Message and Trace loggers**

The message and trace loggers provided by the standalone JRas logging toolkit cannot be directly used in the WebSphere Application Server - Express environment. The JRas extensions provide the appropriate logger implementation classes. Instances of these message and trace logger classes are obtained directly and exclusively from the WebSphere Application Server - Express Manager class, described below. You cannot directly instantiate message and trace loggers. Obtaining loggers in any manner other than directly from the Manager is not allowed. Doing so is a direct violation of the programming model.

The message and trace loggers instances obtained from the WebSphere Application Server - Express Manager class are subclasses of the `RASMessageLogger()` and `RASTraceLogger()` classes provided by the standalone JRas logging toolkit. The `RASMessageLogger()` and `RASTraceLogger()` classes define the set of methods that are directly available. Public methods introduced by the JRas extensions logger subclasses cannot be called directly by user code. Doing so is a violation of the programming model.

Loggers are named objects and are identified by name. When the Manager class is called to obtain a logger, the caller is required to specify a name for the logger. The Manager class maintains a name-to-logger instance mapping. Only one instance of a named logger will ever be created within the lifetime of a process. The first call to the Manager with a particular name will result in the logger being created and configured by the Manager. The Manager will cache a reference to the instance, then return it to the caller. Subsequent calls to the Manager that specify the same name will result in a reference to the cached logger being returned. Separate name spaces are maintained for message and trace loggers. This means a single name can be used to obtain both a message logger and a trace logger from the Manager, without ambiguity, and without causing a name space collision.

In general, loggers have no predefined granularity or scope. A single logger could be used to instrument an entire application. Or users may determine that having a logger per class is more desirable. Or the appropriate granularity may lie somewhere in between. Partitioning an application into logging domains is rightfully determined by the application writer.

The WebSphere Application Server - Express logger classes obtained from the Manager are thread-safe. Although the loggers provided as part of the standalone JRas logging toolkit implement the serializable interface, in fact loggers are not serializable. Loggers are stateful objects, tied to a Java virtual machine instance and are not serializable. Attempting to serialize a logger is a violation of the programming model.

Please note that there is no provision for allowing users to provide their own logger subclasses for use in a WebSphere Application Server - Express environment.

- **WebSphere Application Server - Express handlers**

WebSphere Application Server - Express provides the appropriate handler class that is used to write message and trace events to the WebSphere Application Server - Express runtime logs. You cannot configure the WebSphere Application Server - Express handler to write to any other destination. The creation of a WebSphere Application Server - Express handler is a restricted operation and not available to user code. Every logger obtained from the Manager comes preconfigured with an instance of this handler already installed. You can remove the WebSphere Application Server - Express handler from a logger when you want to run in standalone mode. Once you have removed it, you cannot re-add the WebSphere Application Server - Express handler to the logger from which it was removed (or any other logger). Also, you cannot directly call any method on the WebSphere Application Server - Express handler. Attempting to create an instance of the WebSphere Application Server - Express handler, to call methods on the WebSphere Application Server - Express handler or to add a WebSphere Application Server - Express handler to a logger by user code is a violation of the programming model.

- **WebSphere Application Server - Express formatters**

The WebSphere Application Server - Express handler comes preconfigured with the appropriate formatter for data that is written to WebSphere Application Server - Express logs. The creation of a WebSphere Application Server - Express formatter is a restricted operation and not available to user code. No mechanism exists that allows the user to obtain a reference to a formatter installed in a WebSphere Application Server - Express handler, or to change the formatter a WebSphere Application Server - Express handler is configured to use.

- **WebSphere Application Server - Express manager**

WebSphere Application Server - Express provides a Manager class located in the `com.ibm.websphere.ras` package. All message and trace loggers must be obtained from this Manager. A reference to the Manager is obtained by calling the static `Manager.getManager()` method. Message loggers are obtained by calling the `createRASMessageLogger()` method on the Manager. Trace loggers are obtained by calling the `createRASTraceLogger()` method on the Manager class.

The manager also supports a group abstraction that is useful when dealing with trace loggers. The group abstraction allows multiple, unrelated trace loggers to be registered as part of a named entity called a group. WebSphere Application Server - Express provides the appropriate systems management facilities to manipulate the trace setting of a group, similar to the trace settings of an individual trace logger.

For example, suppose component A consist of 10 classes. Suppose each class is configured to use a separate trace logger. Suppose all 10 trace loggers in the component are registered as members of the same group (for example `Component_A_Group`). You can then turn on trace for a single class. Or you can turn on trace for all 10 classes in a single operation using the group name if you want a component trace. Group names are maintained within the name space for trace loggers.

Create JRas resource bundles and message files: The WebSphere Application Server - Express message logger provides the `message()` and `msg()` methods to allow the user to log localized messages. In addition, it provides the `textMessage()` method for logging of messages that are not localized. Applications can use either or both, as appropriate.

The mechanism for providing localized messages is the Resource Bundle support provided by the Java Development Kit (JDK). If you are not familiar with resource bundles as implemented by the JDK, you can get more information from various texts, or by reading the javadoc for the `java.util.ResourceBundle`, `java.util.ListResourceBundle` and `java.util.PropertyResourceBundle` classes, as well as the `java.text.MessageFormat` class.

The `PropertyResourceBundle` is the preferred mechanism to use. In addition, note that the JRas extensions do not support the extended formatting options such as `{1, date}` or `{0,number, integer}` that are provided by the `MessageFormat` class.

You can forward messages that are written to the internal WebSphere Application Server - Express logs to other processes for display. For example, messages displayed on the administration console can be localized using the late binding process. Late binding means that WebSphere Application Server - Express does not localize messages when they are logged, but defers localization to the process that displays the message.

To properly localize the message, the displaying process must have access to the resource bundle where the message text is stored. This means that you must package the resource bundle separately from the application, and install it in a location where the viewing process can access it. If you do not want to take these steps, you can use the early binding technique to localize messages as they are logged.

The techniques are described as follows:

- **Early binding**

The application must localize the message before logging it. The application looks up the localized text in the resource bundle and formats the message. When formatting is complete, the application logs the message using the `textMessage()` method. Use this technique to package the application's resource bundles with the application.

- **Late binding**

The application can choose to have the WebSphere Application Server - Express runtime localize the message in the process where it is displayed. Using this technique, the resource bundles are packaged in a standalone `.jar` file, separately from the application. You must then install the resource bundle `.jar` file on every machine in the installation from which an administrator's console or log viewing program might be run. You must install the `.jar` file in a directory that is part of the extensions classpath. In addition, if you forward logs to IBM service, you must also forward the `.jar` file containing the resource bundles.

To create a resource bundle, perform the following steps.

1. Create a text properties file that lists message keys and the corresponding messages.

The properties file must have the following characteristics:

- Each property in the file is terminated with a line-termination character.
- If a line contains only white space, or if the first non-white space character of the line is the symbol `#` (pound sign) or `!` (exclamation mark), the line is ignored. The `#` and `!` characters can therefore be used to put comments into the file.
- Each line in the file, unless it is a comment or consists only of white space, denotes a single property. A backslash (`\`) is treated as the line-continuation character.
- The syntax for a property file consists of a key, a separator, and an element. Valid separators include the equal sign (`=`), colon (`:`), and white space ().
- The key consists of all characters on the line from the first non-white space character to the first separator. Separator characters can be included in the key by escaping them with a backslash (`\`), but doing this is not recommended, because escaping characters is error prone and confusing. It is instead recommended that you use a valid separator character that does not appear in any keys in the properties file.

- White space after the key and separator is ignored until the first non-white space character is encountered. All characters remaining before the line-termination character define the element. See the Java documentation for the `java.util.Properties` class for a full description of the syntax and construction of properties files.
2. The file can then be translated into localized versions of the file with language-specific file names (for example, a file named `DefaultMessages.properties` can be translated into `DefaultMessages_de.properties` for German and `DefaultMessages_ja.properties` for Japanese).
 3. When the translated resource bundles are available, write them to a system-managed persistent storage medium. Resource bundles are then used to convert the messages into the requested national language and locale.
 4. When a message logger is obtained from the JRas manager, it can be configured to use a particular resource bundle. Messages logged via the `message()` API will use this resource bundle when message localization is performed. At run time, the user's locale setting is used to determine the properties file from which to extract the message specified by a message key, thus ensuring that the message is delivered in the correct language.
 5. **Optional:** If the message loggers `msg()` method is called, a resource bundle name must be explicitly provided.

The application locates the resource bundle based on the file's location relative to any directory in the classpath. For instance, if the property resource bundle named `DefaultMessages.properties` is located in the `baseDir/subDir1/subDir2/resources` directory and `baseDir` is in the class path, the name `subdir1.subdir2.resources.DefaultMessage` is passed to the message logger to identify the resource bundle.

For more information on creating JRas resource bundles, see "Develop JRas resource bundles."

Develop JRas resource bundles: You can create resource bundles in several ways. The best and easiest way is to create a properties file that supports a `PropertiesResourceBundle`. This sample shows how to create such a properties file.

For this sample, four localizable messages are provided. The properties file is created and the key-value pairs inserted into it. All the normal properties files conventions and rules apply to this file. In addition, the creator must be aware of other restrictions imposed on the values by the Java `MessageFormat` class. For example, apostrophes must be "escaped" or they will cause a problem. Also avoid use of non-portable characters. WebSphere Application Server - Express does not support usage of extended formatting conventions that the `MessageFormat` class supports, such as `{1, date}` or `{0,number, integer}`.

Assume that the base directory for the application that uses this resource bundle is "baseDir" and that this directory is not in the classpath. Assume that the properties file is stored in a subdirectory of `baseDir` that is not in the classpath (e.g. `baseDir/subDir1/subDir2/resources`). In order to allow the messages file to be resolved, the name `subDir1.subDir2.resources.DefaultMessage` is used to identify the `PropertyResourceBundle` and is passed to the message logger.

For this sample, the properties file is named `DefaultMessages.properties`.

```
# Contents of DefaultMessages.properties file
MSG_KEY_00=A message with no substitution parameters.
MSG_KEY_01=A message with one substitution parameter: parm1={0}
MSG_KEY_02=A message with two substitution parameters: parm1={0}, parm2 = {1}
MSG_KEY_03=A message with three parameter: parm1={0}, parm2 = {1}, parm3={2}
```

Once the file `DefaultMessages.properties` is created, the file can be sent to a translation center where the localized versions will be generated.

Create JRas manager and logger instances: You can use the JRas extensions in integrated, standalone, or combined mode. Configuration of the application will vary depending on the mode of operation, but usage of the loggers to log message or trace entries is identical in all modes of operation.

Integrated mode is the default mode of operation. In this mode, message and trace events are sent to the WebSphere Application Server - Express logs. See "Set up for integrated JRas operation" on page 155 for information on configuring for this mode of operation.

In the combined mode, message and trace events are logged to both WebSphere Application Server - Express and user-defined logs. See "Set up for combined JRas operation" on page 155 for more information on configuring for this mode of operation.

In the standalone mode, message and trace events are logged only to user-defined logs. See "Set up for standalone JRas operation" on page 156 for more information on configuring for this mode of operation.

Using the message and trace loggers

Regardless of the mode of operation, the use of message and trace loggers is the same. See "Create JRas resource bundles and message files" on page 151 for more information on using message and trace loggers.

Using a message logger

The message logger is configured to use the DefaultMessages resource bundle. Message keys must be passed to the message loggers if the loggers are using the message() API.

```
msgLogger.message(RASIMessageEvent.TYPE_WARNING, this, methodName, "MSG_KEY_00");
... msgLogger.message(RASIMessageEvent.TYPE_WARN, this, methodName, "MSG_KEY_01",
    "some string");
```

If message loggers use the msg() API, you can specify a new resource bundle name.

```
msgLogger.msg(RASIMessageEvent.TYPE_ERR, this, methodName, "ALT_MSG_KEY_00",
    "alternateMessageFile");
```

You can also log a text message. If you are using the textMessage API, no message formatting is done.

```
msgLogger.textMessage(RASIMessageEvent.TYPE_INFO, this, methodName,
    "String and Integer", "A String", new Integer(5));
```

Using a trace logger

Since trace is normally disabled, trace methods should be guarded for performance reasons.

```
private void methodX(int x, String y, Foo z) {
    // trace an entry point. Use the guard to make sure tracing is enabled.
    // Do this checking before we waste cycles gathering parameters to be traced.
    if (trcLogger.isLoggable(RASITraceEvent.TYPE_ENTRY_EXIT) {
        // since I want to trace 3 parameters, package them up in an Object[]
        Object[] parms = {new Integer(x), y, z};
        trcLogger.entry(RASITraceEvent.TYPE_ENTRY_EXIT, this, "methodX", parms);
    }
    ... logic
    // a debug or verbose trace point
    if (trcLogger.isLoggable(RASITraceEvent.TYPE_MISC_DATA) {
        trcLogger.trace(RASITraceEvent.TYPE_MISC_DATA, this, "methodX" "reached here");
    }
    ...
    // Another classification of trace event. Here an important state
    // change has been detected, so a different trace type is used.
    if (trcLogger.isLoggable(RASITraceEvent.TYPE_SVC) {
        trcLogger.trace(RASITraceEvent.TYPE_SVC, this, "methodX", "an important event");
    }
    ...
    // ready to exit method, trace. No return value to trace
```

```

    if (trcLogger.isLoggable(RASITraceEvent.TYPE_ENTRY_EXIT)) {
        trcLogger.exit(RASITraceEvent.TYPE_ENTRY_EXIT, this, "methodX");
    }
}

```

Set up for integrated JRas operation: In the integrated mode of operation, message and trace events are sent to WebSphere Application Server - Express logs. This is the default mode of operation.

1. Import the requisite JRas extensions classes:

```

import com.ibm.ras.*;
import com.ibm.websphere.ras.*;

```

2. Declare logger references:

```

private RASMessageLogger msgLogger = null;
private RASTraceLogger trcLogger = null;

```

3. Obtain a reference to the Manager and create the loggers.

Since loggers are named singletons, you can do this in a variety of places. One logical candidate for servlets is the servlet `init()` method. For example, for the servlet named “myTestServlet”, place the following code in the servlet `init()` method.

```

com.ibm.websphere.ras.Manager mgr = com.ibm.websphere.ras.Manager.getManager();
msgLogger = mgr.createRASMessageLogger("Acme", "WidgetCounter", "RasTest",
myTestServlet.class.getName());
// Configure the message logger to use the message file created for this application.
msgLogger.setMessageFile("acme.widgets.DefaultMessages");
trcLogger = mgr.createRASTraceLogger("Acme", "Widgets", "RasTest",
myTestServlet.class.getName());
mgr.addLoggerToGroup(trcLogger, groupName);

```

Set up for combined JRas operation: In combined mode, messages and trace are logged to both WebSphere Application Server - Express logs and user-defined logs. The following sample assumes that you have written a user defined handler named `SimpleFileHandler` and a user defined formatter named `SimpleFormatter`. It also assumes that you are not using user defined types or events.

1. Import the requisite JRas extensions classes:

```

import com.ibm.ras.*;
import com.ibm.websphere.ras.*;

```

2. Import the user handler and formatter:

```

import com.ibm.ws.ras.test.user.*;

```

3. Declare the logger references:

```

private RASMessageLogger msgLogger = null;
private RASTraceLogger trcLogger = null;

```

4. Obtain a reference to the Manager, create the loggers and add the user handlers.

Since loggers are named singletons, you can obtain a reference to the loggers in a number of places. One logical candidate for servlets is the servlet `init()` method. Make sure that multiple instances of the same user handler are not accidentally inserted into the same logger. Your initialization code must handle this. The following sample is a message logger sample. The procedure for a trace logger is similar.

```

com.ibm.websphere.ras.Manager mgr = com.ibm.websphere.ras.Manager.getManager();
msgLogger = mgr.createRASMessageLogger("Acme", "WidgetCounter",
"RasTest", myTestServlet.class.getName());

// Configure the message logger to use the message file
// defined in the ResourceBundle sample.

msgLogger.setMessageFile("acme.widgets.DefaultMessages");

// Create the user handler and formatter. Configure the formatter,
// then add it to the handler.

RASHandler handler = new SimpleFileHandler("myHandler", "FileName");

```

```

RASIFormatter formatter = new SimpleFormatter("simple formatter");
formatter.addEventClass("com.ibm.ras.RASMessageEvent");
handler.addFormatter(formatter);

// Add the Handler to the logger. Add the logger to the list of
// the handlers listeners, then set the handlers
// mask, which will update the loggers composite mask appropriately.
// WARNING - there is an order dependency here that must be followed.

msgLogger.addHandler(handler);
handler.addMaskChangeListener(msgLogger);
handler.setMessageMask(RASIMessageEvent.DEFAULT_MESSAGE_MASK);

```

Set up for standalone JRas operation: In standalone mode, messages and traces are logged only to user-defined logs. The following sample assumes that you have a user-defined handler named SimpleFileHandler and a user-defined formatter named SimpleFormatter. It is also assumed that no user-defined types or events are being used.

1. Import the requisite JRas extensions classes:

```

import com.ibm.ras.*;
import com.ibm.websphere.ras.*;

```

2. Import the user handler and formatter:

```

import com.ibm.ws.ras.test.user.*;

```

3. Declare the logger references:

```

private RASMessageLogger msgLogger = null;
private RASTraceLogger trcLogger = null;

```

4. Obtain a reference to the Manager, create the loggers and add the user handlers.

Since loggers are named singletons, you can obtain a reference to the loggers in a number of places. One logical candidate for servlets is the servlet init() method. Make sure that multiple instances of the same user handler are not accidentally inserted into the same logger. Your initialization code must handle this. The following sample is a message logger sample. The procedure for a trace logger is similar.

```

com.ibm.websphere.ras.Manager mgr = com.ibm.websphere.ras.Manager.getManager();
msgLogger = mgr.createRASMessageLogger("Acme", "WidgetCounter",
    "RasTest", myTestServlet.class.getName());

```

```

// Configure the message logger to use the message file
// defined in the ResourceBundle sample.

```

```

msgLogger.setMessageFile("acme.widgets.DefaultMessages");

```

```

// Get a reference to the Handler and remove it from the logger.

```

```

RASHandler aHandler = null;
Enumeration enum = msgLogger.getHandlers();
while (enum.hasMoreElements()) {
    aHandler = (RASHandler)enum.nextElement();
    if (aHandler instanceof WsHandler)
        msgLogger.removeHandler(wsHandler);
}

```

```

// Create the user handler and formatter. Configure the formatter,
// then add it to the handler.

```

```

RASHandler handler = new SimpleFileHandler("myHandler", "FileName");
RASIFormatter formatter = new SimpleFormatter("simple formatter");
formatter.addEventClass("com.ibm.ras.RASMessageEvent");
handler.addFormatter(formatter);

```

```

// Add the Handler to the logger. Add the logger to the list of the
// handlers listeners, then set the handlers
// mask, which will update the loggers composite mask appropriately.

```

```
// WARNING - there is an order dependency here that must be followed.  
  
msgLogger.addHandler(handler);  
handler.addMaskChangeListener(msgLogger);  
handler.setMessageMask(RASIMessageEvent.DEFAULT_MESSAGE_MASK);
```

Extending the JRas framework

Since the JRas extensions classes do not provide the flexibility and behavior required for many scenarios, a variety of extension points have been defined. You are allowed to write your own implementation classes to obtain the required behavior.

In general, the JRas extensions require you to call the Manager class to obtain a message logger or trace logger. No provision is made to allow you to provide your own message or trace logger subclasses. In general, user-provided extensions cannot be used to affect the integrated mode of operation. The behavior of the integrated mode of operation is solely determined by the WebSphere Application Server runtime and the JRas extensions classes.

Handlers

The standalone JRas logging toolkit defines the RASISHandler interface. All handlers must implement this interface. You can write your own handler classes that implement the RASISHandler interface. You should directly create instances of user-defined handlers and add them to the loggers obtained from the Manager.

The standalone JRas logging toolkit provides several handler implementation classes. These handler classes are inappropriate for usage in the J2EE environment. You cannot directly use or subclass any of the Handler classes provided by the standalone JRas logging toolkit. Doing so is a violation of the programming model.

Formatters

The standalone JRas logging toolkit defines the RASIFormatter interface. All formatters must implement this interface. You can write your own formatter classes that implement the RASIFormatter interface. You can only add these classes to a user-defined handler. WebSphere Application Server - Express handlers cannot be configured to use user-defined formatters. Instead, directly create instances of your formatters and add them to the your handlers appropriately.

As with handlers, the standalone JRas logging toolkit provides several formatter implementation classes. Direct usage of these formatter classes is not supported.

Message event types

The standalone JRas toolkit defines message event types in the RASIMessageEvent interface. In addition, the WebSphere Application Server - Express reserves a range of message event types for future use. The RASIMessageEvent interface defines three types, with values of 0x01, 0x02, and 0x04. The values 0x08 through 0x8000 are reserved for future use. You can provide your own message event types by extending this interface appropriately. User-defined message types must have a value of 0x1000 or greater.

Message loggers retrieved from the Manager have their message masks set to pass or process all message event types defined in the RASIMessageEvent interface. In order to process user-defined message types, you must manually set the message logger mask to the appropriate state by user code after the message logger has been obtained from the Manager. WebSphere Application Server - Express does not provide any built-in systems management support for managing any message types.

Message event objects

The standalone JRas toolkit provides a `RASMessageEvent` implementation class. When a message logging method is called on the message logger, and the message type is currently enabled, the logger creates and distributes an event of this class to all handlers currently registered with that logger.

You can provide your own message event classes, but they must implement the `RASIEvent` interface. You must directly create instances of such user-defined message event classes. Once it is created, pass your message event to the message logger by calling the message logger's `fireRASEvent()` method directly. WebSphere Application Server - Express message loggers cannot directly create instances of user-defined types in response to calling a logging method (`msg()`, `message()`...) on the logger. In addition, instances of user-defined message types are never processed by the WebSphere Application Server - Express handler. You cannot create instances of the `RASMessageEvent` class directly.

Trace event types

The standalone JRas toolkit defines trace event types in the `RASITraceEvent` interface. You can provide your own trace event types by extending this interface appropriately. In such a case you must ensure that the values for the user-defined trace event types do not collide with the values of the types defined in the `RASITraceEvent` interface.

Trace loggers retrieved from the Manager typically have their trace masks set to reject all types. A different starting state can be specified by using WebSphere Application Server - Express systems management facilities. In addition, the state of the trace mask for a logger can be changed at runtime using WebSphere Application Server - Express systems management facilities.

In order to process user-defined trace types, the trace logger mask must be manually set to the appropriate state by user code. WebSphere Application Server - Express systems management facilities cannot be used to manage user-defined trace types, either at start time or runtime.

Trace event objects

The standalone JRas toolkit provides a `RASTraceEvent` implementation class. When a trace logging method is called on the WebSphere Application Server - Express trace logger and the type is currently enabled, the logger creates and distributes an event of this class to all handlers currently registered with that logger.

You can provide your own trace event classes. Such trace event classes must implement the `RASIEvent` interface. You must create instances of such user-defined event classes directly. Once it is created, pass the trace event to the trace logger by calling the trace logger's `fireRASEvent()` method directly. WebSphere Application Server - Express trace loggers cannot directly create instances of user-defined types in response to calling a trace method (`entry()`, `exit()`, `trace()`) on the trace logger. In addition, instances of user-defined trace types are never processed by the WebSphere Application Server - Express handler. You cannot create instances of the `RASTraceEvent` class directly.

User defined types, user defined events and WebSphere Application Server - Express

By definition, the WebSphere Application Server - Express handler will process user-defined message or trace types, or user-defined message or trace event classes. Message and trace entries of either a user-defined type or user-defined event class cannot be written to the WebSphere Application Server - Express runtime logs.

For more information about extending the JRas framework, see the following topics:

- **“Writing User Extensions” on page 159**
This topic describes how to write user written handlers and formatters.
- **“Example: user written handler” on page 161**
This topic gives an example of a user written handler class that writes formatted events to a file.

- **“Example: user written formatter” on page 173**

This topic gives an example of a user written formatter class.

Writing User Extensions: You can configure the WebSphere Application Server - Express to use Java 2 security to restrict access to protected resources such as the file system and sockets. Since user written extensions typically access such protected resources, user written extensions must contain the appropriate security checking calls, using AccessController doPrivileged() calls. In addition, the user written extensions must contain the appropriate policy file. In general, it is recommended that you locate user written extensions in a separate package. It is your responsibility to restrict access to the user written extensions appropriately.

Writing a handler

User written handlers must implement the RASHandler interface. The RASHandler interface extends the RASMaskChangeGenerator interface, which extends the RASObject interface. A short discussion of the methods introduced by each of these interfaces follows, along with implementation pointers.

RASObject interface

The RASObject interface is the base interface for standalone JRas logging toolkit classes that are stateful or configurable, such as loggers, handlers and formatters.

- The standalone JRas logging toolkit supports rudimentary properties-file based configuration. To implement this configuration support, the configuration state is stored as a set of key-value pairs in a properties file. The methods public Hashtable getConfig() and public void setConfig(Hashtable ht) are used to get and set the configuration state. The JRas extensions do not support properties based configuration and it is recommended that these methods be implemented as no-operations. You can implement your own properties based configuration using these methods.
- Loggers, handlers and formatters can be named objects. For example, the JRas extensions require the user to provide a name for the loggers that are retrieved from the manager. You can name your handlers. The methods public String getName() and public void setName(String name) are provided to get or set the name field. The JRas extensions currently do not call these methods on user handlers. You can implement these methods as you want, including as no operations.
- Loggers, handlers and formatters can also contain a description field. The methods public String getDescription() and public void setDescription(String desc) can be used to get or set the description field. The JRas extensions currently do not use the description field. You can implement these methods as you want, including as no operations.
- The method public String getGroup() is provided for usage by the RASManager. Since the JRas extensions provide their own Manager class, this method is never called. It is recommended you implement this as a no-operation.

RASMaskChangeGenerator interface

The RASMaskChangeGenerator interface is the interface that defines the implementation methods for filtering of events based on a mask state. This means that it is currently implemented by both loggers and handlers. By definition, an object that implements this interface contains both a message mask and a trace mask, although both need not be used. For example, message loggers contain a trace mask, but the trace mask is never used since the message logger never generates trace events. Handlers however can actively use both mask values. For example a single handler could handle both message and trace events.

- The methods public long getMessageMask() and public void setMessageMask(long mask) are used to get or set the value of the message mask. The methods public long getTraceMask() and public void setTraceMask(long mask) are used to get or set the value of the trace mask.

In addition, this interface introduces the concept of calling back to interested parties when a mask changes state. The callback object must implement the RASMaskChangeListener interface.

- The methods `public void addMaskChangeListener(RASIMaskChangeListener listener)` and `public void removeMaskChangeListener(RASIMaskChangeListener listener)` are used to add or remove listeners to the handler. The method `public Enumeration getMaskChangeListeners()` returns an Enumeration over the list of currently registered listeners. The method `public void fireMaskChangedEvent(RASMaskChangeEvent mc)` is used to call back all the registered listeners to inform them of a mask change event.

For efficiency reasons, the Jras extensions message and trace loggers implement the `RASIMaskChangeListener` interface. The logger implementations maintain a “composite mask” in addition to the logger’s own mask. The logger’s composite mask is formed by logically using the OR keyword to join the appropriate masks of all handlers that are registered to that logger, then using the AND keyword to join the result with the logger’s own mask. For example, the message logger’s composite mask is formed by or’ing the message masks of all handlers registered with that logger, then and’ing the result with the logger’s own message mask.

This means that all handlers are required to properly implement these methods. In addition, when a user handler is instantiated, the logger it is to be added to should be registered with the handler using the `addMaskChangeListener()` method. When either the message mask or trace mask of the handler is changed, the logger must be called back to inform it of the mask change. This allows the logger to dynamically maintain the composite mask.

The `RASMaskChangedEvent` class is defined by the standalone Jras logging toolkit. Direct usage of that class by user code is allowed in this context.

In addition the `RASIMaskChangeGenerator` introduces the concept of caching the names of all message and trace event classes that the implementing object will process. The intent of these methods is to allow a management program such as a GUI to retrieve the list of names, introspect the classes to determine the event types that they might possibly process and display the results. The Jras extensions do not ever call these methods, so they can be implemented as no operations, if desired.

- The methods `public void addMessageEventClass(String name)` and `public void removeMessageEventClass(String name)` can be called to add or remove a message event class name from the list. The method `public Enumeration getMessageEventClasses()` will return an enumeration over the list of message event class names. Similarly, the `public void addTraceEventClass(String name)` and `public void removeTraceEventClass(String name)` can be called to add or remove a trace event class name from the list. The method `public Enumeration getTraceEventClasses()` will return an enumeration over the list of trace event class names.

RASHandler interface

The `RASHandler` interface introduces the methods that are specific to the behavior of a handler.

The `RASHandler` interface as provided by the standalone Jras logging toolkit supports handlers that run in either a synchronous or asynchronous mode. In asynchronous mode, events are typically queued by the calling thread and then written by a worker thread. Since spawning of threads is not allowed in the WebSphere Application Server environment, it is expected that handlers will not queue or batch events, although this is not expressly prohibited.

- The methods `public int getMaximumQueueSize()` and `public void setMaximumQueueSize(int size)` throw `IllegalStateException` are provided to manage the maximum queue size. The method `public int getQueueSize()` is provided to query the actual queue size.
- The methods `public int getRetryInterval()` and `public void setRetryInterval(int interval)` support the notion of error retry, which again implies some type of queuing.
- The methods `public void addFormatter(RASIFormatter formatter)`, `public void removeFormatter(RASIFormatter formatter)` and `public Enumeration getFormatters()` are provided to manage the list of formatters that the handler can be configured with. Different formatters can be provided for different event classes, if appropriate.

- The methods `public void openDevice()`, `public void closeDevice()` and `public void stop()` are provided to manage the underlying device that the handler abstracts.
- The methods `public void logEvent(RASIEvent event)` and `public void writeEvent(RASIEvent event)` are provided to actually pass events to the handler for processing.

Writing a formatter

User written formatters must implement the `RASIFormatter` interface. The `RASIFormatter` interface extends the `RASIObject` interface. The implementation of the `RASIObject` interface is the same for both handlers and formatters. A short discussion of the methods introduced by the `RASIFormatter` interface follows.

RASIFormatter interface

- The methods `public void setDefault(boolean flag)` and `public boolean isDefault()` are used by the concrete `RASHandler` classes provided by the standalone JRas logging toolkit to determine if a particular formatter is the default formatter. Since these `RASHandler` classes must never be used in a WebSphere Application Server - Express environment, the semantic significance of these methods can be determined by the user.
- The methods `public void addEventClass(String name)`, `public void removeEventClass(String name)` and `public Enumeration getEventClasses()` are provided to determine which event classes a formatter can be used to format. You can provide the appropriate implementations as you see fit.
- The method `public String format(RASIEvent event)` is called by handler objects and returns a formatted String representation of the event.

Example: user written handler: The following is a very simple sample of a Handler class that writes formatted events to a file. This class is functional, but is intended solely to demonstrate concepts. For simplicity and clarity, much code (including appropriate boundary condition checking logic) has been ignored. This sample is not intended to be an example of good programming practice.

```
package com.ibm.ws.ras.test.user;

import com.ibm.ras.*;
import java.io.*;
import java.util.*;
import java.security.AccessController;
import java.security.PrivilegedAction;
import java.security.PrivilegedExceptionAction;
import java.security.PrivilegedActionException;

/**
 * The <code>SimpleFileHandler</code> is a class that
 * implements the {@link RASHandler} interface. It is a simple
 * Handler that writes to a file. The name of the file
 * must be specified in the constructor.
 * <p>
 * If the file includes a path, the path separator
 * may be a front-slash ('/') or the
 * platform-specific path separator character. For example:
 *
 * /Dir1/Dir2/Dir3/MyStuff.log
 *
 */
public class SimpleFileHandler implements RASHandler
{
    /**
     * A public boolean that can be inspected by the caller to determine
     * if an error has occurred during an operation.
     * This boolean can only be changed when the device synchronizer is held.
     */
    public boolean errorHasOccurred = false;
```

```

/**
 * The name of the Handler
 */
private String ivName = "";

/**
 * The message mask which determines the types of messages
 * that will be processed.
 */
private long ivMessageMask;

/**
 * The trace mask which determines the types of trace points
 * that will be processed.
 */
private long ivTraceMask;

/**
 * The names of the message event classes which this object processes.
 */
private Vector ivMessageEventClasses;

/**
 * The names of the trace event classes which this object processes.
 */
private Vector ivTraceEventClasses;

/**
 * The set of {@link RASIMaskChangeListener} which want to be informed of changes to the
 * <code>RASIMaskChangeGenerator</code> message
 * or trace mask configuration.
 */
private Vector ivMaskChangeListeners;

/**
 * The fully-qualified, normalized name of the file to which the log entries are written.
 */
private String ivFqFileName;

/**
 * A boolean flag which indicates whether the device to which this handler sends log
 * entries is open. It is set to true when the device is open and false otherwise.
 */
private boolean ivDeviceOpen = false;

/**
 * A Hashtable of RASIFormatters keyed by the name of the event class they format.
 * Each event type can have exactly
 * one formatter. Different event classes can have different formatters.
 */
private Hashtable ivFormatters;

/**
 * An object on which the {@link #closeDevice closeDevice} and
 * {@link #writeEvent writeEvent} methods can synchronize.
 */
private Object ivDeviceLock = new Object();

/**
 * The stream to which formatted log events are written. This stream will wrap a file.
 */
private PrintWriter ivWriter = null;

/**
 * Create a SimpleFileHandler.

```

```

* <p>
* The constructor will attempt to open a stream in append mode over the
* specified file. If the operation does not complete
* successfully, the errorHasOccurred boolean is set to true. If no exceptions are
* thrown by this constructor and the
* errorHasOccurred booleans state is false, the stream is open
* and the handler is usable.
* <p>
* @param name the name assigned to this handler object. Null is tolerated.
* @param fileName a non-null file name. Caller must guarantee this name is not null.
* A fully qualified file name is preferred.
*/
public SimpleFileHandler(String name, String fileName) throws Exception {
    setName(name);
    ivMessageMask = RASIMessageEvent.DEFAULT_MESSAGE_MASK;
    ivTraceMask = RASITraceEvent.DEFAULT_TRACE_MASK;
    // Allocate the Hashtables and Vectors required.
    ivMaskChangeListeners = new Vector();
    ivMessageEventClasses = new Vector();
    ivTraceEventClasses = new Vector();
    ivFormatters = new Hashtable();
    // Add the default event classes that this handler will process
    addMessageEventClass("com.ibm.ras.RASMessageEvent");
    addTraceEventClass("com.ibm.ras.RASTraceEvent");

    // Get the fully qualified, normalized file name. Open the stream
    File x = new File(fileName);
    ivFqFileName = x.getAbsolutePath();
    openDevice();
}

////////////////////////////////////
//
// The following methods are required by the RASIOobject interface
//
////////////////////////////////////

/**
 * Return this objects configuration as a set of Properties in a Hashtable.
 * <p>
 * This handler does not support properties-based configuration.
 * Therefore a call to this method always returns null
 * @return null is always returned.
 */
public Hashtable getConfig() {
    return null;
}

/**
 * Set this objects configuration from the properties in the specified Hashtable.
 * <p>
 * This handler does not support properties-based configuration.
 * This method is a no-operation.
 * @param hashTable a Hashtable containing the properties. Input is ignored.
 */
public void setConfig(Hashtable ht) {
    return;
}

/**
 * Return the name by which this object is known.
 * <p>
 * @return a String containing the name of this object, or an
 * empty string ("") if the name has not been set.
 */
public String getName() {
    return ivName;
}

```

```

}

/**
 * Set the name by which this object is known.
 * If the specified name is <code>null</code>, the current name is not changed.
 * <p>
 * @param name The new name for this object. Null is tolerated.
 */
public void setName(String name) {
    if (name != null)
        ivName = name;
}

/**
 * Return the description field of this object.
 * <p>
 * This handler does not use a description field.
 * An empty String is always returned.
 * <p>
 * @return an empty String.
 */
public String getDescription() {
    return "";
}

/**
 * Set the description field for this object.
 * <p>
 * This handler does not use a description field.
 * Input is ignored and this method does nothing.
 * <p>
 * @param desc The description of this object. Input is ignored.
 */
public void setDescription(String desc) {
    return;
}

/**
 * Return the name of the {@link com.ibm.ras.mgr.RASManager RASManager} group
 * with which this object is associated. This method is only used by the RAS Manager.
 * <p>
 * This object does not support RASManager configuration. Null is always returned.
 * @return null is always returned.
 */
public String getGroup() {
    return null;
}

////////////////////////////////////
//
// Methods required by the RASIMaskChangeGenerator interface
//
////////////////////////////////////

/**
 * Return the message mask which defines the set of message
 * types that will be processed by this Handler. The set of possible
 * types is identified in the {@link RASIMessageEvent}
 * <code>TYPE_XXXX</code> constants.
 * <p>
 * @return The current message mask.
 */
public long getMessageMask() {
    return ivMessageMask;
}

/**

```

```

* Set the message mask which defines the set of message types
* that will be processed by this Handler. The set of possible
* types is identified in the {@link RASIMessageEvent}
* <code>TYPE_XXXX</code> constants.
* The mask value is not validated against these types.
* <p>
* @param mask The message mask.
*/
public void setMessageMask(long mask) {
    RASMaskChangeEvent mc = new RASMaskChangeEvent(this, ivMessageMask, mask, true);
    ivMessageMask = mask;
    fireMaskChangedEvent(mc);
}

/**
* Return the trace mask which defines the set of trace types that will be
* processed by this Handler. The set of possible
* types is identified in the {@link RASITraceEvent}
* <code>TYPE_XXXX</code> constants.
* <p>
* @return The current trace mask.
*/
public long getTraceMask() {
    return ivTraceMask;
}

/**
* Set the trace mask which defines the set of trace types that will
* be processed by this Handler. The set of possible types
* is identified in the {@link RASITraceEvent}
* <code>TYPE_XXXX</code> constants. The specified trace mask value is not validated.
* <p>
* @param mask The trace mask.
*/
public void setTraceMask(long mask) {
    RASMaskChangeEvent mc = new RASMaskChangeEvent(this, ivTraceMask, mask, false);
    ivTraceMask = mask;
    fireMaskChangedEvent(mc);
}

/**
* Add a {@link RASIMaskChangeListener} object to the set of listeners
* which wish to be identified of a change in the message
* or trace mask configuration. If the specified listener is
* <code>null</code> or is already registered, this method does nothing.
* <p>
* @param listener The mask change listener.
*/
public void addMaskChangeListener(RASIMaskChangeListener listener) {
    if (listener != null && (!ivMaskChangeListeners.contains(listener)))
        ivMaskChangeListeners.addElement(listener);
}

/**
* Remove a {@link RASIMaskChangeListener} object from
* the list of registered listeners that wish to be informed of changes
* in the message or trace mask configuration. If the listener is
* <code>null</code> or is not registered, this method does nothing.
* <p>
* @param listener The mask change listener.
*/
public void removeMaskChangeListener(RASIMaskChangeListener listener) {
    if (listener != null && ivMaskChangeListeners.contains(listener))
        ivMaskChangeListeners.removeElement(listener);
}

/**

```

```

* Return an enumeration over the set of listeners currently registered
* to be informed of changes in the message or trace mask configuration.
* <p>
* @return An Enumeration of mask change listeners. If no listeners
* are registered, the Enumeration is empty.
*/
public Enumeration getMaskChangeListeners() {
    return ivMaskChangeListeners.elements();
}

/**
* Inform all registered <code>RASIMaskChangeListener</code>s
* that the message or trace mask has been changed.
*<p>
* @param mc A mask change event, indicating what has changed.
*/
public void fireMaskChangedEvent(RASMaskChangeEvent mc) {
    RASIMaskChangeListener c;
    Enumeration e = getMaskChangeListeners();
    while (e.hasMoreElements()) {
        c = (RASIMaskChangeListener) e.nextElement();
        c.maskValueChanged(mc);
    }
}

/**
* Add the name of a message event class to the list of message
* event classes which this handler processes. If the specified
* event class is null or is already registered, this method does nothing.
* <p>
* @param name The event class name.
*/
public void addMessageEventClass(String name) {
    if (name != null && (! ivMessageEventClasses.contains(name)))
        ivMessageEventClasses.addElement(name);
}

/**
* Remove the name of a message event class from the list of names
* of classes which this handler processes. If the specified event
* class is null or is not registered, this method does nothing.
* <p>
* @param name The event class name.
*/
public void removeMessageEventClass(String name) {
    if ((name != null) && (ivMessageEventClasses.contains(name)))
        ivMessageEventClasses.removeElement(name);
}

/**
* Return an enumeration over the set of names of the message event
* classes which this handler processes.
* <p>
* @return An Enumeration of RAS event class names.
* If no event classes are registered, the Enumeration is empty.
*/
public Enumeration getMessageEventClasses() {
    return ivMessageEventClasses.elements();
}

/**
* Add the name of a trace event class to the list of trace event classes
* which this handler processes. If the specified event
* class is null or is already registered, this method does nothing.
* <p>
* @param name The event class name.
*/

```

```

public void addTraceEventClass(String name) {
    if ((name != null) && (!ivTraceEventClasses.contains(name)))
        ivTraceEventClasses.addElement(name);
}

/**
 * Remove the name of a trace event class from the list of names
 * of classes which this handler processes. If the
 * specified event class is null or is not registered, this method does nothing.
 * <p>
 * @param name The event class name.
 */
public void removeTraceEventClass(String name) {
    if ((name != null) && (ivTraceEventClasses.contains(name)))
        ivTraceEventClasses.removeElement(name);
}

/**
 * Return an enumeration over the set of names of the trace
 * event classes which this handler processes
 * <p>
 * @return An Enumeration of RAS event class names.
 * If no event classes are registered, the Enumeration is empty.
 */
public Enumeration getTraceEventClasses() {
    return ivTraceEventClasses.elements();
}

////////////////////////////////////
//
// Methods required by the RASHandler interface
//
////////////////////////////////////

/**
 * Return the maximum number of {@link RASIEvent RASIEvents}
 * which this handler will queue before writing.
 * <p>
 * In the WebSphere Application Server environment, handlers
 * may not start threads. All writes will be done
 * synchronously and never queued. This handler does not queue
 * events for later retry if a write operation fails.
 * <p>
 * @return zero is always returned.
 */
public int getMaximumQueueSize() {
    return 0;
}

/**
 * Set the maximum number of {@link RASIEvent RASIEvents}
 * which the handler will queue before writing.
 * <p>
 * This handler does not queue events. This method is a no-operation
 * <p>
 * @param size The maximum queue size. Input is ignored.
 */
public void setMaximumQueueSize(int size) throws IllegalStateException {
    return;
}

/**
 * Return the amount of time (in milliseconds) that this
 * handler will wait before retrying a failed write
 * <p>
 * This handler does not retry or queue failed writes. If a write operation
 * fails, the event is simply discarded.

```

```

* <p>
* @return The retry interval. Zero is always returned.
*/
public int getRetryInterval() {
    return 0;
}

/**
* Set the amount of time (in milliseconds) that this handler will
* wait before retrying a failed write.
* <p>
* This handler does not queue or retry failed writes. This method is a no-operation.
* <p>
* @param interval The retry interval. Input is ignored.
*/
public void setRetryInterval(int interval) {
    return;
}

/**
* Return the current number of {@link RASIEvent RASIEvents} in the handler's queue.
* <p>
* This handler does not queue events. Zero is always returned.
* <p>
* @return The current queue size. Zero is always returned.
*/
public int getQueueSize() {
    return 0;
}

/**
* Add a RASIFormatter to the set of formatters which are currently
* registered to this handler. The specified formatter
* must be fully configured. Specifically, the formatter must be configured
* with the set of {@link RASIEvent} classes which it
* knows how to format.
* <p>
* @param formatter The event formatter. Null is tolerated. If the specified
* formatter supports formatting an event class which
* already has an associated formatter, the existing formatter
* is replaced with this one.
**/
public void addFormatter(RASIFormatter formatter) {
    if (formatter != null) {
        Enumeration e = formatter.getEventClasses();
        while (e.hasMoreElements()) {
            String name = (String) e.nextElement();
            ivFormatters.put(name, formatter);
        }
    }
}

/**
* Remove a RASIFormatter from the set of formatters currently
* registered with this handler.
* <p>
* @param formatter The event formatter. If the specified formatter is
* null or is not registered, this method does nothing.
*/
public void removeFormatter(RASIFormatter formatter) {
    if (formatter != null) {
        Enumeration e = formatter.getEventClasses();
        while (e.hasMoreElements()) {
            String name = (String) e.nextElement();
            ivFormatters.remove(name);
        }
    }
}

```

```

}

/**
 * Return an enumeration over the set of RASIFormatters currently
 * registered with this handler.
 * <p>
 * @return An Enumeration over the set of registered formatters. If no formatters are
 * currently registered, the Enumeration is empty.
 */
public Enumeration getFormatters() {
    return ivFormatters.elements();
}

/**
 * Close the stream to which this handler is currently writing its entries,
 * if the stream is currently open.
 */
public void closeDevice() {
    synchronized(ivDeviceLock) {
        if (ivWriter == null)
            return;
        ivWriter.flush();
        ivWriter.close();
        ivWriter = null;
    }
}

/**
 * Stop the handler, closing the stream to which this handler is
 * currently writing its entries
 * <p>
 * This method must be called when a handler is no longer needed.
 * Be careful not to call
 * this method if other loggers may still be using this handler.
 */
public void stop() {
    // This handler does not have any queues to flush or preprocessing to do.
    // Simply call closeDevice().
    closeDevice();
}

/**
 * Asynchronously process a RAS event passed from a logger to this handler.
 * <p>
 * WebSphere Application Server loggers always operate synchronously.
 * It is expected that no events will be delivered via this method. This
 * handler also only supports synchronous operations. If events are
 * delivered via this method, simply process them synchronously
 * <p>
 * @param event A RAS event. Null is tolerated
 */
public void logEvent(RASIEvent event) {
    writeEvent(event);
}

/**
 * Synchronously process a RAS event passed from a logger to this handler.
 * <p>
 * WebSphere Application Server loggers always operate synchronously.
 * It is expected that all
 * events will be delivered via this method. This handler also only supports
 * synchronous operations.
 * <p>
 * @param event A RAS event. Null is tolerated
 */
public void writeEvent(RASIEvent event) {
    if (event == null)

```

```

        return;
    synchronized(ivDeviceLock) {
        if (ivWriter == null)
            return;
        RASIFormatter formatter = findFormatter(event);
        if (formatter != null) {
            String msg = formatter.format(event);
            ivWriter.println(msg);
            // If an error occurs, simply set the boolean that caller can check
            if (ivWriter.checkError())
                errorHasOccurred = true;
        }
    }
}

////////////////////////////////////
//
// Methods introduced by this implementation
//
////////////////////////////////////

/**
 * Return the fully-qualified, normalized name of the file which this handler is
 * currently configured to write events to.
 * <p>
 * @return The fully-qualified, normalized name of the output file.
 */
public String getFileName() {
    return ivFqFileName;
}

/**
 * Set this handler to write to a file other than the file it is currently writing to.
 * <p>
 * The current stream that the handler is writing to is closed.
 * A new stream is opened over the specified file.
 * <p>
 * @param name name of the file. May not be null. A fully-qualified file
 * name is recommended.
 * @exception An exception is thrown if the specified name is null, the file
 * cannot be created or some other error
 * occurs. If an exception is thrown, the handlers state is indeterminate.
 */
public void setFileName(String name) throws Exception {
    if (name == null)
        throw new Exception("Null passed for name");
    synchronized(ivDeviceLock) {
        closeDevice();
        File x = new File(name);
        ivFqFileName = x.getAbsolutePath();
        openDevice();
    }
}

/**
 * Open a stream over the file to which this handler will write formatted log entries.
 * The stream will always be opened in append mode.
 * <p>
 * If a stream is already open over the file, the current stream is closed.
 * If an error occurs during this operation, the errorHasOccurred boolean is set to true
 * and a plain text error message is written to System.err along with the exception
 * stack trace, if any. If the operation is successful, the errorHasOccurred boolean is
 * set to false.
 * <p>
 */
public void openDevice() {
    synchronized(ivDeviceLock) {

```

```

    try {
        closeDevice();
        errorHasOccurred = false;
        // The file name may have been changed. Create the directory for the file
        // if it doesn't already exist.
        File x = new File(ivFqFileName);
        String dir = x.getParent();
        File dirs = new File(dir);
        if (fileExists(dirs) == false) {
            boolean result = makeDirectories(dirs);
            if (result == false) {
                errorHasOccurred = true;
                return;
            }
        }
        // Open a file output stream over the file in append mode.
// Wrap the FileOutputStream in an
// OutputStreamWriter. Finally wrap the OutputStreamWriter in a
// BufferedPrintWriter with line flushing enabled.
        FileOutputStream fos = createFileOutputStream(ivFqFileName, true);
        OutputStreamWriter osw = new OutputStreamWriter(fos);
        ivWriter = new PrintWriter(new BufferedWriter(osw), true);
    }
    catch (Throwable t) {
        // not much we can do here except set the error boolean.
        errorHasOccurred = true;
        System.err.println("Error occurred in openDevice() for handler "+ivName);
        t.printStackTrace();
    }
}
}

/**
 * Return a reference to the formatter associated with the specified
 * event class. If the specified event class is not
 * registered, the superclasses of the event class will be checked
 * for a registered formatter.
 * <p>
 * @param event A RAS event. Must not be null.
 * @return formatter The formatter associated with the specified event class.
 * Null is returned if the event class is not registered.
 */
private RASIFormatter findFormatter(RASIEvent event) {
    Class eventClass = event.getClass();
    RASIFormatter formatter = null;

    while (eventClass != null) {
        String className = eventClass.getName();
        if (ivFormatters.containsKey(className)) {
            return (RASIFormatter) ivFormatters.get(className);
        }
        else
            eventClass = eventClass.getSuperclass();
    }
    return null;
}

/**
 * A worker method that wraps the creation of a
 * FileOutputStream in a doPrivileged block.
 * <p>
 * @param fileName the name of the file to create the stream over.
 * @param append a boolean, when true indicates the file should be opened in append
 * mode
 * @ return the FileOutputStream.
 * @exception SecurityException A security violation has occurred.
 * This class is not authorized to access the specified file.

```

```

* @exception PrivilegedActionException a checked exception was
* thrown in the course of running the privileged action.
* The checked exception is contained within the
* PrivilegedActionException. Most likely the wrapped exception is a FileNotFoundException.
*/
private FileOutputStream createFileOutputStream(String fileName, boolean append) throws
PrivilegedActionException
{
    final String tempFileName = fileName;
    final boolean tempAppend = append;
    FileOutputStream fs = (FileOutputStream) AccessController.doPrivileged(
        new PrivilegedExceptionAction() {
            public Object run() throws IOException {
                return new FileOutputStream(tempFileName, tempAppend);
            }
        }
    );
    return fs;
}

/**
* A worker method that wraps the check for the existence of a file in a
* doPrivileged block.
* <p>
* @param fileToCheck a <code>File</code> object whose abstract pathname corresponds
* to the physical file whose existence is to be checked.
* @return true if and only if the file exists. Otherwise false.
* @exception SecurityException A security violation has occurred.
* This class is not authorized to access the specified file.
*/
private boolean fileExists(File fileToCheck) throws SecurityException
{
    final File tempFileToCheck = fileToCheck;
    Boolean exists = (Boolean) AccessController.doPrivileged(
        new PrivilegedAction() {
            public Object run() {
                return new Boolean(tempFileToCheck.exists());
            }
        }
    );
    return exists.booleanValue();
}

/**
* A worker method that wraps the creation of directories in a doPrivileged block.
* <p>
* @param dirToMake a non-null <code>File</code> object whose
* abstract pathname represents
* the fully qualified directory to create.
* @return true is returned if and only if all necessary directories were created.
* false is returned.
* @exception SecurityException A security violation has occurred. This class
* Otherwise is not authorized
* to access at least one of the specified directories.
*/
private boolean makeDirectories(File dirToMake) throws SecurityException
{
    final File tempDirToMake = dirToMake;
    Boolean result = (Boolean) AccessController.doPrivileged(
        new PrivilegedAction() {
            public Object run() {
                return new Boolean(tempDirToMake.mkdirs());
            }
        }
    );
};

```

```

    return result.booleanValue();
}
}

```

Example: user written formatter: The following is a very simple sample of a Formatter class. This class is functional, but is intended solely to demonstrate concepts. For simplicity and clarity, much code (including appropriate boundary condition checking logic) has been ignored. This sample is not intended to be an example of good programming practice.

```

package com.ibm.ws.ras.test.user;
import com.ibm.ras.*;
import java.text.*;
import java.util.*;

/**
 * The <code>SimpleFormatter</code> implements
 * the RASIFormatter interface. It is a
 * simple implementation used for demonstration purposes only. It does not do any
 * advanced formatting, it simply formats the message and parameters in an event.
 * It does not include the timestamp in the formatted result, for example.
 */
public class SimpleFormatter implements RASIFormatter
{

    /**
     * The name of the formatter
     */
    private String ivName = "";

    /**
     * A vector containing the event classes this Formatter knows how to process.
     */
    private Vector ivEventClasses = new Vector();

    /**
     * Create a <code>SimpleFormatter</code>.
     */
    public SimpleFormatter(String name) {
        setName(name);
    }

    ////////////////////////////////////////////////////////////////////
    //
    // Methods required by the RASIObjct Interface
    //
    ////////////////////////////////////////////////////////////////////

    /**
     * Return this objects configuration as a set of Properties in a Hashtable.
     * <p>
     * This formatter does not support properties-based configuration.
     * Therefore a call to this method always returns null
     * @return null is always returned.
     */
    public Hashtable getConfig() {
        return null;
    }

    /**
     * Set this objects configuration from the properties in the specified Hashtable.
     * <p>
     * This formatter does not support properties-based configuration.
     * This method is a no-operation.
     * @param hashTable a Hashtable containing the properties. Input is ignored.
     */
    public void setConfig(Hashtable ht) {

```

```

    return;
}

/**
 * Return the name by which this formatter is known.
 * <p>
 * @return a String containing the name of this object, or an
 * empty string ("") if the name has not been set.
 */
public String getName() {
    return ivName;
}

/**
 * Set the name by which this formatter is known. If the specified
 * name is <code>null</code>, the current name is not changed.
 * <p>
 * @param name The new name for this object. Null is tolerated.
 */
public void setName(String name) {
    if (name != null)
        ivName = name;
}

/**
 * Return the description field of this formatter.
 * <p>
 * This formatter does not use a description field.
 * An empty String is always returned.
 * <p>
 * @return an empty String.
 */
public String getDescription() {
    return "";
}

/**
 * Set the description field for this formatter.
 * <p>
 * This formatter does not use a description field. Input is ignored and
 * this method does nothing.
 * <p>
 * @param desc The description of this object. Input is ignored.
 */
public void setDescription(String desc) {
    return;
}

/**
 * Return the name of the {@link com.ibm.ras.mgr.RASManager RASManager} group
 * with which this formatter is associated. This method is
 * only used by the RAS Manager.
 * <p>
 * This formatter does not support RASManager configuration. Null is always returned.
 * @return null is always returned.
 */
public String getGroup() {
    return null;
}

////////////////////////////////////
//
// Methods required by the RASIFormatter Interface
//
////////////////////////////////////

/**

```

```

* Set a flag that indicates whether this formatter is the default formatter used by
* {@link com.ibm.ras.RASHandler} objects to format events.
*<p>
* Instances of com.ibm.ras.RASHandler are not allowed to be
* instantiated in the WebSphere Application Server environment.
* This formatter cannot be the default formatter for handlers of this type.
* This method does nothing.
*<p>
* @param flag input is ignored, since this formatter cannot be the default formatter.
*/
public void setDefault(boolean flag) {
    return;
}

/**
* Return a boolean that indicates whether or not this is the default formatter
* used by a {@link com.ibm.ras.RASHandler} to format the RAS events.
*<p>
* com.ibm.ras.RASHandlers will never be instantiated in a
* WebSphere Application Server environment so this method always returns false.
*<p>
* @return false is always returned.
*/
public boolean isDefault() {
    return false;
}

/**
* Add the name of a {@link com.ibm.ras.RASIEvent} class to the list
* of classes which this formatter can process.
* If the specified class name is null or it is already registered, this method does nothing.
*<p>
* @param name The event class name. Null is tolerated.
*/
public void addEventClass(String name) {
    if ((name != null) && (! ivEventClasses.contains(name)))
        ivEventClasses.addElement(name);
}

/**
* Remove the name of a {@link com.ibm.ras.RASIEvent} class
* from the list of classes which this formatter
* can process. If the specified class name is null or is not
* registered, this method does nothing.
*<p>
* @param name The event class name.
*/
public void removeEventClass(String name) {
    if ((name != null) && (ivEventClasses.contains(name)))
        ivEventClasses.removeElement(name);
}

/**
* Return an enumeration over the set of names of
* {@link com.ibm.ras.RASIEvent} classes which this formatter can process.
*<p>
* @return An enumeration of RAS event class names. If no event classes
* are registered, the enumeration is empty.
*/
public Enumeration getEventClasses() {
    return ivEventClasses.elements();
}

/**
* Format the specified {@link com.ibm.ras.RASIEvent} object and
* return a String containing the formatted output.
*<p>

```

```

* @param event The event to format. Null is tolerated.
* @return The formatted event contents. Null may be returned.
*/
public String format(RASIEvent event) {
    if (event == null)
        return null;
    if (event.isMessageEvent())
        return formatMessage(event);
    else
        return formatTrace(event);
}

/**
* Format a message event.
*<p>
* If a message key is used and that key is not found in any
* message file, the message text becomes an error message
* indicating that the key was not found.
*<p>
* @param event The event to format.
* @return The formatted event.
*/
public String formatMessage(RASIEvent event) {
    String messageKey = "null";
    try {
        messageKey = event.getText();
        String[] messageInserts = event.getParameters();
        if (event instanceof com.ibm.ras.RASMessageEvent) {
            // RASMessageEvents usually contain localizable messages.
            RASMessageEvent rme = (RASMessageEvent)event;
            String bundleName = rme.getMessageFile();
            if (bundleName != null) {
                // Not a text message, get localized message and return
                ResourceBundle bundle =
                    ResourceBundle.getBundle(bundleName, Locale.getDefault());
                String localizedKey = bundle.getString(messageKey);
                return MessageFormat.format(localizedKey, messageInserts);
            }
            else {
                // Text message
                for (int i=0; i<messageInserts.length; ++i){messageKey =
                    messageKey + " " + messageInserts[i];
                }
                return messageKey;
            }
        }
        else {
            // A User defined type. Append paramaters to key and return
            for (int i=0; i<messageInserts.length; ++i){
                messageKey = messageKey + " " + messageInserts[i];
            }
            return messageKey;
        }
    }
    catch (Throwable t) {
        t.printStackTrace();
        return "SimpleFormattter: Error while formatting message "+messageKey;
    }
}

/**
* Format a trace event.
*<p>
* Append the parameters (in order of specification) to the text
* message in the trace event object.
*<p>
* @param event The event to format.

```

```

    * @return The formatted event.
    */
private String formatTrace(RASIEvent event) {
    String text = "null";
    try {
        text = event.getText();
        String[] parms = event.getParameters();
        if (parms != null) {
            for (int i=0; i<parms.length; ++i){
                text = text + " " + parms[i];
            }
        }
        return text;
    }
    catch (Throwable t) {
        t.printStackTrace();
        return "SimpleFormatter: Error while formatting trace "+text;
    }
}
}

```

JRas messages and trace event types

This topic describes JRas message and trace event types.

Event types

The base message and trace event types defined by the standalone JRas logging toolkit are not the same as the “native” types recognized by the WebSphere Application Server - Express runtime. Instead the basic JRas types are mapped onto the native types. This mapping may vary by platform or edition. The mapping is discussed below.

Platform Message Event Types

The message event types that are recognized and processed by the WebSphere Application Server - Express runtime are defined in the RASIMessageEvent interface provided by the standalone JRas logging toolkit. These message types are mapped onto the native message types as follows.

| WebSphere Application Server - Express native type | JRas RASIMessageEvent type |
|--|-----------------------------|
| Audit | TYPE_INFO, TYPE_INFORMATION |
| Warning | TYPE_WARN, TYPE_WARNING |
| Error | TYPE_ERR, TYPE_ERROR |

Platform Trace Event Types

The trace event types recognized and processed by the WebSphere Application Server - Express runtime are defined in the RASITraceEvent interface provided by the standalone JRas logging toolkit. The RASITraceEvent interface provides a rich and overly complex set of types. This interface defines both a simple set of levels, as well as a set of enumerated types.

- For a user who prefers a simple set of levels, RASITraceEvent provides TYPE_LEVEL1, TYPE_LEVEL2, and TYPE_LEVEL3. The implementations provide support for this set of levels. The levels are hierarchical (that is, enabling level 2 will also enable level 1, enabling level 3 also enables levels 1 and 2).
- For users who prefer a more complex set of values that can be OR'd together, RASITraceEvent provides TYPE_API, TYPE_CALLBACK, TYPE_ENTRY_EXIT, TYPE_ERROR_EXC, TYPE_MISC_DATA, TYPE_OBJ_CREATE, TYPE_OBJ_DELETE, TYPE_PRIVATE, TYPE_PUBLIC, TYPE_STATIC, and TYPE_SVC.

The trace event types are mapped onto the native trace types. Mapping WebSphere Application Server - Express trace types to JRas RASITraceEvent "Level" types is as follows:

| WebSphere Application Server - Express native type | JRas RASITraceEvent level type |
|--|--------------------------------|
| Event | TYPE_LEVEL1 |
| EntryExit | TYPE_LEVEL2 |
| Debug | TYPE_LEVEL3 |

Mapping WebSphere Application Server - Express trace types to JRas RASITraceEvent enumerated types is as follows:

| WebSphere Application Server native type | JRas RASITraceEvent enumerated types |
|--|--|
| Event | TYPE_ERROR_EXC, TYPE_SVC, TYPE_OBJ_CREATE, TYPE_OBJ_DELETE |
| EntryExit | TYPE_ENTRY_EXIT, TYPE_API, TYPE_CALLBACK, TYPE_PRIVATE, TYPE_PUBLIC, TYPE_STATIC |
| Debug | TYPE_MISC_DATA |

For simplicity, it is recommended that one or the other of the tracing type methodologies is used consistently throughout the application. For users who decide to use the non-level types, it is further recommended that you choose one type from each category and use those consistently throughout the application to avoid confusion.

Message and Trace parameters

The various message logging and trace method signatures accept parameter types of Object, Object[] and Throwable. WebSphere Application Server - Express processes and formats the various parameter types as follows.

- **Primitives**

Primitives, such as int and long are not recognized as subclasses of Object and cannot be directly passed to one of these methods. A primitive value must be transformed to a proper Object type (Integer, Long) before being passed as a parameter.

- **Object**

toString() is called on the object and the resulting String is displayed. The toString() method should be implemented appropriately for any object passed to a message logging or trace method. It is the responsibility of the caller to guarantee that the toString() method does not display confidential data such as passwords in clear text, and does not cause infinite recursion.

- **Object[]**

The Object[] is provided for the case when more than one parameter is passed to a message logging or trace method. toString() is called on each Object in the array. Nested arrays are not handled. (i.e. none of the elements in the Object array should be an array).

- **Throwable**

The stack trace of the Throwable is retrieved and displayed.

- **Array of Primitives**

An array of primitive (e.g. byte[], int[]) is recognized as an Object, but is treated somewhat as a second cousin of Object by Java. In general, arrays of primitives should be avoided, if possible. If arrays of primitives are passed, the results are indeterminate and may change depending on the type of array passed, the API used to pass the array and the release of the product. For consistent results, user code should preprocess and format the primitive array into some type of String form before passing it to the method. If such preprocessing is not performed, the following may result.

- [B@924586a0b - This is deciphered as “a byte array at location X”. This is typically returned when an array is passed as a member of an Object[]. It is the result of calling toString() on the byte[]. Illegal trace argument : array of long. This is typically returned when an array of primitives is passed to a method taking an Object.
- 01040703... : the hex representation of an array of bytes. Typically this may be seen when a byte array is passed to a method taking a single Object. This behavior is subject to change and should not be relied on. “1” “2” ... : The String representation of the members of an int[] formed by converting each element to an Integer and calling toString() on the Integers. This behavior is subject to change and should not be relied on.
- [Ljava.lang.Object;@9136fa0b : An array of objects. Typically this is seen when an array containing nested arrays is passed.

- **Controlling message logging**

Writing a message to a WebSphere Application Server - Express log requires that the message type passes three levels of filtering or screening.

1. The message event type must be one of the message event types defined in the RASIMessageEvent interface.
2. Logging of that message event type must be enabled by the state of the message logger’s mask.
3. The message event type must pass any filtering criteria established by the WebSphere Application Server - Express runtime itself.

When a WebSphere Application Server - Express logger is obtained from the Manager, the initial setting of the mask is to forward all native message event types to the WebSphere Application Server - Express handler. It is possible to control what messages get logged by programmatically setting the state of the message logger’s mask.

Some editions of the product allow the user to specify a message filter level for a server process. When such a filter level is set, only messages at the specified severity levels are written to WebSphere Application Server - Express logs. This means that messages types that pass the message logger’s mask check may be filtered out by the WebSphere Application Server - Express itself.

- **Controlling Tracing**

Each edition of the product provides a mechanism for enabling or disabling trace. The various editions may support static trace enablement (trace settings are specified before the server is started), dynamic trace enablement (trace settings for a running server process can be dynamically modified) or both.

Writing a trace record to a WebSphere Application Server - Express requires that the trace type passes three levels of filtering or screening.

1. The trace event type must be one of the trace event types defined in the RASITraceEvent interface.
2. Logging of that trace event type must be enabled by the state of the trace logger’s mask.
3. The trace event type must pass any filtering criteria established by the WebSphere Application Server runtime itself.

When a logger is obtained from the Manager, the initial setting of the mask is to suppress all trace types. The exception to this rule is the case where the WebSphere Application Server - Express runtime supports static trace enablement and a non-default startup trace state for that trace logger has been specified. Unlike message loggers, the WebSphere Application Server - Express may dynamically modify the state of a trace loggers trace mask. WebSphere Application Server - Express only modifies the portion of the trace logger’s mask corresponding to the values defined in the RASITraceEvent interface. WebSphere Application Server - Express does not modify undefined bits of the mask that may be in use for user defined types.

When the dynamic trace enablement feature available on some platforms is used, the trace state change is reflected both in the Application Server runtime and the trace loggers trace mask. If user code programmatically changes the bits in the trace mask corresponding to the values defined by in the RASITraceEvent interface, the trace logger’s mask state and the runtime state becomes unsynchronized and unexpected results occur. Therefore, programmatically changing the bits of the mask corresponding to the values defined in the RASITraceEvent interface is not allowed.

Step 4: Assemble your application

You must assemble application modules including Web application archives (WAR) and resource adapter archives (RAR) for your application components before you can install them. Application assembly is the process of creating archive files that bundle all of the components belonging to an application and configuring the run-time behavior of these applications.

Before assembling, gather all of the code you need to package into your assembled modules including:

- Servlets, JavaServer Pages (JSP) files and other Web components
- Resource adapter (connector) implementations
- Other supporting classes and files

To complete the assembly process, perform the following steps:

- Create one or more modules.
- Create a deployment descriptor for the module.
- Package the modules into an archive file.

To assemble your application, use the WebSphere Development Studio for iSeries. For more information, see the WebSphere Development Studio Client Help.

Step 5: Deploy your application

After you develop and assemble your application, you can deploy it in your application server. Application deployment involves installing the application files on the application server and configuring the application for the particular operational environment.

When you assemble the application, you must generate a deployment descriptor. A deployment descriptor is an XML file that contains instructions on how to deploy the application, without regard to the operational environment. When you deploy the application, you provide the specific information that is required for the application to run in your environment.

For example, when you develop the application, you can define security roles in the deployment descriptor. When you deploy the application into your application server run time, you map these security roles to specific users or groups that exist in your environment.

After you deploy an application, you may decide to change the application. For example, if you want to add a new Web module, you must assemble the application with the new module. After you assemble the changed application, use the HTTP Server Administration interface to install the changes into the run time. The changed version includes its own deployment descriptor, and may require that you specify additional configuration information. When you deploy the changed version, the HTTP Server Administration interface merges configuration information for both versions.

For information on how to deploy applications, see *Deploy and start a new application in the Administration* topic.

Appendix. Notices

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

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

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

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

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

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

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

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Software Interoperability Coordinator, Department YBWA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, IBM License Agreement for Machine Code, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

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

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

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

COPYRIGHT LICENSE:

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

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming Interface Information

This WebSphere Application Server - Express publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM i5/OS.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX
AIX 5L
e(logo)server
eServer
i5/OS
IBM
IBM (logo)
iSeries
pSeries
WebSphere
xSeries
zSeries

Intel, Intel Inside (logos), MMX, and Pentium are trademarks of Intel Corporation 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.

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

Linux is a trademark of Linus Torvalds 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.

Terms and conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

Personal Use: You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works of these publications, or any portion thereof, without the express consent of IBM.

Commercial Use: You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the p <?Pub Caret?>ublications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Code license and disclaimer information

IBM grants you a nonexclusive copyright license to use all programming code examples from which you can generate similar function tailored to your own specific needs.

SUBJECT TO ANY STATUTORY WARRANTIES WHICH CANNOT BE EXCLUDED, IBM, ITS PROGRAM DEVELOPERS AND SUPPLIERS MAKE NO WARRANTIES OR CONDITIONS EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT, REGARDING THE PROGRAM OR TECHNICAL SUPPORT, IF ANY.

UNDER NO CIRCUMSTANCES IS IBM, ITS PROGRAM DEVELOPERS OR SUPPLIERS LIABLE FOR ANY OF THE FOLLOWING, EVEN IF INFORMED OF THEIR POSSIBILITY:

1. LOSS OF, OR DAMAGE TO, DATA;
2. DIRECT, SPECIAL, INCIDENTAL, OR INDIRECT DAMAGES, OR FOR ANY ECONOMIC CONSEQUENTIAL DAMAGES; OR
3. LOST PROFITS, BUSINESS, REVENUE, GOODWILL, OR ANTICIPATED SAVINGS.

SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF DIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, SO SOME OR ALL OF THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.



Printed in USA