

Development -- table of contents

4: Developing applications

4.1: Programming model and environment

4.1.1: Finding supported specifications

4.1.1.1: Supported programming languages

4.1.1.2: Supported XML/XSL APIs and specifications

4.1.2: Tools for developing Web applications

4.1.2.1: IBM Distributed Debugger and Object Level Trace

4.1.2.2: Tips for using VisualAge for Java

4.1.2.3: Tips for using IBM WebSphere Studio

4.2: Building Web applications

4.2.1: Developing servlets

4.2.1.1: Servlet lifecycle

4.2.1.2: Servlet support and environment in WebSphere

4.2.1.2.1a: Features of Java Servlet API 2.2

4.2.1.2.2: IBM extensions to the Servlet API

4.2.1.2.3a: Invoking sevlets by classname and serving files

4.2.1.2.3b: Security risk example of invoking servlets by class name

4.2.1.3: Servlet content, examples, and samples

4.2.1.3.1: Creating HTTP servlets

Overriding HttpServlet methods

4.2.1.3.2: Inter-servlet communication

Example: Servlet communication by forwarding

4.2.1.3.4: Filtering and chaining servlets

Servlet filtering with MIME types

Servlet filtering with servlet chains

4.2.1.3.5: Enhancing servlet error reporting

Public methods of the ServletErrorReport class

4.2.1.3.6: Serving servlets by classname

4.2.1.3.7: Serving all files from application servers

4.2.1.3.8: Obtaining the Web application classpath from within a servlet

4.2.1.3.9: PageListServlet support

Extending PageListServlet

Configuring page lists using the Application Assembly Tool

Configuring page lists using an XML servlet configuration file

Example of the XML servlet configuration file

PageListServlet client type configuration file

Example of a client type configuration file

4.2.2: Developing JSP files

4.2.2.1: JavaServer Pages (JSP) lifecycle

4.2.2.1a: JSP access models

4.2.2.2: JSP support and environment in WebSphere

4.2.2.2.2: JSP processors

4.2.2.2.3: Java Server Page attributes

4.2.2.2.4: Batch compiling JSP files

Compiling JSP 1.1 files as a batch

4.2.2.3: Overview of JSP file content

4.2.2.3.2: JSP syntax: Class-wide variables and methods

4.2.2.3.3: JSP syntax: Inline Java code (scriptlets)

4.2.2.3.4: JSP syntax: Java expressions

4.2.2.3.5: JSP syntax: useBean tags

JSP syntax: <useBean> tag syntax

JSP syntax: Accessing bean properties

JSP syntax: Setting useBean properties

4.2.2.3.7: IBM extensions to JSP syntax

JSP syntax: Tags for variable data

JSP syntax: <tsx:getProperty> tag syntax and examples

JSP syntax: <tsx:repeat> tag syntax

JSP syntax: The repeat tag results set and the associated bean

JSP syntax: Tags for database access

JSP syntax: <tsx:dbconnect> tag syntax

JSP syntax: <tsx:userid> and <tsx:passwd> tag syntax

JSP syntax: <tsx:dbquery> tag syntax

Example: JSP syntax: <tsx:dbquery> tag syntax

JSP syntax: <tsx:dbmodify> tag syntax

Example: JSP syntax: <tsx:dbmodify> tag syntax

Example: JSP syntax: <tsx:repeat> and <tsx:getProperty> tags

4.2.2.3a: JSP examples

4.2.2.3a01: JSP code example - login

4.2.2.3a02: JSP code example - view employee records

4.2.2.3a03: JSP code example - EmployeeRepeatResults

4.2.3: Incorporating XML

4.2.3.2: Specifying XML document structure

4.2.3.3: Providing XML document content

4.2.3.4: Rendering XML documents

4.2.3.6: Using DOM to incorporate XML documents into applications

4.2.3.6.1: Quick reference to DOM object interfaces

4.2.3.7: SiteOutliner sample

4.2.4: Accessing data

4.2.4.2: Obtaining and using database connections

4.2.4.2.1: Accessing data with the JDBC 2.0 Optional Package APIs

Creating datasources with the WebSphere connection pooling API

Tips for using connection pooling

Handling data access exceptions

4.2.4.2.2: Accessing data with the JDBC 2.0 Core API

4.2.4.2.3: Accessing relational databases with the IBM data access beans

Example: Servlet using data access beans

4.2.4.2.4: Database access by servlets and JSP files

4.2.4.4.1: Providing Web clients a way to invoke JSP files

Invoking servlets and JSP files by URLs

Invoking servlets and JSP files within HTML forms

Example: Invoking servlets within HTML forms

4.2.4.4.2: Providing Web clients access to servlets

Invoking servlets within JSP files

4.2.5: Using the Bean Scripting Framework

4.2.5.1: BSF examples and samples

4.3: Developing enterprise beans

Writing Enterprise Beans

About this book

An introduction to enterprise beans

An architectural overview of the EJB programming environment

WebSphere Programming Model Extensions

More-advanced programming concepts for enterprise beans

Enabling transactions and security in enterprise beans

Developing enterprise beans

Developing EJB clients

Developing servlets that use enterprise beans

Tools for developing and deploying enterprise beans in the EJB server (AE) environment

Appendix A. Changes for version 1.1 of the EJB specification

Appendix B. Example code provided with WebSphere Application Server

Appendix D. Extensions to the EJB Specification

4.4: Personalizing applications

4.4.1: Tracking sessions

4.4.1.1: Session programming model and environment

4.4.1.1.1: Deciding between session tracking approaches

Using cookies to track sessions

Using URL rewriting to track sessions

Using SSL information to track sessions

4.4.1.1.2: Controlling write operations to persistent store

4.4.1.1.3: Securing sessions

4.4.1.1.4: Deciding between single-row and multirow schema for sessions

4.4.1.1.7: Tuning session support

Tuning session support: Session persistence

Tuning session support: Multirow schema

Tuning session support: Write frequency

Tuning session support: Base in-memory session pool size

Tuning session support: Write contents

Tuning session support: Scheduled invalidation

Tuning session support: Tablespace and page sizes

4.4.1.1.8: Best practices for session programming

4.4.2: Keeping user profiles

4.4.2.1: Data represented in the base user profile

4.4.2.2: Customizing the base user profile support

4.4.2.2.1: Extending data represented in user profiles

4.4.2.2.2: Adding columns to the base user profile implementation

4.4.2.2.3: Extending the User Profile enterprise bean and importing legacy databases

4.4.2.3: Accessing user profiles from a servlet

4.5: Dynamic fragment cache

4.5.0: Getting started with Dynamic fragment cache

4.5.1: Custom ID and MetaData generators

4.5.2: External caching

4.5.3: Dynamic fragment cache frequently asked questions

4.6: Java Technologies

4.6.1: Using JavaMail

4.6.1.1: Writing JavaMail applications

4.6.1.2: Configuring JavaMail

4.6.1.3: Debugging JavaMail

4.6.1.4: Running the JavaMail sample

4.6.2: JNDI (Java Naming and Directory Interface) overview

4.6.2.1: JNDI implementation in WebSphere Application Server

4.6.2.2: Using JNDI

4.6.2.3: JNDI caching

4.6.2.4: JNDI helpers and utilities

4.6.2.4.1: JNDI helper class

4.6.2.4.2: JNDI Name Space Dump utility

4.6.3: Java Message Service (JMS) overview

4.6.3.1: Using the JMS point-to-point messaging approach

4.6.3.2: Using the JMS publish/subscribe messaging approach

4.6.3.3: Support of Java Message Service resources

4.6.3.4: Support for the use of MQSeries Java Message Service resources

4.7: Java Clients

4.7.1: Applet client programming model

4.7.1.1: Developing an Applet client

4.7.2: J2EE application client programming model

4.7.2.1: Resources referenced by a J2EE application client

4.7.2.2: Developing a J2EE application client

4.7.2.3: Troubleshooting guide for the J2EE application client

4.7.2.4: J2EE application client classloading overview

4.7.3: Java thin application client programming model

4.7.3.1: Developing a Java application thin client

4.7.3.2: Java thin application client code example

4.7.4: Quick reference to Java client functions

4.7.5: Quick reference to Java client topics

4.7.6: Packaging and distributing Java client applications

4.7.7: Tracing and logging for the Java clients

4.8: Web services

4.8.1: Web services components

4.8.1.1: UDDI4J Overview

4.8.1.1.1: UDDI4J samples

4.8.1.2: SOAP support

4.8.1.2.1: SOAP samples

4.8.1.2.2: Building a SOAP client

Accessing enterprise beans through SOAP

4.8.1.2.3: Deploying a programming artifact as a SOAP accessible Web service

4.8.2: Apache SOAP deployment descriptors

4.8.2.1: SOAP deployment descriptors

4.8.3: Quick reference of Web services resources

4.8.4: Securing SOAP services

4.8.4.1: Running the security samples

4.8.4.2: SOAP signature components

4.8.4.2.1: Keystore files for testing purposes

4.8.4.2.2: Envelope Editor

4.8.4.2.3: Signature Header Handler

4.8.4.2.4: Verification Header Handler

4.10: Developing custom services

Samples

4: Developing applications

For IBM WebSphere Application Server, applications are combinations of building blocks that work together to perform a business logic function. Synonymous with *enterprise* applications, applications can contain enterprise beans, but do not have to. At most:

enterprise applications = enterprise beans + Web applications

Web applications are groups of one or more servlets, plus static content.

Web applications = servlets + JSP files + XML files + HTML files + graphics

The J2EE (Java™2 Platform Enterprise Edition) model introduces a number of new programming concepts including:

- [Thin clients](#)
- [WAR files](#)
- [EAR files](#)

Thin clients are remote clients that pass data for processing to an enterprise bean running on the application server. See article [Java clients](#) for more information.

The J2EE model packages enterprise and Web applications into the new categories of EAR files and WAR files.

- WAR files or Web Archive Resource files are combinations of servlets, JSP files, HTML files, graphics, and a Web [deployment descriptor](#). The file extension for these files is `.war`.
- EAR files or Enterprise Archive Resource files can consist of Web modules (`.war` files), EJB modules (`.jar` files), client modules (`.jar` files), and an application deployment descriptor. The file extension for these files is `.ear`

The [Application Assembly Tool](#) (AAT) creates the WAR, EAR, and JAR files, and assembles application components into Web modules.

View the supported specification levels for servlet, JSP, and EJB APIs at the [WebSphere Application Server prerequisites Web site](#).

See article 4.1 to review the WebSphere application programming model and environment, including information on various tools to help you develop and test your application components.

4.1: Programming model and environment

IBM WebSphere Application Server supports a three-tier programming model in which the application server and its contents -- your applications -- reside in the middle tier.

In this multi-tiered programming model, tier 0 represents Applets which run in a Web browser; tier 1, some application resources such as JSP files and servlets, which respond to HTTP requests; tier 2, the enterprise beans that run on the EJB server; and tier 3, the databases that store the business data. With version 4.0, WebSphere Application Server provides tier 0 support by shipping a "thin" remote client. See article [Java clients](#) for more information.

This documentation is geared towards the following layered approach to application development:

1. Determine what the application should do
2. Plan the application building blocks and their interactions
3. Create the Web application building blocks
4. Write the Web application deployment descriptor
5. Combine the Web application components and deployment descriptor into a Web module
6. Create the enterprise beans
7. Write the EJB deployment descriptor
8. Combine the enterprise beans and the deployment descriptor into an EJB module
9. Package the Web module and EJB module into a J2EE application.

A Web developer working in the J2EE environment is therefore responsible for the following tasks:

- Writing, compiling, and testing the source code
- Writing the JSP and HTML files
- Specifying the deployment descriptor
- Bundling the `servlet.class`, `.jsp`, `.html` and deployment descriptor files into a Web application archive or WAR file
- Bundling the `ejb.class` and deployment descriptor file into a JAR file
- Assembling the EJB JAR and WAR files into a J2EE application enterprise archive resource or EAR file

4.1.1: Finding supported APIs and specifications

Finding supported specification levels

See the [WebSphere Application Server prerequisites Web page](#) for the supported levels of specifications such as the Java Servlet and JavaServer Pages (JSP) specifications from Sun Microsystems.

Refer to the Sun Microsystems Web site for additional information about Java specifications:

<http://java.sun.com/products>

Finding API documentation (Javadoc) pertaining to IBM WebSphere Application Server

Access the Javadoc index for the packages included with IBM WebSphere Application Server (though not necessarily produced by IBM) from the fullInfoCenter:

[Index to API documentation \(Javadoc\)](#)

4.1.1.1: Supported programming languages

WebSphere Application Server is designed and tested to support applications and clients based on the **Java** programming language and technologies.

4.1.1.2: Supported XML/XSL APIs and specifications

IBM WebSphere Application Server provides document parsers, document validators, and document generators for server-side XML processing. The product supports the following XML-related recommendations:

- [W3C Extensible Markup Language \(XML\) 1.0](#)
- [W3C Namespaces in XML](#) (Recommendation January 14, 1999)
- [W3C Level 1 Document Object Model Specification \(DOM\) 1.0](#) (Recommendation October 1, 1998)
- [XSL Transformations Version 1.0](#)
- [XML Path Language Version 1.0](#)

IBM WebSphere Application Server supports the following XML/XSL APIs:

- XML4J Version 3.1 or Xerces Version 1.2.1
- LotusXSL Version 2.0 or Xalan Version 2.0.1

Distributions of XML4J and LotusXSL are shipped with Version 4.0 for immediate use. However, if you prefer to use implementations other than those shipped, you can easily override the default parser, because the order of class resolution has been reversed from that of previous versions of the product. Version 4.0 uses any parser classes specified in a module or application first; then the product uses the classes provided in the run-time environment.

Note: Support is offered only for the parser implementations that are shipped with the product.

XML parsing and validation support

The components of XML for Java provide support for parsing, validating, and generating XML data. The processor implements the base XML, namespace, and DOM W3C recommendations and SAX *de facto* standard. For more information, see the product Javadoc.

xml4j.jar and its open-source version, xerces.jar, can be found in the [product_installation_root](#)\lib directory.

To obtain updates and source code for XML4J and other XML-related resources, visit the IBM alphaWorks site at <http://alphaworks.ibm.com/>. To obtain updates and source code for Xerces, visit the Apache site at <http://xml.apache.org/>.

XSL processing support

This includes APIs for formatting and transforming XML documents at the server.

lotusxsl.jar and its open-source version, xalan.jar, can be found in the [product_installation_root](#)\lib directory.

To obtain updates and source code for LotusXSL, visit the IBM alphaWorks site at the URL provided previously. To obtain updates and source code for Xalan, visit the Apache site at the URL provided previously.

4.1.2: Tools for developing Web applications

When you install IBM WebSphere Application Server from the product CD, the installation program provides options to install IBM Distributed Debugger (DD) and Object Level Trace (OLT).

In addition, the following products can help you develop components for Web applications:

- IBM VisualAge for Java, Enterprise Edition
- IBM WebSphere Studio

These products are available separately.

4.1.2.1: IBM Distributed Debugger and Object Level Trace

The IBM Distributed Debugger (DD) enables you to detect and diagnose errors in your code. Its client/server design enables you to debug programs over a network connection. You can also debug programs running on your local workstation.

Object Level Trace (OLT), which works closely with the IBM Distributed Debugger, enables you to monitor the flow of a distributed application and debug code from a single workstation.

Tips for using OLT/DD

In order to trace and debug the application server, you must install the debugger on the machine on which the application server is running. For remote tracing and debugging, you must also install the debugger on the machine from which you plan to run the OLT tool and the debugger. For example, only remote debugging is supported on Solaris, so if your application server is running on Solaris, you must install the Solaris component of the debugger on that same machine. In addition, you must install OLT and the debugger on the AIX or Windows NT (or Windows 2000) machine from which you plan to run the tools remotely.

For the latest information about OLT/DD, see the [IBM Distributed Debugger](#) and [OLT](#) documentation.

4.1.2.2: IBM VisualAge for Java

VisualAge for Java Enterprise Edition provides the following tools for developing Web application components:

- EJB Development Environment - Enables you to design and package enterprise beans as well as database schemas to support persistent features.
- JSP Execution Monitor - Enables you to monitor the execution of JSP source code, generated servlets, and HTML source code as it is generated. This tool is available for Windows NT systems.
- Servlet Launcher - Enables you to start a Web server, open your Web browser, and launch a servlet. This tool is available for AIX and Windows NT systems.
- WebSphere Test Environment - Enables you to test deployment of Web application components without a full-fledged WebSphere Application Server installation. You can set breakpoints within servlet code, dynamically update the servlet at breakpoints, and continue running the servlet with the changes incorporated. These tasks can be performed without restarting the servlet.

For more information about this product, visit the following Web site:

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

More about the WebSphere Test Environment

IBM VisualAge for Java provides a subset of the WebSphere Application Server run-time environment in a component called the *WebSphere Test Environment* (WTE). The WebSphere Test Environment offers the following:

- A lightweight run-time environment with no dependency on WebSphere Application Server availability
- No dependency on an external database unless entity bean support is required

As a subset of the WebSphere Application Server, the WTE does *not* offer certain features that the application server product does, as follows:

- Secure Socket Layer (SSL) and secure HTTP (HTTPS).
- HTTP-style user ID/password authentication challenge.
- Administrative server and services.
- The XMLConfig tool. Older XML grammar is used in the WTE configuration.
- Personalization APIs
- Security context and API for enterprise beans.
- Security APIs for servlet sessions, or other security classes typically involved in sign-on, authentication, or authorization.
- Support for running multiple Web applications in addition to the default Web application

Tips for using VisualAge for Java

When you are ready to move from the WTE to deployment on the WebSphere Application Server, verify that application class paths are properly set in the new environment.

4.1.2.3: IBM WebSphere Studio

IBM WebSphere Studio Professional Edition offers the following features:

- Create Web applications for various devices, such as voice browsers and handheld devices.
- Select from two Web application models - Servlet or JSP.
- Close integration with IBM VisualAge for Java.
- Graphical display of the links between files in a project.
- Automatic updating of links whenever your files are changed or moved.
- Wizards that jump-start creation of dynamic pages that use databases and Java beans. Use the wizard output as is or tailor it to your needs.
- An import feature to quickly transfer existing Web site content into a Studio project.
- Staging and publishing your project to different (and to multiple) servers.
- The ability to archive a Web site into a single compressed file.
- Full-function visual editing of HTML and JSP files.
- Companion tools:
 - AnimatedGif Designer, for building GIF animations
 - Applet Designer, a visual authoring tool for building Java applets
 - WebArt Designer, for creating buttons, masthead images, and other graphics

For more information about this product, visit the following Web site:

<http://www.ibm.com/software/webservers/studio/index.html>

Tips for using WebSphere Studio

WebSphere Studio provides the `com.ibm.servlet.PageListServlet` class to call JSP files. Servlets generated by the WebSphere Studio wizards are subclasses of this class. Such a servlet must have an associated servlet configuration file (`.servlet`) that specifies all JSP files that the servlet might call. For more information, see *Servlet and JSP Programming with IBM WebSphere Studio and VisualAge for Java* (SG24-5755), available from [the IBM Redbooks Web site](#).

4.2: Building Web applications

Different types of Web applications exist, ranging from static document Web sites to database-backed systems. Some Web applications are front ends to traditional, non-Web applications.

See the [What are enterprise applications?](#) article for a description of applications supported by WebSphere Application Server.

The J2EE™ architecture organizes applications into reusable components, and provides underlying services (in the form of containers) for different component types. The process of assembling the various pieces of a J2EE application involves specifying such container settings as security, Java Naming and Directory Interface lookups, and remote connectivity.

J2EE applications also require deployment descriptors that are Extensible Markup Language (XML) text files to define the operating parameters and the components that comprise the application.

This section provides considerations, instructions, and tips for creating the building blocks that comprise Web applications.

View article [6.6.8: Administering Web modules \(overview\)](#) for information on configuring such Web application settings as:

- Classpaths
- Web paths
- Welcome pages
- Servlet filtering parameters
- Context attributes

View article [6.6: Tools and resources quick reference](#) for the list of new tools to assemble, deploy, and launch your J2EE Web applications.

4.2.1: Developing 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 HTTP, the HyperText Transfer Protocol.

Because servlets are written in Java, they can be ported without modification to different operating systems. 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.

Servlet coding fundamentals

In order to create an HTTP servlet, you should extend the `javax.servlet.HttpServlet` class and override any methods that you wish to implement in the servlet. For example, a servlet would override the `doGet` method to handle GET requests from clients.

For more information on the `HttpServlet` class and methods, review articles:

- [4.2.1.3.1: Creating HTTP Servlets](#)
- [4.2.1.3.1.1: Overriding HttpServlet methods](#)
- [4.2.1.3.2: Inter-servlet communication](#)

The `doGet` and `doPost` methods take two arguments:

- [HttpServletRequest](#)
- [HttpServletResponse](#)

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

The `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.

Since `doGet` and `doPost` throw two exceptions (`javax.servlet.ServletException` and `java.io.IOException`), you must include them in the declaration. You must also import classes in the following packages:

Package names	Functions/Objects
<code>java.io</code>	<code>PrintWriter</code>
<code>javax.servlet</code>	<code>HttpServlet</code>
<code>javax.servlet.http</code>	<code>HttpServletRequest</code> and <code>HttpServletResponse</code>

The beginning of your servlet might look like the following example:

```
import java.io.*;import javax.servlet.*;import javax.servlet.http.*;import java.util.*;public class
MyServlet extends HttpServlet {  public void doGet(HttpServletRequest request,
HttpServletRequest response)      throws ServletException, IOException {
```

After you create your servlet, you must:

1. Compile your servlet using the `javac` command, as for example:
`javac MyServlet.java`
2. Invoke your servlet using one of the methods described in article:
[6.6.1.5.1: Creating an application](#)

You can also compile your servlet using the `-classpath` option on the `javac` compiler. To access the classes that were extended, reference the `j2ee.jar` file in the [product_installation_root\lib](#) directory. Using this method, you issue the following command to compile your servlet:

```
javac -classpath product\_installation\_root\lib\j2ee.jar  MyServlet.java
```

Now that you successfully created, compiled, and tested your servlet on your local machine, you must install it in the WebSphere Application Server runtime. View article [6: Administer applications](#) for this information.

Servlet lifecycle

The [javax.servlet.http.HttpServlet](#) class defines methods to:

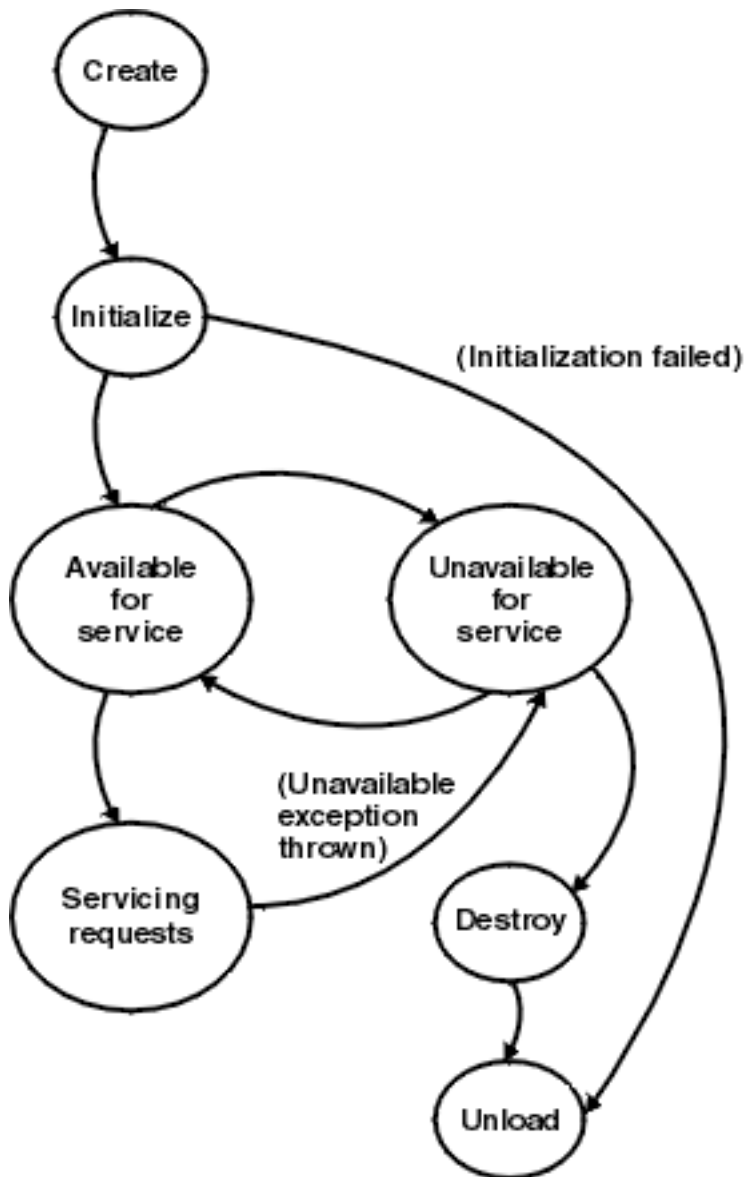
- Initialize a servlet
- Service requests
- Remove a servlet from the server

These are known as life-cycle methods and are called in the following sequence:

1. The servlet is constructed
2. It is initialized with the init method
3. Calls from clients to the service method are handled
4. The servlet is taken out of service
5. It is destroyed with the destroy method
6. The servlet is finalized and the garbage is collected.

Review article 4.2.1.1 for more life cycle information.

4.2.1.1: Servlet lifecycle



Instantiation and initialization

The Web container (the Application Server entity that processes servlets, JSP files, and other types of server-side include coding) creates an instance of the servlet. The Web container creates the servlet configuration object and uses it to pass the servlet initialization parameters to the init method. The servlet configuration object persists until the servlet is destroyed and are applied to all invocations of that servlet until the servlet is destroyed.

If the initialization is successful, the servlet is available for service. If the initialization fails, the Web container unloads the servlet. The administrator can set an application and its servlets to be unavailable for service. In such cases, the application and servlets remain unavailable until the administrator changes them to available.

Servicing requests

A client request arrives at the Application Server. The Web container creates a request object and a response object. The Web container 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 Web container invokes the servlet's `destroy()` method when appropriate and unloads the servlet. The Java Virtual Machine performs garbage collection after the destroy.

More on the initialization and termination phases

A Web container creates an instance of a servlet at the following times:

- Automatically at the application startup, if that option is configured for the servlet
- At the first client request for the servlet after the application startup
- When the servlet is reloaded

The `init` method executes only one time during the lifetime of the servlet. It executes when the Web container loads the servlet. The `init` method is not repeated regardless of how many clients access the servlet.

The `destroy()` method executes only one time during the lifetime of the servlet. That happens when the Web container stops the servlet. Typically, servlets are stopped as part of the process of stopping the application.

4.2.1.2: Servlet support and environment in WebSphere

IBM WebSphere Application Server supports the Java ServletAPI from Sun Microsystems. The product builds upon the specification in two ways.

Article [4.2.1.2.2](#) describes several IBM extensions to the specification to make it easier to manage session state, create personalized Web pages, generate better servlet error reports, and access databases.

See [article 4.2.1.2.1a](#) for a description of the Servlet API 2.2 specification.

4.2.1.2.1a: Features of Java Servlet API 2.2

WebSphere Application Server supports Java Servlet API 2.2 and JSP 1.1.

Java Servlet API 2.2 contains many enhancements intended to make servlets part of a complete application framework

The Servlet 2.2 specification is available at java.sun.com/products/servlet/index.html

No new classes were added to the Java Servlet API 2.2. specification. The following table provides more information on 27 new methods, 2 new constants and 6 deprecated methods supported by WebSphere Application Server:

New methods	Description
getServletName()	Returns the servlet's registered name
getNamedDispatcher(java.lang.String name)	Returns a dispatcher located by resource name
getInitParameter(java.lang.String name)	Returns the value for the named context parameter
getInitParameterNames()	Returns an enumeration of all the context parameter names
removeAttribute(java.lang.String name)	Added for completeness
getLocale()	Gets the client's most preferred locale
getLocales()	Gets a list of the client's preferred locales as an enumeration of locale objects
isSecure()	Returns true if the request was made using a secure channel
getRequestDispatcher(java.lang.String name)	Gets a <code>RequestDispatcher</code> using what can be a relative path
setBufferSize(int size)	Sets the minimum response buffer size
getBufferSize()	Gets the current response buffer size
reset()	Empties the response buffer, clears the response headers
isCommitted()	Returns true if part of the response has already been sent
flushBuffer()	Flushes and commits the response
setLocale(Locale locale)	Sets the response locale, including headers and charset
getLocale()	Gets the current response locale
UnavailableException(String message)	Replaces <code>UnavailableException(Servlet servlet, String message)</code>
UnavailableException(String message, int sec)	Replaces <code>UnavailableException(int sec, Servlet servlet, String message)</code>
getHeader(String message)	Returns all the values for a given header, as an enumeration of strings
getContextPath()	Returns the context path of this request
addHeader(String name, String value)	Adds to the response another value for this header name

addDateHeader(String name, long date)	Adds to the response another value for this header name
addIntHeader(String name, int value)	Adds to the response another value for this header name
getAttribute(String name)	Object HttpSession.getValue(String name)
getAttributeNames()	Replaces String[] HttpSession.getValueNames()
setAttribute(String name, Object value)	Replaces void HttpSession.setValue(String name, Object value)
removeAttribute(String name)	Replaces void HttpSession.removeValue(String name)
New constants	Description
SC_REQUESTED_RANGE_NOT_SATISFIABLE	New mnemonic for status code 416
SC_EXPECTATION_FAILED	New mnemonic for status code 417
Newly deprecated methods	Description
UnavailableException(Servlet servlet, String message)	Replaced by UnavailableException(String message)
UnavailableException(int sec, Servlet servlet, String message)	Replaced by UnavailableException(string message, int sec)
getValue(String name)	Replaced by Object HttpSession.getAttribute(String name)
getValueNames()	Replaced by numeration HttpSession.getAttributeNames()
putValue(String message, Object value)	Replaced by void HttpSession.setAttribute(String name, Object value)
removeValue(String message)	Replaced by void HttpSession removeAttribute(String name)

4.2.1.2.2: IBM extensions to the Servlet API

The Application Server includes its own packages that extend and add to the Java Servlet API. Those extensions and additions make it easier to manage session state, create personalized Web pages, generate better servlet error reports, and access databases. The Javadoc for the Application Server APIs is installed in the product *product_installation_root*\web\apidocs directory.

The Application Server API packages and classes are:

- `com.ibm.servlet.personalization.sessiontracking` package

This Application Server extension to the Java Servlet API records the referral page that led a visitor to your Web site, tracks the visitor's position within the site, and associates user identification with the session. IBM has also added session clustering support to the API.

- `com.ibm.websphere.servlet.session`. `IBMSession` interface

Extends `HttpSession` for session support and increased Web administrators' control in a session cluster environment.

- `com.ibm.servlet.personalization.userprofile` package

Provides an interface for maintaining detailed information about your Web visitors and incorporate it in your Web applications, so that you can provide a personalized user experience. This information is made persistent by storing it in a database.

- `com.ibm.websphere.userprofile` package

User profile enhancements

- `com.ibm.websphere.servlet.error`. `ServletErrorReport` class

A class that enables the application to provide more detailed and tailored messages to the client when errors occur. See the enhanced servlet error reporting article, [4.2.1.3.5](#), for details.

- `com.ibm.websphere.servlet.event` package

Provides listener interfaces for notifications of application lifecycle events, servlet lifecycle events, and servlet errors. The package also includes an interface for registering listeners. See the package Javadoc for details.

- `com.ibm.websphere.servlet.filter` package

Provides classes that support servlet chaining. The package includes the `ChainerServlet`, the `ServletChain` object, and the `ChainResponse` object. See the servlet filtering article, [4.2.1.3.4](#), for more details.

- `com.ibm.websphere.servlet.request` package

Provides an abstract class, `HttpServletRequestProxy`, for overloading the servlet engine's `HttpServletRequest` object. The overloaded request object is forwarded to another servlet for processing. The package also includes the `ServletInputStreamAdapter` class for converting an `InputStream` into a `ServletInputStream` and proxying all method calls to the underlying `InputStream`. See the Javadoc for details and examples.

- `com.ibm.websphere.servlet.response` package

Provides an abstract class, `HttpServletResponseProxy`, for overloading the servlet engine's `HttpServletResponse` object. The overloaded response object is forwarded to another servlet for processing. The package includes the `ServletOutputStreamAdapter` class for converting an `OutputStream` into a `ServletOutputStream` and proxying all method calls to the underlying

OutputStream. The package also includes the StoredResponse object that is useful for caching a servlet response that contains data that is not expected to change for a period of time, for example, a weather forecast. See the Javadoc for details and examples.

4.2.1.2.3a: Invoking servlets by classname and serving files

IBM Application Server provides some optional functions for your Web applications.

The tables below describe the function and how to use the WebSphere ApplicationServer tools to enable the function in your Web application.

Invoke servlets by class name


Objective	Invoke servlets by class or code names (such as MyServletClass)
How to enable the function	<p>Use one of the following facilities:</p> <ul style="list-style-type: none">● If using the Application Assembly Tool (AAT),click serve servlets by classname in the IBM Extensions panel.● In the <code>ibm-web-ext.xml</code> file, change the serveServletsByClassnameEnabled flag from <i>false</i> to <i>true</i>. <p>The <code>ibm-web-ext.xml</code> file is in the WEB-INF directory of theWeb module.</p>

Serve files without specifically configuring them

Objective	<p>Serve HTML, servlets, or other files in the Web application document root without extra configuration steps.</p> <p>For HTML files, you will not need to add a pass rule to the Web server. For servlets, you will not need to explicitlyconfigure the servlets in the WebSphere administrative domain.</p>
How to enable the function	<p>Use one of the following facilities:</p> <ul style="list-style-type: none">● If using the Application Assembly Tool (AAT),click File Serving Enabled in the IBM Extensions panel.● In the <code>ibm-web-ext.xml</code> file, change the fileServingEnabled flag from <i>false</i> to <i>true</i>.

4.2.1.2.3b: Security risk example of invoking servlets by class name

Anyone enabling the "serve files by class name" function in WebSphere Application Server, should take steps to avoid potential security risks. The administrator should remain aware of each and every servlet class placed in the classpath of an application, even if the servlets are to be invoked by their classnames.

 A Web site may inadvertently include malicious HTML tags or scripts in a dynamically generated page based on unvalidated input from untrustworthy sources. By accessing a malicious URL and then accessing an application server, a user may unknowingly execute script code on his machine that has full access to the data and resources on that machine. The browser executes the script on the user machine without the knowledge of the user.

The malicious tags that can be embedded in this way are `<SCRIPT>` and `</SCRIPT>`.

This problem can be prevented if the server generated pages are encoded to prevent the scripts from executing. Developers generating responses containing client data, based on servlet or JSP requests, can encode the response data using the following method:

```
com.ibm.websphere.servlet.response.ResponseUtils.encodeDataString(String)
```

Visit the [Cert advisories Web site](#) for more information.

Protecting servlets

See the article, [Securing Applications](#), for information on securing servlets and Web resources.

4.2.1.3: Servlet content, examples, and samples

Click the related topics to focus on particular aspects of servlet development, including example and sample code.

4.2.1.3.1: Creating HTTP servlets

To create an HTTP servlet, as illustrated in [ServletSample.java](#):

1. Extend the `HttpServlet` abstract class.
2. Override the appropriate methods. The `ServletSample` overrides the `doGet()` method.
3. Get HTTP request information, if any.

Use the `HttpServletRequest` object to retrieve data submitted through HTML forms or as query strings on a URL. The `ServletSample` example receives an optional parameter (`myname`) that can be passed to the servlet as query parameters on the invoking URL. An example is:

```
http://your.server.name/application_URI/ServletSample?myname=Ann
```

The `HttpServletRequest` object has specific methods to retrieve information provided by the client:

- `getParameterNames()`
- `getParameter(java.lang.String name)`
- `getParameterValues(java.lang.String name)`

4. Generate the HTTP response.

Use the `HttpServletResponse` object to generate the client response. Its methods allow you to set the response headers and the response body. The `HttpServletResponse` object also has the `getWriter()` method to obtain a `PrintWriter` object for sending data to the client. Use the `print()` and `println()` methods of the `PrintWriter` object to write the servlet response back to the client.

4.2.1.3.1.1: Overriding HttpServlet methods

HTTP servlets are specialized servlets that can receive HTTP client requests and return a response. To create an HTTP servlet, subclass the `HttpServlet` class. A servlet can be invoked by its URL, from a JavaServer Page (JSP), or from another servlet.

Methods to override

The `javax.servlet.http.HttpServlet` class contains the `init`, `destroy`, and `service` methods. The `init` and `destroy` methods are inherited, while the `service` method implementation is specific to `HttpServlet`. The method behaviors are described below; however, you might want to override methods in order to provide specialized behavior in your servlet.

- **init**

The default `init` method is usually adequate but can be overridden with a custom `init` method, typically to register application-wide resources. For example, you might write a custom `init` method to load GIF images only one time, improving the performance of servlets that return GIF images and have multiple client requests. Other examples are initializing a database connection and registering servlet context attributes.

- **destroy**

The default `destroy` method is usually adequate, but can be overridden. Override the `destroy` method if you need to perform actions during shutdown. For example, if a servlet accumulates statistics while it is running, you might write a `destroy()` method that saves the statistics to a file when the servlet is unloaded. Other examples are closing a database connection and freeing resources created during the initialization.

When the server unloads a servlet, the `destroy` method is called after all `service` method calls complete or after a specified time interval. Where threads have been spawned from within `service` method and the threads have long-running operations, those threads may be outstanding when the `destroy` method is called. Because this is undesirable, make sure those threads are ended or completed when the `destroy` method is called.

- **service**

The `service` method is the heart of the servlet. Unlike the `init` and `destroy` methods, it is invoked for each client request. In the `HttpServlet` class, the `service` method already exists. The default `service` function invokes the `doXXX` method corresponding to the method of the HTTP request. For example, if the HTTP request method is `GET`, `doGet` method is called by default. Because the `HttpServlet.service` method checks the HTTP request method and calls the appropriate handler method, it is usually not desirable to override the `service` method. Rather, override the appropriate `doXXX` methods that the servlet supports.

4.2.1.3.2: Inter-servlet communication

There are three types of servlet communication:

- Accessing data within a servlet's scope
- Forwarding a request and including a response from another servlet using the `RequestDispatcher`
- Application-to-application communication via the `ServletContext`

Sharing data within scope

JavaServerPages (JSPs) use this method to share data through beans. The ability of servlets to share data depends on the scope of the bean. The possible scopes are request, session, and application.

Forwarding and including data

For session-scoped data and attributes, use the `HttpSession.setAttribute` and `getAttribute` methods to set and get attributes in the `HttpSession` object. Session-scoped beans and objects bound to a session are examples of session-scoped objects.

For application-scoped data, use the `RequestDispatcher`'s `forward` and `include` methods to share data among applications. The `forward` method sends the HTTP request from one servlet to a second servlet for additional processing. The calling servlet adds the URL and request parameters in its HTTP request to the request object passed to the target servlet. The forwarding servlet must not have committed any output to the client. The target servlet generates the response and returns it to the client.

The `include` method enables a receiving servlet to include another servlet's response data in its response. The included servlet cannot set response headers. The receiving servlet can fully access the request object but can only write data to the `ServletOutputStream` or `PrintWriter` of the response object. If the servlets use session tracking, you must create the session outside of the included servlet. The `RequestDispatcher.forward` method is similar in function to the `HttpServletResponse.callPage` method previously supported for JSP development.

Application-to-application communication

Web applications share data through the `ServletContext`. A Web application has a single servlet context. A `ServletContext` object is accessible to any Web application associated with a virtual host. Servlet A in application A can obtain the `ServletContext` for application B in the same virtual host. After Servlet A obtains the servlet context for B, it can access the request dispatcher for servlets in application B and call the `getAttribute` and `setAttribute` methods of the servlet context. An example of the coding in Servlet A is:

```
appBcontext = appAcontext.getContext( "/appB" );  
appBcontext.getRequestDispatcher( "/servlet5" );
```

4.2.1.3.2.2: Example: Servlet communication by forwarding

In this example, the forward method is used to send a message to a JSP file (a servlet) that prints the message. The forwarding servlet code is:

```
import java.io.*;import javax.servlet.*;import javax.servlet.http.*;public class UpdateJSPTTest
extends HttpServlet{    public void doGet (HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException    {        String message = "This is a test";
req.setAttribute("message", message);        RequestDispatcher rd =
getServletContext().getRequestDispatcher("/Update.jsp");        rd.forward(req, res);    }}
```

The JSP file is:

```
<html><head></head><body><h1><servlet code=UpdateJSPTTest></servlet></h1><%    String message =
(String) request.getAttribute("message");    out.print("message: <b>" + message +
"</b>");%><p><ul><% for (int i = 0; i < 5; i++)    {        out.println ("<li>" + i);
}%></ul></body></html>
```

4.2.1.3.4: Filtering and chaining servlets

The Application Server supports two kinds of filtering:

- *MIME-based filtering* involves configuring the servlet engine to forward HTTP responses with the specified MIME type to the designated servlet for further processing.
- Servlet chaining involves defining a list (a sequence) of two or more servlets such that the request object and the ServletOutputStream of the first servlet is passed to the next servlet in the sequence. This process is repeated at each servlet in the list until the last servlet returns the response to the client.

4.2.1.3.4.1: Servlet filtering with MIME types

To configure MIME filters, use an administrative client to configure recognized MIME types for virtualhosts containing servlets.

4.2.1.3.4.2: Servlet filtering with servlet chains

To configure a servlet chain, you must use an IBM supplied servlet named `com.ibm.websphere.servlet.filter.ChainerServlet`

1. Add the `com.ibm.websphere.servlet.filter.ChainerServlet` to your Web application during the application assembly stage and assign a servlet URL to the servlet instance.
2. Define the following initialization parameter and value for the ChainerServlet:

Parameter	Value
chainer.pathlist	<i>/first_servlet_URL /next_servlet_URL</i>

The `chainer.pathlist` is a space-delimited list of servlet URLs. For example, if you want the sequence of servlets to be three servlets that you added to the examples application (`servletA`, `servletB`, `servletC`), specify:

Parameter	Value
chainer.pathlist	<code>/servletA /servletB /servletC</code>

3. To invoke a servlet chain, invoke the servlet URL of the ChainerServlet in your application.

4.2.1.3.5: Enhancing servlet error reporting

A servlet can report errors by:

- Calling the `ServletResponse.sendError` method
- Throwing an uncaught exception within its service method

The enhanced servlet error reporting function in IBM WebSphere Application Server provides an easier 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 page to the client, the servlet engine:

1. Gets the `ServletContext.RequestDispatcher` for the URI configured for the application error path.
2. Creates an instance of the error bean (type `com.ibm.websphere.servlet.errorServletErrorReport`). The bean scope is request, so that the target servlet (the servlet that encountered the error) can access the detailed error information.

For the Application Server, the `ServletResponse.sendError()` method has been overridden to provide the functionality previously described. The overridden method is shown below:

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

4.2.1.3.5.1: Public methods of the ServletErrorReport class


To create an error JSP or servlet, you need to know the public methods of the `com.ibm.websphere.servlet.error.ServletErrorReport` class (the error bean), which are:

```
public class ServletErrorReport extends ServletException{           //Get the stacktrace 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. //he 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()}
```

4.2.1.3.6: Serving servlets by classname

To enable serving servlets by classname, you can either:

- Click **serve servlets by classname** in the IBM Extensions panel of the Application Assembly Tool (AAT), or
- Change the **serveServletsByClassnameEnabled** flag in the `ibm-web-ext.xml` file from *false* to *true*.


 The `ibm-web-ext.xml` file is located in the **WEB-INF** directory of the installed Web module

See section [4.2.1.2.3a](#) for details and instructions.

4.2.1.3.7: Serving all files from application servers

Files are served on a *per-web* module, not a *per-appserver* basis. To enable file serving, you can either:

- Click the **File Serving Enabled** checkbox in the IBM extensions panel of the Application Assembly Tool (AAT), or
- Change the **fileServingEnabled** flag from *false* to *true* in the `ibm-web-ext.xml` file.

 The `ibm-web-ext.xml` file is located in the **WEB-INF** directory of the installed Web module

See section [4.2.1.2.3a](#) for details and instructions.

4.2.1.3.8: Obtaining the Web application classpath from within a servlet

To have a servlet or JSP-generated servlet detect the classpath of the Web application to which it belongs, get the

`com.ibm.websphere.servlet.application.classpath`

attribute from the `ServletContext`.

4.2.1.3.9: PageListServlet support

IBM WebSphere Application Server supplies the PageListServlet to call a Java Server Page (JSP) by name. The PageListServlet uses configuration information to map a JSP name to the URI, where the URI specifies a JSP file in the WAR module. This support allows application developers to stop hard-coding URLs in their servlets.


These mappings, or page lists, are logically grouped according to the markup-language type (HTML, WML, and others) the JSP file is going to return to the requesting client. This allows applications, through the use of servlets that extend the PageListServlet class, to call a JSP file that returns the proper markup-language data type of the calling client. For example, if a request comes from a PDA, which requires WML data, and makes a request to a servlet that extends the PageListServlet, then a Java Server Page that returns WML data is called.

PageListServlet configuration information can be defined either in the IBM Web Extensions file or in an XML servlet configuration file. The IBM Web Extensions file is created and stored in the WAR file by the IBM WebSphere Application Assembly Tool. An XML servlet configuration file can be created using IBM WebSphere Studio or manually.

The PageListServlet has a `callPage()` method that invokes a Java Server Page in response to an HTTP request for a page in a page list.

The `callPage()` method can be invoked as follows:

- `callPage(String pageName, HttpServletRequest request, HttpServletResponse response)`
 - **pageName** - A page name defined in the PageListServlet configuration
 - **request** - The HttpServletRequest object
 - **response** - The HttpServletResponse object

 For this method of invocation, the default markup-language is HTML.
- `callPage(String mlName, String pageName, HttpServletRequest req, HttpServletResponse resp)`
 - **mlName** - A markup-language type.
 - **pageName** - A page name defined in the PageListServlet configuration
 - **request** - The HttpServletRequest object
 - **response** - The HttpServletResponse object

See the [Javadoc for the PageListServlet](#) for a complete list of available APIs.

In addition to providing the page list mapping capability, the PageListServlet also has **Client Type Detection** support. Using the configuration information in the `client_types.xml` file, a servlet can determine the markup-language type the calling client requires for the response. This support allows the user's servlet to call an appropriate JSP, based on markup-language type. Use the second version of the `callPage()` method (described above) for **Client Type Detection** support.

In structuring the servlet code, keep in mind that the `PageListServlet.callPage()` method is not an exit. Any servlet code that follows this method call will be executed.

4.2.1.3.9.1: Extending PageListServlet

The HelloPervasiveServlet is an example of a servlet that extends the PageListServlet class and attempts to determine the markup-language type required by the client. The servlet then uses the callPage() method to call the JSP with the page name of "Hello.page".

```
public class HelloPervasiveServlet extends PageListServlet implements Serializable{ /* * doGet --
Process incoming HTTP GET requests */ public void doGet(HttpServletRequest request,
HttpServletRequest 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 contained a markup-language value. For example, if this
// servlet was invoked like this: // http://localhost/servlets/HelloPervasive?mlname=VXML
String mlName = getMLNameFromRequest(request); // If no ML type was provided in the queryString,
then attempt to // determine the client type from the Request and use the ML name as //
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); } }}
```

4.2.1.3.9.2: Configuring page lists using the Application Assembly Tool

PageListServlet configuration information can be defined in the IBM Web Extensions file or in an XML servlet configuration file. The IBM Web Extensions file is created and stored in the WAR file by the IBM WebSphere Application Assembly Tool (AAT). In the AAT, the page list information is configured under **PageList Extensions**.

4.2.1.3.9.3: Configuring page lists using an XML servlet configuration file

An alternative or legacy way of providing PageListServlet configuration information, is using an XML file known as the **XML Servlet Configuration** file. This file provides configuration information for page lists, and additional servlet configuration information. The file has a **.servlet** extension and resides in the same directory as the servlet class file. The XML servlet configuration file must be created with one of the following names:

1. `servlet_class_name.servlet`
2. `servlet_name.servlet`

IBM WebSphere Studio provides wizards that generate servlets with accompanying XML servlet configuration files. If you are not using IBM WebSphere Studio, you can manually create XML servlet configuration files. Each XML configuration file must be a well-formed XML document. The files are not validated against a Document Type Definition (DTD). Although there is no DTD, it is recommended that all elements in the file appear in the same order as the elements described below:

XML Servlet configuration file elements	
Elements	Description
servlet	The root element of an XML servlet configuration file.
code	The class name of the servlet, that extends the PageListServlet, without the <code>.class</code> extension.
description	The description of the servlet.
init-parameter	The attributes of this element specify the name-value pair to be used as an initialization parameter on the servlet. A servlet can have multiple initialization parameters, each within its own init-parameter element.
markup-language	Contains <code><ml-name></code> , <code><ml-mime></code> , and <code><page-list></code> elements. (The root element <code><servlet></code> can contain multiple <code><markup-language></code> elements.)
ml-name	A markup-language type, as for example: HTML, or WML, or VXML, and so forth
ml-mime	A MIME type, as for example: <code>text.html</code> , or <code>text/x-vxml</code> , or <code>text/vnd.wap.wml</code> , and so forth
page-list	Contains <code><default-page></code> , <code><error-page></code> , and <code><page>+</code> elements. (A <code><page-list></code> element can contain multiple <code><page></code> elements.)
default-page	Contains a <code><uri></code> element. The URI specifies the JSP to be called if the requested page does not exist or is not specified on the HTTP request.
error-page	Contains a <code><uri></code> element. The URI specifies the JSP to be called when the <code>handleError()</code> PageListServlet method is called.
page	Contains a <code><uri></code> and <code><page-name></code> element. The URI specifies the JSP file to be called when a PageListServlet method <code>callPage()</code> is called with the same value as <code><page-name></code> .
uri	A JSP file within the WAR Module.
page-name	The name in which a servlet, extending the PageListServlet, will use in the <code>callPage()</code> method to call a JSP.

4.2.1.3.9.4: Example of the XML servlet configuration file

```
<?xml version="1.0"?><servlet>  <code>HelloPervasiveServlet</code>  <description>Shows how to use
PageListServlet class.<:/description>  <init-parameter name="name1" value="value2"/>
<markup-language>    <ml-name>HTML</ml-name>    <ml-mime>text/html</ml-mime>    <page-list>
<error-page>        <uri>/mywebapp/HelloHTMLError.jsp</uri>        </error-page>        <page>
<page-name>Hello.page</page-name>        <uri>/mywebapp/HelloHTML.jsp</uri>        </page>
</page-list>  </markup-language>  <markup-language>    <ml-name>VXML</ml-name>
<ml-mime>text/x-vxml</ml-mime>    <page-list>    <error-page>
<uri>/mywebapp/HelloVXMLError.jsp</uri>    </error-page>    <page>
<page-name>Hello.page</page-name>    <uri>/mywebapp/HelloVXML.jsp</uri>    </page>
</page-list>  </markup-language>  <markup-language>    <ml-name>WML</ml-name>
<ml-mime>text/vnd.wap.wml</ml-mime>    <page-list>    <error-page>
<uri>/mywebapp/HelloWMLError.jsp</uri>    </error-page>    <page>
<page-name>Hello.page</page-name>    <uri>/mywebapp/HelloWML.jsp</uri>    </page>
</page-list>  </markup-language></servlet>
```

4.2.1.3.9.5: PageListServlet client type configuration file

In addition to providing the page list mapping capability, the PageListServlet also has **Client Type Detection** support. Using the configuration information in the `client_types.xml` file, a servlet can determine the markup-language type the calling client requires for the response.

This support allows the servlet, extending PageListServlet, to call an appropriate JSP file, with the `callPage()` method, based on the markup-language type of the request. The client type detection method, `getMLTypeFromRequest(HttpServletRequest request)`, provided by the PageListServlet, inspects the HttpServletRequest object's request headers, and searches for a match in the `client_types.xml` file.

The client type detection method does the following:

1. Using the input HttpServletRequest and the `client_types.xml` file, it checks for a matching HTTP request name and value. If found, it returns the markup-language value configured for the `<client-type>` element.
2. If multiple matches are found, it returns the markup-language for the first `<client-type>` (for which a match was found).
3. If no match was found, it 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 [*product_installation_root*](#)/properties directory.

4.2.1.3.9.6: Example of a client type configuration file

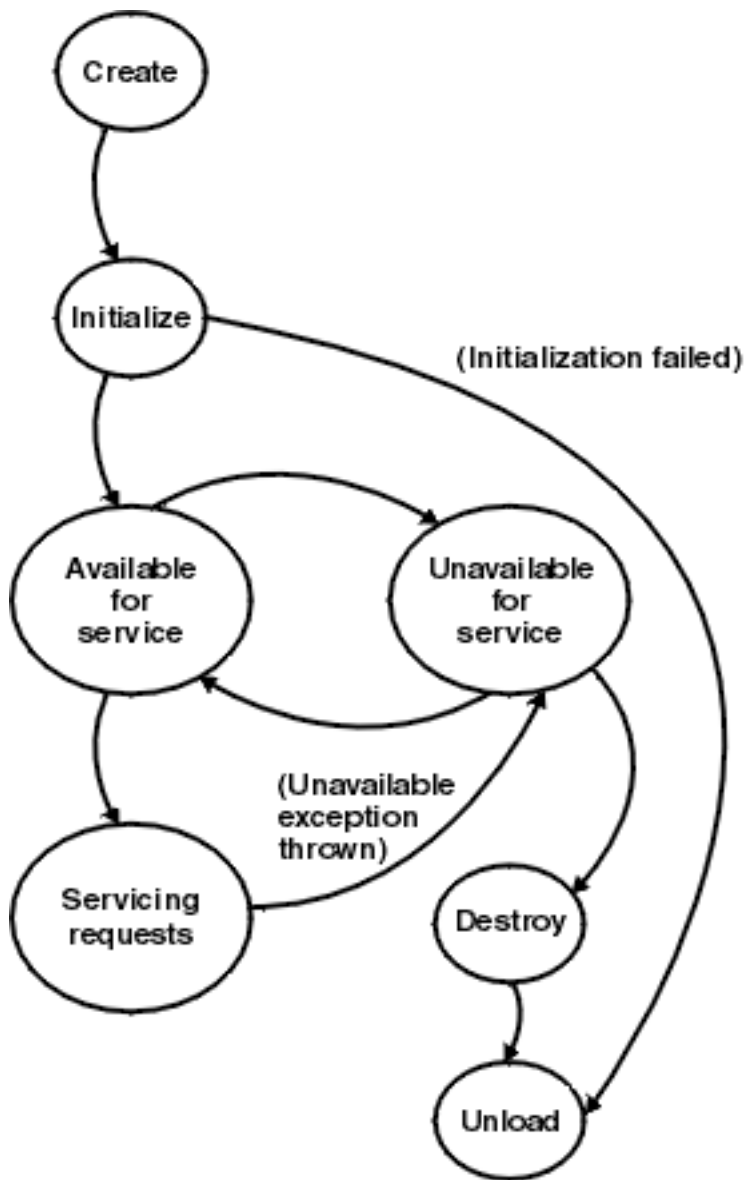
```
<?xml version="1.0"?><!DOCTYPE clients [(<!ELEMENT client-type
(description,markup-language,request-header+)><!ELEMENT description (#PCDATA)><!ELEMENT
markup-language (#PCDATA)><!ELEMENT request-header (name,value)><!ELEMENT clients
(client-type+)><!ELEMENT name (#PCDATA)><!ELEMENT value (#PCDATA)>)]><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>
```

4.2.2: Developing JSP files

If JSP files are fairly new to you, consider reading about their lifecycle and access model. When you are ready to begin writing JSP files, see the article featuring JSP file content. Review the support and environment article for topics such as JSP processors and APIs, recommended development tools, and batch compiling.

4.2.2.1: JavaServer Pages (JSP) lifecycle

JSP files are compiled into servlets. After a JSP is compiled, its lifecycle is similar to the servlet lifecycle:



Java source generation and compilation

When a Web container receives a request for a JSP file, it passes the request to the JSP processor .

If this is the first time the JSP file has been requested or if the compiled copy of the JSP file is not found, the JSP compiler generates and compiles a Java source file for the JSP file. The JSP processor puts the Java source and class file in the JSP processor directory.

By default, the JSP syntax in a JSP file is converted to Java code that is added to the `service()` method of the generated class file. If you need to specify initialization parameters for the servlet or other initialization information, add the method directive set to the value `init`.

Request processing

After the JSP processor places the servlet class file in the JSP processor directory, the Web container creates an instance of the servlet and calls the servlet service() method in response to the request. All subsequent requests for the JSP are handled by that instance of the servlet.

When the Web container receives a request for a JSP file, the engine checks to determine whether the JSP file (.jsp) has changed since it was loaded. If it has changed, the Web container reloads the updated JSP file (that is, generates an updated Java source and class file for the JSP). The newly loaded servlet instance receives the client request.

Termination

When the Web container no longer needs the servlet or a new instance of the servlet is being reloaded, the Web container invokes the servlet's destroy() method. The Web container can also call the destroy() method if the engine needs to conserve resources or a pending call to a servlet service() method exceeds the timeout. The Java Virtual Machine performs garbage collection after the destroy.

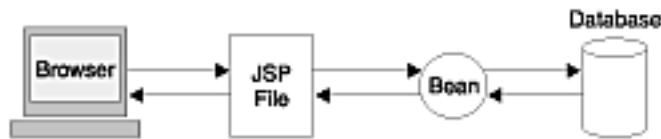
4.2.2.1a: JSP access models

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, as shown:

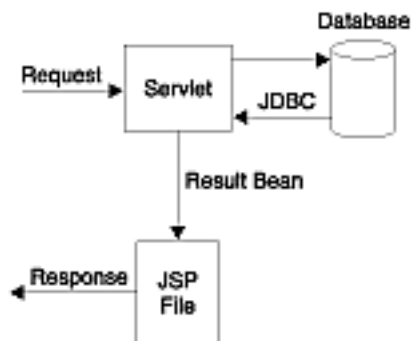
Request for a JSP file



When the Web server receives a request for a JSP file, the server sends the request to the application server. The application server parses the JSP file and generates Java source, which is compiled and executed as a servlet.

- The request is sent to a servlet that generates dynamic content and calls a JSP file to send the content to the browser, as shown:

Request for a servlet



This access model facilitates separating content generation from content display.

The application server supplies a set of methods in the `HttpServletRequest` object and the `HttpServletResponse` object. These methods allow an invoked servlet to place an object (usually a bean) into a request object and pass that request to another page (usually a JSP file) for display. The invoked page retrieves the bean from the request object and generates the client-side HTML.

4.2.2.2: JSP support and environment in WebSphere

IBM WebSphere Application Server supports the [JSP 1.1](#) Specification from Sun Microsystems. If you are going to develop new JSP files for use with IBMWebSphere Application Server, it is recommended you use JSP 1.1.

All APIs described in this section are supported at the JSP 1.1 level.

4.2.2.2.2: JSP processors

When you install the Application Server product on a Web server, the Web server configuration is set to pass HTTP requests for JSP files (files with the extension .jsp) to the Application Server product.

The JSP processor creates and compiles a servlet from each JSP file. The processor produces these files for each JSP file:

- .java file, which contains the Java language code for the servlet
- .class file, which is the compiled servlet
- .dat file, which contains the static part of the original jsp file

The JSP processor puts the .java, the .class file, and the .dat file in the following path:

`<product_installation_root>\temp\<hostname>\<servername>\<webmodulename>`

Like all servlets, a servlet generated from a JSP file extends `javax.servlet.http.HttpServlet`. The servlet Java code contains import statements for the necessary classes and a package statement, if the servlet class is part of a package.

If the JSP file contains JSP syntax (such as directives and scriptlets), the JSP processor converts the JSP syntax to the equivalent Java code. If the JSP file contains HTML tags, the processor adds Java code so that the servlet outputs the HTML character by character.

4.2.2.2.3: Java Server Page attributes

Use the WebSphere Application Assembly Tool (AAT) to set the following Java Server Page attributes. The JSP attributes are stored in the IBM extensions document for Web module, `ibm-web-ext.xml`.

JSP file attribute names	JSP file attribute values (Default values are in bold text)	Purpose
keepgenerated	true false	If true, the generated . javafile will be kept. If the value is false, the . java file is not saved.
dosetattribute	true false	By default, JSP files using the "usebean" tag withScope="session" do not always work properly when session persistence is enabled.
scratchdir	product_installation_root \temp	Set scratchdir to a valid drive and directory which the JSP engine will use to store the generated . class and . java files.
jsp.repeatTag.ignoreException	true false	<p>In previous releases, the <tsx:repeat> tag would iterate until one of the following conditions was met:</p> <ol style="list-style-type: none"> 1. The end value was reached 2. An <code>ArrayIndexOutOfBoundsException</code> was thrown <p>Other types of exceptions were caught but not thrown, which could result in numerous errors being returned to the browser.</p> <p>In version 4.0, the default behavior will now stop the repeat tag iterations when any exception is thrown.</p> <p>To reinstate the old behavior, set this parameter's value to true.</p>
defaultEncoding	<p>Name of the desired character set.</p> <p>The value of the system property file.encoding is the default.</p>	<p>Use this parameter to set the encoding for JSP pages. If a JSP page contains a <code>contentType</code> directive that defines an alternative character set, that character set is used instead of the <code>defaultEncoding</code> parameter's value.</p> <p>The order of precedence is:</p> <ol style="list-style-type: none"> 1. The JSP page's <code>contentType</code> directive's charset. 2. The <code>defaultEncoding</code> parameter's value. 3. The System property <code>file.encoding</code> value 4. ISO-8859-1

4.2.2.2.4: Batch Compiling JSP files

As an IBM enhancement to JSP support, IBM WebSphere Application Server provides a batch JSP compiler. Use this function to batch compile your JSP files and thereby enable faster responses to the initial client requests for the JSP files on your production Web server.

It is best to batch compile all of the JSP files associated with an application. Batch compiling saves system resources and provides security on the application server by specifying if and/or when the server is to check for a classfile or recompile a JSP file. The application server will monitor the compiled JSP file for changes, and will automatically recompile and reload the JSP file whenever the application server detects that the JSP file has changed. By modifying this process, you can eliminate time- and resource-consuming compilations and ensure that you have control over the compilation of your JSP files. It is also useful as a fast way to resynchronize all of the JSP files for an application.

4.2.2.2.4.2: Compiling JSP 1.1 files as a batch

To use the JSP batch compiler for JSP files, enter the following command on a single line at an operating system command prompt:

```
JspBatchCompiler -enterpriseApp<name>-webModule<name>[-filename<jsp name>]  
                [-keepgenerated<true|false>][-configFile<configfile name>]
```

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

where:

- **enterpriseApp**

The name of the Enterprise Application you want to compile.

- **webModule**

The name of the specific Web module that you want to compile.

- **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.

- **keepgenerated**

If set to "yes" WebSphere Application Server will save the generated *.java* files used for compilation on your server. By default, this is set to "no" and the *.java* files are erased after the class files have been compiled.


- **configFile**

The `configFile` parameter is valid only on Advanced Single Server Edition for "Multiplatforms." Use it to specify an alternative server configuration file (the default is `server-cfg.xml`).

4.2.2.3: Overview of JSP file content

JSP files have the extension .jsp. A JSP file contains any combination of the following items. Click an item to learn about its syntax. To learn how to put it all together, see the Related information for examples, samples, and additional syntax references.

JSP syntax

Syntax format	Details
Directives	<p>Use JSP directives (enclosed within <code><%@</code> and <code>%></code>) to specify:</p> <ul style="list-style-type: none">● Scripting language being used● Interfaces a servlet implements● Classes a servlet extends● Packages a servlet imports● MIME type of the generated response <p> See Sun's JSP Syntax Reference for JSP 1.1 syntax descriptions and examples.</p>
Class-wide variable and method declarations	<p>Use the <code><%! declaration(s) %></code> syntax to declare class-wide variables and class-wide methods for the servlet class.</p>
Inline Java code (scriptlets) , enclosed within <code><%</code> and <code>%></code>	<p>You can embed any valid Java language code inline within a JSP file between the <code><%</code> and <code>%></code> tags. Such embedded code is called a <i>scriptlet</i>. If you do not specify the method directive, the generated code becomes the body of the service method.</p> <p>An advantage of embedding Java coding inline in JSP files is that the servlet does not have to be compiled in advance, and placed on the server. This makes it easier to quickly test servlet coding.</p>
Variable text, specified using IBM extensions for variable data or Java expressions enclosed within <code><%=</code> and <code>%></code>	<p>The IBM extensions are the more user-friendly approach to putting variable fields on your HTML pages.</p> <p>A second method for adding variable data is to specify a Java language expression that is resolved when the JSP file is processed. Use the JSP expression tags <code><%=</code> and <code>%></code>. The expression is evaluated, converted into a string, and displayed. Primitive types, such as int and float, are automatically converted to string representation.</p>
<jsp:useBean> tag	<p>Use the <code><jsp:useBean></code> tag to create an instance of a bean that will be accessed elsewhere within the JSP file. Then use JSP tags to access the bean.</p>
JSP tags for database access (JSP 1.1)	<p>The IBM extensions make it easy for non-programmers to create Web pages that access databases.</p>

HTML tags

A JSP file can contain any valid HTML tags. View article [0.70: What is HTML?](#) for more information on HTML. Refer to your favorite HTML reference for a description of HTML tags.

4.2.2.3.2: JSP syntax: Class-wide variables and methods

Use the `<%! declaration(s) %>` syntax to declare class-wide variables and class-wide methods for the servlet class.

An example of specifying class-wide variables and methods:

```
<%! int i=0; String foo = "Hello"; %>    <%! private void foo() { // code for the method } %>
```

4.2.2.3.3: JSP syntax: Inline Java code (scriptlets)

You can embed any valid Java language code inline between the `<%` and `%>` tags. Such embedded code is called a *scriptlet*. If you do not specify the method directive, the generated code becomes the body of the service method.

Be sure to use the braces characters, `{ }`, to enclose `if`, `while`, and `for` statements even if the scope contains a single statement. You can enclose the entire statement with a single scriptlet tag. However, if you use multiple scriptlet tags with the statement, be sure to place the opening brace character, `{`, in the same statement as the `if`, `while`, or `for` keyword. The following examples illustrate these points. The first example is the easiest.

```
<%for (int i = 0; i < 1; i++) {    out.println("<P>This is written when " + i + " is < 1</P>");
}%>...<% for (int i = 0; i < 1; i++) {                                %><%
out.println("<P>This is written when " + i + " is < 1</P>"); %><% }
%>...<% for (int i = 0; i < 1; i++) {
%><%    out.println("<P>This is written when " + i + " is < 1</P>"); %><%    }
%>
```

4.2.2.3.4: JSP syntax: Java expressions

To specify a Java language expression that is resolved when the JSP file is processed, use the JSP expression tags `<%=` and `%>`. The expression is evaluated, converted into a string, and displayed. Primitive types, such as `int` and `float`, are automatically converted to string representation. In this example, `foo` is the class-wide variable declared in the class-wide variables and methods example:

```
<p>Translate the greeting <%= foo %>.</p>
```

When the JSP file is served, the text reads: Translate the greeting Hello.

4.2.2.3.5: JSP syntax: useBean tag

The `<jsp:useBean>` tag locates a Bean or creates an instance of a Bean if it does not exist.

JavaBeans can be class files, serialized beans, or dynamically generated by a servlet. A JavaBean can even be a servlet (that is, provide a service). If a servlet generates dynamic content and stores it in a bean, the bean can then be passed to a JSP file for use within the Web page defined by the file.

See [Sun's JSP Syntax Reference](#) for JSP 1.1 syntax descriptions and examples.

4.2.2.3.5.1: JSP syntax: <jsp:useBean> tag

Use the <jsp:useBean> tag to locate or instantiate a JavaBeans component. The syntax for the <jsp:useBean> tag is:

```
<jsp:useBean
  id="beanSomeName"
  scope="page|request|session|applicaton"
  { class="package_class" |
    type = "package_class" |
    class="package_class" type = "package_class" |
    beanName="{package.class| <%= expression%>}" type = "package_class"
  }
  { /> |
    > other elements
  }
</jsp:useBean>
```

See [Sun's JSP syntax reference](#) for a description of the <jsp:useBean> attributes and examples.

4.2.2.3.5.2: JSP syntax: Accessing bean properties

After specifying the `<jsp:useBean>` tag, you can access the bean at any point within the JSP file using the `<jsp:getProperty>` tag.

For a description of the `<jsp:getProperty>` tag attributes and for coding examples, see [Sun's JSP Syntax Reference](#)

4.2.2.3.5.3: JSP syntax: Setting bean properties

You can set bean properties by using the `<jsp:setProperty>` tag. The `<jsp:setProperty>` tag specifies a list of properties and the corresponding values. The properties are set after the bean is instantiated using the `<jsp:useBean>` tag.

You must declare the bean with `<jsp:useBean>` before you can set a property value.

See the [Sun's JSP syntax reference](#) for `<jsp:setProperty>` syntax details and examples.

4.2.2.3.7: IBM extensions to JSP syntax

Refer to the Sun JSP Specification for the base JavaServer Pages (JSP) APIs. IBMWebSphere Application Server Version 3.5 provided several extensions to the base APIs. The backward compatibility of the JSP 1.1 specification to JSP 1.0 allows users to invoke these APIs without modification.

The extensions belong to these categories:

Extension	Use
Syntax for variable data	Put variable fields in JSP files and have servlets and JavaBeans 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. The user ID and password for the database connection can be provided by the user at request time, or can be hardcoded within the JSP file.

Scope of variables: Because the values specified by syntax apply only to the JSP file in which the syntax is embedded, identifiers and other tag data can be accessed only within the page.

See the Related information for syntax details.

4.2.2.3.7.1: JSP syntax: Tags for variable data

The variable data syntax enables you to put variable fields in your JSP file and have your servlets and JavaBeans dynamically replace the variables with values from a database when the JSP output is returned to the browser.

The table summarizes the tags. Click a tag to link to its syntax description.

Goal	Tag	Details
Get the value of a bean to display in a JSP.	<tsx:getProperty>	<p>This IBM extension of the Sun JSP <code><jsp:getProperty></code> tag implements all of the <code><jsp:getProperty></code> function and adds the ability to introspect a database bean that was created using the IBM extension <code><tsx:dbquery></code> or <code><tsx:dbmodify></code>.</p> <p>Note: You cannot assign the value from this tag to a variable. The value, generated as output from this tag, is displayed in the Browser window.</p>
Repeat a block of HTML tagging that contains the <code><tsx:getProperty></code> syntax and the HTML tags for formatting content.	<tsx:repeat>	<p>Use the <code><tsx:repeat></code> syntax to iterate over a database query results set. The <code><tsx:repeat></code> syntax iterates from the start value to the end value until one of the following conditions is met:</p> <ul style="list-style-type: none">● The end value is reached.● An exception is thrown. <p>The output of a <code><tsx:repeat></code> block is buffered until the block completes. If an exception is thrown before a block completes, no output is written for that block.</p>

4.2.2.3.7.1.1: JSP syntax: <tsx:getProperty> tag syntax and examples

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

where:

- **name**

The name of the JavaBean declared by the `id` attribute of a `<tsx:dbquery>` syntax within the JSP file. See [<tsx:dbquery>](#) for an explanation. 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(sub-property), 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>](#). 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 />
```

4.2.2.3.7.1.2: JSP syntax: <tsx:repeat> tag syntax

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

where:

- **index**

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**

An optional starting index value for this repeat block. The default is 0.

- **end**

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.

4.2.2.3.7.1.2a: JSP syntax: The repeat tag results set and the associated bean

The <tsx:repeat> iterates over a results set. The results set is contained within a JavaBean. The bean can be a static bean (for example, a bean created by using the IBM WebSphere Studio database wizard) or a dynamically generated bean (for example, a bean generated by the <tsx:dbquery> syntax). The following table is a graphic representation of the contents of a bean, 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 section <tsx:dbquery> 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 start row).

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

Static Bean Example	<tsx:repeat> Bean Example
<p>myBean.class</p> <pre>// Code to get a connection// Code to get the data Select * from myTable;// Code to close the connection</pre> <p>JSP file</p> <pre><tsx:repeat index=abc> <tsx:getProperty name="myBean" property="coll(abc)" /></tsx:repeat></pre> <p>ii.</p> <ul style="list-style-type: none">• The bean (myBean.class) is a static bean.• The method to access the bean properties is myBean.get(property(index)).• 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 start row.	<p>JSP file</p> <pre><tsx:dbconnect id="conn"userid="alice"passwd="test"url="jdbc:db2:sample"driver="COM.ibm.db2.jdbc.app.DB2Driver"><tsx:dbquery id="dynamic" connection="conn" > Select * from myTable;</tsx:dbquery><tsx:repeat index=abc> <tsx:repeat name="dynamic" property="coll(abc)" /></tsx:repeat></pre> <p>ii.</p> <ul style="list-style-type: none">• The bean (dynamic) is generated by the <tsx:dbquery> and does not exist until the syntax is executed.• The method to access the bean properties is dynamic.getValue("property", index).• 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>. 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 will display all elements, while Example 3 will show 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 the loop will repeat.

```
<table><tsx:repeat>  <tr><td><tsx:getProperty name="serviceLocationsQuery"  property="city"  /></tr></td>  <tr><td><tsx:getProperty name="serviceLocationsQuery"  property="address"  /></tr></td>  <tr><td><tsx:getProperty name="serviceLocationsQuery"  property="telephone"  /></tr></td></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)  /></tr></td>  <tr><td><tsx:getProperty name="serviceLocationsQuery"  property=address(myIndex)  /></tr></td>  <tr><td><tsx:getProperty name="serviceLocationsQuery"  property=telephone(myIndex)  /></tr></td></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"  /></tr></td>  <tr><td><tsx:getProperty name="serviceLocationsQuery"  property="address(myIndex)"  /></tr></td>  <tr><td><tsx:getProperty name="serviceLocationsQuery"  property="telephone(myIndex)"  /></tr></td></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 sub-properties. 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>  </table>  </tsx:repeat></tsx:repeat>
```

4.2.2.3.7.2: JSP syntax: Tags for database access

Beginning with IBM WebSphere Application Server Version 3.x, the JSP 1.0 support was extended to provide syntax for database access. The syntax makes it simple to add a database connection to a Web page and then use that connection to query or update the database. The user ID and password for the database connection can be provided by the user at request-time or hard coded within the JSP file.

The table summarizes the tags. Click a tag to link to its syntax description.

Goal	Tag	Details and examples
Specify information needed to make a connection to a JDBC or an ODBC database.	<tsx:dbconnect>	<p>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.</p> <p>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.</p>
Avoid hard coding the user ID and password in the <tsx:dbconnect>.	<tsx:userid> and <tsx:passwd>	<p>Use the <tsx:userid> and <tsx:passwd> to accept user input for the values and then add that data to the request object. The request object can be accessed by a JSP file (such as the JSPEmployee.jsp example) that requests the database connection.</p> <p>The <tsx:userid> and <tsx:passwd> must be used within a <tsx:dbconnect> tag.</p>
Establish a connection to a database, submit database queries, and return the results set.	<tsx:dbquery>	<p>The <tsx:dbquery>:</p> <ol style="list-style-type: none">1. References 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>.2. Establishes a new connection3. Retrieves and caches data in the results object4. Closes the connection (releases the connection resource)

<p>Establish a connection to a database and then add records to a database table.</p>	<p><tsx:dbmodify></p>	<p>The <tsx:dbmodify>:</p> <ol style="list-style-type: none"> 1. References a <tsx:dbconnect> in the same JSP file and uses the information provided by that 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>. 2. Establishes a new connection 3. Updates a table in the database 4. Closes the connection (releases the connection resource) <p>Examples: Basic example</p>
<p>Display query results.</p>	<p><tsx:repeat> and <tsx:getProperty></p>	<p>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.</p> <p>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.</p> <p>Examples: Basic example</p>

4.2.2.3.7.2.1: JSP syntax: <tsx:dbconnect> tag syntax

```
<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>
```

where:

- **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 optional, the userid must be provided. See <tsx:userid> and <tsx:passwd> for an alternative to hard coding 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, the password must be provided. See <tsx:userid> and <tsx:passwd> for an alternative to hard coding this attribute in the JSP file.

- **url and driver**

To establish a database connection, the URL and driver must be provided.

The Application Server Version 3 supports connection to JDBC databases and ODBC databases.

JDBC database

For a JDBC database, the URL consists of the following colon-separated elements: jdbc, the sub-protocol name, and the name of the database to be accessed. An example for a connection to the Sample database included with IBM DB2 is:

```
url="jdbc:db2:sample"driver="COM.ibm.db2.jdbc.app.DB2Driver"
```

ODBC database


Use the Sun JDBC-to-ODBC bridge driver included in the Java Development Kit (JDK) or another vendor's ODBC driver.

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.

If the database is an ODBC database, you can use an ODBC driver or the Sun JDBC-to-ODBC bridge included with the JDK. If you want to use an ODBC driver, refer to the driver documentation for instructions on specifying the database location (the url attribute) and the driver name.

In the case of the bridge, the url syntax is jdbc:odbc:database. An example is:

```
url="jdbc:odbc:autos"driver="sun.jdbc.odbc.JdbcOdbcDriver"
```

 To enable the Application Server to access the ODBC database, use the ODBC Data Source Administrator to add the ODBC data source to the System DSN configuration. To access the

ODBC Administrator, click the ODBC icon on the Windows NT Control Panel.

- **jndiname**

An optional attribute that identifies a valid context in the Application Server 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.

If the jndiname is specified, the JSP processor ignores the driver and url attributes on the <tsx:dbconnect> tag.

An empty element (such as <url></url>) is valid.

4.2.2.3.7.2.2: JSP syntax: <tsx:userid> and <tsx:passwd> tag syntax

```
<tsx:dbconnect id="connection_id"          <font color="red"><userid></font><tsx:getProperty  
name="request" property=request.getParameter("userid") /><font color="red"></userid></font>    <font  
color="red"><passwd></font><tsx:getProperty name="request" property=request.getParameter("passwd")  
/><font color="red"></passwd></font>    url="protocol:database_name:database_table"  
driver="JDBC_driver_name"> </tsx:dbconnect>
```

where:

- **<tsx:getProperty>**

This syntax is a mechanism for embedding variable data. See [JSP syntax for variable data](#).

- **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.

4.2.2.3.7.2.3: JSP syntax: <tsx:dbquery> tag syntax

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

where:

- **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> to display query results.

The id becomes the name of a bean that contains the results set. 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. In such a case, the effective limit is determined by the number of records and the system caching capability.

- **SELECT command and JSP syntax**

The only valid SQL command is SELECT because the <tsx:dbquery> must return a results set. Refer to your database documentation for information about the SELECT command. See other sections of this document for a description of JSP syntax for variable data and inline Java code.

4.2.2.3.7.2.3a: Example: JSP syntax: <tsx:dbquery> tag syntax

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 the field is based on user input.

```
<tsx:dbquery id="empqs" connection="conn" >select * from Employee where WORKDEPT='<tsx:getProperty  
name="request" property=request.getParameter("WORKDEPT") />'</tsx:dbquery>
```

4.2.2.3.7.2.4: JSP syntax: <tsx:dbmodify> tag syntax

<%-- Any valid database update commands can be placed within the DBMODIFY tag. --><%-- Any other syntax, including HTML comments, are not valid. --><tsx:dbmodify
connection="connection_id"></tsx:dbmodify>

where:

- **connection**

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

- **Database commands**

Valid database commands. Refer to your database documentation for details

4.2.2.3.7.2.4a: Example: JSP syntax: <tsx:dbmodify> tag syntax

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 <tsx:getProperty>.

```
<tsx:dbmodify connection="conn" >insert into EMPLOYEE
(EMPNO,FIRSTNME,MIDINIT,LASTNAME,WORKDEPT,EDLEVEL)values(' <tsx:getProperty name="request"
property=request.getParameter("EMPNO") />', '<tsx:getProperty name="request"
property=request.getParameter("FIRSTNME") />', '<tsx:getProperty name="request"
property=request.getParameter("MIDINIT") />', '<tsx:getProperty name="request"
property=request.getParameter("LASTNAME") />', '<tsx:getProperty name="request"
property=request.getParameter("WORKDEPT") />', <tsx:getProperty name="request"
property=request.getParameter("EDLEVEL") />)</tsx:dbmodify>
```

4.2.2.3.7.2.5a: Example: JSP syntax: <tsx:repeat> and <tsx:getProperty> tags

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

4.2.2.3a: JSP examples

The example JSP application accesses the Sample database that you can install with IBM DB2. The example application includes:

JSPLogin.jsp	An interface for logging in to the application
JSPEmployee.jsp	A dialog for querying and updating database records
JSPEmployeeRepeatResults.jsp	A dialog for displaying update confirmations and query results

JSP code example - a login

```
<HTML><HEAD><TITLE>JSP:  Login into the Employee Records
Center</TITLE></HEAD><BODY><H1><CENTER>Login into the Employee Records Center</CENTER></H1><FORM
NAME="LoginForm" ACTION="jsp10employee.jsp" METHOD="post"
ENCODE="application/x-www-form-urlencoded"><P>To login to the Employee Records Center, submit a
validuserid and password to access the Sample database installed under IBM DB2.</P><TABLE><TR
VALIGN=TOP ALIGN=LEFT><TD><B><I>Userid:</I></B></TD><TD><INPUT TYPE="text" NAME="USERID"
VALUE="userid"><BR></TD></TR><TR VALIGN=TOP ALIGN=LEFT><TD><B><I>Password:</I></B></TD><TD><INPUT
TYPE="password" NAME="PASSWD" VALUE="password"></TD></TR></TABLE><INPUT TYPE="submit" NAME="Submit"
VALUE="LOGIN"></FORM><HR></BODY></HTML>
```

JSP code example - view employee records

```
<HTML><HEAD><TITLE>JSP: Add and View Employee Records</TITLE></HEAD><BODY><H1><CENTER>Add and View  
Employee Records</CENTER></H1><%-- Get a connection to the Sample DB2 database using parameters from  
Login.jsp --%><tsx:dbconnect id="conn" url="jdbc:db2:sample"  
driver="COM.ibm.db2.jdbc.app.DB2Driver"><userid><tsx:getProperty name="request" property="USERID"  
/></userid><passwd><tsx:getProperty name="request" property="PASSWD"  
/></passwd></tsx:dbconnect><FORM NAME="EmployeeForm" ACTION="employeeRepeatResults.jsp"  
METHOD="post" ENCODE="application/x-www-form-urlencoded"><h2>Add Employee Record</h2><P>To add a new  
employee record to the database, submit the following data:</P><TABLE><TR VALIGN="TOP"  
ALIGN="LEFT"><TD><B><I>Employee Number:<br>(1 to 6 characters)</I></B></TD><TD><INPUT TYPE="text"  
NAME="EMPNO"> </TD></TR><TR VALIGN="TOP" ALIGN="LEFT"><TD><B><I>First name:</I></B></TD><TD><INPUT  
TYPE="text" NAME="FIRSTNAME" VALUE="First Name"><BR></TD></TR><TR VALIGN="TOP"  
ALIGN="LEFT"><TD><B><I>Middle Initial:</I></B></TD><TD><INPUT TYPE="text" NAME="MIDINIT"  
VALUE="M"><BR></TD></TR><TR VALIGN="TOP" ALIGN="LEFT"><TD><B><I>Last Name: </I></B></TD><TD><INPUT  
TYPE="text" NAME="LASTNAME" VALUE="Last Name"><BR></TD></TR><TR VALIGN="TOP" ALIGN="LEFT"><TD><%--  
Query the database to get the list of departments --%><tsx:dbquery id="qs" connection="conn" >  
select * from DEPARTMENT </tsx:dbquery><B><I>Department:</I></B></TD><TD><SELECT NAME="WORKDEPT"  
><tsx:repeat> <OPTION VALUE= "<tsx:getProperty name="qs" property="DEPTNO" />" ><tsx:getProperty  
name="qs" property="DEPTNAME" /></tsx:repeat></SELECT></TD></TR><TR VALIGN="TOP"  
ALIGN="LEFT"><TD><B><I>Education:</I></B></TD><TD><SELECT NAME="EDLEVEL"><OPTION VALUE="1"  
SELECTED>BS<OPTION VALUE="2">MS<OPTION VALUE="3">PhD</SELECT></TD></TR></TABLE><INPUT TYPE="submit"  
NAME="Submit" VALUE="Update"><INPUT TYPE="hidden" NAME="USERID" VALUE="<tsx:getProperty  
name="request" property="USERID" />"><INPUT TYPE="hidden" NAME="PASSWD" VALUE="<tsx:getProperty  
name="request" property="PASSWD" />"></FORM><HR><FORM NAME="EmployeeForm"  
ACTION="jsp10employeeRepeatResults.jsp" METHOD="post"  
ENCODE="application/x-www-form-urlencoded"><h2>View Employees by Department</h2><P>To view records  
for employees by department, select the department and submit the query:</P><TABLE><TR VALIGN="TOP"  
ALIGN="LEFT"><TD><B><I>Department:</I></B></TD><TD><%-- Use the bean generated by earlier QUERY tag  
--%><SELECT NAME="WORKDEPT" ><tsx:repeat> <OPTION VALUE= "<tsx:getProperty name="qs"  
property="DEPTNO" />" ><tsx:getProperty name="qs" property="DEPTNAME"  
/></tsx:repeat></SELECT></TD></TR></TABLE><INPUT TYPE="submit" NAME="Submit" VALUE="Query"><INPUT  
TYPE="hidden" NAME="USERID" VALUE="<tsx:getProperty name="request" property="USERID" />"><INPUT  
TYPE="hidden" NAME="PASSWD" VALUE="<tsx:getProperty name="request" property="PASSWD"  
/>"></FORM><HR></BODY></HTML>
```

JSP code example - EmployeeRepeatResults

```
<HTML><HEAD><TITLE>JSP Employee Results</TITLE></HEAD><H1><CENTER>EMPLOYEE RESULTS</CENTER></H1><BODY><tsx:dbconnect id="conn"
url="jdbc:db2:sample" driver="COM.ibm.db2.jdbc.app.DB2Driver"><userid><tsx:getProperty name="request"
property=request.getParameter("USERID") /></userid><passwd><tsx:getProperty name="request"
property=request.getParameter("PASSWD") /></passwd></tsx:dbconnect><% if ( ( request.getParameter("Submit")).equals("Update") ) >
{ %><tsx:dbmodify connection="conn" > INSERT INTO EMPLOYEE (EMPNO,FIRSTNME,MIDINIT,LASTNAME,WORKDEPT,EDLEVEL) VALUES (
'<tsx:getProperty name="request" property=request.getParameter("EMPNO") />', '<tsx:getProperty name="request"
property=request.getParameter("FIRSTNME") />', '<tsx:getProperty name="request" property=request.getParameter("MIDINIT") />',
'<tsx:getProperty name="request" property=request.getParameter("LASTNAME") />', '<tsx:getProperty name="request"
property=request.getParameter("WORKDEPT") />', '<tsx:getProperty name="request" property=request.getParameter("EDLEVEL") />
</tsx:dbmodify> <B><UL>UPDATE SUCCESSFUL</UL></B> <BR><BR><tsx:dbquery id="qs" connection="conn" > select * from Employee
where WORKDEPT= '<tsx:getProperty name="request" property=request.getParameter("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="FIRSTNME" /></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 Employee where WORKDEPT= '<tsx:getProperty name="request" property=request.getParameter("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="FIRSTNME"
/></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>
```

4.2.3: Incorporating XML

IBM WebSphere Application Server provides XML Document Structure Services, which consist of a document parser, a document validator, and a document generator for server-side XML processing.

See article 4.1.1.2 for all of the details about XML support in the product. If you are just becoming familiar with XML, start with article 0.33, a primer on XML concepts, vocabulary, and uses.

Other related information provides guidance on the following topics:

- Structure -- defining and obeying the syntax for an XML tag set
- Content -- determining the mechanism for filling XML tags with data
- Presentation -- determining the mechanism for formatting and displaying XML content

In addition, some special topics are covered, including DOM objects and manipulation of Channel Definition Format (CDF) files as illustrated by the SiteOutliner example.

When you install IBM WebSphere Application Server, the core XML APIs are automatically added to the appropriate class path, enabling you to serve static XML documents as soon as the product is installed.

To serve XML documents that are dynamically generated, use the core APIs to develop servlets or Web applications that generate XML documents (for example, the applications might read the document content from a database) and then deploy those components on your application server.

4.2.3.2: Specifying XML document structure

The structure of an XML document is governed by syntax rules for its tag set. Those tags are defined formally in an XML-based grammar, such as a Document Type Definition (DTD). At the time of this publication, DTD is the most widely-implemented grammar. Therefore, this article discusses options for using DTDs.

Options for XML document structure include:

Do not use a DTD. Not using a DTD enables maximum flexibility in evolving XML document structure, but this flexibility limits the ability to share the documents among users and applications. An XML document can be parsed without a DTD. If the parser does not find an inline DTD or a reference to an external DTD, the parser proceeds using the actual structure of the tags within the document as an implied DTD. The processor evaluates the document to determine whether it meets the rules for well-formedness.

Use a public DTD. Various industry and other interest groups are developing DTDs for categories of documents, such as chemical data and archival documents. Many of these DTDs are in the public domain and are available over the Internet. Using an industry standard DTD maximizes sharing documents among applications that act on the grammar. If the standard DTD does not accommodate the schema the applications need, flexibility is limited.

Several industry and interest groups have developed and proposed DTD grammars for the types of documents they produce and exchange. To make it easier for you to use those grammars, local copies are installed with the product. Use the grammars as examples in developing your own grammars as well as for creating and validating XML documents of those types. The library is located at [product_installation_root\web\xml\grammar\](#)

Develop a DTD. If none of the public DTDs meet an enterprise's needs and enforcing document validity is a requirement, the XML implementers can develop a DTD. Developing a DTD requires careful analysis of the information (data) that the documents will contain.

For DTD updates, visit the XML Industry Portal. For details about the DTD specifications and sample DTDs, refer to IBM's developerWorks site for education and other DTD resources.

4.2.3.3: Providing XML document content

The content of an XML document is the actual data that appears within the document tags. XML implementers must determine the source and the mechanism for putting the data into the document tags. The options include:

Static content. XML document content is created and stored on the Web server as static files. The XML document author composes the document to include valid XML tags and data in a manner similar to how HTML authors compose static HTML files. This approach works well for data that is not expected to change or that will change infrequently. Examples are journal articles, glossaries, and literature.

Dynamically generated content. XML document content can be dynamically generated from a database and user input. In this scenario, XML-capable servlets, Java beans, and even inline Java code within a JavaServer Page (JSP) file can be used to generate the XML document content.

A hybrid of static and dynamically generated content. This scenario involves a prudent combination of static and dynamically generated content.

You can also use XSL to add to or remove information from existing XML content. For details, see the Related information.

4.2.3.4: Rendering XML documents

Options for presenting XML documents include:

Present the XML document in an XML-enabled browser. An XML-enabled browser can parse a document, apply its XSL stylesheet, and present the document to the user. Searching and enabling users to modify an XML document are other possible functions of XML-enabled browsers.

Present the XML document to a browser that converts XML to HTML. Until XML-enabled browsers are readily available, presenting XML documents to users will involve converting the XML document to HTML. That conversion can be handled by conversion-capable browsers. Another option is to use JavaScript or ActiveX controls embedded within the XML document. Microsoft Internet Explorer Version 5 is an XML-to-HTML converter. HTML is not the only format to which XML documents can be converted. It's just the easiest to implement given the commercially available browsers and user agents.

Send an HTML file to the browser. If the users do not have XML-capable browsers, the XML document must be converted at the server before being transmitted to the browser. The server-side XML application that handles the conversion could also determine the capability of the browser before converting the document to HTML, to avoid unnecessary processing if the browser is XML-capable. The XSL processor included with this product supports such server-side functions.

Using XSL to convert XML documents to other formats

IBM WebSphere Application Server includes the Lotus XSL processor and its open-source version, Xalan, for formatting and converting XML documents. Processing can be done at the server or at the browser, to HTML or to other XML-compliant markup languages. For sample code, see the Xalan documentation.

Converting XML documents at the server

One option for presenting an XML document is for the server to convert the XML document to HTML and return the HTML document to the client. On the server side, this typically requires the creation of a servlet to handle the processing of one data stream (the XML document) with another (the XSL document). The output of that process is then forwarded back to the browser.

Server-side processing often requires the passing in of parameters through the XSL processor to customize the output. For an example, see the Xalan documentation.

4.2.3.6: Using DOM to incorporate XML documents into applications

The Document Object Model (DOM) is an API for representing XML and HTML documents as objects that can be accessed by object-oriented programs, such as business logic, for the purposes of creating, navigating, manipulating, and modifying the documents.

Article 0.33.3 introduces DOM concepts and vocabulary. Article 4.1.1.2 tells you where to find the DOM specification and `org.w3c.dom` package.

Article 4.2.3.6.1 provides a quick reference so that you can jump right into DOM development, referring to the package and specification as needed.

4.2.3.6.1: Quick reference to DOM object interfaces

This section highlights a few of the object interfaces. Refer to the DOM Specification for details (see article 4.1.1.2).

Node methods

Node methods include:

Method	Description
hasChildNodes	Returns a boolean to indicate whether a node has children
appendChild	Appends a new child node to the end of the list of children for a parent node
insertBefore	Inserts a child node before the existing child node
removeChild	Removes the specified child node from the node list and returns the node
replaceChild	Replaces the specified child node with the specified new node and returns the new node

Document methods

The Document object represents the entire XML document. Document methods include:

Method	Description
createElement	Creates and returns an Element (tag) of the type specified. If the document will be validated against a DTD, that DTD must contain an Element declaration for the created element.
createTextNode	Creates a Text node that contains the specified string
createComment	Creates a Comment node with the specified content (enclosed within <!-- and --> tags)
createAttribute	Creates an Attribute node of the specified name. Use the setAttribute method of Element to set the value of the Attribute. If the document will be validated against a DTD, that DTD must contain an Attribute declaration for the created attribute.
createProcessingInstruction	Creates a Processing Instruction with the specified name and data (enclosed within <? and ?> tags). A processing instruction is an instruction to the application (such as an XML document formatter) that receives the XML document.

Element methods

Element node methods include:

Method	Description
getAttribute	Returns the value of the specified attribute or empty string
setAttribute	Adds a new attribute-value pair to the element

removeAttribute	Removes the specified attribute from the element
getElementsByName	Returns a list of the element descendants that have the specified tag name

A Text node can be a child of an Element or Attribute node and contains the textual content (character data) for the parent node. If the content does not include markup, all of the content is placed within a single Text node. If the content includes markup, that markup is placed in one or more Text nodes that are siblings of the Text node that contains the non-markup content.

The Text node extends the CharacterData interface, which has methods for setting, getting, replacing, inserting, and making other modifications to a Text node. In addition to those methods, the Text node adds a method:

Method	Description
splitText	Splits the Text node at the specified offset. Returns a new Text node, which contains the original content starting at the offset. The original Text node contains the content from the beginning to the offset.

4.2.3.7: SiteOutliner sample

The SiteOutliner servlet illustrates how to use the XML Document Structure Services to generate and view a Channel Definition Format (CDF) file for a target directory on the servlet's Web server. Use Lotus Notes 5 (the Headlines page), Microsoft Internet Explorer 4 Channel Bar, PointCast, Netscape Navigator 4.06, or other CDF-capable viewers to view and manipulate the CDF file.

SiteOutliner is part of the WebSphere Samples Gallery. When you open the gallery, follow the links to SiteOutliner to run it on your local machine.

4.2.4: Accessing data

This section discusses data access programming:the JDBC 2.0 specification,WebSphere-specific enhancements,best practices, error handling,and tips for specific databases.

4.2.4.2: Obtaining and using database connections

IBM WebSphere Application Server Version 4.0 provides two options for accessing database connections:

- Programming directly to the connection pooling model through the JDBC 2.0 Optional Package API
- Use of the IBM data access beans, which also use connection pooling but give you additional ability to manipulate result sets

WebSphere Application Server versions earlier than 4.0 also supported the connection manager model, which was based on JDBC 1.0. If your Web applications used the connection manager model, you must migrate these in Version 4.0 to use connection pooling.

IBM WebSphere Application Server also provides data access beans, which offer a rich set of features for working with relational database queries and result sets.

For a comprehensive treatment of WebSphere connection pooling and data access, be sure to read the IBM whitepaper to be published [on the Web](#) during the summer of 2001.

Considerations for DB2/390

Speak with your DB2/390 administrator about setting the RRULOCK parameter to YES. This ensures that `SELECT . . . FOR UPDATE` statements get an update lock rather than a sharable lock. If your database is using sharable locks and you attempt to commit updates later, the database can become deadlocked.

4.2.4.2.1: Accessing data with the JDBC 2.0 Optional Package APIs

In JDBC 1.0 and the JDBC 2.0 Core API, the `DriverManager` class is used exclusively for obtaining a connection to a database. The database URL, user ID, and password are used in the `getConnection()` call. In the JDBC 2.0 Optional Package API, the `DataSource` object provides a means for obtaining connections to a database. The benefit of using `DataSource` is that the creation and management of the connection factory is centralized. Applications do not need to have specific information like the database name, user ID, or password in order to obtain a connection to the database.

The steps for obtaining and using a connection with the JDBC 2.0 Optional Package API differ slightly from those in the JDBC 2.0 Core API example. Using the extensions, you access a relational database as follows:

1. Retrieve a `DataSource` object from the JNDI naming service
2. Obtain a `Connection` object from the `DataSource`
3. Send SQL queries or updates to the database management system
4. Process the results

The connection obtained from the `DataSource` is a pooled connection. This means that the `Connection` object is obtained from a pool of connections managed by IBM WebSphere Application Server. The following code fragment shows how to obtain and use a connection with the JDBC 2.0 Optional Package API:

```
try { // Retrieve a DataSource through the JNDI Naming Service      java.util.Properties parms = new
java.util.Properties();    parms.setProperty(Context.INITIAL_CONTEXT_FACTORY,
"com.ibm.websphere.naming.WsnInitialContextFactory");    // Create the Initial Naming Context
javax.naming.Context ctx = new javax.naming.InitialContext(parms);    // Lookup through the naming
service to retrieve a DataSource object    javax.sql.DataSource ds =
(javax.sql.DataSource)ctx.lookup("java:comp/env/jdbc/SampleDB");    // Obtain a Connection from the
DataSource    java.sql.Connection conn = ds.getConnection(); // query the database
java.sql.Statement stmt = conn.createStatement();    java.sql.ResultSet rs =
stmt.executeQuery("SELECT EMPNO, FIRSTNME, LASTNAME FROM SAMPLE"); // process the results    while
(rs.next()) {        String empno = rs.getString("EMPNO");        String firstnme =
rs.getString("FIRSTNME");        String lastname = rs.getString("LASTNAME");    // work with results
}} catch (java.sql.SQLException sqle) { // handle SQLException} finally {    try {        if (rs !=
null) rs.close();    }    catch (java.sql.SQLException sqle) { // can ignore    }    try {        if
(stmt != null) stmt.close();    }    catch (java.sql.SQLException sqle) { // can ignore    }    try {
if (conn != null) conn.close();    }    catch (SQLException sqle) { // can ignore    }} // end
finally
```

In the previous example, the first action is to retrieve a `DataSource` object from the JNDI namespace. This is done by creating a `Properties` object of parameters used to set up an `InitialContext` object. After a context is obtained, a lookup on the context is performed to find the specific `DataSource` necessary, in this case, `SampleDB`.

(In this example, it is assumed the `DataSource` has already been created and bound into JNDI by the WebSphere administrator. For information about doing this in application code, see the Related information.)

After a `DataSource` object is obtained, the application code calls `getConnection()` on the `DataSource` to get a `Connection` object. After the connection is acquired, the querying and processing steps are the same as for the JDBC 2.0 Core API example.

4.2.4.2.1.1: Creating datasources with the WebSphere connection pooling API

IBM WebSphere Application Server provides a public API to enable you to configure a WebSphere datasource in application code. This is necessary only when the application must create a datasource on demand. Otherwise, the datasource is configured by the administrator in the administrative console.

The complete API specification can be found in javadoc for the class `com.ibm.websphere.advanced.cm.factory.DataSourceFactory`. See the Related information.

To create a datasource on demand in an application, the application must do the following:

1. Create a Properties object with datasource properties
2. Obtain a datasource from the factory
3. Bind the datasource into JNDI

The following code fragment shows how an application would create a datasource and bind it into JNDI:

```
import com.ibm.websphere.advanced.cm.factory.DataSourceFactory;...try {    // Create a properties
file for the DataSource    java.util.Properties prop = new java.util.Properties();
prop.put(DataSourceFactory.NAME, "SampleDB");    prop.put(DataSourceFactory.DATASOURCE_CLASS_NAME,
"COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource");    prop.put(DataSourceFactory.DESCRPTION, "My
sample
datasource");    prop.put("databaseName", "sample");// Obtain a DataSource from
the factory    DataSource ds = DataSourceFactory.getDataSource(prop);// Bind the DataSource into JNDI
DataSourceFactory.bindDataSource(ds);} catch (ClassNotFoundException cnfe) {// check the class path
for all necessary classes} catch (CMFactoryException cmfe) {// Example of exception: incorrect
properties} catch (NamingException ne) {// Example of exception:  datasource by this name may
already exist}
```

To create a datasource for binding into JNDI, the application must first create a Properties object to hold the DataSource configuration properties. The only properties required for the datasource from a WebSphere perspective are:

- **NAME** - The name of the datasource. This is used to identify the datasource when it is bound into JNDI.
- **DATASOURCE_CLASS_NAME** - The complete name of the DataSource class that can be found in the JDBC resource archive file (often referred to as the JDBC provider package). This DataSource class will be used to create connections to the database. The class specified here must implement `javax.sql.ConnectionPoolDataSource` or `javax.sql.XADataSource`.

However, depending on the DataSource class specified in the **DATASOURCE_CLASS_NAME** property, there may be other vendor-specific properties required. In this example, the `databaseName` property is also required, because `DB2ConnectionPoolDataSource` is being used. For more information on these vendor-specific properties, see the vendor's documentation for the complete list of properties supported for a datasource.

After a properties object is created, the application can create a new DataSource object by calling `getDataSource()` on the factory, passing in the Properties object as a parameter. This creates an object of type DataSource, but it is not yet bound into JNDI. To bind a datasource into JNDI, call `bindDataSource()` on the factory. At this point, other applications can share the datasource by retrieving it from JNDI with the name property specified on creation.

All other APIs specific to connection pooling are not public APIs. Applications that use a WebSphere datasource should follow the JDBC 2.0 Core and JDBC 2.0 Optional Package APIs.

4.2.4.2.1.2: Tips for using connection pooling

Most best practices have been documented elsewhere in Related information. The following are additional items that have not been explicitly called out:

Obtain and close connection in the same method. An application should obtain and close its connection in the method that requires the connection. This keeps the application from holding resources not being used and leaves more available connections in the pool for other applications. In addition, this practice removes the temptation to use the same connection in multiple transactions, which, by default, is not allowed. This practice does not cost the application much in performance, because the Connection object is from a pool of connections, where the overhead for establishing the connection has already been incurred. Lastly, make sure to declare the Connection object in the same method as the getConnection() call in a servlet; otherwise, the Connection object works as if it is a static variable (see "Worst Practices" later in this article for problems with this).

If you opened it, close it. All JDBC resources that have been obtained by an application should be explicitly closed by that application. The product tries to clean up JDBC resources on a connection after the connection has been closed. However, this behavior should not be relied upon, especially if the application might be migrated to another platform in the future.

For servlets, obtain DataSource in the init() method. For performance reasons, it is usually a good idea to put the JNDI lookup for the datasource into the init() method of the servlet. Because the datasource is simply a factory for connections that does not typically change, retrieving it in this method ensures that the lookup happens only once.

Worst practices

The following are some very common problems with applications that should be avoided, because they most often result in unexpected failures:

Do not close connections in a finalize() method. If an application waits to close a connection or other JDBC resource until the finalize() method, the connection is not closed until the object that obtained it is garbage-collected. This leads to problems if the application is not very thorough about closing its JDBC resources, such as ResultSet objects. Databases can quickly run out of the memory required to store the information about all of the JDBC resources it currently has open. In addition, the pool can quickly run out of connections to service other requests.

Do not declare connections as static objects. It is never recommended that connections be declared as static objects. If a connection is declared as static, the same connection might get used on different threads at the same time. This causes a great deal of difficulty, within both the product and the database.

In servlets, do not declare Connection objects as instance variables. In a servlet, all variables declared as instance variables act as if they are class variables. For example, in a servlet with an instance variable

```
Connection conn = null;
```

this variable acts as if it were static. In this case, all instances of the servlet use the same Connection object. This is because a single servlet instance can be used to serve multiple Web requests in different threads.

In CMP beans, do not manage data access. If a Container Managed Persistence (CMP) bean is written so that it manages its own data access, this data access may be part of a global transaction. Generally, if specialized data access is required, use a BMP session bean.

4.2.4.2.1.3: Handling data access exceptions

For data access, the standard Java exception class to catch is `java.sql.SQLException`. IBM WebSphere Application Server monitors for specific SQL exceptions thrown from the database. Several of these exceptions are mapped to WebSphere-specific exceptions. The product provides WebSphere-specific exceptions to ease development by not requiring you to know all of the database-specific SQL exceptions that could be thrown in typical situations. In addition, monitoring SQL exceptions enables the product and application to recover from common problems like intermittent network or database outages.

ConnectionWaitTimeoutException

This exception (`com.ibm.ejs.cm.pool.ConnectionWaitTimeoutException`) indicates that the application has waited for the `connectionTimeout` (`CONN_TIMEOUT`) number of seconds and has not been returned a connection. This can occur when the pool is at its 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 user ID and password are different or it is in a different transaction. The following code fragment shows how to use this exception:

```
java.sql.Connection conn = null;
javax.sql.DataSource ds = null;
...try { // Retrieve a DataSource
through the JNDI Naming Service      java.util.Properties parms = new java.util.Properties();
setProperty.put(Context.INITIAL_CONTEXT_FACTORY,
"com.ibm.websphere.naming.WsnInitialContextFactory"); // Create the Initial Naming Context
javax.naming.Context ctx = new      javax.naming.InitialContext(parms); // Lookup through the
naming service to retrieve a DataSource object      javax.sql.DataSource ds =
(javax.sql.DataSource)ctx.lookup("java:comp/env/jdbc/SampleDB"); conn = ds.getConnection();
// work on connection} catch (com.ibm.ejs.cm.pool.ConnectionWaitTimeoutException cw) { // notify the
user that the system could not provide a // connection to the database} catch
(java.sql.SQLException sqle) { // deal with exception}
```

In all cases in which the `ConnectionWaitTimeoutException` is caught, there is very little to do in terms of recovery. It usually doesn't make sense to retry the `getConnection()` method, because if a longer wait time is required, the `connectionTimeout` datasource property should be set higher. Therefore, if this exception is caught by the application, the administrator should review the expected usage of the application and tune the connection pool and the database accordingly.

StaleConnectionException

This exception (`com.ibm.websphere.ce.cm.StaleConnectionException`) indicates that the connection currently being held is no longer valid. This can occur for numerous reasons, including:

- The application fails to get a connection because of a problem such as the database not being started.
- A connection is no longer usable because of a database failure. When an application tries to use a connection it previously obtained, the connection is no longer valid. In this case, all connections currently in use by the application may prompt this exception.
- The application using the connection has already called `close()` and then tries to use the connection again.
- The connection has been orphaned, and the application tries to use the orphaned connection.
- The application tries to use a JDBC resource, such as `Statement`, obtained on a now-stale connection.

When application code catches `StaleConnectionException`, it should take explicit steps to handle the exception. `StaleConnectionException` extends `SQLException`, so it can be thrown from any method that is declared to throw `SQLException`. The most common occasion for a `StaleConnectionException` to be thrown is the first time a connection is used, just after it has been 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 with 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. Typically, this occurs because all connections currently allocated to an application are marked stale; the more connections the application has, the more connections can be stale.

Generally, when a `StaleConnectionException` is caught, the transaction in which the connection was involved needs to be rolled back and a new transaction begun with a new connection.

For more information and detailed code samples, be sure to read the IBM whitepaper to be published [on the Web](#) during the summer of 2001.

4.2.4.2.2: Accessing data with the JDBC 2.0 Core API

WebSphere applications can access a relational database directly through a JDBC provider that uses the JDBC 2.0 Core API. You access a relational database in this manner as follows:

1. Establish a connection through the DriverManager class
2. Send SQL queries or updates to the database management system
3. Process the results

Only a single connection is obtained. This connection does not belong to a pool of connections and is not managed by IBM WebSphere Application Server. It is the responsibility of the application to manage the use of this connection.

The following code fragment shows a simple example of obtaining and using a connection directly through a JDBC provider:

```
try { // establish a connection through the DriverManager
Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");   String url = "jdbc:db2:sample";   String username
= "dbuser";   String password = "passwd";   java.sql.Connection conn =
java.sql.DriverManager.getConnection(url, username, password); // query the database
java.sql.Statement stmt = conn.createStatement();   java.sql.ResultSet rs =
stmt.executeQuery("SELECT EMPNO, FIRSTNME, LASTNAME FROM SAMPLE");   // process the results   while
(rs.next()) {       String empno = rs.getString("EMPNO");       String firstnme =
rs.getString("FIRSTNME");       String lastname = rs.getString("LASTNAME");   // work with results
}} catch (java.sql.SQLException sqle) { // handle the SQLException} finally {   try {       if(rs !=
null) rs.close();   }   catch (SQLException sqle) {       // can ignore   }   try {       if(stmt !=
null) stmt.close();   }   catch (SQLException sqle) {       // can ignore   }   try {       if(conn !=
null) conn.close();   }   catch (SQLException sqle) {       // can ignore   }}
```

In the previous example, the first action is to establish a connection to the database. This is done by loading and registering the JDBC driver and then requesting a connection from DriverManager. DriverManager, a JDBC 1.0 class, is the basic service for managing a set of JDBC drivers. It is necessary to load the driver class before the call to getConnection(), because DriverManager can establish a connection only to a driver that has registered with it. Loading the driver class also registers it with DriverManager.

After a connection has been obtained, the database is queried by creating a statement and executing a query on that statement. The query results are put into a ResultSet object.

Lastly, the results are processed by stepping through the result set and pulling the data from each record retrieved.

According to the JDBC 2.0 Core API specification, the DriverManager class has been deprecated. Therefore, any application using this class should be rewritten to use WebSphere connection pooling, which uses the datasource method described in the JDBC 2.0 Optional Package API to obtain connections to the database. For more information, see the JDBC 2.0 Core API specification.

4.2.4.2.3: Accessing relational databases with the IBM data access beans

Java programs that access JDBC-compliant relational databases typically use the classes and methods in the `java.sql` package to access data. Instead of using the `java.sql` package, you can use the classes and methods in the package `com.ibm.db`, the IBM data access beans. This gives you additional features for data access beyond those available in the `java.sql` package.

The Related information discusses what the data access beans are, their advantages, and how to use them. A data access bean uses a connection that you provide to it, such as a connection from a connection pool that you get through a `DataSource` object.

4.2.4.2.3.1: Example: Servlet using data access beans

The sample servlet uses the data access beans and is based on the sample servlet discussed in Article 4.2.4.2.1.1. The connection pooling sample servlet uses classes such as `Connection`, `Statement`, and `ResultSet` from the `java.sql` package to interact with the database. In contrast, this sample servlet uses the data access beans, instead of the classes in the `java.sql` package, to interact with the database. For convenience, call this sample servlet the DA (for data access beans) and call the sample servlet on which it is based the CP (for connection pooling).

The CP and DA sample servlets benefit from the performance and resource management enhancements made possible by connection pooling. The programmer coding the DA sample servlet benefits from the additional features and functions provided by the data access beans.

The DA sample servlet differs slightly from the CP sample servlet. This discussion covers only the changes. See [Article 4.2.4.2.1.1](#) for the discussion of the CP sample servlet. The DA sample servlet shows the basics of connection pooling and the data access beans, but keeps other code to a minimum. Therefore, the servlet is not entirely realistic. You are expected to be familiar with basic servlet and JDBC coding.

The changes

This section describes how the DA sample servlet differs from the CP sample servlet. To view the coding in one or both of the samples while you read this section, click these links:

- [DA sample](#)
- [CP sample](#)

Steps 1 through 6 of the CP sample servlet are mostly unchanged in the DA sample servlet. The main changes to the DA sample servlet are:

- New package

The `com.ibm.db` package (containing the data access beans classes) must be imported. The classes are in the `databeans.jar` file, found in the `lib` directory under the Application Server root install directory. You will need this jar file in your `CLASSPATH` in order to compile a servlet using the data access beans.

- The `metaData` variable

This variable is declared in the Variables section at the start of the code, outside of all methods. This allows a single instance to be used by all incoming user requests. The full specification of the variable is completed in the `init()` method.

- The `init()` method

New code has been appended to the `init()` method to do a one-time initialization on the `metaData` object when the servlet is first loaded. The new code begins by creating the base query object `sqlQuery` as a `String` object. Note the two `"?"` parameter placeholders. The `sqlQuery` object specifies the base query within the `metaData` object. Finally, the `metaData` object is provided higher levels of data (`metadata`), in addition to the base query, that will help with running the query and working with the results. The code sample shows:

- The `addParameter()` method notes that when running the query, the parameter `idParm` is supplied as a Java Integer datatype, for the convenience of the servlet, but that `idParm` should be converted (through the `metaData` object) to do a query on the SMALLINT relational datatype of the underlying relational data when running the query.

A similar use of the `addParameter()` method for the `deptParm` parameter notes that for the same underlying SMALLINT relational datatype, the second parameter will exist as a different Java

datatype within the servlet - as a String rather than as an Integer. Thus parameters can be Java datatypes convenient for the Java application and can automatically be converted by the metaData object to be consistent with the required relational datatype when the query is run.

Note that the "?" parameter placeholders in the sqlQuery object and the addParameter() methods are related. The first addParameter() attaches idParm to the first "?", and so on. Later, a setParameter() will use idParm as an argument to replace the first "?" in the sqlQuery object with an actual value.

- The addColumn() method performs a function somewhat similar to the addParameter() method. For each column of data to be retrieved from the relational table, the addColumn() method maps a relational datatype to the Java datatype most convenient for use within the Java application. The mapping is used when reading data out of the result cache and when making changes to the cache (and then to the underlying relational table).
- The addTable() method explicitly specifies the underlying relational table. This information is needed if changes to the result cache are to be propagated to the underlying relational table.

● Step 5

Step 5 has been rewritten to use the data access beans to do the SQL query instead of the classes in the java.sql package. The query is run using the selectStatement object, which is a SelectStatement data access bean.

Step 5 is part of the process of responding to the user request. When steps 1 through 4 have run, the conn Connection object from the connection pool is available for use. The code shows:

1. The dataAccessConn object (a DatabaseConnection bean) is created to establish the link between the data access beans and the database connection - the conn object.
2. The selectStatement object (a SelectStatement bean) is created, pointing to the database connection through the dataAccessConn object, and pointing to the query through the metaData object.
3. The query is "completed" by specifying the parameters using the setParameter() method. The "?" placeholders in the sqlQuery string are replaced with the parameter values specified.
4. The query is executed using the execute() method.
5. The result object (a SelectResult bean) is a cache containing the results of the query, created using the getResult() method.
6. The data access beans offer a rich set of features for working with the result cache - at this point the code shows how the first row of the result cache (and the underlying relational table) can be updated using standard Java coding, without the need for SQL syntax.
7. The close() method on the result cache breaks the link between the result cache and the underlying relational table, but the data in the result cache is still available for local access within the servlet. After the close(), the database connection is unnecessary. Step 6 (which is unchanged from the CP sample servlet) closes the database connection (in reality, the connection remains open but is returned to the connection pool for use by another servlet request).

● Step 7

Step 7 has been entirely rewritten (with respect to the CP sample servlet) to use the query result cache retrieved in Step 5 to prepare a response to the user. The query result cache is a SelectResult data access bean.

Although the result cache is no longer linked to the underlying relational table, the cache can still be accessed for local processing. In this step, the response is prepared and sent back to the user. The code shows the following:

- The nextRow() and previousRow() methods are used to navigate through the result cache.

Additional navigation methods are available.

- The getColumnValue() method is used to retrieve data from the result cache. Because of properties set earlier in creating the metaData object, the data can be easily cast to formats convenient for the needs of the servlet.

A possible simplification

If you do not need to update the relational table, you can simplify the sample servlet:

- At the end of the init() method, you can drop the lines with the addColumn() and addTable() methods, since the metaData object does not need to know as much if there are no relational table updates.
- You will also need to drop the lines with the setColumnValue() and updateRow() methods at the end of step 5, because you are no longer updating the relational table.
- Finally, you can remove most of the type casts associated with the getColumnValue() methods in step 7. You will, however, need to change the type cast to (Short) for the "ID" and "DEPT" use of the getColumnValue() method.

4.2.4.2.4: Database access by servlets and JSP files

Servlets using getConnection() to access a data source

When used without parameters, getConnection() assumes the default user ID and password for a data source. The WebSphere administrative clients do not offer a way to configure a default user ID and password for a data source to be used by a servlet.

Therefore, servlets using getConnection() to access a data source should specify a user ID and password:

```
getConnection(userid,password);
```

4.2.4.4.1: Providing Web clients access to JSP files

Suppose an application contains one or more JSP files -- how does the application developer allow a user at a Web client (browser) to invoke the JSP files? The table summarizes the available approaches. Click an approach for details.


Programming approach	How user accesses JSP file
Provide the JSP file URL to users for direct access, or include an HREF link to the JSP file on the Web site	Type the JSP URL in a browser, or follow a link to it
Call JSP file from an HTML form	Fill out an HTML form and submit it to the JSP file for processing

4.2.4.4.1.1: Invoking servlets and JSP files by URLs

Users can invoke a servlet or JSP file by its URL, using a browser to open:

`http://your.server.name/application_web_path/servlet_or_JSP_web_path`

Users must be provided with the URL to use in order to invoke the servlet. See the Related information to learn how to determine the URL.

 Appending `/$/foo` to the URL allows you to access the servlet URL, but the URL is then misunderstood by the runtime environment. This type of URL may create a security exposure. The best practice for securing servlets is to follow the Java security specifications documented in the [Securing applications](#) section.

Note that in order for servlets to be invoked by their class names, the administrator must have manually enabled the option while configuring the Web application to which the servlet belongs.

4.2.4.4.1.2: Invoking servlets and JSP files within HTML forms

A Web page can be designed so that users can invoke a servlet or JSP file from an HTML form. An HTML form enables a user to enter data on a Web page (from a browser) and submit the data to a servlet, or a servlet generated by a JSP file.

The HTML FORM tag has attributes for specifying how to invoke the servlet or JSP file:

FORM attribute	Description
METHOD	Indicates how user information is to be submitted.
ACTION	Indicates the URL used to invoke the servlet or JSP file

If the information entered by the user is to be submitted to a *servlet* by a GET or POST method, the servlet must override the doGet() method or doPost() method. For JSP files, the override is not necessary. The same service method that is called whether the form is submitted using GET or POST.

Examples

Using GET:

```
<FORM METHOD="GET" ACTION="/application_Web_path/servlet_Web_path"><!-- HTML tags for text entry areas, buttons, and other prompts go here --></FORM>
```

Using POST:

```
<FORM METHOD="POST" ACTION="application_Web_path/servlet_Web_path"><!-- HTML tags for text entry areas, buttons, and other prompts go here --></FORM>
```

4.2.4.4.1.2.1: Example: Invoking servlets within HTML forms

Suppose the application programmer uses an HTML form to provide users access to a servlet. Assuming the METHOD attribute on the FORM tag is "GET," the flow is as follows:

1. The user views the form in a browser. The user provides information requested by the form and specifies to submit the form (usually by clicking a Submit button or other button visible on the form).
2. The form encodes the user-supplied information into a URL-encoded query string. It appends the query string to the servlet URL and submits the entire URL.
3. The servlet processes the information. The `getParameterNames()`, `getParameter()`, and `getParameterValues()` methods of the `HttpServletRequest` object provide access to the form parameter names and values in the client request. The extraction process also decodes the names and values.
4. Often, the final action of the servlet is to dynamically create an `HTMLresponse` (based on parameter input from the form) and pass it back to the user through the server. Methods of the `HttpServletResponse` object are used to send the response, which is sent back to the client as a complete HTML page.

4.2.4.4.2: Providing Web clients access to servlets

Suppose an application contains one or more servlets -- how does the application developer allow a user at a Web client (browser) to invoke the servlets? The table summarizes the available approaches. Click an approach for details.

Programming approach	How user accesses servlet
Provide the servlet URL to users for direct access, or include an HREF link to the servlet URL on the Web site	Type the servlet URL in a browser, or follow a link to it
Call servlet from an HTML form	Fill out an HTML form and submit it to the servlet for processing
Call servlet from a JSP file	Open a JSP page that invokes the servlet

4.2.4.4.2.2: Invoking servlets within JSP files

Users can invoke servlets from within JavaServer Page (JSP)files. Application developers should consult the JavaServer Pages (JSP)reference for a complete description of the JSP syntax.

To invoke a JSP file, a user can either:

- Use a Web browser to open the JSP file
- Use a Web browser to invoke a servlet that invokes the JSP file

4.2.5: Using the Bean Scripting Framework

Most Web developers are familiar with using scripting languages to generate user-cued HTML pages or to create new browser windows.

The *Bean Scripting Framework* (BSF) enables developers to use scripting language functions in their Java, server-side applications. It also extends scripting languages so that existing Java classes and Java beans can now be invoked from that language.

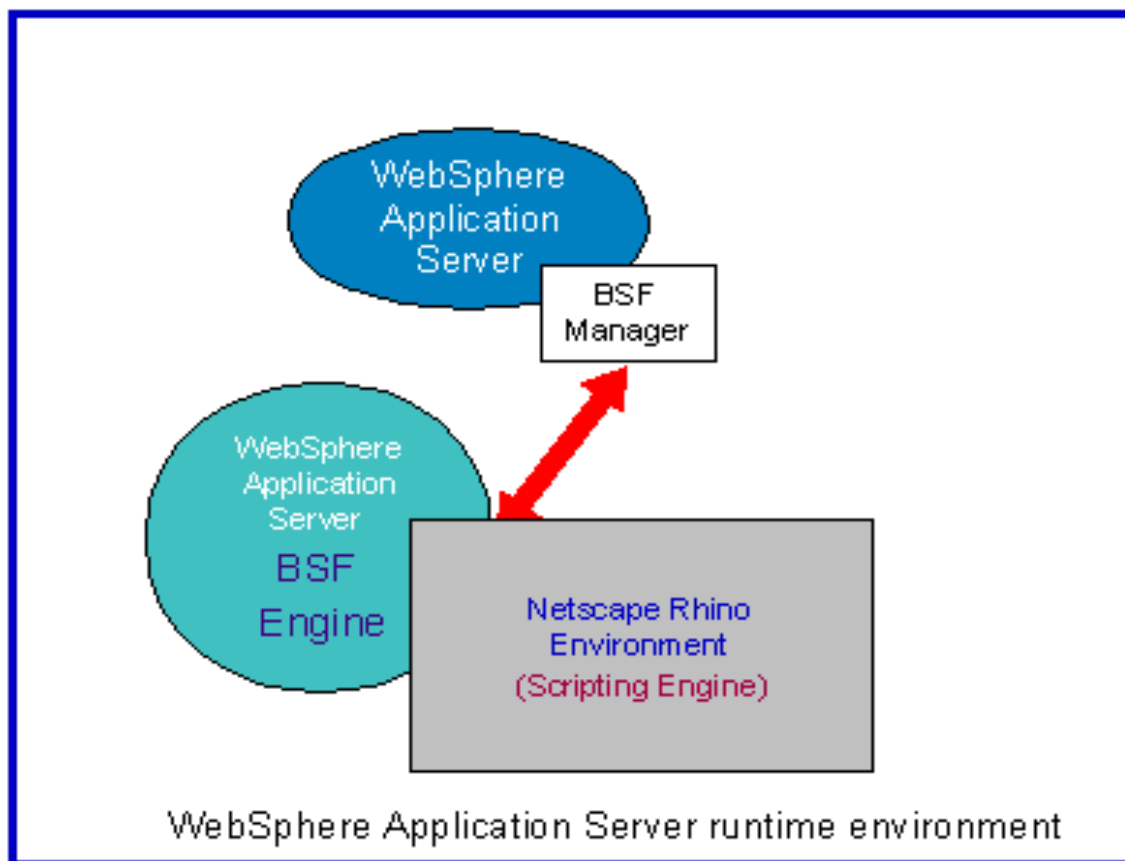
With BSF, scripts can now create, manipulate and access values from Java objects and, conversely, Java programs can now evaluate and access results from scripts.

BSF components:

WebSphere Application Server provides the *Bean Scripting Framework* (BSF), which consists of a BSF Manager and a BSF Engine, and a scripting engine which is the *Rhino* version 1.5 environment from Netscape.

JavaScript from Netscape is the only language supported by WebSphere Application Server's implementation of BSF.

The relationship of the BSF components is illustrated in the following graphic:



Features of BSF:

The *BSF Manager* is a bean that provides scripting services for the application and support services for the scripting engine to enable it to interact with the JVM.

The *BSF Engine* is an interface that allows a specific scripting language, in this case Netscape's JavaScript, to become part of the bean scripting framework.

Visit the [BSF project Web site](#) for news on the latest updates to BSF functionality.

See article "BSF examples and samples" when you are ready to delve into programming examples.

4.2.5.1: BSF examples and samples

There are no WebSphere Application Server implementation restrictions on using BSF. Invoke BSF as you would any other Web application, using the instructions in the article [Installing application files](#) to administer your application.

To test these code samples, from a Browser window, copy the code samples and paste them into your own file. You can use any file name, but the file extension must be **.jsp**. To see the results, the file must be served from a server with a JSP engine, such as WebSphere Application Server.

The following steps and code samples describe how to implement BSF:

1. [Create a JSP file](#)
2. [Change the Java code to JavaScript](#)
3. [Add the required BSF tag](#) as illustrated in the [View 2 sample](#)
4. [Add the file to the Web application document root directory](#)
5. Invoke the code.

See the file [JSP access models](#) for more JSP information.

1. Create a JSP file that looks like this next example:

```
<html> <head> <title> Temperature Table using Java >/title> </head> <body> <h1>Temperature Table using Java</h1> <p>American tourists visiting Canada can use this handy temperature table which converts from Fahrenheit to Celsius: <br> <br> <table BORDER COLS=2 WIDTH="20%" > <tr BGCOLOR="#FFFF00"> <th>Fahrenheit</th> <th>Celsius</th> </tr> <% for (int i=0; i<101; i+=10) { out.println ("<tr ALIGN=RIGHT BGCOLOR=\"#CCCCC\">"); out.println ("<td>" + i + "</td>"); out.println ("<td>" + ((i - 32)*5/9) + "</td>"); out.println ("</tr>"); } %> </table> <p><i> <%= new java.util.Date () %> </i></p> </body> </html>
```

2. Change the Java code in the previous file to JavaScript so the file now looks like the following example:

```
<%@ page language="javascript" %>
<html> <head> <title> Temperature Table using JavaScript >/title> </head> <body>
<h1>Temperature Table using JavaScript</h1> <p>American tourists visiting Canada can use this handy temperature table which converts from Fahrenheit to Celsius: <br> <br> <table BORDER COLS=2 WIDTH="20%" > <tr BGCOLOR="#FFFF00"> <th>Fahrenheit</th> <th>Celsius</th> </tr> <% for (var i=0; i<101; i+=10) { out.println ("<tr ALIGN=RIGHT BGCOLOR=\"#CCCCC\">"); out.println ("<td>" + i + "</td>"); out.println ("<td>" + Math.round((i - 32)*5/9) + "</td>"); out.println ("</tr>"); } %> </table> <p><i> <%= new java.util.Date () %> </i></p> </body> </html>
```

3. The only BSF-specific tag that is required in your file is

<%@ page language="javascript" %>

This tag identifies the language to BSF. [View 2](#) illustrates where this tag is located in the file.

4.3: Developing enterprise beans

Enterprise applications are applications that typically use enterprise beans. To develop enterprise applications, you must:

1. [Develop any session or entity beans your application will use](#)
2. [Create the deployment descriptor and the EJB JAR file.](#)
3. [Deploy the enterprise beans.](#)

Enterprise applications support both [transactions and security](#).

Writing Enterprise Beans is a programming guide for developing, packaging, and deploying enterprise beans in IBM WebSphere Application Server. It discusses both the Advanced Edition and Enterprise Edition of the product.

Format
PDF
HTML

About this book

This document focuses on the development of enterprise beans written to the Sun Microsystems Enterprise JavaBeans^(TM) specification in the WebSphere^(TM) Application Server programming environment. It also discusses development of EJB clients that can access enterprise beans.

Who should read this book

This document is written for developers and system architects who want an introduction to programming enterprise beans and EJB clients in WebSphere Application Server. It is assumed that programmers are familiar with the concepts of object-oriented programming, distributed programming, and Web-based programming. Knowledge of the Sun Microsystems Java^(TM) programming language is also assumed.

Document organization

This document is organized as follows:

- [An architectural overview of the EJB programming environment](#) provides a high-level introduction to the EJB server environment in WebSphere Application Server.
 - [An introduction to enterprise beans](#) explains the main concepts associated with enterprise beans.
 - [Tools for developing and deploying enterprise beans](#) explains how to set up and use the tools used in developing and deploying enterprise beans.
 - [Developing enterprise beans](#) explains how to develop entity beans with container-managed persistence (CMP) and session beans. It also provides information on how to package enterprise beans for later deployment.
 - [Enabling transactions and security in enterprise beans](#) explains how to enable transactions in enterprise beans by using the appropriate deployment descriptor attributes.
 - [Developing EJB clients](#) explains the basic code required by an EJB client to use an enterprise bean. This chapter covers generic issues relevant to enterprise beans, Java applications, and Java servlets that use enterprise beans.
 - [Developing servlets that use enterprise beans](#) discusses the basic code required in a servlet that accesses an enterprise bean.
 - [More-advanced programming concepts for enterprise beans](#) explains how to develop a simple entity bean with bean-managed persistence and discusses the basic code required of an enterprise bean that manages its own transactions.
 - [Appendix A, Changes for version 1.1 of the EJB specification](#) describes features that are new or have changed in version 1.1 of the EJB specification and discusses migration issues for enterprise beans written to version 1.0 of the EJB specification.
 - [Appendix B, Example code provided with WebSphere Application Server](#) describes the major example used throughout this book and the additional examples that are delivered with the various editions of WebSphere Application Server.
 - [Appendix C, Extensions to the EJB Specification](#) describes the extensions to the EJB Specification that are specific to WebSphere Application Server. Use of these extensions is supported in VisualAge for Java only.
-

Related information

For further information on the topics discussed in this manual, see the following documents:

- [Getting Started with WebSphere Application Server](#)
 - [Building Business Solutions with WebSphere](#)
-

How to send your comments

Your feedback is important in helping to provide the most accurate and highest quality information. If you have any comments about this book, send your comments by e-mail to wasdoc@us.ibm.com. Be sure to include the name of the book, the document number of the book, the edition and version of WebSphere Application Server, and, if applicable, the specific location of the information you are commenting on (for example, a page number or table number).

An introduction to enterprise beans

This chapter looks at the characteristics and purpose of enterprise beans. It describes the two basic types of enterprise beans and their life cycles, and it provides an example of how enterprise beans can be combined to create distributed, three-tiered applications.

Bean basics

An enterprise bean is a Java component that can be combined with other enterprise beans and other Java components to create a distributed, three-tiered application. There are two types of enterprise beans:

- An *entity* bean encapsulates permanent data, which is stored in a data source such as a database or a file system, and associated methods to manipulate that data. In most cases, an entity bean must be accessed in some transactional manner. Instances of an entity bean are unique and they can be accessed by multiple users.

For example, the information about a bank account can be encapsulated in an entity bean. An account entity bean might contain an account ID, an account type (checking or savings), and a balance variable and methods to manipulate these variables.

- A *session* bean encapsulates ephemeral (nonpermanent) data associated with a particular EJB client. Unlike the data in an entity bean, the data in a session bean is not stored in a permanent data source, and no harm is caused if this data is lost. However, a session bean can update data in an underlying database, usually by accessing an entity bean. A session bean can also participate in a transaction.

When created, instances of a session bean are identical, though some session beans can store semipermanent data that makes them unique at certain points in their life cycle. A session bean is always associated with a single client; attempts to make concurrent calls result in an exception being thrown.

For example, the task associated with transferring funds between two bank accounts can be encapsulated in a session bean. Such a transfer session bean can find two instances of an account entity bean (by using the account IDs), and then subtract a specified amount from one account and add the same amount to the other account.

Entity beans

This section discusses the basics of entity beans.

Basic components of an entity bean

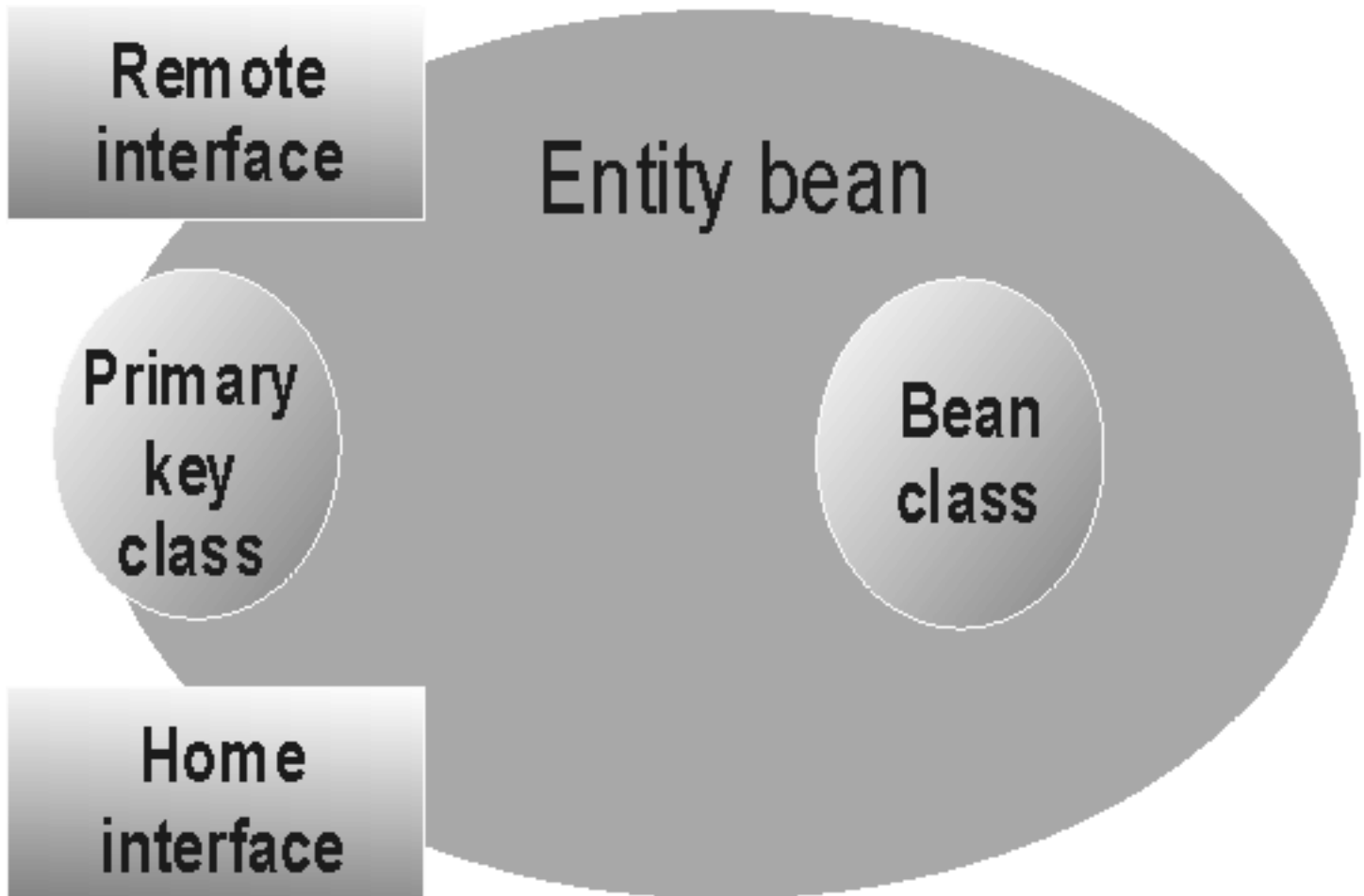
Every entity bean must have the following components, which are illustrated in [Figure 3](#):

- *Bean class*--This class encapsulates the data for the entity bean and contains the developer-implemented business methods that access the data. It also contains the methods used by the container to manage the life cycle of an entity bean instance. EJB clients (whether they are other enterprise beans or user components such as servlets) *never* access objects of this class directly; instead, they use the container-generated classes associated with the home and remote interfaces to manipulate the entity bean instance.
- *Home interface*--This interface defines the methods used by the client to create, find, and remove instances of the entity bean. This interface is implemented by the container during deployment in a class known generically as the *EJB home class*; instances are referred to as *EJB home objects*.
- *Remote interface*--Once the client has used the home interface to gain access to an entity bean, it uses this interface to invoke indirectly the business methods implemented in the bean class. This interface is

implemented by the container during deployment in a class known generically as the *EJB object class*; instances are referred to as *EJB objects*.

- *Primary key* -- One or more variables that uniquely identify a specific entity bean instance. A primary key that consists of a single variable of a primitive Java data type can be specified at deployment. A *primary key class* is used to encapsulate primary keys that consist of multiple variables or more complex Java data types. The primary key class also contains methods to create primary key objects and manipulate those objects.

Figure 3. The components of an entity bean



Data persistence

Entity beans encapsulate and manipulate *persistent* (or permanent) business data. For example, at a bank, entity beans can be used to model customer profiles, checking and savings accounts, car loans, mortgages, and customer transaction histories.

To ensure that this important data is not lost, the entity bean stores its data in a data source such as a database. When the data in an enterprise bean instance is changed, the data in the data source is synchronized with the bean data. Of course, this synchronization takes place within the context of the appropriate type of transaction, so that if a router goes down or a server fails, permanent changes are not lost. When you design an entity bean, you must decide whether you want the enterprise bean to handle this data synchronization or whether you want the container to handle it. An enterprise bean that handles its own data synchronization is said to implement *bean-managed persistence* (BMP), while an enterprise bean whose data synchronization is handled by the container is said to implement *container-managed persistence* (CMP).

Unless you have a good reason for implementing BMP, it is recommended that you design your entity beans to

use CMP. The code for an enterprise bean with CMP is easier to write and does not depend on any particular data storage product, making it more portable between EJB servers. However, you must use entity beans with BMP if you want to use a data source that is not supported by the EJB server.

Session beans

This section discusses the basics of session beans.

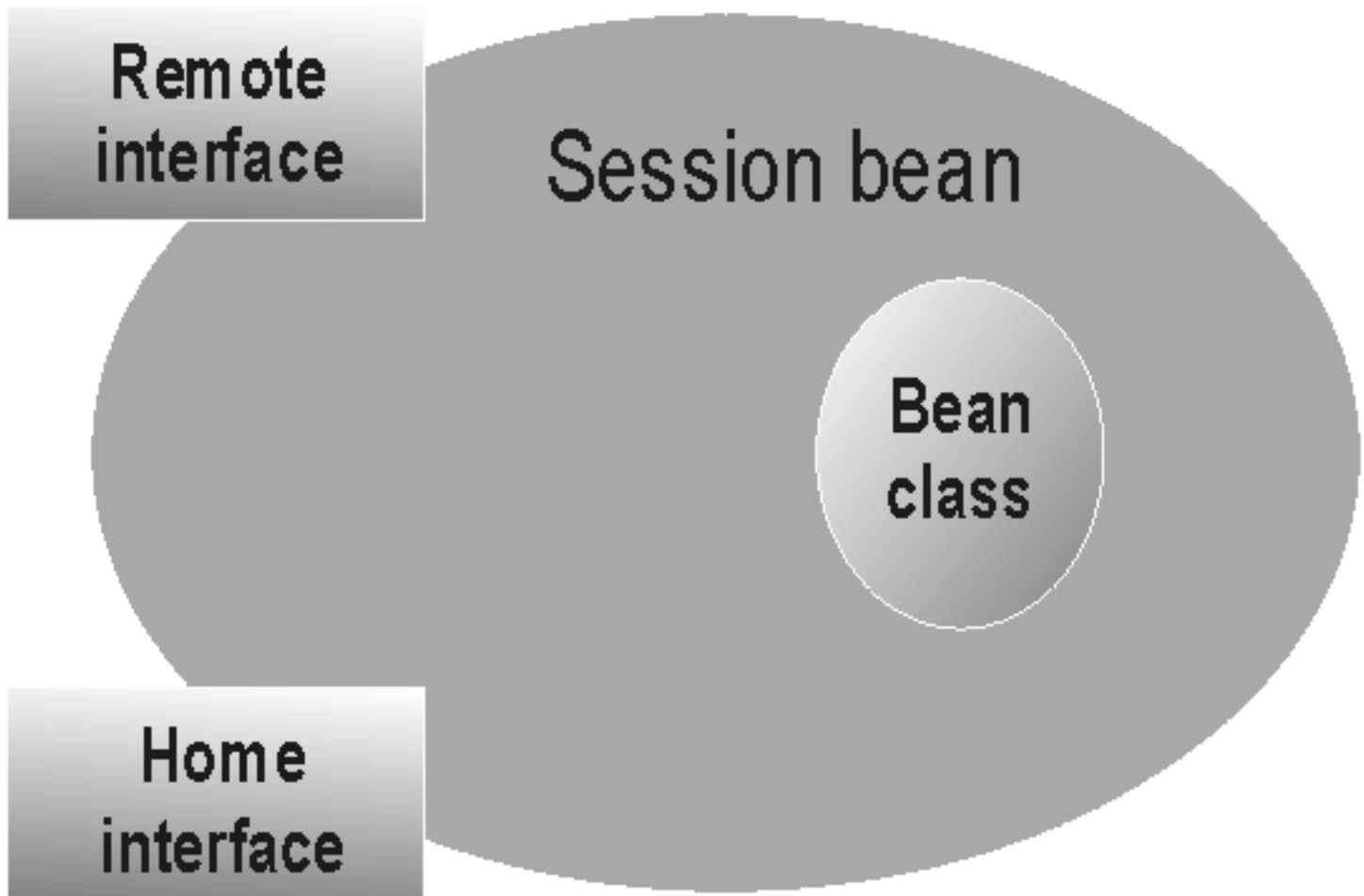
Basic components of a session bean

Every session bean must have the following components, which are illustrated in [Figure 4](#):

- *Bean class*--This class encapsulates the data associated with the session bean and contains the developer-implemented business methods that access this data. It also contains the methods used by the container to manage the life cycle of an session bean instance. EJB clients (whether they are other enterprise beans or user applications) *never* access objects of this class directly; instead, they use the container-generated classes associated with the home and remote interfaces to manipulate the session bean.
- *Home interface*--This interface defines the methods used by the client to create and remove instances of the session bean. This interface is implemented by the container during deployment in a class known generically as the *EJB home class*; instances are referred to as *EJB home object*.
- *Remote interface*--After the client has used the home interface to gain access to an session bean, it uses this interface to invoke indirectly the business methods implemented in the bean class. This interface is implemented by the container during deployment in a class known generically as the *EJB object class*; instances are referred to as *EJB objects*.

Unlike an entity bean, a session bean does not have a primary key class. A session bean does not require a primary key class because you do not need to search for specific instances of session beans.

Figure 4. The components of a session bean



Stateless versus stateful session beans

Session beans encapsulate data and methods associated with a user session, task, or ephemeral object. By definition, the data in a session bean instance is ephemeral; if it is lost, no real harm is done. For example, at a bank, session beans can represent a funds transfer, the creation of a customer profile or new account, and a withdrawal or deposit. If information about a fund transfer is already typed (but not yet committed), and a server fails, the balances of the bank accounts remains the same. Only the transfer data is lost, which can always be retyped.

The manner in which a session bean is designed determines whether its data is shorter lived or longer lived:

- If a session bean needs to maintain specific data across methods, it is referred to as a *stateful* session bean. When a session bean maintains data across methods, it is said to have a *conversational state*. A Web-based shopping cart is a classic use of a stateful session bean. As the shopping cart user adds items to and subtracts items from the shopping cart, the underlying session bean instance must maintain a record of the contents of the cart. After a particular EJB client begins using an instance of a stateful session bean, the client must continue to use that instance as long as the specific state of that instance is required. If the session bean instance is lost before the contents of the shopping cart are committed to an order, the shopper must load a new shopping cart.
- If a session bean does not need to maintain specific data across methods, it is referred to as a *stateless* session bean. The example Transfer session bean developed in [Developing session beans](#) provides an example of a stateless session bean. For stateless session beans, a client can use any instance to invoke any of the session bean's methods because all instances are the same.

Creating an EJB module

The last step in the development of an enterprise bean is the creation of an EJB module. An EJB module consists of the following:

- One or more deployable enterprise beans.
- A deployment descriptor, stored in an Extensible Markup Language (XML) file. This file contains information about the structure and external dependencies of the beans in the module, and application assembly information describing how the beans are to be used in an application.

The EJB module can be created by using the tools within an integrated development environment (IDE) like IBM's VisualAge for Java Enterprise Edition or by using the tools contained in WebSphere. For more information, see [Tools for developing and deploying enterprise beans](#).

The EJB module

The *EJB module* is used to assemble enterprise beans into a single deployable unit; this file uses the standard Java archive file format. The EJB module can contain individual enterprise beans or multiple enterprise beans. For more information, see [Creating an EJB module and deployment descriptor](#).

The deployment descriptor

The EJB module contains one or more deployable enterprise beans and one *deployment descriptor*. The deployment descriptor contains attribute and environment settings for each bean in the module, and it defines how the container invokes functionality for all beans in the module. The deployment descriptor attributes can be set for the entire enterprise bean or for the individual methods in the bean. The container uses the definition of the bean-level attribute unless a method-level attribute is defined, in which case the latter is used. The deployment descriptor contains the following information about entity and session beans. These attributes can be set on the bean only; they cannot be set on a specific method of the bean.

- The bean's name, class, home interfaces, remote interfaces, and bean type (entity or session).
- *Primary key class* attribute--Identifies the primary key class for the bean. For more information, see [Writing the primary key class \(entity with CMP\)](#) or [Writing or selecting the primary key class \(entity with BMP\)](#).
- *Persistence management*. Specifies whether persistence management is performed by the enterprise bean or by the container.
- *Container-managed fields* attribute--Lists those persistent variables in the bean class that the container must synchronize with fields in a corresponding data source to ensure that this data is persistent and consistent. For more information, see [Defining variables](#).
- *Reentrant* attribute--Specifies whether an enterprise bean can invoke methods on itself or call another bean that invokes a method on the calling bean. Only entity beans can be reentrant. For more information, see [Using threads and reentrancy in enterprise beans](#).
- *State management* attribute--Defines the conversational state of the session bean. This attribute must be set to either STATEFUL or STATELESS. For more information on the meaning of these conversational states, see [Stateless versus stateful session beans](#).
- *Timeout* attribute--Defines the idle timeout value in seconds associated with this session bean. (This attribute is an extension to the standard deployment descriptor.)
- References to external resources, such as resource connection factories, to the homes of other enterprise beans, and to security roles.

The deployment descriptor contains the following application assembly information:

- A display name and icons for identifying the module.
- The location of class files needed for a client program to access the beans in the module.

- *Security roles*-- Define a logical grouping of principals. Access to operations (such as EJB methods) is controlled by granting access to a role.
- *Method permissions*--Define a mapping between one or more security roles and one or more methods that a member of the role can invoke. This value is set per method.
- *Transaction* attributes--Define the transactional manner in which the container invokes a method for enterprise beans that require container-managed transaction demarcation. This value is set per method. The values for this attribute are described in [Enabling transactions and security in enterprise beans](#).
- *Transaction isolation level* attribute--Defines the degree to which transactions are isolated from each other by the container. This value is set per method. The values for this attribute are described in [Enabling transactions and security in enterprise beans](#). (This attribute is an extension to the standard deployment descriptor.)
- *RunAsMode* and *RunAsIdentity* attributes--The *RunAsMode* attribute defines the identity used to invoke the method. If a specific identity is required, the *RunAsIdentity* attribute is used to specify that identity. This value is set per bean. The values for the *RunAsMode* attribute are described in [Enabling transactions and security in enterprise beans](#). (This attribute is an extension to the standard deployment descriptor.)

The following binding attribute is stored in the repository (it is not part of the deployment descriptor):

- *JNDI home name* attribute--Defines the Java Naming and Directory Interface (JNDI) home name that is used to locate instances of an EJB home object. This value is set per bean. The values for this repository attribute are described in [Creating and getting a reference to a bean's EJB object](#).

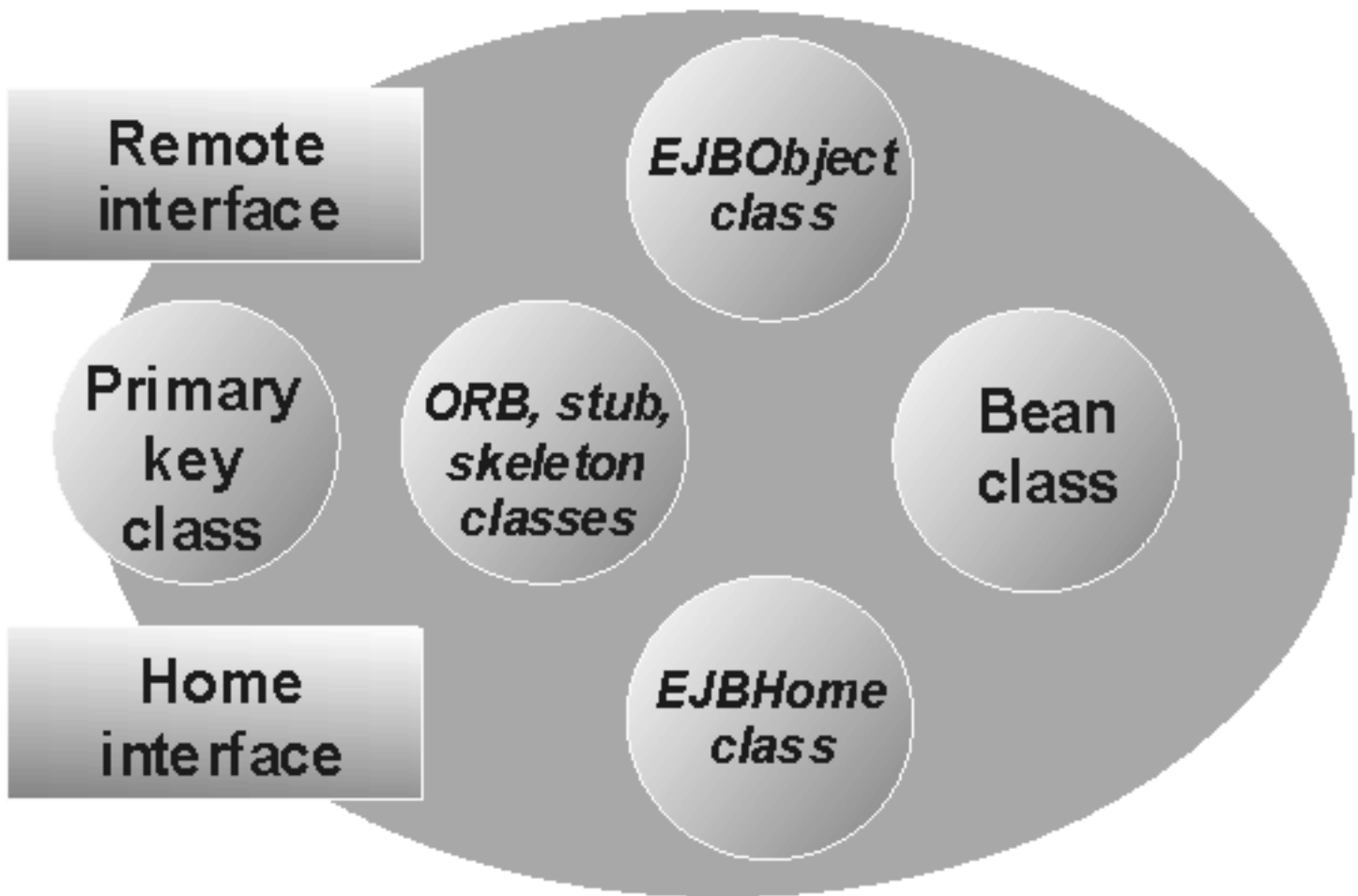
Deploying an EJB module

When you deploy an EJB module, the deployment tool creates or incorporates the following elements:

- The container-implemented *EJBObject* and *EJBHome* classes (hereafter referred to as the EJB object and EJB home classes) from the enterprise bean's home and remote interfaces (and the persister and finder classes for entity beans with CMP).
- The stub and skeleton files required for remote method invocation (RMI).

[Figure 5](#) shows a simplified version of a deployed entity bean.

Figure 5. The major components of a deployed entity bean

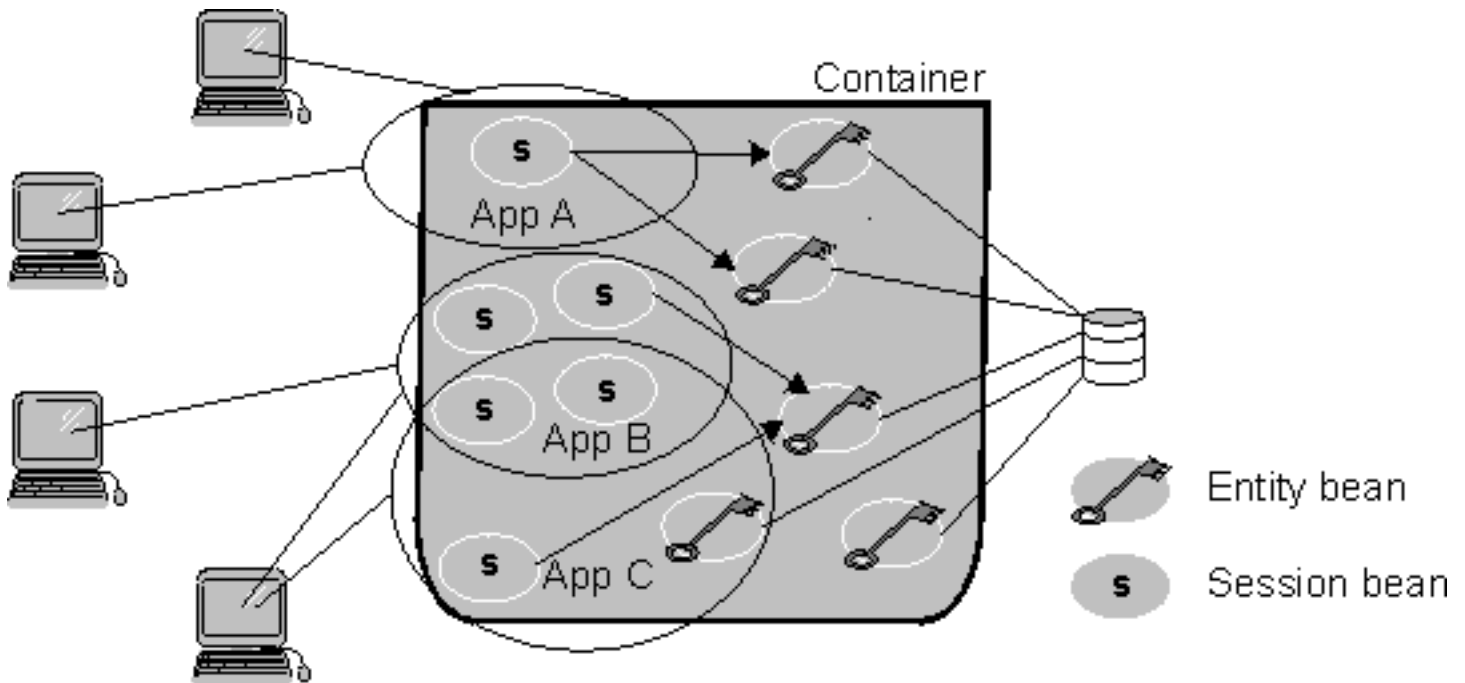


You can deploy an EJB module with a variety of different tools. For more information, see [Tools for developing and deploying enterprise beans](#).

Developing EJB applications

To create EJB applications, create the enterprise beans and EJB clients that encapsulate your business data and functionality and then combine them appropriately. [Figure 6](#) provides a conceptual illustration of how EJB applications are created by combining one or more session beans, one or more entity beans, or both. Although individual entity beans and session beans can be used directly in an EJB client, session beans are designed to be associated with clients and entity beans are designed to store persistent data, so most EJB applications contain session beans that, in turn, access entity beans.

Figure 6. Conceptual view of EJB applications



This section provides an example of the ways in which enterprise beans can be combined to create EJB applications.

An example: enterprise beans for a bank

If you develop EJB applications for the banking industry, you can develop the following entity beans to encapsulate your business data and associated methods:

- Account bean--An entity bean that contains information about customer checking and savings accounts.
- CarLoan bean--An entity bean that contains information about an automobile loan.
- Customer bean--An entity bean that contains information about a customer, including information on accounts held and loans taken out by the customer.
- CustomerHistory bean--An entity bean that contains a record of customer transactions for specified accounts.
- Mortgage bean--An entity bean that contains information about a home or commercial mortgage.

An EJB client can directly access entity beans or session beans; however, the EJB Specification suggests that EJB clients use session beans to in turn access entity beans, especially in more complex applications. Therefore, as an EJB developer for the banking industry, you can create the following session beans to represent client tasks:

- LoanApprover bean--A session bean that allows a loan to be approved by using instances of the CarLoan bean, the Mortgage bean, or both.
- CarLoanCreator bean--A session bean that creates a new instance of a CarLoan bean.
- MortgageCreator bean--A session bean that creates a new instance of a Mortgage bean.
- Deposit bean--A session bean that credits a specified amount to an existing instance of an Account bean.
- StatementGenerator bean--A session bean that generates a statement summarizing the activities associated with a customer's accounts by using the appropriate instances of the Customer and CustomerHistory entity beans.
- Payment bean--A session bean that credits a payment to a customer's loan by using instances of the CarLoan bean, the Mortgage bean, or both.
- NewAccount bean--A session bean that creates a new instance of an Account bean.

- NewCustomer bean--A session bean that creates a new instance of a Customer bean.
- LoanReviewer bean--A session bean that accesses information about a customer's outstanding loans (instances of the CarLoan bean, the Mortgage bean, or both).
- Transfer bean--A session bean that transfers a specified amount between two existing instances of an Account bean.
- Withdraw bean--A session bean that debits a specified amount from an existing instance of an Account bean.

This example is simplified by necessity. Nevertheless, by using this set of enterprise beans, you can create a variety of EJB applications for different types of users by combining the appropriate beans within that application. One or more EJB clients can then be built to access the application.

Using the banking beans to develop EJB banking applications

When using beans built to the Sun Microsystems JavaBeans^(TM) Specification (as opposed to the EJB Specification), you combine predefined components such as buttons and text fields to create GUI applications. When using enterprise beans, you combine predefined components such as the banking beans to create three-tiered applications.

For example, you can use the banking enterprise beans to create the following EJB applications:

- Home Banking application--An Internet application that allows a customer to transfer funds between accounts (with the Transfer bean), to make payments on a loan by using funds in an existing account (with the Payment bean), to apply for a car loan or home mortgage (with the CarLoanCreator bean or the MortgageCreator bean).
- Teller application--An intranet application that allows a teller to create new customer accounts (with the NewCustomer bean and the NewAccount bean), transfer funds between accounts (with the Transfer bean), and record customer deposits and withdrawals (with the Withdraw bean and the Deposit bean).
- Loan Officer application--An intranet application that allows a loan officer to create and approve car loans and home mortgages (with the CarLoanCreator, MortgageCreator, LoanReviewer, and LoanApprover beans).
- Statement Generator application--A batch application that prints monthly customer statements related to account activity (with the StatementGenerator bean).

These examples represent only a subset of the possible EJB applications that can be created with the banking beans.

Life cycles of enterprise bean instances

After an enterprise bean is deployed into a container, clients can create and use instances of that bean as required. Within the container, instances of an enterprise bean go through a defined life cycle. The events in an enterprise bean's life cycle are derived from actions initiated by either the EJB client or the container in the EJB server. You must understand this life cycle because for some enterprise beans, you must write some of the code to handle the different events in the enterprise bean's life cycle.

The methods mentioned in this section are discussed in greater detail in [Developing enterprise beans](#).

Session bean life cycle

This section describes the life cycle of a session bean instance. Differences between stateful and stateless session beans are noted.

Creation state

A session bean's life cycle begins when a client invokes a create method defined in the bean's home interface. In response to this method invocation, the container does the following:

1. Creates a new memory object for the session bean instance.
2. Invokes the session bean's `setSessionContext` method. (This method passes the session bean instance a reference to a session context interface that can be used by the instance to obtain container services and get information about the caller of a client-invoked method.)
3. Invokes the session bean's `ejbCreate` method corresponding to the create method called by the EJB client.

Ready state

After a session bean instance is created, it moves to the ready state of its life cycle. In this state, EJB clients can invoke the bean's business methods defined in the remote interface. The actions of the container at this state are determined by whether a method is invoked transactionally or nontransactionally:

- *Transactional method invocations*--When a client invokes a transactional business method, the session bean instance is associated with a transaction. After a bean instance is associated with a transaction, it remains associated until that transaction completes. (Furthermore, an error results if an EJB client attempts to invoke another method on the same bean instance if invoking that method causes the container to associate the bean instance with another transaction or with no transaction.)

The container then invokes the following methods:

1. The `afterBegin` method, if that method is implemented by the bean class.
2. The business method in the bean class that corresponds to the business method defined in the bean's remote interface and called by the EJB client.
3. The bean instance's `beforeCompletion` method, if that method is implemented by the bean class and if a commit is requested prior to the container's attempt to commit the transaction.

The transaction service then attempts to commit the transaction, resulting either in a commit or a roll back. When the transaction completes, the container invokes the bean's `afterCompletion` method, passing the completion status of the transaction (either commit or rollback).

If a rollback occurs, a stateful session bean can roll back its conversational state to the values contained in the bean instance prior to beginning the transaction. Stateless session beans do not maintain a conversational state, so they do not need to be concerned about rollbacks.

- *Nontransactional method invocations*--When a client invokes a nontransactional business method, the container simply invokes the corresponding method in the bean class.

Pooled state

The container has a sophisticated algorithm for managing which enterprise bean instances are retained in memory. When a container determines that a stateful session bean instance is no longer required in memory, it invokes the bean instance's `ejbPassivate` method and moves the bean instance into a reserve pool. A stateful session bean instance cannot be passivated when it is associated with a transaction.

If a client invokes a method on a passivated instance of a stateful session bean, the container activates the instance by restoring the instance's state and then invoking the bean instance's `ejbActivate` method. When this method returns, the bean instance is again in the ready state.

Because every stateless session bean instance of a particular type is the same as every other instance of that

type, stateless session bean instances are not passivated or activated. These instances exist in a ready state at all times until their removal.

Removal state

A session bean's life cycle ends when an EJB client or the container invokes a remove method defined in the bean's home interface and remote interface. In response to this method invocation, the container calls the bean instance's `ejbRemove` method.

If you attempt to remove a bean instance while it is associated with a transaction, the `javax.ejb.RemoveException` is thrown. After a bean instance is removed, any attempt to invoke a method on that instance causes the `java.rmi.NoSuchObjectException` to be thrown.

A container can implicitly call a remove method on an instance after the lifetime of the EJB object has expired. The lifetime of a session EJB object is set in the deployment descriptor with the *timeout* attribute.

For more information on the remove methods, see [Removing a bean's EJB object](#).

Entity bean life cycle

This section describes the life cycle of entity bean instances. Differences between entity beans with CMP and BMP are noted.

Creation State

An entity bean instance's life cycle begins when the container creates that instance. After creating a new entity bean instance, the container invokes the instance's `setEntityContext` method. This method passes the bean instance a reference to an entity context interface that can be used by the instance to obtain container services and get information about the caller of a client-invoked method.

Pooled State

After an entity bean instance is created, it is placed in a pool of available instances of the specified entity bean class. While the instance is in this pool, it is not associated with a specific EJB object. Every instance of the same enterprise bean class in this pool is identical. While an instance is in this pooled state, the container can use it to invoke any of the bean's finder methods.

Ready State

When a client needs to work with a specific entity bean instance, the container picks an instance from the pool and associates it with the EJB object initialized by the client. An entity bean instance is moved from the pooled to the ready state if there are no available instances in the ready state.

There are two events that cause an entity bean instance to be moved from the pooled state to the ready state:

- When a client invokes the create method in the bean's home interface to create a new and unique entity of the entity bean class (and a new record in the data source). As a result of this method invocation, the container calls the bean instance's `ejbCreate` and `ejbPostCreate` methods, and the new EJB object is associated with the bean instance.
- When a client invokes a finder method to manipulate an existing instance of the entity bean class (associated with an existing record in the data source). In this case, the container calls the bean instance's `ejbActivate` method to associate the bean instance with the existing EJB object.

When an entity bean instance is in the ready state, the container can invoke the instance's `ejbLoad` and `ejbStore`

methods to synchronize the data in the instance with the corresponding data in the data source. In addition, the client can invoke the bean instance's business methods when the instance is in this state. All interactions required to handle an entity bean instance's business methods in the appropriate transactional (or nontransactional) manner are handled by the container.

When a container determines that an entity bean instance in the ready state is no longer required, it moves the instance to the pooled state. This transition to the pooled state results from either of the following events:

- When the container invokes the `ejbPassivate` method.
- When the EJB client invokes a remove method on the EJB object or on the EJB home object. When one of these methods is called, the underlying entity is removed permanently from the data source.

Removal State

An entity bean instance's life cycle ends when the container invokes the `unsetEntityContext` method on an entity bean instance in the pooled state. Do not confuse the removal of an entity bean instance with the removal of the underlying entity whose data is stored in the data source. The former simply removes an uninitialized object; the latter removes data from the data source.

For more information on the remove methods, see [Removing a bean's EJB object](#).

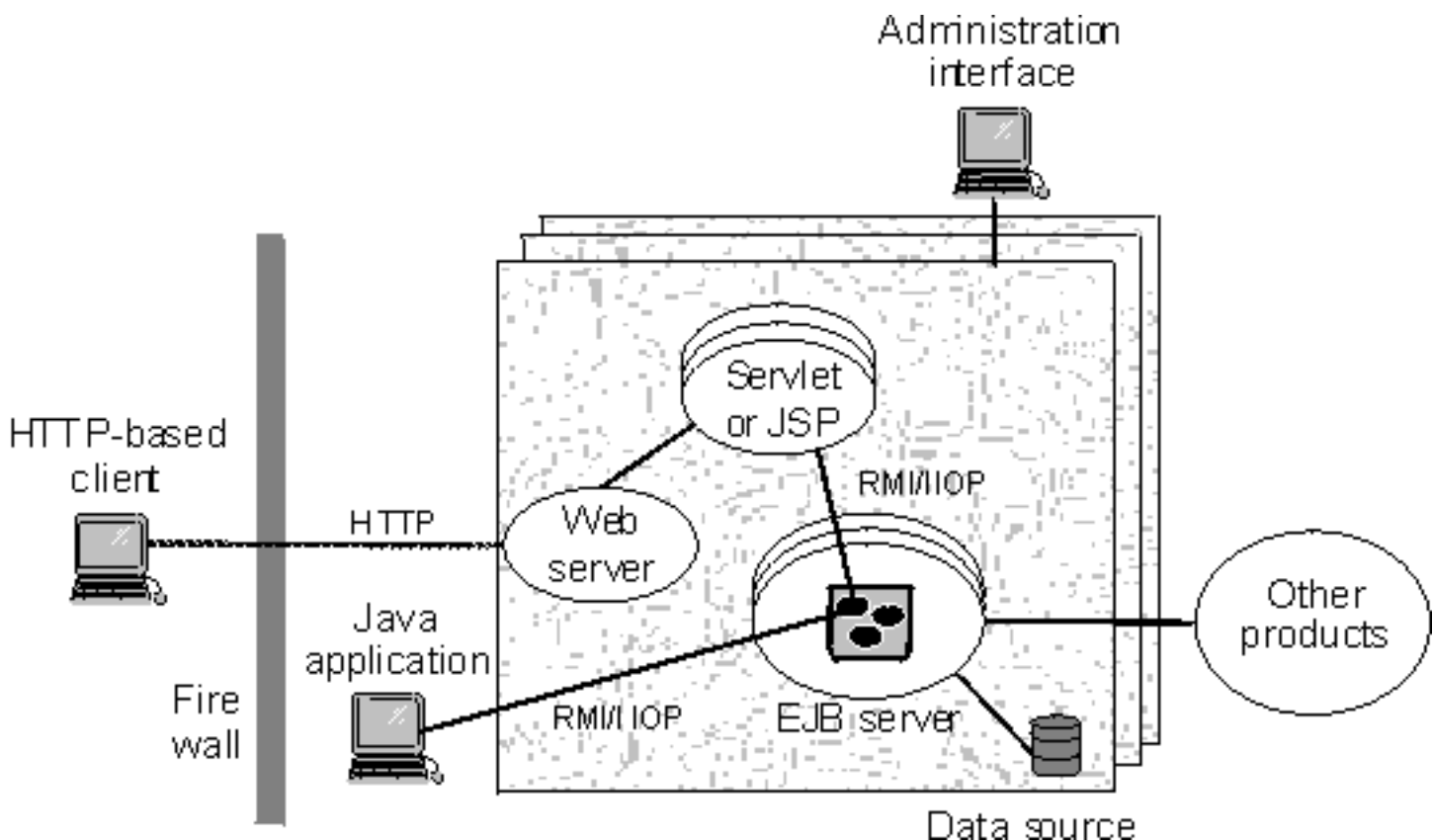
An architectural overview of the EJB programming environment

The World Wide Web (the Web) has transformed the way in which businesses work with their customers. At first, it was good enough just to have a Web home page. Then, businesses began to deploy active Web sites that allowed customers to order products and services. Today, businesses not only need to use the Web in all of these ways, they need to integrate their Web-based systems with their other business systems. The IBM^(R) WebSphere Application Server, and specifically the support for enterprise beans, provides the model and the tools to accomplish this integration.

Components of the EJB environment

IBM's implementation of the Sun Microsystems Enterprise JavaBeans (EJB) Specification enables users of the WebSphere Application Server to integrate their Web-based systems with their other business systems. A major part of this implementation is the WebSphere EJB server and its associated components, which are illustrated in [Figure 1](#).

Figure 1. The components of the EJB environment



The WebSphere EJB server environment contains the following components, which are discussed in more detail in the specified sections:

- **EJB server**--A WebSphere EJB server contains and runs one or more *enterprise beans*, which encapsulate the business logic and data used and shared by EJB clients. The enterprise beans installed in an EJB server do not communicate directly with the server; instead, an *EJB container* provides an interface between the enterprise beans and the EJB server, providing many low-level services such as threading, support for transactions, and management of data storage and retrieval. For more information,

see [The EJB server](#).

- *Data source*--There are two types of enterprise beans: session beans, which encapsulate short-lived, client-specific tasks and objects, and entity beans, which encapsulate permanent or *persistent* data. The EJB server stores and retrieves this persistent data in a data source, which can be a database, another application, or even a file. For more information, see [The data source](#).
- *EJB clients*--There are two general types of EJB clients:
 - *HTTP-based clients* that interact with the EJB server by using either Java servlets or JavaServer Pages^(TM) (JSP) by way of the Hypertext Transfer Protocol (HTTP).
 - *Java applications* that interact directly with the EJB server by using Java remote method invocation over the Internet Inter-ORB Protocol (RMI/IIOP).

For more information, see [The EJB clients](#).

- The *administration interface*--The administrative interface allows you to manage the EJB server environment. For more information, see [The administration interface](#).
-

The EJB server

The EJB server is the application server tier of WebSphere Application Server's three-tier architecture. The EJB server has three components: the EJB server runtime, the EJB containers, and the enterprise beans. EJB containers insulate the enterprise beans from the underlying EJB server and provide a standard application programming interface (API) between the beans and the container. The EJB Specification defines this API. Together, the EJB server and container components provide or give access to the following services for the enterprise beans that are deployed into it:

- A tool that deploys enterprise beans. When a bean is deployed, the deployment tool creates several classes that implement the interfaces that make up the predeployed bean. In addition, the deployment tool generates Java ORB, stub, and skeleton classes that enable remote method invocation. For entity beans, the tool also generates persister and finder classes to handle interaction between the bean and the data source that stores the bean's persistent data. Before an enterprise bean can be deployed, the developer must create an *EJB module* and associated *deployment descriptor*. The deployment descriptor provides information about each enterprise bean in the module and instructions for the container on how to handle the beans. For more information on deployment, see [Deploying an EJB module](#).
- A security service that handles authentication and authorization for principals that need to access resources in an EJB server environment. For more information, see [The security service](#).
- A workload management service that ensures that resources are used efficiently. For more information, see [The workload management service](#).
- A persistence service that handles interaction between an entity bean and its data source to ensure that persistent data is properly managed. For more information, see [The persistence service](#).
- A naming service that exports a bean's name, as defined in the deployment descriptor, into the name space. The EJB server uses the Java Naming and Directory Interface^(TM) (JNDI) to implement a naming service. For more information, see [The naming service](#).
- A transaction service that implements the transactional attributes in a bean's deployment descriptor. For more information, see [The transaction service](#).

The security service

When enterprise computing was handled solely by a few powerful mainframes located at a centralized site, ensuring that only authorized users obtained access to computing services and information was a fairly

straightforward task. In distributed computing systems where users, application servers, and resource managers can be spread out across the world, securing computing resources has become a much more complicated task. Nevertheless, the underlying issues are basically the same.

Authentication and authorization

A good security service provides two main functions: authentication and authorization.

Authentication takes place when a *principal* (a user or a computer process) initially attempts to gain access to a computing resource. At that point, the security service challenges the principal to prove that the principal is who it claims to be. Human users typically prove who they are by entering a user ID and password; a process normally presents an encrypted key. If the password or key is valid, the security service gives the user a *token* or *ticket* that identifies the principal and indicates that the principal has been authenticated. After a principal is authenticated, it can then attempt to use any of the resources within the boundaries of the computing system protected by the security service; however, a principal can use a particular computing resource only if it has been authorized to do so. *Authorization* takes place when an authenticated principal requests the use of a resource and the security service determines if the user has been granted permission to use that resource. Typically, authorization is handled by associating access control lists (ACLs) with resources that define which principal (or groups of principals) are authorized to use the resource. If the principal is authorized, it gains access to the resource.

In a distributed computing environment, principals and resources must be mutually suspicious of each other's identity until both have proven that they are who they say they are. This is necessary because principals can attempt to falsify an identity to get access to a resource, and a resource can be a trojan horse, attempting to get valuable information from the principal. To solve this problem, the security service contains a security server that acts as a *trusted third party*, authenticating principals and resources so that these entities can prove their identities to each other. This security protocol is known as *mutual authentication*.

Using the security server

The security service does *not* use the *access control* and *run-as identity* security attributes defined in the deployment descriptor. However, it does use the *run-as mode* attribute as the basis for mapping a user identity to a user security context. For more information on this attribute, see [The deployment descriptor](#).

The main component of the security service is an EJB server that contains security enterprise beans. When system administrators administer the security service, they manipulate the security beans in the security EJB server.

Once an EJB client is authenticated, it can attempt to invoke methods on the enterprise beans that it manipulates. A method is successfully invoked if the principal associated with the method invocation has the required permissions to invoke the method. These permissions can be set at the application level (an administrator-defined set of Web and object resources) and at the method group level (an administrator-defined set of Java interface/method pairs). An application can contain multiple method groups.

In general, the principal under which a method is invoked is associated with that invocation across multiple Web servers and EJB servers (this association is known as *delegation*). Delegating the method invocations in this way ensures that the user of an EJB client needs to authenticate only once. HTTP cookies are used to propagate a user's authentication information across multiple Web servers. These cookies have a lifetime equal to the life of the browser session, and a logout method is provided to destroy these cookies when the user is finished.

For information on administering security, see the WebSphere InfoCenter and the online help available with the WebSphere Administrative Console.

The workload management service

The workload management service improves the scalability of the EJB server environment by grouping multiple EJB servers into *server groups*. Clients then access these server groups as if they are a single EJB server, and the workload management service ensures that the workload is evenly distributed across the EJB servers in the server groups. An EJB server can belong to only one server group. The creation of server groups is an administrative task that is handled from within the WebSphere Administrative Console. For more information on workload management, consult the WebSphere InfoCenter and the online help for the appropriate administrative interface.

The persistence service

There are two types of enterprise beans: session beans and entity beans. Session beans encapsulate temporary data associated with a particular client. Entity beans encapsulate permanent data that is stored in a data source. For more information, see [An introduction to enterprise beans](#).

The persistence service ensures that the data associated with entity beans is properly synchronized with their corresponding data in the data source. To accomplish this task, the persistence service works with the transaction service to insert, update, extract, and remove data from the data source at the appropriate times.

There are two types of entity beans: those with container-managed persistence (CMP) and those with bean-managed persistence (BMP). In entity beans with CMP, the persistence service handles nearly all of the tasks required to manage persistent data. In entity beans with BMP, the bean itself handles most of the tasks required to manage persistent data.

The persistence service uses the following components to accomplish its task:

- The Java Database Connectivity (JDBCTM) API, which gives entity beans a common interface to relational databases.
- Java transaction support, which is discussed in [Using transactions in the EJB server environment](#). The EJB server ensures that persistent data is always handled within the appropriate transactional context.

The naming service

In an object-oriented distributed computing environment, clients must have a mechanism to locate and identify objects so that the clients, objects, and resources appear to be on the same machine. A naming service provides this mechanism. In the EJB server environment, JNDI is used to mask the actual naming service and provide a common interface to the naming service.

JNDI provides naming and directory functionality to Java applications, but the API is independent of any specific implementation of a naming and directory service. This implementation independence ensures that different naming and directory services can be used by accessing them by way of the JNDI API. Therefore, Java applications can use many existing naming and directory services such as the Lightweight Directory Access Protocol (LDAP), the Domain Name Service (DNS), or the DCE Cell Directory Service (CDS).

JNDI was designed for Java applications by using Java's object model. Using JNDI, Java applications can store and retrieve named objects of any Java object type. JNDI also provides methods for executing standard directory operations, such as associating attributes with objects and searching for objects by using their attributes.

In the EJB server environment, the deployment descriptor is used to specify the JNDI name for an enterprise bean. When an EJB server is started, it registers these names with JNDI.

The transaction service

A *transaction* is a set of operations that transforms data from one consistent state to another. This set of

operations is an indivisible unit of work, and in some contexts, a transaction is referred to as a *logical unit of work* (LUW). A transaction is a tool for distributed systems programming that simplifies failure scenarios. Transactions provide the *ACID properties*:

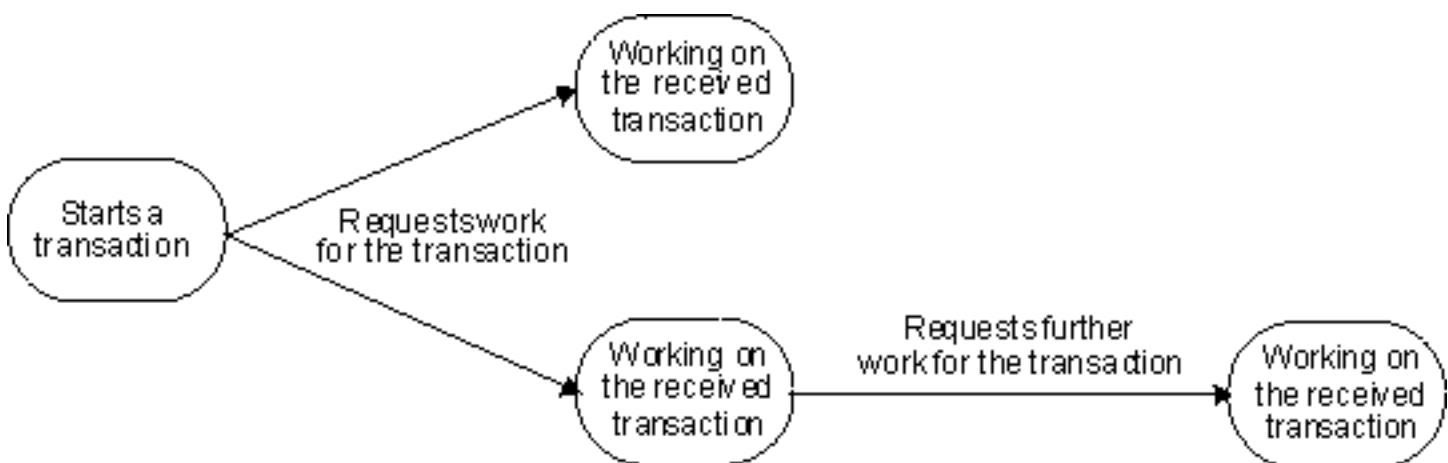
- *Atomicity*: A transaction's changes are atomic: either all operations that are part of the transaction happen or none happen.
- *Consistency*: A transaction moves data between consistent states.
- *Isolation*: Even though transactions can run (or be executed) concurrently, no transaction sees another's work in progress. The transactions appear to run serially.
- *Durability*: After a transaction completes successfully, its changes survive subsequent failures.

As an example, consider a transaction that transfers money from one account to another. Such a transfer involves money being deducted from one account and deposited in the other. Withdrawing the money from one account and depositing it in the other account are two parts of an *atomic* transaction: if both cannot be completed, neither must happen. If multiple requests are processed against an account at the same time, they must be *isolated* so that only a single transaction can affect the account at one time. If the bank's central computer fails just after the transfer, the correct balance must still be shown when the system becomes available again: the change must be *durable*. Note that *consistency* is a function of the application; if money is to be transferred from one account to another, the application must subtract the same amount of money from one account that it adds to the other account. Transactions can be completed in one of two ways: they can commit or roll back. A successful transaction is said to *commit*. An unsuccessful transaction is said to *roll back*. Any data modifications made by a rolled back transaction must be completely undone. In the money-transfer example, if money is withdrawn from one account but a failure prevents the money from being deposited in the other account, any changes made to the first account must be completely undone. The next time any source queries the account balance, the correct balance must be shown.

Distributed transactions and the two-phase commit process

A *distributed transaction* is one that runs in multiple processes, often on several machines. Each process participates in the transaction. This is illustrated in [Figure 2](#), where each oval indicates work being done on a different machine, and each arrow indicates a remote method invocation (RMI).

Figure 2. Example of a distributed transaction



Distributed transactions, like local transactions, must adhere to the ACID properties. However, maintaining these properties is greatly complicated for distributed transactions because a failure can occur in any process, and in the event of such a failure, each process must undo any work already done on behalf of the transaction.

A distributed transaction processing system maintains the ACID properties in distributed transactions by using two features:

- *Recoverable processes*: Recoverable processes are those that can restore earlier states if a failure occurs.
- *A commit protocol*: A commit protocol enables multiple processes to coordinate the committing or rolling back (aborting) of a transaction. The most common commit protocol, and the one used by the EJB server, is the two-phase commit protocol.

Transaction state information must be stored by all recoverable processes. However, only processes that manage application data (such as resource managers) must store descriptions of changes to data. Not all processes involved in a distributed transaction need to be recoverable. In general, clients are not recoverable because they do not interact directly with a resource manager. Processes that are not recoverable are referred to as *ephemeral* processes. The *two-phase commit protocol*, as the name implies, involves two phases: a prepare phase and a resolution phase. In each transaction, one process acts as the coordinator. The *coordinator* oversees the activities of the other participants in the transaction to ensure a consistent outcome. In the *prepare phase*, the coordinator sends a message to each process in the transaction, asking each process to prepare to commit. When a process prepares, it guarantees that it can commit the transaction and makes a permanent record of its work. After guaranteeing that it can commit, it can no longer unilaterally decide to roll back the transaction. If a process cannot prepare (that is, if it cannot guarantee that it can commit the transaction), it must roll back the transaction. In the *resolution phase*, the coordinator tallies the responses. If all participants are prepared to commit, the transaction commits; otherwise, the transaction is rolled back. In either case, the coordinator informs all participants of the result. In the case of a commit, the participants acknowledge that they have committed.

Using transactions in the EJB server environment

The enterprise bean transaction model corresponds in most respects to the OMG OTS version 1.1. An enterprise bean instance that is transaction enabled corresponds to an object of the OTS TransactionalObject interface. However, the enterprise bean transaction model does not support transaction nesting.

In the EJB server environment, transactions are handled by three main components of the transaction service:

- A transaction manager interface that enables the EJB server to control transaction boundaries within its enterprise beans based on the transactional attributes specified for the beans.
- An interface (UserTransaction) that allows an enterprise bean or an EJB client to manage transactions. The container makes this interface available to enterprise beans and EJB clients by way of the name service.
- Coordination by way of the X/Open XA interface that enables a transactional resource manager (such as a database) to participate in a transaction controlled by an external transaction manager.

For most purposes, the enterprise bean developers can delegate the tasks involved in managing a transaction to the container. The developer performs this delegation by setting the deployment descriptor attributes for transactions. These attributes and their values are described in [Setting transactional attributes in the deployment descriptor](#).

In other cases, the enterprise bean developer will want or need to manage the transactions at the bean level or involve the EJB client in the management of transactions. For more information on this approach, see [Using bean-managed transactions](#).

The data source

Entity beans contain persistent data that must be permanently stored in a recoverable data source. Although the EJB Specification often refers to databases as the place to store persistent data associated with an entity bean, it leaves open the possibility of using other data sources, including operating system files and other applications. If you want to let the container handle the interaction between an entity bean and a data source, you must use the data sources supported by that container.

If you write the additional code required to handle the interaction between a BMP entity bean and the data source, you can use any data source that meets your needs and is compatible with the persistence service. For more information, see [Developing entity beans with BMP](#).

The EJB clients

An EJB client can take one of the following forms: it can be a Java application, a Java servlet, a Java applet-servlet combination, or a JSP file. The EJB client code required to access and manipulate enterprise beans is very similar across the different Java EJB clients. EJB client developers must consider the following issues:

- *Naming and communications*--A Java EJB client must use either HTTP or RMI to communicate with enterprise beans. Fortunately, there is very little difference in the coding required to enable communications between the EJB client and the enterprise bean, because JNDI masks the interaction between the EJB client and the name service.
 - Java applications communicate with enterprise beans by using RMI/IIOP.
 - Java servlets and JSP files communicate with enterprise beans by using HTTP. To use servlets with an EJB server, a Web server must be installed and configured on a machine in the EJB server environment. For more information, see [The Web server](#).
- *Threading*--Java clients can be either single-threaded or multithreaded depending on the tasks that the client needs to perform. Each client thread that uses a service provided by a session bean must create or find a separate instance of that bean and maintain a reference to that bean until the thread completes; multiple client threads can access the same entity bean.
- *Security* - EJB clients that access an EJB server over HTTP (for example, servlets and JSP files) encounter the following two layers of security:
 1. Universal Resource Locator (URL) security enforced by the WebSphere Application Server Security Plug-in attached to the Web server in collaboration with the security service.
 2. Enterprise bean security enforced at the server working with the security service.

When the user of an HTTP-based EJB client attempts to access an enterprise bean, the Web server (using the WebSphere Server plug-in) authenticates the user. This authentication can take the form of a request for a user ID and password or it can happen transparently in the form of a certificate exchange followed by the establishment of a Secure Sockets Layer (SSL) session.

The authentication policy is governed by an additional option: secure channel constraint. If the secure channel constraint is required, an SSL session must be established as the final phase of authentication; otherwise, SSL is optional.

- *Transactions*--Both types of Java clients can use the transaction service by way of the JTA interfaces to manage transactions. The code required for transaction management is identical in the two types of clients. For general information on transactions and the Java transaction service, see [The transaction service](#). For information on managing transactions in a Java EJB client, see [Managing transactions in an EJB client](#).

The Web server

To access the functionality in the EJB server, Java servlets and JSP files must have access to a Web server. The Web server enables communication between a Web client and the EJB server. The EJB server, Web server, and Java servlet can each reside on different machines.

For information on the Web servers supported by the EJB servers, see the Advanced Application Server *Getting Started* document.

The administration interface

The EJB server uses the WebSphere Administrative Console. For more information on this interface, consult the WebSphere InfoCenter and the online help available with the WebSphere Administrative Console. You can also administer the EJB server using the **wscp** command-line tool. For more information, see the Advanced Edition Information Center.

WebSphere Programming Model Extensions

This section discusses facilities that are provided as part of the Programming Model Extensions in WebSphere Application Server:

- The exception-chaining package, which can be used by distributed applications to capture a sequence of exceptions. For more information, see [The distributed-exception package](#).
- The command package, which can be used by distributed applications to reduce the number of remote invocations they must make. For more information, see [The command package](#).
- The localizable-text package, which can be used by distributed applications spanning locales to deliver output in a user-specified language. For more information, see [The localizable-text package](#).

The exception-chaining and command packages are available as part of WebSphere Application Server Advanced Edition and Enterprise Edition; the localizable-text package is available as part of WebSphere Application Server Advanced Edition. All three packages are general-purpose utilities, designed to provide common functions in a reusable way. Although these facilities are described in the context of enterprise beans, they are available to any WebSphere Application Server Java application. They are not restricted to use with enterprise beans.

The distributed-exception package

Distributed applications require a strategy for exception handling. As applications become more complex and are used by more participants, handling exceptions becomes problematic. To capture the information contained in every exception, methods have to rethrow every exception they catch. If every method adopts this approach, the number of exceptions can become unmanageable, and the code itself becomes less maintainable. Furthermore, if a new method introduces a new exception, all existing methods that call the new method have to be modified to handle the new exception. Trying to explicitly manage every possible exception in a complex application quickly becomes intractable.

In order to keep the number of exceptions manageable, some programmers adopt a strategy in which methods catch all exceptions in a single clause and throw one exception in response. This reduces the number of exceptions each method must recognize, but it also means that the information about the originating exception is lost. This loss of information can be desirable, for example, when you wish to hide implementation details from end users. However, this strategy can make applications much more difficult to debug.

The distributed-exception package provides a facility that allows you to build chains of exceptions. An *exception chain* encapsulates the stack of previous exceptions. With an exception chain, you can throw one exception in response to another without discarding the previous exceptions, so you can manage the number of exceptions without losing the information they carry. Exceptions that support chaining are called *distributed exceptions*.

Distributed exceptions are packaged in the `ras.jar` file, which must be included in the application's `CLASSPATH` variable.

Overview

Support for chaining distributed exceptions is provided by the `com.ibm.websphere.exception` Java package. The following classes and interfaces make up this package:

- `DistributedException`--This class provides access to the methods on the `DistributedExceptionInfo` object. It acts as the root class for exceptions in a distributed application. For more information, see [The DistributedException class](#).
- `DistributedExceptionEnabled`--This interface allows exceptions that cannot inherit from the `DistributedException` class to be used in exception chains, so that exceptions based on predefined exceptions can be captured. For more information, see [The DistributedExceptionEnabled interface](#).
- `DistributedExceptionInfo`--This class encapsulates the work necessary for distributed exceptions. An exception class that extends the `DistributedException` class automatically gets access to this class. A class that implements the `DistributedExceptionEnabled` interface must explicitly declare a `DistributedExceptionInfo` attribute. For more information, see [The DistributedExceptionInfo class](#).
- `ExceptionInstantiationException`--This class defines the exception that is thrown if an exception chain cannot be created. This exception is instantiated internally, but you can catch and re-throw it.

This section provides a general description of the interfaces and classes in the exception-chaining package.

The DistributedException class

The DistributedException class provides the root exception for exception hierarchies defined by applications. With this class, you build chains of exceptions by saving a caught exception and bundling it into the new exception to be thrown. This way, the information about the old exception is forwarded along with the new exception. The class declares six constructors; [Figure 55](#) shows the signatures for these constructors. When your exception is a subclass of the DistributedException class, you must provide corresponding constructors in your exception class.

Figure 55. Code example: Constructors for the DistributedException class

```
...
public class DistributedException extends Exception
implements DistributedExceptionEnabled
{
    // Constructors
    public DistributedException() {...}
    public DistributedException(String message) {...}
    public DistributedException(Throwable exception) {...}
    public DistributedException(String message, Throwable exception) {...}
    public DistributedException(String resourceBundleName,
                                String resourceKey,
                                Object[] formatArguments,
                                String defaultText)
    {
        {...}
    }
    public DistributedException(String resourceBundleName,
                                String resourceKey,
                                Object[] formatArguments,
                                String defaultText,
                                Throwable exception)
    {
        {...}
    }
    // Other methods
    ...
}
```

The class also provides methods for extracting exceptions from the chain and querying the chain. These methods include:

- `getMessage`--This method returns the message string associated with the current exception.
- `getPreviousException`--This method returns the preceding exception in a chain as a `Throwable` object. If there are no previous exceptions, it returns `null`.
- `getOriginalException`--This method returns the original exception in a chain as a `Throwable` object. If there is no prior exception, it returns `null`.
- `getException`--This method returns the most recent instance of the named exception from the chain as a `Throwable` object. If there are no instances present, it returns `null`.
- `getExceptionInfo`--This method returns the `DistributedExceptionInfo` object for the exception.
- `printStackTrace`--These methods print the stack trace for the current exception, which includes the stack traces of all previous exceptions in the chain.

Localization support

Support for localized messages is provided by two of the constructors for distributed exceptions. These constructors take arguments representing a resource bundle, a resource key, a default message, and the set of replacement strings for variables in the message. A resource bundle is a collection of resources or resource names representing information associated with a specific locale. Resource bundles are provided as either a subclass of the `ResourceBundle` class or in a properties file. The resource key indicates which resource in the bundle to retrieve. The default message is returned if either the name of the resource bundle or the key is `null` or invalid.

The DistributedExceptionEnabled interface

Use the `DistributedExceptionEnabled` interface to create distributed exceptions when your exception cannot extend the `DistributedException` class. Because Java does not permit multiple inheritance, you cannot extend multiple exception classes. If you are extending an existing exception class, for example, `javax.ejb.CreateException`, you cannot also extend the `DistributedException` class. To allow your new exception class to chain other exceptions, you must implement the `DistributedExceptionEnabled` interface instead. The `DistributedExceptionEnabled` interface declares eight methods you must implement in your exception class:

- `getMessage`--This method returns the message string associated with the current exception.
- `getPreviousException`--This method returns the preceding exception in a chain as a `Throwable` object. If there are no previous exceptions, it returns null.
- `getOriginalException`--This method returns the original exception in a chain as a `Throwable` object. If there is no prior exception, it returns null.
- `getException`--This method returns the most recent instance of the named exception from the chain as a `Throwable` object. If there are no instances present, it returns null.
- `getExceptionInfo`--This method returns the `DistributedExceptionInfo` object for the exception.
- `printStackTrace`--These methods print the stack trace for the current exception, which includes the stack traces of all previous exceptions in the chain.
- `printSuperStackTrace`--This method is used by a `DistributedExceptionInfo` object to retrieve and save the current stack trace.

When implementing the `DistributedExceptionEnabled` interface, you must declare a `DistributedExceptionInfo` attribute. This attribute provides implementations for most of these methods, so implementing them in your exception class consists of calling the corresponding methods on the `DistributedExceptionInfo` object. For more information, see [Implementing the methods from the DistributedExceptionEnabled interface](#).

The `DistributedExceptionInfo` class

The `DistributedExceptionInfo` class provides the functionality required for distributed exceptions. It must be used by any exception that implements the `DistributedExceptionEnabled` interface (which includes the `DistributedException` class). A `DistributedExceptionInfo` object contains the exception itself, and it provides constructors for creating exception chains and methods for retrieving the information within those chains. It also provides the underlying methods for managing chained exceptions.

Extending the `DistributedException` class

The `DistributedException` class provides the root exception for exception hierarchies defined by applications. The class also provides methods for extracting exceptions from the chain and querying the chain. You must provide constructors corresponding to the constructors in the `DistributedException` class (see [Figure 55](#)). The constructors can simply pass arguments to the constructor in the `DistributedException` class by using super methods, as illustrated in [Figure 56](#).

Figure 56. Code example: Constructors in an exception class that extends the `DistributedException` class

```
...
import com.ibm.websphere.exception.*;
public class MyDistributedException extends DistributedException
{
    // Constructors
    public MyDistributedException() {
        super();
    }
    public MyDistributedException(String message) {
        super(message);
    }
    public MyDistributedException(Throwable exception) {
        super(exception);
    }
    public MyDistributedException(String message, Throwable exception) {
        super(message, exception);
    }
}
```

```

    }
    public MyDistributedException(String resourceBundleName,
                                String resourceKey, Object[] formatArguments,
                                String defaultText)
    {
        super(resourceBundleName, resourceKey, formatArguments, defaultText);
    }
    public MyDistributedException(String resourceBundleName,
                                String resourceKey, Object[] formatArguments,
                                String defaultText, Throwable exception)
    {
        super(resourceBundleName, resourceKey, formatArguments, defaultText,
              exception);
    }
}

```

Implementing the DistributedExceptionEnabled interface

Use the DistributedExceptionEnabled interface to create distributed exceptions when your exception cannot extend the DistributedException class. To allow your new exception class to be chained, you must implement the DistributedExceptionEnabled interface instead. [Figure 57](#) shows the structure of an exception class that extends the existing javax.ejb.CreateException class and implements the DistributedExceptionEnabled interface. The class also declares the required DistributedExceptionInfo object.

Figure 57. Code example: The structure of an exception class that implements the DistributedExceptionEnabled interface

```

...
import javax.ejb.*;
import com.ibm.websphere.exception.*;
public class AccountCreateException extends CreateException
implements DistributedExceptionEnabled
{
    DistributedExceptionInfo exceptionInfo = null;
    // Constructors
    ...
    // Methods from the DistributedExceptionEnabled interface
    ...
}

```

Implementing the constructors for the exception class

The exception-chaining package supports six different ways of creating instances of exception classes (see [Figure 55](#)). When you write an exception class by implementing the DistributedExceptionEnabled interface, you must implement these constructors. In each one, you must use the DistributedExceptionInfo object to capture the information for chaining the exception. [Figure 58](#) shows standard implementations for the six constructors.

Figure 58. Code example: Constructors for an exception class that implements the DistributedExceptionEnabled interface

```

...
public class AccountCreateException extends CreateException
implements DistributedExceptionEnabled
{
    DistributedExceptionInfo exceptionInfo = null;
    // Constructors
    AccountCreateException() {
        super ();
        exceptionInfo = new DistributedExceptionInfo(this);
    }
    AccountCreateException(String msg) {

```

```

        super (msg);
        exceptionInfo = new DistributedExceptionInfo(this);
    }
    AccountCreateException(Throwable e) {
        super ();
        exceptionInfo = new DistributedExceptionInfo(this, e);
    }
    AccountCreateException(String msg, Throwable e) {
        super (msg);
        exceptionInfo = new DistributedExceptionInfo(this, e);
    }
    AccountCreateException(String resourceBundleName, String resourceKey,
                           Object[] formatArguments, String defaultText)
    {
        super ();
        exceptionInfo = new DistributedExceptionInfo(resourceBundleName,
                                                    resourceKey, formatArguments, defaultText, this);
    }
    AccountCreateException(String resourceBundleName, String resourceKey,
                           Object[] formatArguments, String defaultText,
                           Throwable exception)
    {
        super ();
        exceptionInfo = new DistributedExceptionInfo(resourceBundleName,
                                                    resourceKey, formatArguments, defaultText, this, exception);
    }
    // Methods from the DistributedExceptionEnabled interface
    ...
}

```

Implementing the methods from the DistributedExceptionEnabled interface

The DistributedExceptionInfo object provides implementations for most of the methods in the DistributedExceptionEnabled interface, so you can implement the required methods in your exception class by calling the corresponding methods on the DistributedExceptionInfo object. [Figure 59](#) illustrates this technique. The only two methods that do not involve calling a corresponding method on the DistributedExceptionInfo object are the getExceptionInfo method, which returns the object, and the printSuperStackTrace method, which calls the super.printStackTrace method.

Figure 59. Code example: Implementations of the methods in the DistributedExceptionEnabled interface

```

...
public class AccountCreateException extends CreateException
implements DistributedExceptionEnabled
{
    DistributedExceptionInfo exceptionInfo = null;
    // Constructors
    ...
    // Methods from the DistributedExceptionEnabled interface
    String getMessage() {
        if (exceptionInfo != null)
            return exceptionInfo.getMessage();
        else return null;
    }
    Throwable getPreviousException() {
        if (exceptionInfo != null)
            return exceptionInfo.getPreviousException();
        else return null;
    }
    Throwable getOriginalException() {
        if (exceptionInfo != null)
            return exceptionInfo.getOriginalException();
    }
}

```

```

        else return null;
    }
    Throwable getException(String exceptionClassName) {
        if (exceptionInfo != null)
            return exceptionInfo.getException(exceptionClassName);
        else return null;
    }
    DistributedExceptionInfo getExceptionInfo() {
        if (exceptionInfo != null)
            return exceptionInfo;
        else return null;
    }
    void printStackTrace() {
        if (exceptionInfo != null)
            return exceptionInfo.printStackTrace();
        else return null;
    }
    void printStackTrace(PrintWriter pw) {
        if (exceptionInfo != null)
            return exceptionInfo.printStackTrace(pw);
        else return null;
    }
    void printSuperStackTrace(PrintWriter pw)
        if (exceptionInfo != null)
            return super.printStackTrace(pw);
        else return null;
    }
}

```

Using distributed exceptions

Defining a distributed exception gives you the ability to chain exceptions together. The `DistributedExceptionInfo` class provides methods for adding information to an exception chain and for extracting information from the chain. This section illustrates the use of distributed exceptions.

Catching distributed exceptions

You can catch exceptions that extend the `DistributedException` class or implement the `DistributedExceptionEnabled` interface separately. You can also test a caught exception to see if it has implemented the `DistributedExceptionEnabled` interface. If it has, you can treat it as any other distributed exception. [Figure 60](#) shows the use of the `instanceof` method to test for exception chaining.

Figure 60. Code example: Testing for an exception that implements the `DistributedExceptionEnabled` interface

```

....
try {
    someMethod();
}
catch (Exception e) {
    ...
    if (e instanceof DistributedExceptionEnabled) {
        ...
    }
}
...

```

Adding an exception to a chain

To add an exception to a chain, you must call one of the constructors for your distributed-exception class. This captures the previous exception information and packages it with the new exception. [Figure 61](#) shows the use of the `MyDistributedException(Throwable)` constructor.

Figure 61. Code example: Adding an exception to a chain

```
void someMethod() throws MyDistributedException {
    try {
        someOtherMethod();
    }
    catch (DistributedExceptionEnabled e) {
        throw new MyDistributedException(e);
    }
    ...
}...
```

Retrieving information from a chain

Chained exceptions allow you to retrieve information about prior exceptions in the chain. For example, the `getPreviousException`, `getOriginalException`, and `getException(String)` methods allow you to retrieve specific exceptions from the chain. You can retrieve the message associated with the current exception by calling the `getMessage` method. You can also get information about the entire chain by calling one of the `printStackTrace` methods. [Figure 62](#) illustrates calling the `getPreviousException` and `getOriginalException` methods.

Figure 62. Code example: Extracting exceptions from a chain

```
...
try {
    someMethod();
}
catch (DistributedExceptionEnabled e) {
    try {
        Throwable prev = e.getPreviousException();
    }
    catch (ExceptionInstantiationException eie) {
        DistributedExceptionInfo prevExInfo = e.getPreviousExceptionInfo();
        if (prevExInfo != null) {
            String prevExName = prevExInfo.getClassName();
            String prevExMsg = prevExInfo.getClassMessage();
            ...
        }
    }
    try {
        Throwable orig = e.getOriginalException();
    }
    catch (ExceptionInstantiationException eie) {
        DistributedExceptionInfo origExInfo = null;
        DistributedExceptionInfo prevExInfo = e.getPreviousExceptionInfo();
        while (prevExInfo != null) {
            origExInfo = prevExInfo;
            prevExInfo = prevExInfo.getPreviousExceptionInfo();
        }
        if (origExInfo != null) {
            String origExName = origExInfo.getClassName();
            String origExMsg = origExInfo.getClassMessage();
            ...
        }
    }
}
...
```

Distributed applications are defined by the ability to utilize remote resources as if they were local, but this remote work affects the performance of distributed applications. Distributed applications can improve performance by using remote calls sparingly. For example, if a server does several tasks for a client, the application can run more quickly if the client bundles requests together, reducing the number of individual remote calls. The command package provides a mechanism for collecting sets of requests to be submitted as a unit.

In addition to giving you a way to reduce the number of remote invocations a client makes, the command package provides a generic way of making requests. A client instantiates the command, sets its input data, and tells it to run. The command infrastructure determines the target server and passes a copy of the command to it. The server runs the command, sets any output data, and copies it back to the client. The package provides a common way to issue a command, locally or remotely, and independently of the server's implementation. Any server (an enterprise bean, a Java Database Connectivity (JDBC) server, a servlet, and so on) can be a target of a command if the server supports Java access to its resources and provides a way to copy the command between the client's Java Virtual Machine (JVM) and its own JVM.

Overview

The command facility is implemented in the `com.ibm.websphere.command` Java package. The classes and interfaces in the command package fall into four general categories:

- Interfaces for creating commands. For more information, see [Facilities for creating commands](#).
- Classes and interfaces for implementing commands. For more information, see [Facilities for implementing commands](#).
- Classes and interfaces for determining where the command is run. For more information, see [Facilities for setting and determining targets](#).
- Classes defining package-specific exceptions. For more information, see [Exceptions in the command package](#).

This section provides a general description of the interfaces and classes in the command package.

Facilities for creating commands

The Command interface specifies the most basic aspects of a command. This interface is extended by both the TargetableCommand interface and the CompensableCommand interface, which offer additional features. To create commands for applications, you must:

- Define an interface that extends one or more of interfaces in the command package.
- Provide an implementation class for your interface.

In practice, most commands implement the TargetableCommand interface, which allows the command to be executed remotely. [Figure 63](#) shows the structure of a command interface for a targetable command.

Figure 63. Code example: The structure of an interface for a targetable command

```
...
import com.ibm.websphere.command.*;
public interface MySimpleCommand extends TargetableCommand {
    // Declare application methods here
}
```

The CompensableCommand interface allows the association of one command with another that can undo the work of the first. Compensable commands also typically implement the TargetableCommand interface. [Figure 64](#) shows the structure of a command interface for a targetable, compensable command.

Figure 64. Code example: The structure of an interface for a targetable, compensable command

```
...
import com.ibm.websphere.command.*;
public interface MyCommand extends TargetableCommand, CompensableCommand {
    // Declare application methods here
}
```


Facilities for implementing commands

Commands are implemented by extending the class `TargetableCommandImpl`, which implements the `TargetableCommand` interface. The `TargetableCommandImpl` class is an abstract class that provides some implementations for some of the methods in the `TargetableCommand` interface (for example, setting return values) and declares additional methods that the application itself must implement (for example, how to execute the command).

You implement your command interface by writing a class that extends the `TargetableCommandImpl` class and implements your command interface. This class contains the code for the methods in your interface, the methods inherited from extended interfaces (the `TargetableCommand` and `CompensableCommand` interfaces), and the required (abstract) methods in the `TargetableCommandImpl` class. You can also override the default implementations of other methods provided in the `TargetableCommandImpl` class. [Figure 65](#) shows the structure of an implementation class for the interface in [Figure 64](#).

Figure 65. Code example: The structure of an implementation class for a command interface

```
...
import java.lang.reflect.*;
import com.ibm.websphere.command.*;
public class MyCommandImpl extends TargetableCommandImpl
implements MyCommand {
    // Set instance variables here
    ...
    // Implement methods in the MyCommand interface
    ...
    // Implement methods in the CompensableCommand interface
    ...
    // Implement abstract methods in the TargetableCommandImpl class
    ...
}
```

Facilities for setting and determining targets

The object that is the target of a `TargetableCommand` must implement the `CommandTarget` interface. This object can be an actual server-side object, like an entity bean, or it can be a client-side adapter for a server. The implementor of the `CommandTarget` interface is responsible for ensuring the proper execution of a command in the desired target server environment. This typically requires the following steps:

1. Copying the command to the target server by using a server-specific protocol.
2. Running the command in the server.
3. Copying the executed command from the target server to the client by using a server-specific protocol.

Common ways to implement the `CommandTarget` interface include:

- A local target, which runs in the client's JVM.
- A client-side adapter for a server. For an example that implements the target as a client-side adapter, see [Writing a command target \(client-side adapter\)](#).
- An enterprise bean (either a session bean or an entity bean). [Figure 66](#) shows the structure of the remote interface and enterprise bean class for an entity bean that implements the `CommandTarget` interface. An enterprise bean is provided with WebSphere that can be deployed and used as a `CommandTarget`. See [Using the WebSphere EJBCommandTarget bean as a command target](#).

Figure 66. Code example: The structure of a command-target entity bean

```
...
import java.rmi.RemoteException;
import java.util.Properties;
import javax.ejb.*;
import com.ibm.websphere.command.*;
// Remote interface for the MyBean enterprise bean (also a command target)
public interface MyBean extends EJBObject, CommandTarget {
    // Declare methods for the remote interface
```



```

    ...
}
// Entity bean class for the MyBean enterprise bean (also a command target)
public class MyBeanClass implements EntityBean, CommandTarget {
    // Set instance variables here
    ...
    // Implement methods in the remote interface
    ...
    // Implement methods in the EntityBean interface
    ...
    // Implement the method in the CommandTarget interface
    ...
}

```

Since targetable commands can be run remotely in another JVM, the command package provides mechanisms for determining where to run the command. A *target policy* associates a command with a target and is specified through the TargetPolicy interface. You can design customized target policies by implementing this interface, or you can use the provided TargetPolicyDefault class. For more information, see [Targets and target policies](#).

Exceptions in the command package

The command package defines a set of exception classes. The CommandException class extends the DistributedException class and acts as the base class for the additional command-related exceptions: UnauthorizedAccessException, UnsetInputPropertiesException, and UnavailableCompensableCommandException. Applications can extend the CommandException class to define additional exceptions, as well.

Although the CommandException class extends the DistributedException class, you do not have to import the distributed-exception package, com.ibm.websphere.exception, unless you need to use the features of the DistributedException class in your application. For more information on distributed exceptions, see [The distributed-exception package](#).

Writing command interfaces

To write a command interface, you extend one or more of the three interfaces included in the command package. The base interface for all commands is the Command interface. This interface provides only the client-side interface for generic commands and declares three basic methods:

- **isReadyToCallExecute**--This method is called on the client side before the command is passed to the server for execution.
- **execute**--This method passes the command to the target and returns any data.
- **reset**--This method reverts any output properties to the values they had before the execute method was called so that the object can be reused.

The implementation class for your interface must contain implementations for the isReadyToCallExecute and reset methods. The execute method is implemented for you elsewhere; for more information, see [Implementing command interfaces](#). Most commands do not extend the Command interface directly but use one of the provided extensions: the TargetableCommand interface and the CompensableCommand interface.

The TargetableCommand interface

The TargetableCommand interface extends the Command interface and provides for remote execution of commands. Most commands will be targetable commands. The TargetableCommand interface declares several additional methods:

- **setCommandTarget**--This method allows you to specify the target object to a command.
- **setCommandTargetName**--This method allows you to specify the target by name to a command.
- **getCommandTarget**--This method returns the target object of the command.
- **getCommandTargetName**--This method returns the name of the target object of the command.
- **hasOutputProperties**--This method indicates whether or not the command has output that must be copied back to the

client. (The implementation class also provides a method, `setHasOutputProperties`, for setting the output of this method. By default, `hasOutputProperties` returns true.)

- `setOutputProperties`--This method saves output values from the command for return to the client.
- `performExecute`--This method encapsulates the application-specific work. It is called for you by the `execute` method declared in the `Command` interface.

With the exception of the `performExecute` method, which you must implement, all of these methods are implemented in the `TargetableCommandImpl` class. This class also implements the `execute` method declared in the `Command` interface.

The `CompensableCommand` interface

The `CompensableCommand` interface also extends the `Command` interface. A compensable command is one that has another command (a compensator) associated with it, so that the work of the first can be undone by the compensator. For example, a command that attempts to make an airline reservation followed by a hotel reservation can offer a compensating command that allows the user to cancel the airline reservation if the hotel reservation cannot be made.

The `CompensableCommand` interface declares one method:

- `getCompensatingCommand`--This method returns the command that can be used to undo the effects of the original command.

To create a compensable command, you write an interface that extends the `CompensableCommand` interface. Such interfaces typically extend the `TargetableCommand` interface as well. You must implement the `getCompensatingCommand` method in the implementation class for your interface. You must also implement the compensating command.

The example application

The example used throughout the remainder of this discussion uses an entity bean with container-managed persistence (CMP) called `CheckingAccountBean`, which allows a client to deposit money, withdraw money, set a balance, get a balance, and retrieve the name on the account. This entity bean also accepts commands from the client. The code examples illustrate the command-related programming. For a servlet-based example, see [Writing a command target \(client-side adapter\)](#).

[Figure 67](#) shows the interface for the `ModifyCheckingAccountCmd` command. This command is both targetable and compensable, so the interface extends both `TargetableCommand` and `CompensableCommand` interfaces.

Figure 67. Code example: The `ModifyCheckingAccountCmd` interface

```
...
import com.ibm.websphere.exception.*;
import com.ibm.websphere.command.*;
public interface ModifyCheckingAccountCmd
extends TargetableCommand, CompensableCommand {
    float getAmount();
    float getBalance();
    float getOldBalance();           // Used for compensating
    float setBalance(float amount);
    float setBalance(int amount);
    CheckingAccount getCheckingAccount();
    void setCheckingAccount(CheckingAccount newCheckingAccount);
    TargetPolicy getCmdTargetPolicy();
    ...
}
```

Implementing command interfaces

The command package provides a class, `TargetableCommandImpl`, that implements all of the methods in the `TargetableCommand` interface except the `performExecute` method. It also implements the `execute` method from the `Command` interface. To implement an application's command interface, you must write a class that extends the `TargetableCommandImpl` class and implements your command interface. [Figure 68](#) shows the structure of the

ModifyCheckingAccountCmdImpl class.

Figure 68. Code example: The structure of the ModifyCheckingAccountCmdImpl class

```
...
public class ModifyCheckingAccountCmdImpl extends TargetableCommandImpl
implements ModifyCheckingAccountCmd
{
    // Variables
    ...
    // Methods
    ...
}
```

The class must declare any variables and implement these methods:

- Any methods you defined in your command interface.
- The `isReadyToCallExecute` and `reset` methods from the `Command` interface.
- The `performExecute` method from the `TargetableCommand` interface.
- The `getCompensatingCommand` method from the `CompensableCommand` interface, if your command is compensable. You must also implement the compensating command.

You can also override the nonfinal implementations provided in the `TargetableCommandImpl` class. The most likely candidate for reimplementation is the `setOutputProperties` method, since the default implementation does not save final, transient, or static fields.

Defining instance and class variables

The `ModifyCheckingAccountCmdImpl` class declares the variables used by the methods in the class, including the remote interface of the `CheckingAccount` entity bean; the variables used to capture operations on the checking account (balances and amounts); and a compensating command. [Figure 69](#) shows the variables used by the `ModifyCheckingAccountCmd` command.

Figure 69. Code example: The variables in the ModifyCheckingAccountCmdImpl class

```
...
public class ModifyCheckingAccountCmdImpl extends TargetableCommandImpl
implements ModifyCheckingAccountCmd
{
    // Variables
    public float balance;
    public float amount;
    public float oldBalance;
    public CheckingAccount checkingAccount;
    public ModifyCheckingAccountCompensatorCmd
        modifyCheckingAccountCompensatorCmd;
    ...
}
```

Implementing command-specific methods

The `ModifyCheckingAccountCmd` interface defines several command-specific methods in addition to extending other interfaces in the command package. These command-specific methods are implemented in the `ModifyCheckingAccountCmdImpl` class.

You must provide a way to instantiate the command. The command package does not specify the mechanism, so you can choose the technique most appropriate for your application. The fastest and most efficient technique is to use constructors. The most flexible technique is to use a factory. Also, since commands are implemented internally as JavaBeans components, you can use the standard `Beans.instantiate` method. The `ModifyCheckingAccountCmd` command uses constructors.

[Figure 70](#) shows the two constructors for the command. The difference between them is that the first uses the default target

policy for determining the target of the command and the second allows you to specify a custom policy. (For more information on targets and target policies, see [Targets and target policies](#).)

Both constructors take a `CommandTarget` object as an argument and cast it to the `CheckingAccount` type. The `CheckingAccount` interface extends both the `CommandTarget` interface and the `EJBObject` (see [Figure 80](#)). The resulting `checkingAccount` object routes the command to the desired server by using the bean's remote interface. (For more information on `CommandTarget` objects, see [Writing a command target \(server\)](#).)

Figure 70. Code example: Constructors in the `ModifyCheckingAccountCmdImpl` class

```
...
public class ModifyCheckingAccountCmdImpl extends TargetableCommandImpl
implements ModifyCheckingAccountCmd
{
    // Variables
    ...
    // Constructors
    // First constructor: relies on the default target policy
    public ModifyCheckingAccountCmdImpl(CommandTarget target,
                                         float newAmount)
    {
        amount = newAmount;
        checkingAccount = (CheckingAccount)target;
        setCommandTarget(target);
    }
    // Second constructor: allows you to specify a custom target policy
    public ModifyCheckingAccountCmdImpl(CommandTarget target,
                                         float newAmount,
                                         TargetPolicy targetPolicy)
    {
        setTargetPolicy(targetPolicy);
        amount = newAmount;
        checkingAccount = (CheckingAccount)target;
        setCommandTarget(target);
    }
    ...
}
```

[Figure 71](#) shows the implementation of the command-specific methods:

- `setBalance`--This method sets the balance of the account.
- `getAmount`--This method returns the amount of a deposit or withdrawal.
- `getOldBalance`, `getBalance`--These methods capture the balance before and after an operation.
- `getCmdTargetPolicy`--This method retrieves the current target policy.
- `setCheckingAccount`, `getCheckingAccount`--These methods set and retrieve the current checking account.

Figure 71. Code example: Command-specific methods in the `ModifyCheckingAccountCmdImpl` class

```
...
public class ModifyCheckingAccountCmdImpl extends TargetableCommandImpl
implements ModifyCheckingAccountCmd
{
    // Variables
    ...
    // Constructors
    ...
    // Methods in ModifyCheckingAccountCmd interface
    public float getAmount() {
        return amount;
    }
    public float getBalance() {
```

```

        return balance;
    }
    public float getOldBalance() {
        return oldBalance;
    }
    public float setBalance(float amount) {
        balance = balance + amount;
        return balance;
    }
    public float setBalance(int amount) {
        balance += amount ;
        return balance;
    }
    public TargetPolicy getCmdTargetPolicy() {
        return getTargetPolicy();
    }
    public void setCheckingAccount(CheckingAccount newCheckingAccount) {
        if (checkingAccount == null) {
            checkingAccount = newCheckingAccount;
        }
        else
            System.out.println("Incorrect Checking Account (" +
                newCheckingAccount + ") specified");
    }
    public CheckingAccount getCheckingAccount() {
        return checkingAccount;
    }
    ...
}

```

The ModifyCheckingAccountCmd command operates on a checking account. Because commands are implemented as JavaBeans components, you manage input and output properties of commands using the standard JavaBeans techniques. For example, initialize input properties with set methods (like setCheckingAccount) and retrieve output properties with get methods (like getCheckingAccount). The get methods do not work until after the command's execute method has been called.

Implementing methods from the Command interface

The Command interface declares two methods, isReadyToCallExecute and reset, that must be implemented by the application programmer. [Figure 72](#) shows the implementations for the ModifyCheckingAccountCmd command. The implementation of the isReadyToCallExecute method ensures that the checkingAccount variable is set. The reset method sets all of the variables back to starting values.

Figure 72. Code example: Methods from the Command interface in the ModifyCheckingAccountCmdImpl class

```

...
public class ModifyCheckingAccountCmdImpl extends TargetableCommandImpl
implements ModifyCheckingAccountCmd
{
    ...
    // Methods from the Command interface
    public boolean isReadyToCallExecute() {
        if (checkingAccount != null)
            return true;
        else
            return false;
    }
    public void reset() {
        amount = 0;
        balance = 0;
        oldBalance = 0;
    }
}

```

```

        checkingAccount = null;
        targetPolicy = new TargetPolicyDefault();
    }
    ...
}

```

Implementing methods from the TargetableCommand interface

The TargetableCommand interface declares one method, performExecute, that must be implemented by the application programmer. [Figure 73](#) shows the implementation for the ModifyCheckingAccountCmd command. The implementation of the performExecute method does the following:

- Saves the current balance (so the command can be undone by a compensator command)
- Calculates the new balance
- Sets the current balance to the new balance
- Ensures that the hasOutputProperties method returns true so that the values are returned to the client

In addition, the ModifyCheckingAccountCmdImpl class overrides the default implementation of the setOutputProperties method.

Figure 73. Code example: Methods from the TargetableCommand interface in the ModifyCheckingAccountCmdImpl class

```

...
public class ModifyCheckingAccountCmdImpl extends TargetableCommandImpl
implements ModifyCheckingAccountCmd
{
    ...
    // Method from the TargetableCommand interface
    public void performExecute() throws Exception {
        CheckingAccount checkingAccount = getCheckingAccount();
        oldBalance = checkingAccount.getBalance();
        balance = oldBalance+amount;
        checkingAccount.setBalance(balance);
        setHasOutputProperties(true);
    }
    public void setOutputProperties(TargetableCommand fromCommand) {
        try {
            if (fromCommand != null) {
                ModifyCheckingAccountCmd modifyCheckingAccountCmd =
                    (ModifyCheckingAccountCmd) fromCommand;
                this.oldBalance = modifyCheckingAccountCmd.getOldBalance();
                this.balance = modifyCheckingAccountCmd.getBalance();
                this.checkingAccount =
                    modifyCheckingAccountCmd.getCheckingAccount();
                this.amount = modifyCheckingAccountCmd.getAmount();
            }
        }
        catch (Exception ex) {
            System.out.println("Error in setOutputProperties.");
        }
    }
    ...
}

```

Implementing the CompensableCommand interface

The CompensableCommand interface declares one method, getCompensatingCommand, that must be implemented by the application programmer. [Figure 74](#) shows the implementation for the ModifyCheckingAccountCmd command. The implementation simply returns an instance of the ModifyCheckingAccountCompensatorCmd command associated with the

current command.

Figure 74. Code example: Method from the CompensableCommand interface in the ModifyCheckingAccountCmdImpl class

```
...
public class ModifyCheckingAccountCmdImpl extends TargetableCommandImpl
implements ModifyCheckingAccountCmd
{
    ...
    // Method from CompensableCommand interface
    public Command getCompensatingCommand() throws CommandException {
        modifyCheckingAccountCompensatorCmd =
            new ModifyCheckingAccountCompensatorCmd(this);
        return (Command)modifyCheckingAccountCompensatorCmd;
    }
}
```

Writing the compensating command

An application that uses a compensable command requires two separate commands: the primary command (declared as a `CompensableCommand`) and the compensating command. In the example application, the primary command is declared in the `ModifyCheckingAccountCmd` interface and implemented in the `ModifyCheckingAccountCmdImpl` class. Because this command is also a compensable command, there is a second command associated with it that is designed to undo its work. When you create a compensable command, you also have to write the compensating command.

Writing a compensating command can require exactly the same steps as writing the original command: writing the interface and providing an implementation class. In some cases, it may be simpler. For example, the command to compensate for the `ModifyCheckingAccountCmd` does not require any methods beyond those defined for the original command, so it does not need an interface. The compensating command, called `ModifyCheckingAccountCompensatorCmd`, simply needs to be implemented in a class that extends the `TargetableCommandImpl` class. This class must:

- Provide a way to instantiate the command; the example uses a constructor
- Implement the three required methods:
 - `isReadyToCallExecute` and `reset`--both from the `Command` interface
 - `performExecute`--from the `TargetableCommand` interface

Figure 75 shows the structure of the implementation class, its variables (references to the original command and to the relevant checking account), and the constructor. The constructor simply instantiates the references to the primary command and account.

Figure 75. Code example: Variables and constructor in the ModifyCheckingAccountCompensatorCmd class

```
...
public class ModifyCheckingAccountCompensatorCmd extends TargetableCommandImpl
{
    public ModifyCheckingAccountCmdImpl modifyCheckingAccountCmdImpl;
    public CheckingAccount checkingAccount;

    public ModifyCheckingAccountCompensatorCmd(
        ModifyCheckingAccountCmdImpl originalCmd)
    {
        // Get an instance of the original command
        modifyCheckingAccountCmdImpl = originalCmd;
        // Get the relevant account
        checkingAccount = originalCmd.getCheckingAccount();
    }
    // Methods from the Command and Targetable Command interfaces
    ....
}
```


Figure 76 shows the implementation of the inherited methods. The implementation of the `isReadyToCallExecute` method ensures that the `checkingAccount` variable has been instantiated.

The `performExecute` method verifies that the actual checking-account balance is consistent with what the original command returns. If so, it replaces the current balance with the previously stored balance by using the `ModifyCheckingAccountCmd` command. Finally, it saves the most-recent balances in case the compensating command needs to be undone. The `reset` method has no work to do.

Figure 76. Code example: Methods in `ModifyCheckingAccountCompensatorCmd` class

```
...
public class ModifyCheckingAccountCompensatorCmd extends TargetableCommandImpl
{
    // Variables and constructor
    ....
    // Methods from the Command and TargetableCommand interfaces
    public boolean isReadyToCallExecute() {
        if (checkingAccount != null)
            return true;
        else
            return false;
    }
    public void performExecute() throws CommandException
    {
        try {
            ModifyCheckingAccountCmdImpl originalCmd =
                modifyCheckingAccountCmdImpl;
            // Retrieve the checking account modified by the original command
            CheckingAccount checkingAccount = originalCmd.getCheckingAccount();
            if (modifyCheckingAccountCmdImpl.balance ==
                checkingAccount.getBalance()) {
                // Reset the values on the original command
                checkingAccount.setBalance(originalCmd.oldBalance);
                float temp = modifyCheckingAccountCmdImpl.balance;
                originalCmd.balance = originalCmd.oldBalance;
                originalCmd.oldBalance = temp;
            }
            else {
                // Balances are inconsistent, so we cannot compensate
                throw new CommandException(
                    "Object modified since this command ran.");
            }
        }
        catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
    public void reset() {}
}
```

Using a command

To use a command, the client creates an instance of the command and calls the command's `execute` method. Depending on the command, calling other methods can be necessary. The specifics will vary with the application.

In the example application, the server is the `CheckingAccountBean`, an entity enterprise bean. In order to use this enterprise bean, the client gets a reference to the bean's home interface. The client then uses the reference to the home interface and one of the bean's finder methods to obtain a reference to the bean's remote interface. If there is no appropriate bean, the client can create one using a `create` method on the home interface. All of this work is standard enterprise bean programming covered elsewhere in this document.

Figure 77 illustrates the use of the `ModifyCheckingAccountCmd` command. This work takes place after an appropriate `CheckingAccount` bean has been found or created. The code instantiates a command, setting the input values by using one of the constructors defined for the command. The null argument indicates that the command should look up the server using the default target policy, and 1000 is the amount the command attempts to add to the balance of the checking account. (For more information on how the command package uses defaults to determine the target of a command, see [The default target policy](#).) After the command is instantiated, the code calls the `setCheckingAccount` method to identify the account to be modified. Finally, the `execute` method on the command is called.

Figure 77. Code example: Using the `ModifyCheckingAccountCmd` command

```
{
    ...
    CheckingAccount checkingAccount
    ...
    try {
        ModifyCheckingAccountCmd cmd =
            new ModifyCheckingAccountCmdImpl(null, 1000);
        cmd.setCheckingAccount(checkingAccount);
        cmd.execute();
    }
    catch (Exception e) {
        System.out.println(e.getMessage());
    }
    ...
}
```

Using a compensating command

To use a compensating command, you must retrieve the compensator associated with the primary command and call its `execute` method. Figure 78 shows the code used to run the original command and to give the user the option of undoing the work by running the compensating command.

Figure 78. Code example: Using the `ModifyCheckingAccountCompensator` command

```
{
    ...
    CheckingAccount checkingAccount
    ....
    try {
        ModifyCheckingAccountCmd cmd =
            new ModifyCheckingAccountCmdImpl(null, 1000);
        cmd.setCheckingAccount(checkingAccount);
        cmd.execute();
        ...
        System.out.println("Would you like to undo this work? Enter Y or N");
        try {
            // Retrieve and validate user's response
            ...
        }
        ...
        if (answer.equalsIgnoreCase(Y)) {
            Command compensatingCommand = cmd.getCompensatingCommand();
            compensatingCommand.execute();
        }
    }
    catch (Exception e) {
        System.out.println(e.getMessage());
    }
    ...
}
```

Using the WebSphere EJBCCommandTarget bean as a command target

WebSphere ships a CommandTarget enterprise bean to allow administrators to execute a command in a designated server without providing their own implementation of CommandTarget. The EJBCCommandTarget class, along with the EJBCCommandTarget bean (CommandServerSessionBean), are located in the EJBCCommandTarget.jar file in the lib directory under the WebSphere installation directory. This is a deployed jar file. You can use this JAR file in a new application or add it into an existing application.

The EJBCCommandTarget class serves as a wrapper for a CommandTarget bean. CommandServerSessionBean is the WebSphere implementation of this CommandTarget bean. A command developer can set this EJBCCommandTarget object into the Command. [Figure 79](#) shows an example.

Figure 79. Code example: Using an EJBCCommandTarget bean

```
EJBCCommandTarget target = new EJBCCommandTarget();

MyCommand cmd = new MyCommandImpl(Arguments...);
cmd.setCommandTarget(target);
cmd.execute();
```

In this example, the client creates a MyCommand object. It is then executed in the application server. When the execute method is performed, the target (EJBCCommandTarget) looks up the CommandServerSessionHome from the InitialContext and executes the executeCommand method on the CommandServerSessionBean. The EJBCCommandTarget object ensures that there is only one CommandServerSessionBean per object to avoid extra naming lookup.

An EJBCCommandTarget object can be created using four different constructors:

- EJBCCommandTarget("MyNamingServerName", "PortNumber", "JNDIName")
- EJBCCommandTarget(InitialContext, "JNDIName")
- EJBCCommandTarget("JNDIName")
- EJBCCommandTarget()

The first constructor allows the application to specify the naming server name and the port. The JNDI name of the CommandServerSessionBean can also be specified. The EJBCCommandTarget constructs a provider URL of "iiop://MyNamingServerName:PortNumber" and looks up the CommandServerSessionBean with the given JNDI name. If null values are passed in for any of the parameters the WebSphere defaults for server and port and a default JNDI name of CommandServerSession are used.

The second constructor allows the application to specify its own initial context. The EJBCCommandTarget object then uses this initial context to look up the CommandServerSession bean with the specified JNDI name.

The third constructor allows the application to set up the naming server (the provider URL) in property files.

The default constructor uses the default values for the provider URL and default JNDI name for the CommandServerSession bean (CommandServerSession).

You do not need to use the EJBCCommandTarget class. You can instead create your own custom target policy that uses the EJBCCommandTarget bean (CommandServerSessionBean). The EJBCCommandTarget object is a convenience class and attempts to address most usage scenarios

Writing a command target (server)

In order to accept commands, a server must implement the CommandTarget interface and its single method, executeCommand.

The example application implements the CommandTarget interface in an enterprise bean. (For a servlet-based example, see [Writing a command target \(client-side adapter\)](#).) The target enterprise bean can be a session bean or an entity bean. You can write a target enterprise bean that forwards commands to a specific server, such as another entity bean. In this case, all commands directed at a specific target go through the target enterprise bean. You can also write a target enterprise bean that does the work of the command locally.

Make an enterprise bean the target of a command by:

- Extending the `CommandTarget` interface when you define the bean's remote interface, which must also extend the `EJBObject` interface
- Implementing the `CommandTarget` interface when you implement the bean class, which must also implement either the `SessionBean` or `EntityBean` interface

The target of the example application is an enterprise bean called `CheckingAccountBean`. This bean's remote interface, `CheckingAccount`, extends the `CommandTarget` interface in addition to the `EJBObject` interface. The methods declared in the remote interface are independent of those used by the command. The `executeCommand` is declared in neither the bean's home nor remote interfaces. [Figure 80](#) shows the `CheckingAccount` interface.

Figure 80. Code example: The remote interface for the `CheckingAccount` entity bean, also a command target

```
...
import com.ibm.websphere.command.*;
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
public interface CheckingAccount extends CommandTarget, EJBObject {
    float deposit (float amount) throws RemoteException;
    float deposit (int amount) throws RemoteException;
    String getAccountName() throws RemoteException;
    float getBalance() throws RemoteException;
    float setBalance(float amount) throws RemoteException;
    float withdrawal (float amount) throws RemoteException, Exception;
    float withdrawal (int amount) throws RemoteException, Exception;
}
```

The enterprise bean class, `CheckingAccountBean`, implements the `EntityBean` interface as well as the `CommandTarget` interface. The class contains the business logic for the methods in the remote interface, the necessary life-cycle methods (`ejbActivate`, `ejbStore`, and so on), and the `executeCommand` declared by the `CommandTarget` interface. The `executeCommand` method is the only command-specific code in the enterprise bean class. It attempts to run the `performExecute` method on the command and throws a `CommandException` if an error occurs. If the `performExecute` method runs successfully, the `executeCommand` method uses the `hasOutputProperties` method to determine if there are output properties that must be returned. If the command has output properties, the method returns the command object to the client. [Figure 81](#) shows the relevant parts of the `CheckingAccountBean` class.

Figure 81. Code example: The bean class for the `CheckingAccount` entity bean, also a command target

```
...
public class CheckingAccountBean implements EntityBean, CommandTarget {
    // Bean variables
    ...
    // Business methods from remote interface
    ...
    // Life-cycle methods for CMP entity beans
    ...
    // Method from the CommandTarget interface
    public TargetableCommand executeCommand(TargetableCommand command)
    throws RemoteException, CommandException
    {
        try {
            command.performExecute();
        }
        catch (Exception ex) {
            if (ex instanceof RemoteException) {
                RemoveException remoteException = (RemoteException)ex;
                if (remoteException.detail != null) {
                    throw new CommandException(remoteException.detail);
                }
                throw new CommandException(ex);
            }
        }
    }
}
```

```

        if (command.hasOutputProperties()) {
            return command;
        }
        return null;
    }
}

```

Targets and target policies

A targetable command extends the `TargetableCommand` interface, which allows the client to direct a command to a particular server. The `TargetableCommand` interface (and the `TargetableCommandImpl` class) provide two ways for a client to specify a target: the `setCommandTarget` and `setCommandTargetName` methods. (These methods were introduced in [The TargetableCommand interface](#).) The `setCommandTarget` methods allows the client to set the target object directly on the command. The `setCommandTargetName` method allows the client to refer to the server by name; this approach is useful when the client is not directly aware of server objects. A targetable command also has corresponding `getCommandTarget` and `getCommandTargetName` methods.

The command package needs to be able to identify the target of a command. Because there is more than one way to specify the target and because different applications can have different requirements, the command package does not specify a selection algorithm. Instead, it provides a `TargetPolicy` interface with one method, `getCommandTarget`, and a default implementation. This allows applications to devise custom algorithms for determining the target of a command when appropriate.

The default target policy

The command package provides a default implementation of the `TargetPolicy` interface in the `TargetPolicyDefault` class. If you use this default implementation, the command determines the target by looking through an ordered sequence of four options:

1. The `CommandTarget` value
2. The `CommandTargetName` value
3. A registered mapping of a target for a specific command
4. A defined default target

If it finds no target, it returns null. The `TargetPolicyDefault` class provides methods for managing the assignment of commands with targets (`registerCommand`, `unregisterCommand`, and `listMappings`), and a method for setting a default name for the target (`setDefaultTargetName`). The default target name is `com.ibm.websphere.command.LocalTarget`, where `LocalTarget` is a class that runs the command's `performExecute` method locally. [Figure 82](#) shows the relevant variables and the methods in the `TargetPolicyDefault` class.

Figure 82. Code example: The `TargetPolicyDefault` class

```

...
public class TargetPolicyDefault implements TargetPolicy, Serializable
{
    ...
    protected String defaultTargetName = "com.ibm.websphere.command.LocalTarget";
    public CommandTarget getCommandTarget(TargetableCommand command) {
        ... }
    public Dictionary listMappings() {
        ... }
    public void registerCommand(String commandName, String targetName) {
        ... }
    public void unregisterCommand(String commandName) {
        ... }
    public void seDefaultTargetName(String defaultTargetName) {
        ... }
}

```

Setting the command target

The `ModifyCheckingAccountImpl` class provides two command constructors (see [Figure 70](#)). One of them takes a command target as an argument and implicitly uses the default target policy to locate the target. The constructor used in [Figure 77](#) passes a null target, so that the default target policy traverses its choices and eventually finds the default target name, `LocalTarget`.

The example in [Figure 83](#) uses the same constructor to set the target explicitly. This example differs from [Figure 77](#) as follows:

- The command target is set to the checking account rather than null. The default target policy starts to traverse its choices and finds the target in the first place it looks.
- It does not have to call the `setCheckingAccount` method to indicate the account on which the command should operate; the constructor uses the target variable as both the target and the account.

Figure 83. Code example: Identifying a target with `CommandTarget`

```
{
    ...
    CheckingAccount checkingAccount
    ....
    try {
        ModifyCheckingAccountCmd cmd =
            new ModifyCheckingAccountCmdImpl(checkingAccount, 1000);
        cmd.execute();
    }
    catch (Exception e) {
        System.out.println(e.getMessage());
    }
    ...
}
```

Setting the command target name

If a client needs to set the target of the command by name, it can use the command's `setCommandTargetName` method. [Figure 84](#) illustrates this technique. This example compares with [Figure 77](#) as follows:

- Both explicitly set the command target in the constructor to null.
- Both use the `setCheckingAccount` method to indicate the account on which the command should operate.
- This example sets the target name explicitly by using the `setCommandTargetName` method. When the default target policy traverses its choices, it finds a null for the first choice and a name for the second.

Figure 84. Code example: Identifying a target with `CommandTargetName`

```
{
    ...
    CheckingAccount checkingAccount
    ....
    try {
        ModifyCheckingAccountCmd cmd =
            new ModifyCheckingAccountCmdImpl(null, 1000);
        cmd.setCheckingAccount(checkingAccount);
        cmd.setCommandTargetName("com.ibm.sfc.cmd.test.CheckingAccountBean");
        cmd.execute();
    }
    catch (Exception e) {
        System.out.println(e.getMessage());
    }
    ...
}
```

Mapping the command to a target name

The default target policy also permits commands to be registered with targets. Mapping a command to a target is an administrative task that most appropriately done through a configuration tool. The WebSphere Application Server administrative console does not yet support the configuration of mappings between commands and targets. Applications that require support for the registration of commands with targets must supply the tools to manage the mappings. These tools can be visual interfaces or command-line tools.

Figure 85 shows the registration of a command with a target. The names of the command class and the target are explicit in the code, but in practice, these values would come from fields in a user interface or arguments to a command-line tool. If a program creates a command as shown in Figure 77, with a null for the target, when the default target policy traverses its choices, it finds a null for the first and second choices and a mapping for the third.

Figure 85. Code example: Mapping a command to a target in an external application

```
{
    ...
    targetPolicy.registerCommand(
        "com.ibm.sfc.cmd.test.ModifyCheckingAccountImpl",
        "com.ibm.sfc.cmd.test.CheckingAccountBean");
    ...
}
```

Customizing target policies

You can define custom target policies by implementing the `TargetPolicy` interface and providing a `getCommandTarget` method appropriate for your application. The `TargetableCommandImpl` class provides `setTargetPolicy` and `getTargetPolicy` methods for managing custom target policies.

So far, the target of all the commands has been a checking-account entity bean. Suppose that someone introduces a session enterprise bean (`MySessionBean`) that can also act as a command target. Figure 86 shows a simple custom policy that sets the target of every command to `MySessionBean`.

Figure 86. Code example: Creating a custom target policy

```
...
import java.io.*;
import java.util.*;
import java.beans.*;
import com.ibm.websphere.command.*;
public class CustomTargetPolicy implements TargetPolicy, Serializable {
    public CustomTargetPolicy {
        super();
    }
    public CommandTarget getCommandTarget(TargetableCommand command) {
        CommandTarget = null;
        try {
            target = (CommandTarget)Beans.instantiate(null,
                "com.ibm.sfc.cmd.test.MySessionBean");
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Since commands are implemented as JavaBeans components, using custom target policies requires importing the `java.beans` package and writing some elementary JavaBeans code. Also, your custom target-policy class must also implement the `java.io.Serializable` interface.

Using a custom target policy

The `ModifyCheckingAccountImpl` class provides two command constructors (see [Figure 70](#)). One of them implicitly uses the default target policy; the other takes a target policy object as an argument, which allows you to use a custom target policy. The example in [Figure 87](#) uses the second constructor, passing a null target and a custom target policy, so that the custom policy is used to determine the target. After the command is executed, the code uses the `reset` method to return the target policy to the default.

Figure 87. Code example: Using a custom target policy

```
{
    ...
    CheckingAccount checkingAccount
    ....
    try {
        CustomTargetPolicy customPolicy = new CustomTargetPolicy();
        ModifyCheckingAccountCmd cmd =
            new ModifyCheckingAccountCmdImpl(null, 1000, customPolicy);
        cmd.setCheckingAccount(checkingAccount);
        cmd.execute();
        cmd.reset();
    }
    catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
```

Writing a command target (client-side adapter)

Commands can be used with any Java application, but the means of sending the command from the client to the server varies. The application described in [The example application](#) used enterprise beans. The example in this section shows how you can send a command to a servlet over the HTTP protocol.

In this example, the client implements the `CommandTarget` interface locally. [Figure 88](#) shows the structure of the client-side class; it implements the `CommandTarget` interface by implementing the `executeCommand` method.

Figure 88. Code example: The structure of a client-side adapter for a target

```
...
import java.io.*;
import java.rmi.*;
import com.ibm.websphere.command.*;
public class ServletCommandTarget implements CommandTarget, Serializable
{
    protected String hostName = "localhost";
    public static void main(String args[]) throws Exception
    {
        ....
    }
    public TargetableCommand executeCommand(TargetableCommand command)
        throws CommandException
    {
        ....
    }
    public static final byte[] serialize(Serializable serializable)
        throws IOException {
        ... }
    public String getHostName() {
        ... }
    public void setHostName(String hostName) {
        ... }
    private static void showHelp() {
        ... }
}
```



```
}
```

The main method in the client-side adapter constructs and initializes the CommandTarget object, as shown in [Figure 89](#).

Figure 89. Code example: Instantiating the client-side adapter

```
public static void main(String args[]) throws Exception
{
    String hostName = InetAddress.getLocalHost().getHostName();
    String fileName = "MyServletCommandTarget.ser";
    // Parse the command line
    ...
    // Create and initialize the client-side CommandTarget adapter
    ServletCommandTarget servletCommandTarget = new ServletCommandTarget();
    servletCommandTarget.setHostName(hostName);
    ...
    // Flush and close output streams
    ...
}
```

Implementing a client-side adapter

The CommandTarget interface declares one method, executeCommand, which the client implements. The executeCommand method takes a TargetableCommand object as input; it also returns a TargetableCommand. [Figure 90](#) shows the implementation of the method used in the client-side adapter. This implementation does the following:

- Serializes the command it receives
- Creates an HTTP connection to the servlet
- Creates input and output streams, to handle the command as it is sent to the server and returned
- Places the command on the output stream
- Sends the command to the server
- Retrieves the returned command from the input stream
- Returns the returned command to the caller of the executeCommand method

Figure 90. Code example: A client-side implementation of the executeCommand method

```
public TargetableCommand executeCommand(TargetableCommand command)
    throws CommandException
{
    try {
        // Serialize the command
        byte[] array = serialize(command);
        // Create a connection to the servlet
        URL url = new URL
            ("http://" + hostName +
             "/servlet/com.ibm.websphere.command.servlet.CommandServlet");
        HttpURLConnection httpURLConnection =
            (HttpURLConnection) url.openConnection();
        // Set the properties of the connection
        ...
        // Put the serialized command on the output stream
        OutputStream outputStream = httpURLConnection.getOutputStream();
        outputStream.write(array);
        // Create a return stream
        InputStream inputStream = httpURLConnection.getInputStream();
        // Send the command to the servlet
        httpURLConnection.connect();
        ObjectInputStream objectInputStream =
            new ObjectInputStream(inputStream);
        // Retrieve the command returned from the servlet
```



```

        Object object = objectInputStream.readObject();
        if (object instanceof CommandException) {
            throw ((CommandException) object);
        }
        // Pass the returned command back to the calling method
        return (TargetableCommand) object;
    }
    // Handle exceptions
    ....
}

```

Running the command in the servlet

The servlet that runs the command is shown in [Figure 91](#). The service method retrieves the command from the input stream and runs the performExecute method on the command. The resulting object, with any output properties that must be returned to the client, is placed on the output stream and sent back to the client.

Figure 91. Code example: Running the command in the servlet

```

...
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.ibm.websphere.command.*;
public class CommandServlet extends HttpServlet {
    ...
    public void service(HttpServletRequest request,
                        HttpServletResponse response)
        throws ServletException, IOException
    {
        try {
            ...
            // Create input and output streams
            InputStream inputStream = request.getInputStream();
            OutputStream outputStream = response.getOutputStream();
            // Retrieve the command from the input stream
            ObjectInputStream objectInputStream =
                new ObjectInputStream(inputStream);
            TargetableCommand command = (TargetableCommand)
                objectInputStream.readObject();
            // Create the command for the return stream
            Object returnObject = command;

            // Try to run the command's performExecute method
            try {
                command.performExecute();
            }
            // Handle exceptions from the performExecute method
            ...

            // Return the command with any output properties
            ObjectOutputStream objectOutputStream =
                new ObjectOutputStream(outputStream);
            objectOutputStream.writeObject(returnObject);
            // Flush and close output streams
            ...
        }
        catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}

```

```
}
```

In this example, the target invokes the `performExecute` method on the command, but this is not always necessary. In some applications, it can be preferable to implement the work of the command locally. For example, the command can be used only to send input data, so that the target retrieves the data from the command and runs a local database procedure based on the input. You must decide the appropriate way to use commands in your application.

The localizable-text package

Overview

Users of distributed applications can come from widely varying areas; they can speak different languages, represent dates and times in regionally specific ways, and use different currencies. An application intended to be used by such an audience must either force them all to use the same interface (for example, an English-based interface), or it can be written in such a way that it can be configured to the linguistic conventions of the users, so English-speaking users can use the English interface but French-speaking users can interact with the application through a French interface.

An application that can present information to users in formats that abide by the users' linguistic conventions is said to be *localizable*: the application can be configured to interact with users from different localities in linguistically appropriate ways. In a localized application, a user in one region sees error messages, output, and interface elements (like menu options) in the requested language. Additionally, other elements that are not strictly linguistic, like date and time formats and currencies, are presented in the appropriate style 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 localizable applications has been restricted to large corporations writing complex systems. The strategies for writing localizable code, collectively called *internationalization techniques*, have traditionally been expensive and difficult to implement, so they have been applied only to major development efforts. However, given the rise in distributed computing and in use of the World Wide Web, application developers have been pressured to make a much wider variety of applications localizable. This requires making internationalization--the techniques for writing localizable programs--much more accessible to application developers. The WebSphere localizable-text package is a set of Java classes and interfaces that can be used by WebSphere application developers to localize distributed WebSphere applications easily. Language catalogs for distributed WebSphere applications can be stored centrally, so the catalogs can be maintained and administered efficiently.

Writing localizable programs

In a nonlocalizable application, parts of the application that a user sees are unalterably coded into the application. For example, a routine that prints an error message simply prints a string, probably in English, to a file or the console. A localizable program adds a layer of abstraction into the design. Instead of going simply from error condition to output string, a localizable program represents error messages with some language-neutral information; in the simplest case, each error condition corresponds to a key. In order to print a usable error string for the user, the application looks up the key in the configured message catalog. A message catalog is a list of keys with corresponding strings. Different message catalogs provide the strings in different languages. The application looks up the key in the appropriate catalog, retrieves the corresponding error message in the desired language, and prints this string for the user.

The technique of localization can be used for far more than translating error messages. For example, by using keys to represent each element--button, label, menu item, and so forth--in a graphical user interface and by providing a message catalog containing translations of the button names, labels, and menu items, the graphical interface can be automatically translated into multiple languages. In addition, extending support to additional languages requires providing message catalogs for those languages; the application itself requires no modification.

Localization of an application is driven by two variables, the time zone and the locale. The time zone variable indicates how to compute the local time as an offset from a standard time like Greenwich Mean Time. The locale is a collection of information that indicates a geographic, political, or cultural region. It provides information on language, currency, and the conventions for presenting information like dates, and in a localizable program, the locale also indicates the message catalog from which an application retrieves messages. 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.

Identifying localizable text

To write a localizable application, an application developer must determine which aspects of the application need to be translatable. These are typically the parts of an application a user must read and understand. Application developers must consider the parts of an application with which all users directly interact, like the application's interface, and the parts serving more specialized purposes, like messages in log files. Good candidates for localization include:

- Elements in graphical user interfaces
 - Title bars for windows
 - Menu names, and the items on the menus (for example, "select File > Open")
 - Labels on buttons (for example, "click the OK button")
 - Instructions directing users to fill in fields (for example, "enter the account number")
 - Any other elements that users must read
- Prompts in command-line interfaces
- Output from the program
 - Responses to user input
 - Error messages
 - Text returned when exceptions are thrown
 - Other status messages (warnings, audit messages, and others)

After identifying each element of the application to be localized, application developers must assign a unique key to each element and provide a message catalog for each language to be supported. Each message catalog consists of keys and the corresponding language-specific strings. The key, therefore, is the link between the program and the message catalog; the program internally refers to localizable elements by key and uses the message catalog to generate the output seen by the user. Translated strings are generated by calling the format method on a `LocalizableTextFormatter` object, which represents a key and a resource bundle (a set of message catalogs). The locale setting of the program determines the message catalog in which to search for the key.

Creating message catalogs

After identifying each element to be localized, message catalogs must be created for each language to be supported. These catalogs, which are implemented as Java resource bundles, can be created in two ways, either as subclasses of the `ResourceBundle` class or as Java properties files. Resource bundles have a variety of uses in Java; for message catalogs, the properties-file approach is more common. If properties files are used, support for languages to be added or removed without modifying the application code, and catalogs can be prepared by people without programming expertise.

A message catalog implemented in a properties file consists of a line for each key, where a key identifies a localizable element. Each line in the file has the following structure:

```
key = String corresponding to the key
```

For example, a graphical user interface for a banking system can have a pull-down menu to be used for selecting a type of account, like savings or checking. The label for the pull-down menu and the account types on the menu are good choices for localization. There are three elements that require keys: the label for the account menu and the two items on the menu. If the keys are `accountString`, `savingsString`, and `checkingString`, the English properties file associates each with an English string.

Figure 92. Three elements in an English message catalog

```
accountString = Accounts
savingsString = Savings
checkingString = Checking
...
```

In the German properties files, each key is given a corresponding German value.

Figure 93. Three elements in a German message catalog

```
accountString = Konten
savingsString = Sparkonto
```

```
checkingString = Girokonto  
...
```

Properties files can be added for any other needed languages, as well.

Naming the properties files

To enable resolution to a specific properties file, Java specifies naming conventions for the properties files in a resource bundle: `resourceBundleName_localeID.properties`

Each file takes a fixed extension, `.properties`. The set of files making up the resource bundle is given a collective name; for a simple banking application, an obvious name, like `BankingResources`, suffices for the resource bundle. Each file is given the name of the resource bundle with a locale identifier; the specific value of the locale ID varies with the locale. These are used internally by the `Java.util.ResourceBundle` class to match files in a resource bundle to combinations of locale and time-zone settings. The details of the algorithm vary with the release of the JDK; see your Java documentation for information specific to your installation.

In the banking application, typical files in the `BankingResources` resource bundle include `BankingResources_en.properties` for the English message catalog and `BankingResources_de.properties` for the German catalog. Additionally, a default catalog, `BankingResources.properties`, is provided for use when the requested catalog cannot be found. The default catalog is often the English-language catalog.

Resource bundles containing message catalogs for use with localizable text need to be installed only on the systems where the formatting of strings is actually performed. The resource bundles are typically placed in an application's JAR file. See [WebSphere support](#) for more information.

Localization support in WebSphere and Java

The Java package `com.ibm.websphere.i18n.localizabletext` contains the classes and interfaces constituting the localizable-text package. This package makes extensive use of the internationalization and localization features of the Java language; programmers using the WebSphere localizable-text package must understand the underlying Java support, which are not documented in any detail here.

Java support

The WebSphere localizable-text package relies primarily on the following Java components:

- `java.util.Locale`
- `java.util.TimeZone`
- `java.util.ResourceBundle`
- `java.text.MessageFormat`

This list is not exhaustive. WebSphere and these Java classes can also use related Java classes, but the related classes--for example, `java.util.Calendar`--are typically special-purposes classes. This section briefly describes only the primary classes.

Locale

A `Locale` object in Java encapsulates a language and a geographic region, for example, the `java.util.Locale.US` object contains locale information for the United States. An application that specifies a locale can then take advantage of the locale-sensitive formatters built into the Java language. These formatters, in the `java.text` package, handle the presentation of numbers, currency values, dates, and times.

TimeZone

A `TimeZone` object in Java encapsulates a representation of the time and provides methods for tasks like reporting the time and accommodating seasonal time shifts. Applications use the time zone to determine the local date and time.

ResourceBundle

A resource bundle is a named collection of resources--information used by the application, for example, strings, fonts, and images--used by a specific locale. The `ResourceBundle` class allows an application to retrieve the named resource bundle appropriate to the locale. Resource bundles are used to hold the messages catalogs, as described in [Writing localizable programs](#). Resource bundles can be implemented in two ways, either as subclasses of the `ResourceBundle` class or as Java properties files.

MessageFormat

The `MessageFormat` class can be used to construct strings based on parameters. As a simple example, suppose a localized application represents a particular error condition with a numeric key. When the application reports the error condition, it uses a message formatter to convert the numeric key into a meaningful string. The message formatter constructs the output string by looking up the code (the parameter) in an appropriate resource bundle and retrieving the corresponding string from the message catalog. Additional parameters--for example, another key representing the program module--can also be used in assembling the output message.

WebSphere support

The WebSphere localizable-text package wraps the Java support and extends it for efficient and simple use in a distributed environment. The primary class used by application programmers is the `LocalizableTextFormatter` class. Objects of this class are created, typically in server programs, but clients can also create them. `LocalizableTextFormatter` objects are created for specific resource-bundle names and keys. Client programs that receive `LocalizableTextFormatter` objects call the object's `format` method. This method uses the locale of the client application to retrieve the appropriate resource bundle and assemble the locale-specific message based on the key.

For example, suppose that a WebSphere client-server application supports both French and English locales; the server is using an English locale and the client, a French locale. The server creates two resource bundles, one for English and one for French. When the client makes a request that triggers a message, the server creates a `LocalizableTextFormatter` object containing the name of the resource bundle and the key for the message, and passes the object back to the client.

When the client receives the `LocalizableTextFormatter` object, it calls the object's `format` method, which returns the message corresponding to the key from the French resource bundle. The `format` method retrieves the client's locale and, using the locale and name of the resource bundle, determines the resource bundle corresponding to the locale. (If the client has set an English locale, calling the `format` method results in the retrieval of an English message.) The formatting of the message is transparent to the client. In this simple client-server example, the resource bundles reside centrally with the server. The client machine does not have to install them. Part of what the WebSphere localizable-text package provides is the infrastructure to support centralized catalogs. WebSphere uses an enterprise bean, a stateless session bean provided with the localizable-text package, to access the message catalogs. When the client calls the `format` method on the `LocalizableTextFormatter` object, the following events occur internally:

1. The client application sets the time zone and locale values in the `LocalizableTextFormatter` object, either by passing them explicitly or through defaults.
2. A call, `LocalizableTextFormatterEJBFinder`, is made to retrieve a reference to the formatting enterprise bean.
3. Information from the `LocalizableTextFormatter` object, including the client's time zone and locale, is sent to the formatting bean.
4. The formatting bean uses the name of the resource bundle, the message key, the time zone, and the locale to assemble the language-specific message.
5. The enterprise bean returns the formatted message to the client.
6. The formatted message is inserted into the `LocalizableTextFormatter` object and returned by the `format` method.

A call to a `LocalizableTextFormatter.format` method requires at most one remote invocation, to contact the formatting enterprise bean. However, the `LocalizableTextFormatter` object can optionally cache formatted messages, eliminating the formatting call for subsequent uses. It also allows the application to set a fallback string; this means the application can still return a readable string even if it cannot access a message catalog to retrieve the language-specific string. Additionally, the resource bundles can be stored locally. The localizable-text package provides a static variable that indicates whether the bundles are stored locally (`LocalizableConfiguration.LOCAL`) or remotely (`LocalizableConfiguration.REMOTE`), but the setting of this variable applies to all applications running within a Java Virtual Machine (JVM).

The LocalizableTextFormatter class

The `LocalizableTextFormatter` class, found in the package `com.ibm.websphere.i18n.localizabletext`, is the primary programming interface for using the localizable-text package. Objects of this class contain the information needed to create language-specific strings from keys and resource bundles.

Location of message catalogs and the `ApplicationName` value

Applications written with the WebSphere localizable-text package can store message catalogs locally or remotely. In a distributed environment, the use of remote, centrally stored catalogs is appropriate. All applications can use the same catalogs, and administration and maintenance of the catalogs are simplified; each component does not need to store and maintain copies of the message catalogs. Local formatting is useful in test situations and appropriate under some circumstances. In order to support both local and remote formatting, a `LocalizableTextFormatter` object must indicate the name of the formatting application. For example, when an application formats a message by using remote, centrally stored catalogs, the message is actually formatted by a simple enterprise bean (see [WebSphere support](#) for more information). Although the localizable-text package contains the code to automate looking up the enterprise bean and issuing a call to it, the application needs to know the name of the formatting enterprise bean. Several methods in the `LocalizableTextFormatter` class use a value described as *application name*; this refers to the name of the formatting application, which is not necessarily the name of the application in which the value is set.

Caching messages

The `LocalizableTextFomatter` object can optionally cache formatted messages so that they do not have to be reformatted when needed again. By default, caching is not used, but the `LocalizableTextFormatter.setCacheSetting` method can be used to enable caching. When caching is enabled and the `LocalizableTextFormatter.format` method is called, the method determines whether the message has already been formatted. If so, the cached message is returned. If the message is not found in the cache, the message is formatted and returned to the caller, and a copy of the message is cached for future use.

If caching is disabled after messages have been cached, those messages remain in the cache until the cache is cleared by a call to the `LocalizableTextFormatter.clearCache` method. The cache can be cleared at any time. The cache within a `LocalizableTextFormatter` object is automatically cleared when any of the following methods are called on the object:

- `setResourceBundleName(String resourceBundleName)`
- `setPatternKey(String patternKey)`
- `setArguments(Object[] args)`
- `setApplicationName(String appName)`

Fallback information

Under some circumstances, it can be impossible to format a message. The localizable-text package implements a fallback strategy, making it possible to get some information even if a message cannot be correctly formatted into the desired language. The `LocalizableTextFomatter` object can optionally store a fallback value for a message string, the time zone, and the locale. These can be ignored unless the `LocalizableTextFormatter` object throws an exception.

Application-specific variables

The localizable-text package provides native support for localization based on time zone and locale, but application developers can construct messages on the basis of other values as well. The localizable-text package provides an illustrative class, `LocalizableTextDateTimeArgument`, which reports the date and time. The date and time information is localized by using the locale and time-zone values, but the class also uses additional variables to determine how the output is presented. The date and time information can be requested in a variety of styles, from the fully detailed to the terse. In this example, the construction of message strings is driven by three variables: the locale, the time zone, and the style. Applications can use any number of variables in addition to locale and time zone for constructing messages. See [Using optional arguments](#) for more information.

Writing a localizable application

To develop a WebSphere application that uses localizable text, application developers must do the following:

- Determine the parts of the application to be localized.
 - Identify the application elements to be localized and assign each a key.

- Create message catalogs for each language by associating a string with each key.

These tasks were described previously. See [Identifying localizable text](#) and [Creating message catalogs](#) for more information.

- Assemble language-specific strings from keys, resource bundles, and other arguments.
 - Create a `LocalizableTextFormatter` object.
 - Set the values within the object for the key, the name of the resource bundle, the name of the remote formatting application, and any optional arguments.
 - Call the `format` method on the `LocalizableTextObject`, which returns the assembled string.

This section describes these tasks.

Creating a `LocalizableTextFormatter` object

Server programs typically create `LocalizableTextFormatter` objects, which are sent to clients as the result of some operation; clients format the objects at the appropriate time. Less typically, clients can create `LocalizableTextFormatter` objects locally. To create a `LocalizableTextFormatter` object, applications use one of the constructors in the `LocalizableTextFormatter` class:

- `LocalizableTextFormatter()`
- `LocalizableTextFormatter(String resourceBundleName, String patternKey, String appName)`
- `LocalizableTextFormatter(String resourceBundleName, String patternKey, String appName, Object[] args)`

The `LocalizableTextFormatter` object must have values set for the name of the resource bundle, the key, the name of the formatting application, and for any optional values so the object can be formatted. The `LocalizableTextFormatter` object can be created and the values set in one step by using the constructor that takes the necessary arguments, or the object can be created and the values set in separate steps. Values are set by using methods on the `LocalizableTextFormatter` object; for setting the values manually, rather than by using a constructor, use these methods:

- `setResourceBundleName(String resourceBundleName)`
- `setPatternKey(String patternKey)`
- `setApplicationName(String appName)`
- `setArguments(Object[] args)`

Note:

When values in the array of optional arguments are set within a `LocalizableTextFormatter` object, they are copied into the object, not referenced. If an array variable holding a value is changed after the value has been copied into the `LocalizableTextFormatter` object, the value in the `LocalizableTextFormatter` object will not reflect the change unless it is also reset.

A `LocalizableTextFormatter` object also has methods that can be used to set values that cannot be set when the object is created, for example:

- To toggle the cache setting for the `LocalizableTextFormatter` object, use the `setCacheSetting(boolean setting)` method (See [Caching messages](#) for more information.)
- To clear the cache, use the `clearLocalizableTextFormatter` method
- To set fallback values, use these methods:
 - `setFallBackString`
 - `setFallBackLocale`
 - `setFallBackTimeZone`

(See [Fallback information](#) for more information.)

Each of these set methods also has a corresponding get method for retrieving the value. The `clearLocalizableTextFormatter` method unsets all values, returning the `LocalizableTextFormatter` object to a blank state. After clearing the object, reuse the object by setting new values and calling the `format` method again.

[Figure 94](#) creates a `LocalizableTextFormatter` object by using the default constructor and uses methods on the new object to set values for the key, name of the resource bundle, name of the formatting application, and fallback string on the object.

Figure 94. Code example: Creating a `LocalizableTextFormatter` object and setting values on it

```
import com.ibm.websphere.i18n.localizabletext.LocalizableException;
import com.ibm.websphere.i18n.localizabletext.LocalizableTextFormatter;
import java.util.Locale;
public void drawAccountNumberGUI(String accountType) {
    ...
    LocalizableTextFormatter ltf = new LocalizableTextFormatter();
    ltf.setPatternKey("accountNumber");
    ltf.setResourceBundleName("BankingSample.BankingResources");
    ltf.setApplicationName("BankingSample");
    ltf.setFallBackString("Enter account number: ");
    ...
}
```

Setting localization values

The application requesting a localized message can specify the locale and time zone for which the message is to be formatted, or the application can use the default values set for the JVM. For example, a graphical user interface can allow users to select the language in which to display the menus. A default value must be set, either in the environment or programmatically, so the menus can be generated when the application first starts, but users can then change the menu language to suit their needs. [Figure 95](#) illustrates how to change the locale used by the application based on the selection of a menu item.

Figure 95. Code example: Setting the locale programmatically

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
...
import java.util.Locale;
public void actionPerformed(ActionEvent event) {
    String action = event.getActionCommand();
    ...
    if (action.equals("en_us")) {
        applicationLocale = new Locale("en", "US");
        ...
    }
    else if (action.equals("de_de")) {
        applicationLocale = new Locale("de", "DE");
        ...
    }
    else if (action.equals("fr_fr")) {
        applicationLocale = new Locale("fr", "FR");
        ...
    }
    ...
}
```

When an application calls a format method, it can specify no arguments, which causes the message to be formatted using the JVM's default values for locale and time zone, or a combination of locale and time zone can be specified to override the JVM's defaults. (See [Generating the localized text](#) for more information on the arguments to the format methods.)

Generating the localized text

After the `LocalizableTextFormatter` object has been created and the appropriate values set, the object can be formatted to generate the string appropriate to the locale and time zone. The format methods in the `LocalizableTextFormatter` class perform the work necessary to generate a string from a set of message keys and resource bundles, based on locale and time zone. The `LocalizableTextFormatter` class provides four format methods. Each format method returns the formatted

message string. The methods take a combination of `java.util.Locale` and `java.util.TimeZone` objects and throw `LocalizableException` objects:

- `String format();`
- `String format(locale);`
- `String format(timeZone);`
- `String format(locale, timeZone);`

The `format` method with no arguments uses the locale and time-zone values set as defaults for the JVM. The other `format` methods can be used to override either or both of these values.

[Figure 96](#) shows the creation of a localized string for the `LocalizableTextFormatter` object created in [Figure 94](#); formatting is based on the locale set in [Figure 95](#). If the formatting fails, the application retrieves and uses the fallback string instead of the localized string.

Figure 96. Code example: Formatting a `LocalizableTextFormatter` object

```
import com.ibm.websphere.i18n.localizabletext.LocalizableException;
import com.ibm.websphere.i18n.localizabletext.LocalizableTextFormatter;
import java.util.Locale;
public void drawAccountNumberGUI(String accountType) {
    ...
    LocalizableTextFormatter ltf = new LocalizableTextFormatter();
    ltf.setPatternKey("accountNumber");
    ltf.setResourceBundleName("BankingSample.BankingResources");
    ltf.setApplicationName("BankingSample");
    ltf.setFallBackString("Enter account number: ");
    try {
        msg = new Label (ltf.format(this.applicationLocale) , Label.CENTER);
    }
    catch (LocalizableException le) {
        msg = new Label(ltf.getFallBackString(), Label.CENTER);
    }
    ...
}
```

Using optional arguments

The `localizable-text` package allows users to specify an array of optional arguments in a `LocalizableTextFormatter` object. These optional arguments can greatly enhance the kinds of localization done in WebSphere applications. This section describes two ways in which applications can use the optional arguments:

- To assemble and format complex strings with variable substrings
- To customize the formatting of strings, taking variables other than locale and time zone into account

Assembling complex strings

All of the keys discussed so far have represented flat strings; during localization, a string in the appropriate language is substituted for the key. The `localizable-text` package also supports substitution into the strings, which can include variables as placeholders. For example, an application that needs to report that an operation on a specified account was successful must provide a string like "The operation on account number was successful"; the variable number is to be replaced by the actual account number. Without support for creating strings with variable pieces, each possible string would need its own key, or the strings would have to be built phrase by phrase.

Both of these approaches quickly become intractable if a variable can take many values or if a string has several variable components. Instead, the `localizable-text` package supports substitution of variables in strings with optional arguments. A string in a message catalog uses integers in braces--for example, `{0}` or `{1}`--to represent variable components. [Figure 97](#) shows an example from an English message catalog for a string with a single variable substitution. (The same key in message catalogs for other languages has a translation of this string with the variable in the appropriate location for the language.)

Figure 97. A message-catalog entry with a variable substring

successfulTransaction = The operation on account {0} was successful.

The values that are substituted into the string come from an array of optional arguments. One of the constructors for `LocalizableTextFormatter` objects takes an array of objects as an argument, and such an array of objects can be set within any `LocalizableTextFormatter` object. The array is used to hold values for variable parts of a string. When a format method is called on the object, the array is passed to the format method, which takes an element of the array and substitutes it into a placeholder with the matching index in the string. The value at index 0 in the array replaces the {0} variable in the string, the value at index 1 replaces {1}, and so forth.

Figure 98 shows the creation of a single-element argument array and the creation and use of a `LocalizableTextFormatter`. The element in the argument array is the account number entered by the user. The `LocalizableTextFormatter` is created by using a constructor that takes the array of optional arguments; this can also be set directly by using the `setArguments` method on the `LocalizableTextFormatter` object. Later in the code, the application calls the format method. The format method automatically substitutes values from the array of arguments into the string returned from the appropriate message catalog.

Figure 98. Code example: Formatting a message with a variable substring

```
public void updateAccount(String transactionType) {  
    ...  
    Object[] arg = { new String(this.accountNumber)};  
    ...  
    LocalizableTextFormatter successLTF =  
        new LocalizableTextFormatter("BankingResources",  
                                     "successfulTransaction",  
                                     "BankingSample",  
                                     arg);  
    ...  
    successLTF.format(this.applicationLocale);  
    ...  
}
```

Nesting LocalizableTextFormatter objects

The ability to substitute variables into the strings in message catalogs adds a level of flexibility to the localizable-text package, but the additional flexibility is limited, at least in an international environment, unless the substituted arguments themselves can be localized. For example, if an application needs to report that an operation on a specific account was successful, a string like "The operation on account number was successful"--where the only variable is an account number--can be translated and used in message catalogs for multiple languages. A string in which a variable is also a string, for example, "The type operation on account number was successful"--where the new type variable takes values like "deposit" and "withdrawal"--cannot be as easily translated. The values assumed by the type variable also need to be localized.

Figure 99 shows a message string in an English catalog with two variables, one of which will be localized, and the keys for two possible values. (The second variable in the string, the account number, is simply a number that must be substituted into the string; it does not need to be localized.)

Figure 99. A message-catalog entry with two variable substrings

```
sucessfulTransaction = The {0} operation on account {1} was successful.  
depositOpString = deposit  
withdrawOpString = withdrawal
```

To support localization of substrings, the localizable-text package allows the nesting of `LocalizableTextFormatter` objects. This is done simply by inserting a `LocalizableTextFormatter` object into the array of arguments for another `LocalizableTextFormatter`. When the format method does the variable substitution, it formats any `LocalizableTextFormatter` objects as it substitutes array elements for variables. This allows substrings to be formatted independently of the string in which they are embedded.

Figure 100 modifies the example in Figure 98 to format a message with a localizable substring. First, a `LocalizableTextFormatter` object for the localizable substring (referring to a deposit operation) is created. This object is

inserted, along with the account-number information, into the array of arguments. The array of arguments is then used in constructing the `LocalizableTextFormatter` object for the complete string; when the format method is called, the embedded `LocalizableTextFormatter` object is formatted to replace the first variable, and the account number is substituted for the second variable.

Figure 100. Code example: Formatting a message with a localizable variable substring

```
public void updateAccount(String transactionType) {
    ...
    // Successful Deposit.
    LocalizableTextFormatter opLTF =
        new LocalizableTextFormatter("BankingResources",
                                     "depositOpString", "BankingSample");
    Object[] args = {opLTF, new String(this.accountNumber)};
    LocalizableTextFormatter successLTF =
        new LocalizableTextFormatter("BankingResources",
                                     "successfulTransaction",
                                     "BankingSample",
                                     args);
    ...
    successLTF.format(this.applicationLocale);
    ...
}
```

Customizing the behavior of a format method

The array of optional arguments can contain simple values, like an account number to be substituted into a formatted string, and other `LocalizableTextFormatter` objects, representing localizable substrings to be substituted into a larger formatted string. These techniques are described in [Assembling complex strings](#). In addition, the optional-argument array can contain objects of user-defined classes.

User-defined classes used as optional arguments provide application-specific format methods, which programmers can use to perform localization on the basis of any number of values, not just locale and time zone. These user-defined classes need to be available only on the systems where they are constructed and inserted into `LocalizableTextFormatter` objects and where the actual formatting is done; client applications do not need to install these classes.

The localizable-text package provides an example of such a user-defined class in the `LocalizableTextDateTimeArgument` class. This class allows date and time information to be selectively formatted according to the style values defined in the `java.text.DateFormat` class and according to the constants defined by the `LocalizableTextDateTimeArgument` class.

The `DateFormat` styles determine how information is reported about a date. For example, when the `DateFormat.FULL` style is chosen, the twenty-second day of February in 2000 is represented in English as *Tuesday, February 22, 2000*. When the `DateFormat.SHORT` style is used, the same date is represented as *2/22/00*. The valid values are:

- `DateFormat.FULL`
- `DateFormat.LONG`
- `DateFormat.MEDIUM`
- `DateFormat.SHORT`
- `DateFormat.DEFAULT`

The `LocalizableTextDateTimeArgument` class defines constants that can be used to request only date or time information, or both, either in date-time order or in time-date order. The defined values are:

- `LocalizableTextDateTimeArgument.TIME`
- `LocalizableTextDateTimeArgument.DATE`
- `LocalizableTextDateTimeArgument.TIMEANDDATE`
- `LocalizableTextDateTimeArgument.DATEANDTIME`

An object of a user-defined class like the `LocalizableTextDateTimeArgument` class can be set in the optional-argument array of a `LocalizableTextFormatter` object, and when the `LocalizableTextFormatter` object attempts to format the

user-defined object, it calls the format method on that object. That format method, written by the application developer, can do whatever is appropriate with the application-specific values. In the case of the `LocalizableTextDateTimeArgument` class, the format method determines if date, time, or both are required, formats them according to the `DateFormat` value, and assembles them in the order requested in the `LocalizableTextDateTimeArgument` style. The date and time information are also affected by the locale and time-zone values, but the refinements in the formatting are accomplished by the `DateFormat` class and the user-defined values.

The string assembled from a user-defined class like the `LocalizableTextDateTimeArgument` class can then be substituted into a larger string, just as the return values of nested `LocalizableTextFormatter` objects can be. When writing such user-defined classes, it is helpful to think of them as specialized versions of the generic `LocalizableTextFormatter` class, and the way in which the `LocalizableTextFormatter` class is written provides a model for writing user-defined classes.

Structure of the LocalizableTextFormatter class

The `LocalizableTextFormatter` class is a general-purpose class for localizable text. It extends the `java.lang.Object` class and implements the `java.io.Serializable` interface and four localizable-text interfaces:

- LocalizableTextLTZ
- LocalizableTextL
- LocalizableTextTZ
- LocalizableText

Each of the localizable-text interfaces implemented by the `LocalizableTextFormatter` class implements the `Localizable` interface (which simply extends the `Serializable` interface) and defines a single `format` method:

- The `LocalizableTextLTZ` interface defines `format(locale, timezone)`.
- The `LocalizableTextL` defines `format(locale)`.
- The `LocalizableTextTZ` defines `format(timezone)`.
- The `LocalizableText` defines `format()`.

Because the `LocalizableTextFormatter` class implements all four of these interfaces, it must provide an implementation for each of these format methods.

Writing a user-defined class

A user-defined class must implement at least one of the localizable-text interfaces and its corresponding format method, as well as the Serializable interface. If the class implements more than one of the localizable-text interfaces and format methods, the order of evaluation of the interfaces is:

1. LocalizableTextLTZ
2. LocalizableTextL
3. LocalizableTextTZ
4. LocalizableText

For example, the `LocalizableTextDateTimeArgument` class implements only the `LocalizableTextLTZ` interface, as shown in [Figure 101](#).

Figure 101. Code example: The structure of the LocalizableTextDateTimeArgument class

[illegible]

A user-defined class must contain a constructor and an implementation of the format methods as defined in the localizable-text interfaces that the class implements. It can also contain other methods as needed. The `LocalizableTextDateTimeArgument` class contains a constructor, a single format method, an equality method, a hash-code generator, and a string-conversion method.

Figure 102. Code example: The methods in the `LocalizableTextDateTimeArgument` class

```
...
public class LocalizableTextDateTimeArgument implements LocalizableTextLTZ,
                                                         Serializable
{
    public final static int DATE = 1;
    public final static int TIME = 2;
    public final static int DATEANDTIME = 3;
    public final static int TIMEANDDATE = 4;
    private Date date = null;
    private dateTimeStyle = LocalizableTextDateTimeArgument.DATE;
    private int dateFormatStyle = DateFormat.FULL;
    ...
    public LocalizableTextDateTimeArgument(Date date, int dateTimeStyle,
                                           int dateFormatStyle)
    { ... }
    public boolean equals(Object param)
    { ... }
    public format (Locale locale, TimeZone timeZone)
        throws IllegalArgumentException
    { ... }

    public int hashCode()
    { ... }

    public String toString()
    { ... }
}
```

Each format method in the user-defined class can do whatever is appropriate for the application. In the `LocalizableTextDateTimeArgument` class, the format method (see [Figure 103](#) for the implementation) examines the setting of the date-time style set within the object, for example, `DATEANDTIME`. It then assembles the requested information in the requested order, according to the date-format value.

Figure 103. Code example: The format method in the `LocalizableTextDateTimeArgument` class

```
public format (Locale locale, TimeZone timeZone)
    throws IllegalArgumentException
{
    String returnString = null;

    switch(dateTimeStyle) {
        case LocalizableTextDateTimeArgument.DATE :
        {
            returnString = DateFormat.getDateInstance(dateFormatStyle,
                                                         locale).format(date);
            break;
        }
        case LocalizableTextDateTimeArgument.TIME :
        {
            df = DateFormat.getTimeInstance(dateFormatStyle, locale);
            df.setTimeZone(timeZone);
            returnString = df.format(date);
            break;
        }
        case LocalizableTextDateTimeArgument.DATEANDTIME :
```

```

    {
        dateString = DateFormat.getDateInstance(dateFormatStyle,
                                                  locale).format(date);
        df = DateFormat.getTimeInstance(dateFormatStyle, locale);
        df.setTimeZone(timeZone);
        timeString = df.format(date);
        returnString = dateString + " " + timeString;
        break;
    }
    case LocalizableTextDateTimeArgument.TIMEANDDATE :
    {
        dateString = DateFormat.getDateInstance(dateFormatStyle,
                                                  locale).format(date);
        df = DateFormat.getTimeInstance(dateFormatStyle, locale);
        df.setTimeZone(timeZone);
        returnString = timeString + " " + dateString;
        break;
    }
    default :
    {
        throw new IllegalArgumentException();
    }
}
return returnString;
}

```

An application can create a `LocalizableTextDateTimeArgument` object (or an object of any other user-defined class) and place it in the optional-argument array of a `LocalizableTextFormatter` object. When the `LocalizableTextFormatter` object reaches the user-defined object, it will attempt to format it by calling the object's `format` method. The returned string is then substituted for a variable as the `LocalizableTextFormatter` processes each element in the array of optional arguments.

Deploying the formatter enterprise bean

The **LocalizableTextEJBDeploy** tool is used by the application deployer to create a deployed `LocalizableText` JAR file for the `LocalizableText` service. You must deploy the enterprise bean for each server per application where the service is to be run. There may be servers for which the `LocalizableText` service does not need to be installed. The same deployed JAR file can be included in several application Enterprise Archive (EAR) files, but additional steps are required when the EAR file is deployed. The application deployer must also make sure that the application resource bundles are added to the application EAR file as files. The server's `CLASSPATH` variable must be adjusted to include the deployed location of the EAR file. This is so that the resource bundles can be located on the host and server.

Setting up the tool

Before the **LocalizableTextEJBDeploy** tool can be used, the following conditions must be met:

- A JAR file called `ltext.jar` must exist in the `lib` directory under the WebSphere installation directory.
- A working directory has to exist for the tool to use. The location is passed to the tool.

Using the LocalizableTextEJBDeploy Tool

After the prerequisites for the tool have been met, the tool can be used to deploy formatting session beans. The tool requires values for five arguments:

```

LocalizableTextEJBDeploy -a <appName>
                        -h <hostName>
                        -i <installationDir>
                        -s <serverName>
                        -w <workingDir>

```

The required arguments, which can be specified in any order, follow:

- **appName:** The name of the formatting session bean. This name is used in `LocalizableTextFormatter` objects to specify where the actual formatting takes place. If a `LocalizableTextFormatter` object specifies a name that cannot be resolved, an exception is thrown by the `format` method.
- **hostName:** The name of the machine on which the formatting session bean is deployed. This value specified here is case sensitive on all platforms.
- **installationDir:** The location at which WebSphere Application Server is installed on the machine.
- **serverName:** The name of the WebSphere Application Server. If this argument is not specified, the value `Default Server` is used.
- **workingDir:** The name of the working directory for the tool to use.

After the tool is run, a deployed JAR file is located in the working directory specified to the tool. This JAR file can be included in the application EAR or WAR file.

Special considerations when deploying a `LocalizableText` enterprise bean

When the application is being deployed onto a host and server, during the deployment process you will be asked if you want to regenerate the deployment code for the `LocalizableText` enterprise bean. Do not redeploy the bean. If the bean is redeployed, the JNDI name will be wrong.

If more than one `LocalizableText` enterprise bean is deployed with an application, there are two ways to handle the situation.

- Run the **`LocalizableTextEJBDeploy`** tool for each host/server combination. The tool generates a unique JNDI name for each enterprise bean. Otherwise, even though the bean has been deployed on multiple hosts and servers, the JNDI name is not changed, and there is only one entry in the naming service.
- During the deployment of the application, change the JNDI name for the localizable-text bean should begin with `com/ibm/websphere/i18n/localizabletext/homes/`. This should be followed by the application and host names, the server name, and by the string `LocalizableTextEJBHome`, all separated by two underscores, as follows:

```
<AppName>/<HostName>__<ServerName> __LocalizableTextEJBHome
```

More-advanced programming concepts for enterprise beans

This chapter discusses some of the more advanced programming concepts associated with developing and using enterprise beans. It includes information on developing entity beans with bean-managed persistence (BMP), writing the code required by a BMP bean to interact with a database, and developing session beans that directly participate in transactions.

Developing entity beans with BMP

In an entity bean with container-managed persistence (CMP), the container handles the interactions between the enterprise bean and the data source. In an entity bean with bean-managed persistence (BMP), the enterprise bean must contain all of the code required for the interactions between the enterprise bean and the data source. For this reason, developing an entity bean with CMP is simpler than developing an entity bean with BMP. However, you must use BMP if any of the following is true about an entity bean:

- The bean's persistent data is stored in more than one data source.
- The bean's persistent data is stored in a data source that is not supported by the EJB server that you are using.

This section examines the development of entity beans with BMP. For information on the tasks required to develop an entity bean with CMP, see [Developing entity beans with CMP](#).

Every entity bean must contain the following basic parts:

- The enterprise bean class. For more information, see [Writing the enterprise bean class \(entity with BMP\)](#).
- The enterprise bean's home interface. For more information, see [Writing the home interface \(entity with BMP\)](#).
- The enterprise bean's remote interface. For more information, see [Writing the remote interface \(entity with BMP\)](#).

In an entity bean with BMP, you can create your own primary key class or use an existing class for the primary key. For more information, see [Writing or selecting the primary key class \(entity with BMP\)](#).

Writing the enterprise bean class (entity with BMP)

In an entity bean with BMP, the bean class defines and implements the business methods of the enterprise bean, defines and implements the methods used to create instances of the enterprise bean, and implements the methods invoked by the container to move the bean through different stages in the bean's life cycle.

By convention, the enterprise bean class is named *NameBean*, where *Name* is the name you assign to the enterprise bean. The enterprise bean class for the example AccountBM enterprise bean is named AccountBMBean. Every entity bean class with BMP must meet the following requirements:

- It must be public, it must *not* be abstract, and it must implement the `javax.ejb.EntityBean` interface. For more information, see [Implementing the EntityBean interface](#).
- It must define instance variables that correspond to persistent data associated with the enterprise bean. For more information, see [Defining instance variables](#).
- It must implement the business methods used to access and manipulate the data associated with the enterprise bean. For more information, see [Implementing the business methods](#).
- It must contain code for getting connections to, interacting with, and releasing connections to the data source (or sources) used to store the persistent data. For more information, see [Using a database with a BMP entity bean](#).
- It must define and implement an `ejbCreate` method for each way in which the enterprise bean can be instantiated. It can, but is not required to, define and implement a corresponding `ejbPostCreate` method for each `ejbCreate` method. For more information, see [Implementing the ejbCreate and ejbPostCreate methods](#).
- It must implement the `ejbFindByPrimaryKey` method that takes a primary key and determines if it is valid and unique. It can also define and implement additional finder methods as required. For more information, see [Implementing the ejbFindByPrimaryKey and other ejbFind methods](#).

Note:

The enterprise bean class can implement the enterprise bean's remote interface, but this is not recommended. If the enterprise bean class implements the remote interface, it is possible to inadvertently pass the *this* variable as a method

argument.

Figure 42 shows the import statements and class declaration for the example AccountBM enterprise bean.

Figure 42. Code example: The AccountBMBean class

```
...
import java.rmi.RemoteException;
import java.util.*;
import javax.ejb.*;
import java.lang.*;
import java.sql.*;
import com.ibm.ejs.doc.account.InsufficientFundsException;
public class AccountBMBean implements EntityBean {
    ...
}
```

Defining instance variables

An entity bean class can contain both persistent and nonpersistent instance variables; however, static variables are not supported in enterprise beans unless they are also final (that is, they are constants). Persistent variables are stored in a database. Unlike the persistent variables in a CMP entity bean class, the persistent variables in a BMP entity bean class can be private.

Nonpersistent variables are *not* stored in a database and are temporary. Nonpersistent variables must be used with caution and must not be used to maintain the state of an EJB client between method invocations. This restriction is necessary because nonpersistent variables cannot be relied on to remain the same between method invocations outside of a transaction because other EJB clients can change these variables or they can be lost when the entity bean is passivated.

The AccountBMBean class contains three instance variables that represent persistent data associated with the AccountBM enterprise bean:

- *accountId*, which identifies the account ID associated with an account
- *type*, which identifies the account type as either savings (1) or checking (2)
- *balance*, which identifies the current balance of the account

The AccountBMBean class contains several nonpersistent instance variables including the following:

- *entityContext*, which identifies the entity context of each instance of an AccountBM enterprise bean. The entity context can be used to get a reference to the EJB object currently associated with the bean instance and to get the primary key object associated with that EJB object.
- *jdbcUrl*, which encapsulates the database universal resource locator (URL) used to connect to the data source. This variable must have the following format: *dbAPI:databaseType:databaseName*. For example, to specify a database named sample in an IBM DB2 database with the Java Database Connectivity (JDBC) API, the argument is *jdbc:db2:sample*.
- *driverName*, which encapsulates the database driver class required to connect to the database.
- *DBLogin*, which identifies the database user ID required to connect to the database.
- *DBPassword*, which identifies password for the specified user ID (*DBLogin*) required to connect to the database.
- *tableName*, which identifies the database table name in which the bean's persistent data is stored.
- *jdbcConn*, which encapsulates a Java Database Connectivity (JDBC) connection to a data source within a *java.sql.Connection* object.

Figure 43. Code example: The instance variables of the AccountBMBean class

```
...
public class AccountBMBean implements EntityBean {
    private EntityContext entityContext = null;
    ...
    private static final String DBURLProp = "DBURL";
    private static final String DriverNameProp = "DriverName";
    private static final String DBLoginProp = "DBLogin";
    private static final String DBPasswordProp = "DBPassword";
    private static final String TableNameProp = "TableName";
```

```

private String jdbcUrl, driverName, DBLogin, DBPassword, tableName;
private long accountId = 0;
private int type = 1;
private float balance = 0.0f;

private Connection jdbcConn = null;
...
}

```

To make the AccountBM bean more portable between databases and database drivers, the database-specific variables (*jdbcUrl*, *driverName*, *DBLogin*, *DBPassword*, and *tableName*) are set by retrieving corresponding environment variables contained in the enterprise bean. The values of these variables are retrieved by the `getEnvProps` method, which is implemented in the `AccountBMBean` class and invoked when the `setEntityContext` method is called. For more information, see [Managing database connections in the EJB server environment](#).

Although [Figure 43](#) shows database access compatible with version 1.0 of the JDBC specification, you can also perform database accesses that are compatible with version 2.0 of the JDBC specification. An administrator binds a `javax.sql.DataSource` reference (which encapsulates the information that was formerly stored in the `jdbcURL` and `driverName` variables) into the JNDI namespace. The entity bean with BMP does the following to get a `java.sql.Connection`:

```

DataSource ds = (DataSource)initialContext.lookup("java:comp/env/jdbc/MyDataSource");
Connection con = ds.getConnection();

```

where *MyDataSource* is the name the administrator assigned to the `datasource`.

Implementing the business methods

The business methods of an entity bean class define the ways in which the data encapsulated in the class can be manipulated. The business methods implemented in the enterprise bean class cannot be directly invoked by an EJB client. Instead, the EJB client invokes the corresponding methods defined in the enterprise bean's remote interface by using an EJB object associated with an instance of the enterprise bean, and the container invokes the corresponding methods in the instance of the enterprise bean.

Therefore, for every business method implemented in the enterprise bean class, a corresponding method must be defined in the enterprise bean's remote interface. The enterprise bean's remote interface is implemented by the container in the EJB object class when the enterprise bean is deployed.

There is no difference between the business methods defined in the `AccountBMBean` bean class and those defined in the CMP bean class `AccountBean` shown in [Figure 10](#).

Implementing the `ejbCreate` and `ejbPostCreate` methods

You must define and implement an `ejbCreate` method for each way in which you want a new instance of an enterprise bean to be created. For each `ejbCreate` method, you can also define a corresponding `ejbPostCreate` method. Each `ejbCreate` method must correspond to a `create` method in the EJB home interface.

Like the business methods of the bean class, the `ejbCreate` and `ejbPostCreate` methods cannot be invoked directly by the client. Instead, the client invokes the `create` method of the enterprise bean's home interface by using the EJB home object, and the container invokes the `ejbCreate` method followed by the `ejbPostCreate` method.

Unlike the method in an entity bean with CMP, the `ejbCreate` method in an entity bean with BMP must contain all of the code required to insert the bean's persistent data into the data source. This requirement means that the `ejbCreate` method must get a connection to the data source (if one is not already available to the bean instance) and insert the values of the bean's variables into the appropriate fields in the data source.

Each `ejbCreate` method in an entity bean with BMP must meet the following requirements:

- It must be public and return the bean's primary key class.
- Its arguments and return type must be valid for Java remote method invocation (RMI).
- It must contain the code required to insert the values of the persistent variables into the data source. For more information, see [Using a database with a BMP entity bean](#).

Each `ejbPostCreate` method must be public, return void, and have the same arguments as the matching `ejbCreate` method. If

necessary, both the `ejbCreate` method and the `ejbPostCreate` method can throw the `java.rmi.RemoteException` exception, the `javax.ejb.CreateException` exception, the `javax.ejb.DuplicateKeyException` exception, and any user-defined exceptions.

Figure 44 shows the two `ejbCreate` methods required by the example `AccountBMBean` bean class. No `ejbPostCreate` methods are required.

As in the `AccountBean` class, the first `ejbCreate` method calls the second `ejbCreate` method; the latter handles all of the interaction with the data source. The second method initializes the bean's instance variables and then ensures that it has a valid connection to the data source by invoking the `checkConnection` method. The method then creates, prepares, and executes an SQL INSERT call on the data source. If the INSERT call is executed correctly, and only one row is inserted into the data source, the method returns an object of the bean's primary key class.

Figure 44. Code example: The `ejbCreate` methods of the `AccountBMBean` class

```
public AccountBMKey ejbCreate(AccountBMKey key) throws CreateException,
    RemoteException {
    return ejbCreate(key, 1, 0.0f);
}
...
public AccountBMKey ejbCreate(AccountBMKey key, int type, float balance)
    throws CreateException, RemoteException
{
    accountId = key.accountId;
    this.type = type;
    this.balance = balance;
    checkConnection();
    // INSERT into database
    try {
        String sqlString = "INSERT INTO " + tableName +
            " (balance, type, accountId) VALUES (?, ?, ?)";
        PreparedStatement sqlStatement = jdbcConn.prepareStatement(sqlString);
        sqlStatement.setFloat(1, balance);
        sqlStatement.setInt(2, type);
        sqlStatement.setLong(3, accountId);
        // Execute query
        int updateResults = sqlStatement.executeUpdate();
        ...
    }
    catch (Exception e) { // Error occurred during insert
        ...
    }
    return key;
}
```

Implementing the `ejbFindByPrimaryKey` and other `ejbFind` methods

At a minimum, each entity bean with BMP must define and implement the `ejbFindByPrimaryKey` method that takes a primary key and determines if it is valid and unique for an instance of an enterprise bean; if the primary key is valid and unique, it returns the primary key. An entity bean can also define and implement other finder methods to find enterprise bean instances. All finder methods can throw the `javax.ejb.FinderException` exception to indicate an application-level error. Finder methods designed to find a single bean can also throw the `javax.ejb.ObjectNotFoundException` exception, a subclass of the `FinderException` class. Finder methods designed to return multiple beans should not use the `ObjectNotFoundException` to indicate that no suitable beans were found; instead, such methods should return empty return values. Throwing the `java.rmi.RemoteException` exception is deprecated; see [Standard application exceptions for entity beans](#) for more information.

Like the business methods of the bean class, the `ejbFind` methods cannot be invoked directly by the client. Instead, the client invokes a finder method on the enterprise bean's home interface by using the EJB home object, and the container invokes the corresponding `ejbFind` method. The container invokes an `ejbFind` method by using a generic instance of that entity bean in the pooled state.

Because the container uses an instance of an entity bean in the pooled state to invoke an `ejbFind` method, the method must do the following:

1. Get a connection to the data source (or sources).
2. Query the data source for records that match specifications of the finder method.
3. Drop the connection to the data source (or sources).

For more information on these data source tasks, see [Using a database with a BMP entity bean](#). Figure 45 shows the `ejbFindByPrimaryKey` method of the example `AccountBMBean` class. The `ejbFindByPrimaryKey` method gets a connection to its data source by calling the `makeConnection` method shown in Figure 45. It then creates and invokes an SQL SELECT statement on the data source by using the specified primary key.

If one and only one record is found, the method returns the primary key passed to it in the argument. If no records are found or multiple records are found, the method throws the `FinderException`. Before determining whether to return the primary key or throw the `FinderException`, the method drops its connection to the data source by calling the `dropConnection` method described in [Using a database with a BMP entity bean](#).

Figure 45. Code example: The `ejbFindByPrimaryKey` method of the `AccountBMBean` class

```
public AccountBMKey ejbFindByPrimaryKey (AccountBMKey key)
    throws FinderException {
    boolean wasFound = false;
    boolean foundMultiples = false;
    makeConnection();
    try {
        // SELECT from database
        String sqlString = "SELECT balance, type, accountid FROM " + tableName
            + " WHERE accountid = ?";
        PreparedStatement sqlStatement = jdbcConn.prepareStatement(sqlString);
        long keyValue = key.accountId;
        sqlStatement.setLong(1, keyValue);

        // Execute query
        ResultSet sqlResults = sqlStatement.executeQuery();

        // Advance cursor (there should be only one item)
        // wasFound will be true if there is one
        wasFound = sqlResults.next();

        // foundMultiples will be true if more than one is found.
        foundMultiples = sqlResults.next();
    }
    catch (Exception e) { // DB error
        ...
    }
    dropConnection();
    if (wasFound && !foundMultiples)
    {
        return key;
    }
    else
    {
        // Report finding no key or multiple keys
        ...
        throw(new FinderException(foundStatus));
    }
}
```

Figure 46 shows the `ejbFindLargeAccounts` method of the example `AccountBMBean` class. The `ejbFindLargeAccounts` method also gets a connection to its data source by calling the `makeConnection` method and drops the connection by using the `dropConnection` method. The SQL SELECT statement is also very similar to that used by the `ejbFindByPrimaryKey` method. (For more information on these data source tasks and methods, see [Using a database with a BMP entity bean](#).)

While the `ejbFindByPrimaryKey` method needs to return only one primary key, the `ejbFindLargeAccounts` method can be expected to return zero or more primary keys in an Enumeration object. To return an enumeration of primary keys, the

ejbFindLargeAccounts method does the following:

1. It uses a while loop to examine the result set (*sqlResults*) returned by the `executeQuery` method.
2. It inserts each primary key in the result set into a hash table named *resultTable* by wrapping the returned account ID in a Long object and then in an AccountBMKey object. (The Long object, *memberId*, is used as the hash table's index.)
3. It invokes the `elements` method on the hash table to obtain the enumeration of primary keys, which it then returns.

Figure 46. Code example: The `ejbFindLargeAccounts` method of the `AccountBMBean` class

```
public Enumeration ejbFindLargeAccounts(float amount) throws FinderException {
    makeConnection();
    Enumeration result;
    try {
        // SELECT from database
        String sqlString = "SELECT accountid FROM " + tableName
            + " WHERE balance >= ?";
        PreparedStatement sqlStatement = jdbcConn.prepareStatement(sqlString);
        sqlStatement.setFloat(1, amount);
        // Execute query
        ResultSet sqlResults = sqlStatement.executeQuery();
        // Set up Hashtable to contain list of primary keys
        Hashtable resultTable = new Hashtable();
        // Loop through result set until there are no more entries
        // Insert each primary key into the resultTable
        while(sqlResults.next() == true) {
            long acctId = sqlResults.getLong(1);
            Long memberId = new Long(acctId);
            AccountBMKey key = new AccountBMKey(acctId);
            resultTable.put(memberId, key);
        }
        // Return the resultTable as an Enumeration
        result = resultTable.elements();
        return result;
    } catch (Exception e) {
        ...
    } finally {
        dropConnection();
    }
}
```

Implementing the EntityBean interface

Each entity bean class must implement the methods inherited from the `javax.ejb.EntityBean` interface. The container invokes these methods to move the bean through different stages in the bean's life cycle. Unlike an entity bean with CMP, in an entity bean with BMP, these methods must contain all of the code for the required interaction with the data source (or sources) used by the bean to store its persistent data.

- **ejbActivate**--This method is invoked by the container when the container selects an entity bean instance from the instance pool and assigns that instance to a specific existing EJB object. This method must contain the code required to activate the enterprise bean instance by getting a connection to the data source and using the bean's `javax.ejb.EntityContext` class to obtain the primary key in the corresponding EJB object.

In the example `AccountBMBean` class, the `ejbActivate` method obtains the bean instance's account ID, sets the value of the *accountId* variable, and invokes the `checkConnection` method to ensure that it has a valid connection to the data source.

- **ejbLoad**--This method is invoked by the container to synchronize an entity bean's persistent variables with the corresponding data in the data source. (That is, the values of the fields in the data source are loaded into the persistent variables in the corresponding enterprise bean instance.) This method must contain the code required to load the values from the data source and assign those values to the bean's instance variables.

In the example `AccountBMBean` class, the `ejbLoad` method obtains the bean instance's account ID, sets the value of the *accountId* variable, invokes the `checkConnection` method to ensure that it has a valid connection to the data source, constructs and executes an SQL SELECT statement, and sets the values of the *type* and *balance* variables to match the

values retrieved from the data source.

- **ejbPassivate**--This method is invoked by the container to disassociate an entity bean instance from its EJB object and place the enterprise bean instance in the instance pool. This method must contain the code required to "passivate" or deactivate an enterprise bean instance. Usually, this passivation simply means dropping the connection to the data source.

In the example `AccountBMBean` class, the `ejbPassivate` method invokes the `dropConnection` method to drop the connection to the data source.

- **ejbRemove**--This method is invoked by the container when a client invokes the `remove` method inherited by the enterprise bean's home interface (from the `javax.ejb.EJBHome` interface) or remote interface (from the `javax.ejb.EJBObject` interface). This method must contain the code required to remove an enterprise bean's persistent data from the data source. This method can throw the `javax.ejb.RemoveException` exception if removal of an enterprise bean instance is not permitted. Usually, removal involves deleting the bean instance's data from the data source and then dropping the bean instance's connection to the data source.

In the example `AccountBMBean` class, the `ejbRemove` method invokes the `checkConnection` method to ensure that it has a valid connection to the data source, constructs and executes an SQL `DELETE` statement, and invokes the `dropConnection` method to drop the connection to the data source.

- **setEntityContext**--This method is invoked by the container to pass a reference to the `javax.ejb.EntityContext` interface to an enterprise bean instance. This method must contain any code required to store a reference to a context.

In the example `AccountBMBean` class, the `setEntityContext` method sets the value of the `entityContext` variable to the value passed to it by the container.

- **ejbStore**--This method is invoked by the container when the container needs to synchronize the data in the data source with the values of the persistent variables in an enterprise bean instance. (That is, the values of the variables in the enterprise bean instance are copied to the data source, overwriting the previous values.) This method must contain the code required to overwrite the data in the data source with the corresponding values in the enterprise bean instance.

In the example `AccountBMBean` class, the `ejbStore` method invokes the `checkConnection` method to ensure that it has a valid connection to the data source and constructs and executes an SQL `UPDATE` statement.

- **unsetEntityContext**--This method is invoked by the container, before an enterprise bean instance is removed, to free up any resources associated with the enterprise bean instance. This is the last method called prior to removing an enterprise bean instance.

In the example `AccountBMBean` class, the `unsetEntityContext` method sets the value of the `entityContext` variable to null.

Writing the home interface (entity with BMP)

An entity bean's home interface defines the methods used by EJB clients to create new instances of the bean, find and remove existing instances, and obtain metadata about an instance. The home interface is defined by the enterprise bean developer and implemented in the EJB home class created by the container during enterprise bean deployment. The container makes the home interface accessible to clients through the Java Naming and Directory Interface (JNDI).

By convention, the home interface is named *NameHome*, where *Name* is the name you assign to the enterprise bean. For example, the `AccountBM` enterprise bean's home interface is named `AccountBMHome`. Every home interface for an entity bean with BMP must meet the following requirements:

- It must extend the `javax.ejb.EJBHome` interface. The home interface inherits several methods from the `javax.ejb.EJBHome` interface. See [The javax.ejb.EJBHome interface](#) for information on these methods.
- Each method in the interface must be either a create method, which corresponds to an `ejbCreate` method (and possibly an `ejbPostCreate` method) in the enterprise bean class, or a finder method, which corresponds to an `ejbFind` method in the enterprise bean class. For more information, see [Defining create methods](#) and [Defining finder methods](#).
- The parameters and return value of each method defined in the home interface must be valid for Java RMI. For more information, see [The java.io.Serializable and java.rmi.Remote interfaces](#). In addition, each method's throws clause must include the `java.rmi.RemoteException` exception class.

Figure 47 shows the relevant parts of the definition of the home interface (`AccountBMHome`) for the example `AccountBM` bean. This interface defines two abstract create methods: the first creates an `AccountBM` object by using an associated `AccountBMKey` object, the second creates an `AccountBM` object by using an associated `AccountBMKey` object and

specifying an account type and an initial balance. The interface defines the required `findByPrimaryKey` method and the `findLargeAccounts` method.

Figure 47. Code example: The AccountBMHome home interface

```
...
import java.rmi.*;
import javax.ejb.*;
import java.util.*;
public interface AccountBMHome extends EJBHome {
    ...
    AccountBM create(AccountBMKey key) throws CreateException,
        RemoteException;
    ...
    AccountBM create(AccountBMKey key, int type, float amount)
        throws CreateException, RemoteException;
    ...
    AccountBM findByPrimaryKey(AccountBMKey key)
        throws FinderException, RemoteException;
    ...
    Enumeration findLargeAccounts(float amount)
        throws FinderException, RemoteException;
}
```

Defining create methods

A create method is used by a client to create an enterprise bean instance and insert the data associated with that instance into the data source. Each create method must be named `create` and it must have the same number and types of arguments as a corresponding `ejbCreate` method in the enterprise bean class. (The `ejbCreate` method can itself have a corresponding `ejbPostCreate` method.) The return types of the create method and its corresponding `ejbCreate` method are always different.

Each create method must meet the following requirements:

- It must be named `create`.
- It must return the type of the enterprise bean's remote interface. For example, the return type for the create methods in the `AccountBMHome` interface is `AccountBM` (as shown in [Figure 13](#)).
- It must have a throws clause that includes the `java.rmi.RemoteException` exception, the `javax.ejb.CreateException` exception, and all of the exceptions defined in the throws clause of the corresponding `ejbCreate` and `ejbPostCreate` methods.

Defining finder methods

A finder method is used to find one or more existing entity EJB objects. Each finder method must be named `findName`, where *Name* further describes the finder method's purpose.

At a minimum, each home interface must define the `findByPrimaryKey` method that enables a client to locate an EJB object by using the primary key only. The `findByPrimaryKey` method has one argument, an object of the bean's primary key class, and returns the type of the bean's remote interface.

Every other finder method must meet the following requirements:

- It must return the type of the enterprise bean's remote interface, the `java.util Enumeration` interface, or the `java.util Collection` interface (when a finder method can return more than one EJB object or an EJB collection).
- It must have a throws clause that includes the `java.rmi.RemoteException` and `javax.ejb.FinderException` exception classes.

Although every entity bean must contain only the default finder method, you can write additional ones if needed. For example, the `AccountBM` bean's home interface defines the `findLargeAccounts` method to find objects that encapsulate accounts with balances of more than a specified dollar amount, as shown in [Figure 47](#). Because this finder method can be expected to return a reference to more than one EJB object, its return type is `java.util Enumeration`.

Unlike the implementation in an entity bean with CMP, in an entity bean with BMP, the bean developer must fully implement the `ejbFindByPrimaryKey` method that corresponds to the `findByPrimaryKey` method. In addition, the bean developer must

write each additional `ejbFind` method corresponding to the finder methods defined in the home interface. The implementation of the `ejbFind` methods in the `AccountBMBean` class is discussed in [Implementing the `ejbFindByPrimaryKey` and other `ejbFind` methods](#).

Writing the remote interface (entity with BMP)

An entity bean's remote interface provides access to the business methods available in the bean class. It also provides methods to remove an EJB object associated with a bean instance and to obtain the bean instance's home interface, object handle, and primary key. The remote interface is defined by the EJB developer and implemented in the EJB object class created by the container during enterprise bean deployment.

By convention, the remote interface is named *Name*, where *Name* is the name you assign to the enterprise bean. For example, the `AccountBM` enterprise bean's remote interface is named `AccountBM`. Every remote interface must meet the following requirements:

- It must extend the `javax.ejb.EJBObject` interface. The remote interface inherits several methods from the `javax.ejb.EJBObject` interface. See [Methods inherited from `javax.ejb.EJBObject`](#) for information on these methods.
- It must define a corresponding business method for every business method implemented in the enterprise bean class.
- The parameters and return value of each method defined in the interface must be valid for Java RMI. For more information, see [The `java.io.Serializable` and `java.rmi.Remote` interfaces](#).
- Each method's throws clause must include the `java.rmi.RemoteException` exception class.

[Figure 48](#) shows the relevant parts of the definition of the remote interface (`AccountBM`) for the example `AccountBM` enterprise bean. This interface defines four methods for displaying and manipulating the account balance that exactly match the business methods implemented in the `AccountBMBean` class. All of the business methods throw the `java.rmi.RemoteException` exception class. In addition, the subtract method must throw the user-defined exception `com.ibm.ejs.doc.account.InsufficientFundsException` because the corresponding method in the bean class throws this exception. Furthermore, any client that calls this method must either handle the exception or pass it on by throwing it.

Figure 48. Code example: The `AccountBM` remote interface

```
...
import java.rmi.*;
import javax.ejb.*;
import com.ibm.ejs.doc.account.InsufficientFundsException;
public interface AccountBM extends EJBObject {
    ...
    float add(float amount) throws RemoteException;
    ...
    float getBalance() throws RemoteException;
    ...
    void setBalance(float amount) throws RemoteException;
    ...
    float subtract(float amount) throws InsufficientFundsException,
        RemoteException;
}
```

Writing or selecting the primary key class (entity with BMP)

Every entity EJB object has a unique identity within a container that is defined by a combination of the object's home interface name and its primary key, the latter of which is assigned to the object at creation. If two EJB objects have the same identity, they are considered identical.

The primary key class is used to encapsulate an EJB object's primary key. In an entity bean (with BMP or CMP), you can write a distinct primary key class or you can use an existing class as the primary key class, as long as that class is serializable. For more information, see [The `java.io.Serializable` and `java.rmi.Remote` interfaces](#).

The example `AccountBM` bean uses a primary key class that is identical to the `AccountKey` class contained in the `Account` bean shown in [Figure 16](#), with the exception that the key class is named `AccountBMKey`.

Note:

The primary key class of an entity bean with BMP must implement the hashCode and equals method. In addition, the variables that make up the primary key must be public.

The java.lang.Long class is also a good candidate for a primary key class for the AccountBM bean.

Using a database with a BMP entity bean

In an entity bean with BMP, each ejbFind method and all of the life cycle methods (ejbActivate, ejbCreate, ejbLoad, ejbPassivate, and ejbStore) must interact with the data source (or sources) used by the bean to maintain its persistent data. To interact with a supported database, the BMP entity bean must contain the code to manage database connections and to manipulate the data in the database. The EJB server uses a set of specialized beans to encapsulate information about databases and an IBM-specific interface to JDBC to allow entity bean interaction with a connection manager. For more information, see [Managing database connections in the EJB server environment](#)

In general, there are three approaches to getting and releasing connections to databases:

- The bean can get a database connection in the setEntityContext method and release it in the unsetEntityContext method. This approach is the easiest for the enterprise bean developer to implement. However, without a connection manager, this approach is not viable because under it bean instances hold onto database connections even when they are not in use (that is, when the bean instance is passivated). Even with a connection manager, this approach does not scale well.
- The bean can get a database connection in the ejbActivate and ejbCreate methods, get and release a database connection in each ejbFind method, and release the database connection in the ejbPassivate and ejbRemove methods. This approach is somewhat more difficult to implement, but it ensures that only those bean instances that are activated have connections to the database.
- The bean can get and release a database connection in each method that requires a connection: ejbActivate, ejbCreate, ejbFind, ejbLoad, and ejbStore. This approach is more difficult to implement than the first approach, but is no more difficult than the second approach. This approach is the most efficient in terms of connection use and also the most scalable.

The example AccountBM bean, uses the second approach described in the preceding text. The AccountBMBean class contains two methods for making a connection to the DB2 database, checkConnection and makeConnection, and one method to drop connections: dropConnection. The code required to make the AccountBM bean work with the connection manager is shown in [Managing database connections in the EJB server environment](#)

The code required to manipulate data in a database is described in [Manipulating data in a database](#).

Managing database connections in the EJB server environment

In the EJB server environment, the administrator creates a specialized set of entity beans that encapsulate information about the database and the database driver. These specialized entity beans are created by using the WebSphere Administrative Console.

An entity bean that requires access to a database must use JNDI to create a reference to an EJB object associated with the right database bean instance. The entity bean can then use the IBM-specific interface (named com.ibm.db2.jdbc.app.stdebt.java.sql.DataSource) to get and release connections to the database.

The DataSource interface enables the entity bean to transparently interact with the connection manager of the EJB server. The connection manager creates a pool of database connections, which are allocated and deallocated to individual entity beans as needed.

Getting an EJB object reference to a data source bean instance

Before a BMP entity bean can get a connection to a database, the entity bean must perform a JNDI lookup on the data source entity bean associated with the database used to store the BMP entity bean's persistent data. [Figure 49](#) shows the code required to create an InitialContext object and then get an EJB object reference to a database bean instance. The JNDI name of the database bean is defined by the administrator; it is recommended that the JNDI naming convention be followed when defining this name. The value of the required database-specific variables are obtained by the getEnvProps method, which accesses the corresponding environment variables from the deployed enterprise bean.

Because the connection manager creates and drops the actual database connections and simply allocates and deallocates these

connections as required, there is no need for the BMP entity bean to load and register the database driver. Therefore, there is no need to define the *driverName* and *jdbcUrl* variables discussed in [Defining instance variables](#).

Figure 49. Code example: Getting an EJB object reference to a data source bean instance in the `setEntityContext` method (rewritten to use `DataSource`)

```
...
# import com.ibm.db2.jdbc.app.stdext.javax.sql.DataSource;
# import javax.naming.*;
...
InitialContext initContext = null;
DataSource ds = null;
...
public void setEntityContext(EntityContext ctx)
    throws EJBException {
    entityContext = ctx;
    try {
        getEnvProps();
        ds = initContext.lookup("jdbc/sample");
    } catch (NamingException e) {
        ...
    }
}
...
```

Allocating and deallocating a connection to a database

After creating an EJB object reference for the appropriate database bean instance, that object reference is used to get and release connections to the corresponding database. Unlike when using the `DriverManager` interface, when using the `DataSource` interface, the BMP entity bean does not really create and close data connections; instead, the connection manager allocates and deallocates connections as required by the entity bean. Nevertheless, a BMP entity bean must still contain code to send allocation and deallocation requests to the connection manager.

In the `AccountBMBean` class, the `checkConnection` method is called within other bean class methods that require a database connection, but for which it can be assumed that a connection already exists. This method checks to make sure that the connection is still available by checking if the *jdbcConn* variable is set to null. If the variable is null, the `makeConnection` method is invoked to get the connection (that is a connection allocation request is sent to the connection manager).

The `makeConnection` method is invoked when a database connection is required. It invokes the `getConnection` method on the data source object. The `getConnection` method is overloaded: it can take either a user ID and password or no arguments, in which case the user ID and password are implicitly set to null (this version is used in [Figure 50](#)).

Figure 50. Code example: The `checkConnection` and `makeConnection` methods of the `AccountBMBean` class (rewritten to use `DataSource`)

```
private void checkConnection() throws EJBException {
    if (jdbcConn == null) {
        makeConnection();
    }
    return;
}
...
private void makeConnection() throws EJBException {
    ...
    try {
        // Open database connection
        jdbcConn = ds.getConnection();
    } catch (Exception e) { // Could not get database connection
        ...
    }
}
```

Entity beans with BMP must also release database connections when a particular bean instance no longer requires it (that is, they must send a deallocation request to the connection manager). The `AccountBMBean` class contains a `dropConnection`

method to handle this task. To release the database connection, the `dropConnection` method does the following (as shown in [Figure 51](#)):

1. Invokes the `close` method on the connection object to tell the connection manager that the connection is no longer needed.
2. Sets the connection object reference to null.

Putting the `close` method inside a `try/catch/finally` block ensures that the connection object reference is always set to null even if the `close` method fails for some reason. Nothing is done in the `catch` block because the connection manager must clean up idle connections; this is not the job of the enterprise bean code.

Figure 51. Code example: The `dropConnection` method of the `AccountBMBean` class (rewritten to use `DataSource`)

```
private void dropConnection() {
    try {
        // Close the connection
        jdbcConn.close();
    } catch (SQLException ex) {
        // Do nothing
    } finally {
        jdbcConn = null;
    }
}
```

Manipulating data in a database

After an instance of a BMP entity bean obtains a connection to its database, it can read and write data. The `AccountBMBean` class communicates with the DB2 database by constructing and executing Java Structured Query Language (JSQL) calls by using the `java.sql.PreparedStatement` interface.

As shown in [Figure 52](#), the SQL call is created as a `String` (`sqlString`). The `String` variable is passed to the `java.sql.Connection.prepareStatement` method; and the values of each variable in the SQL call are set by using the various setter methods of the `PreparedStatement` class. (The variables are substituted for the question marks in the `sqlString` variable.) Invoking the `PreparedStatement.executeUpdate` method executes the SQL call.

Figure 52. Code example: Constructing and executing an SQL update call in an `ejbCreate` method

```
private void ejbCreate(AccountBMKey key, int type, float initialBalance)
    throws CreateException, EJBException {
    // Initialize persistent variables and check for good DB connection
    ...
    // INSERT into database
    try {
        String sqlString = "INSERT INTO " + tableName +
            " (balance, type, accountid) VALUES (?, ?, ?)";
        PreparedStatement sqlStatement = jdbcConn.prepareStatement(sqlString);
        sqlStatement.setFloat(1, balance);
        sqlStatement.setInt(2, type);
        sqlStatement.setLong(3, accountId);
        // Execute query
        int updateResults = sqlStatement.executeUpdate();
        ...
    }
    catch (Exception e) { // Error occurred during insert
        ...
    }
    ...
}
```

The `executeUpdate` method is called to insert or update data in a database; the `executeQuery` method is called to get data from a database. When data is retrieved from a database, the `executeQuery` method returns a `java.sql.ResultSet` object, which must be examined and manipulated using the methods of that class.

Note:

To improve scalability and performance, you do not need to call `PreparedStatement` for each database update. Instead, you can cache the results of the first `PreparedStatement` call.

[Figure 53](#) provides an example of how the data in a `ResultSet` is manipulated in the `ejbLoad` method of the `AccountBMBean` class.

Figure 53. Code example: Manipulating a `ResultSet` object in the `ejbLoad` method

```
public void ejbLoad () throws EJBException {
    // Get data from database
    try {
        // SELECT from database
        ...
        // Execute query
        ResultSet sqlResults = sqlStatement.executeQuery();
        // Advance cursor (there should be only one item)
        sqlResults.next();
        // Pull out results
        balance = sqlResults.getFloat(1);
        type = sqlResults.getInt(2);
    } catch (Exception e) {
        // Something happened while loading data.
        ...
    }
}
```

Using bean-managed transactions

In most situations, an enterprise bean can depend on the container to manage transactions within the bean. In these situations, all you need to do is set the appropriate transactional properties in the deployment descriptor as described in [Enabling transactions and security in enterprise beans](#).

Under certain circumstances, however, it can be necessary to have an enterprise bean participate directly in transactions. By setting the *transaction* attribute in an enterprise bean's deployment descriptor to `BeanManaged`, you indicate to the container that the bean is an active participant in transactions.

Note:

The value `BeanManaged` is not a valid value for the *transaction* deployment descriptor attribute in entity beans. In other words, entity beans cannot manage transactions.

When writing the code required by an enterprise bean to manage its own transactions, remember the following basic rules:

- An instance of a stateless session bean *cannot* reuse the same transaction context across multiple methods called by an EJB client. Therefore, it is recommended that the transaction context be a local variable to each method that requires a transaction context.
- An instance of a stateful session bean can reuse the same transaction context across multiple methods called by an EJB client. Therefore, make the transaction context an instance variable or a local method variable at your discretion. (When a transaction spans multiple methods, you can use the `javax.ejb.SessionSynchronization` interface to synchronize the conversational state with the transaction.)

[Figure 54](#) shows the standard code required to obtain an object encapsulating the transaction context. There are three basics steps involved:

1. The enterprise bean class must set the value of the `javax.ejb.SessionContext` object reference in the `setSessionContext` method.
2. A `javax.transaction.UserTransaction` object is created by calling the `getUserTransaction` method on the `SessionContext` object reference.
3. The `UserTransaction` object is used to participate in the transaction by calling transaction methods such as `begin` and `commit` as needed. If a enterprise bean begins a transaction, it must also complete that transaction either by invoking the `commit` method or the `rollback` method.

Note:

In both EJB servers, the `getUserTransaction` method of the `javax.ejb.EJBContext` interface (which is inherited

by the SessionContext interface) returns an object of type javax.transaction.UserTransaction rather than type javax.jts.UserTransaction. While this is a deviation from the 1.0 version of the EJB Specification, the 1.1 version of the EJB Specification requires that the getUserTransaction method return an object of type javax.transaction.UserTransaction and drops the requirement to return objects of type javax.jts.UserTransaction.

Figure 54. Code example: Getting an object that encapsulates a transaction context

```
...
import javax.transaction.*;
...
public class MyStatelessSessionBean implements SessionBean {
    private SessionContext mySessionCtx = null;
    ...
    public void setSessionContext(SessionContext ctx) throws EJBException {
        mySessionCtx = ctx;
    }
    ...
    public float doSomething(long arg1) throws FinderException, EJBException {
        UserTransaction userTran = mySessionCtx.getUserTransaction();
        ...
        // User userTran object to call transaction methods
        userTran.begin();
        // Do transactional work
        ...
        userTran.commit();
        ...
    }
    ...
}
```

The following methods are available with the UserTransaction interface:

- begin--Begins a transaction. This method takes no arguments and returns void.
- commit--Attempts to commit a transaction; assuming that nothing causes the transaction to be rolled back, successful completion of this method commits the transaction. This method takes no arguments and returns void.
- getStatus--Returns the status of the referenced transaction. This method takes no arguments and returns int; if no transaction is associated with the reference, STATUS_NO_TRANSACTION is returned. The following are the valid return values for this method:
 - STATUS_ACTIVE--Indicates that transaction processing is still in progress.
 - STATUS_COMMITTED--Indicates that a transaction has been committed and the effects of the transaction have been made permanent.
 - STATUS_COMMITTING--Indicates that a transaction is in the process of committing (that is, the transaction has started committing but has not completed the process).
 - STATUS_MARKED_ROLLBACK--Indicates that a transaction is marked to be rolled back.
 - STATUS_NO_TRANSACTION--Indicates that a transaction does not exist in the current transaction context.
 - STATUS_PREPARED--Indicates that a transaction has been prepared but not completed.
 - STATUS_PREPARING--Indicates that a transaction is in the process of preparing (that is, the transaction has started preparing but has not completed the process).
 - STATUS_ROLLEDBACK--Indicates that a transaction has been rolled back.
 - STATUS_ROLLING_BACK--Indicates that a transaction is in the process of rolling back (that is, the transaction has started rolling back but has not completed the process).
 - STATUS_UNKNOWN--Indicates that the status of a transaction is unknown.
- rollback--Rolls back the referenced transaction. This method takes no arguments and returns void.
- setRollbackOnly--Specifies that the only possible outcome of the transaction is rollback. This method takes no arguments and returns void.
- setTransactionTimeout--Sets the timeout (in seconds) associated with the transaction. If some transaction participant has not specifically set this value, a default timeout is used. This method takes a number of seconds (as type int) and returns void.

Enabling transactions and security in enterprise beans

This chapter examines how to enable transactions and security in enterprise beans by setting the appropriate deployment descriptor attributes:

- For transactions, a session bean can either use container-managed transactions or implement bean-managed transactions; entity beans must use container-managed transactions. To enable container-managed transactions, you must set the transaction attribute to any value *except* BeanManaged and set the transaction isolation level attribute. To enable bean-managed transactions, you must set the transaction attribute to BeanManaged and set the transaction isolation level attribute. For more information, see [Setting transactional attributes in the deployment descriptor](#).

If you want a session bean to manage its own transactions, you must write the code that explicitly demarcates the boundaries of a transaction as described in [Using bean-managed transactions](#).

If you want an EJB client to manage its own transactions, you must explicitly code that client to do so as described in [Managing transactions in an EJB client](#).

- For security, the *run-as mode* attribute is used by the EJB server environments. For information on the valid values of this attribute, see [Setting the security attribute in the deployment descriptor](#).

These attributes, like the other deployment descriptor attributes, are set by using one of the tools available. For more information, see [Tools for developing and deploying enterprise beans](#).

Setting transactional attributes in the deployment descriptor

The EJB Specification describes the creation of applications that enforce transactional consistency on the data manipulated by the enterprise beans. However, unlike other specifications that support distributed transactions, the EJB specification does not require enterprise bean and EJB client developers to write any special code to use transactions. Instead, the container manages transactions based on two deployment descriptor attributes associated with the EJB module, and the enterprise bean and EJB application developers are freed to deal with the business logic of their applications.

Enterprise bean developers can specifically design enterprise beans and EJB applications that explicitly manage transactions. For more information, see [Using bean-managed transactions](#).

Under most conditions, transaction management can be handled within the enterprise beans, freeing the EJB client developer of this task. However, EJB clients can participate in transactions if required or desired. For more information, see [Managing transactions in an EJB client](#).

Two attributes determine the way in which an enterprise bean is managed from a transactional perspective:

- The *transaction* attribute defines the transactional manner in which the container invokes a method. This attribute is part of the standard deployment descriptor. [Setting the transaction attribute](#) defines the valid values of this attribute and explains their meanings.
- The *transaction isolation level* attribute defines the manner in which transactions are isolated from each other by the container. This attribute is an extension to the standard deployment descriptor. [Setting the transaction isolation level attribute](#) defines the valid values of this attribute and explains their meanings.

Setting the transaction attribute

The transaction attribute defines the transactional manner in which the container invokes enterprise bean methods. This attribute is set for individual methods in a bean. The following are valid values for this attribute in decreasing order of transactional strictness:

BeanManaged

Notifies the container that the bean class directly handles transaction demarcation. This attribute value can be specified only for session beans and it cannot be specified for individual bean methods. For more information on designing session beans to implement this attribute value, see [Using bean-managed transactions](#).

Mandatory

Directs the container to always invoke the bean method within the transaction context associated with the client. If the client attempts to invoke the bean method without a transaction context, the container throws the `javax.jts.TransactionRequiredException` exception to the client. The transaction context is passed to any EJB object or resource accessed by an enterprise bean method.

EJB clients that access these entity beans must do so within an existing transaction. For other enterprise beans, the enterprise bean or bean method must implement the `BeanManaged` value or use the `Required` or `RequiresNew` value. For non-enterprise bean EJB clients, the client must invoke a transaction by using the `javax.transaction.UserTransaction` interface, as described in [Managing transactions in an EJB client](#).

Required

Directs the container to invoke the bean method within a transaction context. If a client invokes a bean method from within a transaction context, the container invokes the bean method within the client transaction context. If a client invokes a bean method outside of a transaction context, the container creates a new transaction context and invokes the bean method from within that context. The transaction context is passed to any enterprise bean objects or resources that are used by this bean method.

RequiresNew

Directs the container to always invoke the bean method within a new transaction context, regardless of whether the client invokes the method within or outside of a transaction context. The transaction context is passed to any enterprise bean objects or resources that are used by this bean method.

Supports

Directs the container to invoke the bean method within a transaction context if the client invokes the bean method within a transaction. If the client invokes the bean method without a transaction context, the container invokes the bean method without a transaction context. The transaction context is passed to any enterprise bean objects or resources that are used by this bean method.

NotSupported

Directs the container to invoke bean methods without a transaction context. If a client invokes a bean method from within a transaction context, the container suspends the association between the transaction and the current thread before invoking the method on the enterprise bean instance. The container then resumes the suspended association when the method invocation returns. The suspended transaction context is *not* passed to any enterprise bean objects or resources that are used by this bean method.

Never

Directs the container to invoke bean methods without a transaction context.

- If the client invokes a bean method from within a transaction context, the container throws the `java.rmi.RemoteException` exception.
- If the client invokes a bean method from outside a transaction context, the container behaves in the same way as if the `NotSupported` transaction attribute was set. The client must call the method without a transaction context.

Table 1. Effect of the enterprise bean's transaction attribute on the transaction context

Transaction attribute	Client transaction context	Bean transaction context
Mandatory	No transaction	Not allowed
	Client transaction	Client transaction
RequiresNew	No transaction	New transaction
	Client transaction	New transaction
Required	No transaction	New transaction
	Client transaction	Client transaction
Supports	No transaction	No transaction
	Client transaction	Client transaction
NotSupported	No transaction	No transaction
	Client transaction	No transaction
Never	No transaction	No transaction
	No transaction	No transaction

When setting the deployment descriptor for an entity bean, you can mark getter methods as "Read-Only" methods to improve performance. If a transaction unit of work includes no methods other than "Read-Only" designated methods, then the entity bean state synchronization does not invoke store.

Setting the transaction isolation level attribute

The transaction isolation level determines how strongly one transaction is isolated from another. This attribute is set for individual methods in a bean. However, within a transactional context, the isolation level associated with the first method invocation becomes the required isolation level for all other methods invoked within that transaction. If a method is invoked with a different isolation level from that of the first method, the `java.rmi.RemoteException` exception is thrown.

The following are valid values for this attribute, in decreasing order of isolation:

Serializable

This level prohibits all of the following types of reads:

- *Dirty reads*, where a transaction reads a database row containing uncommitted changes from a second transaction.
- *Nonrepeatable reads*, where one transaction reads a row, a second transaction changes the same row, and the first transaction rereads the row and gets a different value.
- *Phantom reads*, where one transaction reads all rows that satisfy an SQL WHERE condition, a second transaction inserts a row that also satisfies the WHERE condition, and the first transaction applies the same WHERE condition and gets the row inserted by the second transaction.

RepeatableRead

This level prohibits dirty reads and nonrepeatable reads, but it allows phantom reads.

ReadCommitted

This level prohibits dirty reads, but allows nonrepeatable reads and phantom reads.

ReadUncommitted

This level allows dirty reads, nonrepeatable reads, and phantom reads.

These isolation levels correspond to the isolation levels defined in the Java Database Connectivity (JDBC)

java.sql.Connection interface.

The container uses the transaction isolation level attribute as follows:

- Session beans and entity beans with bean-managed persistence (BMP)--For each database connection used by the bean, the container sets the transaction isolation level at the start of each transaction.
- Entity beans with container-managed persistence (CMP)--The container generates database access code that implements the specified isolation level.

None of these values permits two transactions to update the same data concurrently; one transaction must end before another can update the same data. These values determine only how locks are managed for reading data. However, risks to consistency can arise from read operations when a transaction does further work based on the values read. For example, if one transaction is updating a piece of data and a second transaction is permitted to read that data after it has been changed but before the updating transaction ends, the reading transaction can make a decision based on a change that is eventually rolled back. The second transaction risks making a decision on transient data.

Deciding which isolation level to use depends on several factors:

- The acceptable level of risk to data consistency
- The acceptable levels of concurrency and performance
- The isolation levels supported by the underlying database

The first two factors, risk to consistency and level of concurrency, are related. Decreasing the risk to consistency requires you to decrease concurrency because reducing the risk to consistency requires holding locks longer. The longer a lock is held on a piece of data, the longer concurrently running transactions must wait to access that data. The Serializable value protects data by eliminating concurrent access to it. Conversely, the ReadUncommitted value allows the highest degree of concurrency but entails the greatest risk to consistency. You need to balance these two factors appropriately for your application.

By default, most developers deploy enterprise beans with the transaction isolation level set to Serializable. This is the default value in IBM VisualAge for Java Enterprise Edition and other deployment tools. It is also the most restrictive and protected transaction isolation level incurring the most overhead. Some workloads do not require the isolation level and protection afforded by Serializable. A given application might never update the underlying data or be run with other applications that also make concurrent updates. In that case, the application would not have to be concerned with dirty, non-repeatable, or phantom reads. The ReadUncommitted isolation level would probably be sufficient.

Because the transaction isolation level is set in the EJB module's deployment descriptor, the same enterprise bean could be reused in different applications with different transaction isolation levels. The isolation level requirements should be reviewed and adjusted appropriately to increase performance.

The third factor, isolation levels supported in the database, means that although the EJB specification allows you to request one of the four levels of transaction isolation, it is possible that the database being used in the application does not support all of the levels. Also, vendors of database products implement isolation levels differently, so the precise behavior of an application can vary from database to database. You need to consider the database and the isolation levels it supports when deciding on the value for the transaction isolation attribute in deployment descriptors. Consult your database documentation for more information on supported isolation levels.

Setting the security attribute in the deployment descriptor

When an EJB client invokes a method on an enterprise bean, the user context of the client principal is encapsulated in a CORBA Current object, which contains credential properties for the principal. The Current object is passed among the participants in the method invocation as required to complete the method.

The security service uses the credential information to determine the permissions that a principal has on various resources. At appropriate points, the security service determines if the principal is authorized to use a particular resource based on the principal's permissions.

If the method invocation is authorized, the security service does the following with the principal's credential properties based on the value of the *run-as mode* attribute of the enterprise bean. If a specific identity is required, the *RunAsIdentity* attribute is used to specify that identity.

Identity of Caller

The security service makes no changes to the principal's credential properties.

Identity of EJB Server

The security service alters the principal's credential properties to match the credential properties associated with the EJB server.

Identity Assigned to Specified Role

A security principal that has been assigned to the specified role is used for the execution of the bean's methods. This association is part of the application binding where the role is associated with a user ID and password of a user who is granted that role.

Developing enterprise beans

This chapter explains the basic tasks required to develop and package the most common types of enterprise beans. Specifically, this chapter focuses on creating stateless session beans and entity beans that use container-managed persistence (CMP); in the discussion of stateless session beans, important information about stateful beans is also provided. For information on developing entity beans that use bean-managed persistence (BMP), see [Developing entity beans with BMP](#).

The information in this chapter is not exhaustive; however, it includes the information you need to develop basic enterprise beans. For information on developing more complicated enterprise beans, consult a commercially available book on enterprise bean development. The example enterprise beans discussed in this chapter and the example Java applications and servlets that use them are described in [Information about the examples described in the documentation](#).

This chapter describes the requirements for building each of the major components of an enterprise bean. If you do *not* intend to use one of the commercially available integrated development environments (IDE), such as IBM's VisualAge for Java, you must build each of these components manually (by using tools in the Java Development Kit and WebSphere). Manually developing enterprise beans is much more difficult and error-prone than developing them in an IDE. Therefore, it is strongly recommended that you choose an IDE with which you are comfortable.

Developing entity beans with CMP

In an entity bean with CMP, the container handles the interactions between the entity bean and the data source. In an entity bean with BMP, the entity bean must contain all of the code required for the interactions between the entity bean and the data source. For this reason, developing an entity bean with CMP is simpler than developing an entity bean with BMP.

This section examines the development of entity beans with CMP. While much of the information in this section also applies to entity beans with BMP, there are some major differences between the two types. For information on the tasks required to develop an entity bean with BMP, see [Developing entity beans with BMP](#).

Every entity bean must contain the following basic parts:

- The enterprise bean class. For more information, see [Writing the enterprise bean class \(entity with CMP\)](#).
- The enterprise bean's home interface. For more information, see [Writing the home interface \(entity with CMP\)](#).
- The enterprise bean's remote interface. For more information, see [Writing the remote interface \(entity with CMP\)](#).
- The enterprise bean's primary key class. For more information, see [Writing the primary key class \(entity with CMP\)](#).

Writing the enterprise bean class (entity with CMP)

In a CMP entity bean, the bean class defines and implements the business methods of the enterprise bean, defines and implements the methods used to create instances of the enterprise bean, and implements the methods used by the container to inform the instances of the enterprise bean of significant events in the instance's life cycle. Enterprise bean clients never access the bean class directly; instead, the classes that implement the home and remote interfaces are used to indirectly invoke the methods defined in the bean class.

By convention, the enterprise bean class is named *NameBean*, where *Name* is the name you assign to the enterprise bean. The enterprise bean class for the example Account enterprise bean is named AccountBean. Every entity bean class with CMP must meet the following requirements:

- It must be public, it must *not* be abstract, and it must implement the javax.ejb.EntityBean interface. For more information, see [Implementing the EntityBean interface](#).
- It must define instance variables that correspond to persistent data associated with the enterprise bean. For more information, see [Defining variables](#).
- It must implement the business methods used to access and manipulate the data associated with the enterprise bean. For more information, see [Implementing the business methods](#).
- It must define and implement an ejbCreate method for each way in which the enterprise bean can be instantiated. A corresponding ejbPostCreate method must be defined for each ejbCreate method. For more information, see [Implementing the ejbCreate and ejbPostCreate methods](#).

Note:

The enterprise bean class can implement the enterprise bean's remote interface, but this is not recommended. If the enterprise bean class implements the remote interface, it is possible to inadvertently pass the *this* variable as a method argument.

An enterprise bean class cannot implement two different interfaces if the methods in the interfaces have the same name, even if the methods have different signatures, due to the Java-IDL mapping specification. Errors can occur when the enterprise bean is

deployed.

[Figure 8](#) shows the main parts of the enterprise bean class for the example Account enterprise bean. (Emphasized code is in bold type.) The sections that follow discuss these parts in greater detail.

Figure 8. Code example: The AccountBean class

```
...
import java.util.Properties;
import javax.ejb.*;
import java.lang.*;
public class AccountBean implements EntityBean {
    // Set instance variables here
    ...
    // Implement methods here
    ...
}
```

Defining variables

An entity bean class can contain both persistent and nonpersistent instance variables; however, static variables are not supported in enterprise beans unless they are also final (that is, they are constants). Static variables are not supported because there is no way to guarantee that they remain consistent across enterprise bean instances.

Container-managed fields (which are persistent variables) are stored in a database. Container-managed fields must be public.

Nonpersistent variables are *not* stored in a database and are temporary. Nonpersistent variables must be used with caution and must not be used to maintain the state of an EJB client between method invocations. This restriction is necessary because nonpersistent variables cannot be relied on to remain the same between method invocations outside of a transaction because other EJB clients can change these variables, or they can be lost when the entity bean is passivated.

The AccountBean class contains three container-managed fields (shown in [Figure 9](#)):

- *accountId*, which identifies the account ID associated with an account
- *type*, which identifies the account type as either savings (1) or checking (2)
- *balance*, which identifies the current balance of the account

Figure 9. Code example: The variables of the AccountBean class

```
...
public class AccountBean implements EntityBean {
    private EntityContext entityContext = null;
    private ListResourceBundle bundle =
        ResourceBundle.getBundle(
            "com.ibm.ejs.doc.account.AccountResourceBundle");
    public long accountId = 0;
    public int type = 1;
    public float balance = 0.0f;
    ...
}
```

The deployment descriptor is used to identify container-managed fields in entity beans with CMP. In an entity bean with CMP, each container-managed field must be initialized by each `ejbCreate` method (see [Implementing the ejbCreate and ejbPostCreate methods](#)).

A subset of the container-managed fields is used to define the primary key class associated with each instance of an enterprise bean. As is shown in [Writing the primary key class \(entity with CMP\)](#), the *accountId* variable defines the primary key for the Account enterprise bean. The AccountBean class contains two nonpersistent variables:

- *entityContext*, which identifies the entity context of each instance of an Account enterprise bean. The entity context can be used to get a reference to the EJB object currently associated with the bean instance and to get the primary key object associated with that EJB object.
- *bundle*, which encapsulates a resource bundle class (`com.ibm.ejs.doc.account.AccountResourceBundle`) that contains locale-specific objects used by the Account bean.

Implementing the business methods

The business methods of an entity bean class define the ways in which the data encapsulated in the class can be manipulated. The business methods implemented in the enterprise bean class cannot be directly invoked by an EJB client. Instead, the EJB client invokes

the corresponding methods defined in the enterprise bean's remote interface, by using an EJB object associated with an instance of the enterprise bean, and the container invokes the corresponding methods in the instance of the enterprise bean.

Therefore, for every business method implemented in the enterprise bean class, a corresponding method must be defined in the enterprise bean's remote interface. The enterprise bean's remote interface is implemented by the container in the EJB object class when the enterprise bean is deployed.

Figure 10 shows the business methods for the AccountBean class. These methods are used to add a specified amount to an account balance and return the new balance (add), to return the current balance of an account (getBalance), to set the balance of an account (setBalance), and to subtract a specified amount from an account balance and return the new balance (subtract). The subtract method throws the user-defined exception `com.ibm.ejs.doc.account.InsufficientFundsException` if a client attempts to subtract more money from an account than is contained in the account balance. The subtract method in the Account bean's remote interface must also throw this exception as shown in Figure 15. User-defined exception classes for enterprise beans are created as are any other user-defined exception class. The message content for the `InsufficientFundsException` exception is obtained from the `AccountResourceBundle` class file by invoking the `getMessage` method on the *bundle* object.

Note:

If an enterprise bean container catches a system exception from the business method of an enterprise bean, and the method is running within a container-managed transaction, the container rolls back the transaction before passing the exception on to the client. However, if the business method is throwing an application exception, then the transaction is not rolled back (it is committed), unless the application has called `setRollbackOnly` function. In this case, the transaction is rolled back before the exception is re-thrown.

Figure 10. Code example: The business methods of the AccountBean class

```
...
public class AccountBean implements EntityBean {
    ...
    public long accountId = 0;
    public int type = 1;
    public float balance = 0.0f;
    ...
    public float add(float amount) {
        balance += amount;
        return balance;
    }
    ...
    public float getBalance() {
        return balance;
    }
    ...
    public void setBalance(float amount) {
        balance = amount;
    }
    ...
    public float subtract(float amount) throws InsufficientFundsException {
        if(balance < amount) {
            throw new InsufficientFundsException(
                bundle.getMessage("insufficientFunds"));
        }
        balance -= amount;
        return balance;
    }
    ...
}
```

Standard application exceptions for entity beans

Version 1.1 of the EJB specification defines several standard application exceptions for use by enterprise beans. All of these exceptions are subclasses of the `javax.ejb.EJBException` class. For entity beans with both container- and bean-managed persistence, the EJB specification defines the following application exceptions:

- `javax.ejb.CreateException`
- `javax.ejb.DuplicateKeyException`
- `javax.ejb.RemoveException`
- `javax.ejb.FinderException`
- `javax.ejb.ObjectNotFoundException`

Application programmers can use the generic `EJBException` class or one of the provided subclassed exceptions, or programmers can define their own exceptions by subclassing any of this family of exceptions. All of these exceptions inherit from the `javax.ejb.RuntimeException` class and do not have to be explicitly declared in throws clauses.

Each exception is discussed in more detail within the relevant section; for more information on:

- `CreateException` and `DuplicateKeyException` (a subclass of the `CreateException` class), see [Implementing the `ejbCreate` and `ejbPostCreate` methods](#).
- `javax.ejb.RemoveException`, see [Implementing the `EntityBean` interface](#).
- `FinderException` and `ObjectNotFoundException` (a subclass of the `FinderException` class), see [Defining finder methods](#).

Note:

Version 1.0 of the EJB specification used the `java.rmi.RemoteException` class to capture application-specific exceptions; the `EJBException` class and its subclasses are new in the 1.1 version of the specification. Therefore, using the `RemoteException` class is now deprecated in favor of the more precise exception classes. Older applications that use the `RemoteException` class can still run, but enterprise beans compliant with version 1.1 of the specification must use the new exception classes.

Implementing the `ejbCreate` and `ejbPostCreate` methods

You must define and implement an `ejbCreate` method for each way in which you want a new instance of an enterprise bean to be created. For each `ejbCreate` method, you must also define a corresponding `ejbPostCreate` method. Each `ejbCreate` and `ejbPostCreate` method must correspond to a create method in the home interface.

Like the business methods of the bean class, the `ejbCreate` and `ejbPostCreate` methods cannot be invoked directly by the client. Instead, the client invokes the create method of the enterprise bean's home interface by using the EJB home object, and the container invokes the `ejbCreate` method followed by the `ejbPostCreate` method. If the `ejbCreate` and `ejbPostCreate` methods are executed successfully, an EJB object is created and the persistent data associated with that object is inserted into the data source.

For an entity bean with CMP, the container handles the required interaction between the entity bean instance and the data source between calls to the `ejbCreate` and `ejbPostCreate` methods. For an entity bean with BMP, the `ejbCreate` method must contain the code to directly handle this interaction. For more information on entity beans with BMP, see [Developing entity beans with BMP](#).

Each `ejbCreate` method in an entity bean with CMP must meet the following requirements:

- It must be public and return the same type as the primary key. The actual return value must be null.
- Its arguments must be valid for Java remote method invocation (RMI). For more information, see [The `java.io.Serializable` and `java.rmi.Remote` interfaces](#).
- It must initialize the container-managed fields of the enterprise bean instance. The container extracts the values of these variables and writes them to the data source after the `ejbCreate` method returns.

Each `ejbPostCreate` method must be public, return void, and have the same arguments as the matching `ejbCreate` method. If necessary, both the `ejbCreate` method and the `ejbPostCreate` method can throw the `javax.ejb.EJBException` exception or one of the creation-related subclasses, the `CreateException` or the `DuplicateKeyException` exceptions. The `DuplicateKeyException` class is a subclass of the `CreateException` class. Throwing the `java.rmi.RemoteException` exception is deprecated; see [Standard application exceptions for entity beans](#) for more information.

[Figure 11](#) shows two sets of `ejbCreate` and `ejbPostCreate` methods required for the example `AccountBean` class. The first set of `ejbCreate` and `ejbPostCreate` methods are wrappers that call the second set of methods and set the *type* variable to 1 (corresponding to a savings account) and the *balance* variable to 0 (zero dollars).

Figure 11. Code example: The `ejbCreate` and `ejbPostCreate` methods of the `AccountBean` class

```
...
public class AccountBean implements EntityBean {
    ...
    public long accountId = 0;
    public int type = 1;
    public float balance = 0.0f;
    ...
    public Integer ejbCreate(AccountKey key) {
        ejbCreate(key, 1, 0.0f);
    }
    ...
    public Integer ejbCreate(AccountKey key, int type, float initialBalance)
    throws EJBException {
        accountId = key.accountId;
```

```

        type = type;
        balance = initialBalance;
    }
    ...
    public void ejbPostCreate(AccountKey key)
    throws EJBException {
        ejbPostCreate(key, 1, 0);
    }
    ...
    public void ejbPostCreate(AccountKey key, int type, float initialBalance) { }
    ...
}

```

Implementing the EntityBean interface

Each entity bean class must implement the methods inherited from the `javax.ejb.EntityBean` interface. The container invokes these methods to inform the bean instance of significant events in the instance's life cycle. (For more information, see [Entity bean life cycle](#).) All of these methods must be public and return void; they can throw the `javax.ejb.EJBException` exception or, in the case of the `ejbRemove` method, the `javax.ejb.RemoveException` exception. Throwing the `java.rmi.RemoteException` exception is deprecated; see [Standard application exceptions for entity beans](#) for more information.

- **ejbActivate**--This method is invoked by the container when the container selects an entity bean instance from the instance pool and assigns that instance to a specific existing EJB object. This method must contain any code that you want to execute when the enterprise bean instance is activated.
- **ejbLoad**--This method is invoked by the container to synchronize an entity bean's container-managed fields with the corresponding data in the data source. (That is, the values of the fields in the data source are loaded into the container-managed fields in the corresponding enterprise bean instance.) This method must contain any code that you want to execute when the enterprise bean instance is synchronized with associated data in the data source.
- **ejbPassivate**--This method is invoked by the container when the container disassociates an entity bean instance from its EJB object and places the enterprise bean instance in the instance pool. This method must contain any code that you want to execute when the enterprise bean instance is "passivated" or deactivated.
- **ejbRemove**--This method is invoked by the container when a client invokes the remove method inherited by the enterprise bean's home interface from the `javax.ejb.EJBHome` interface. This method must contain any code that you want to execute before an enterprise bean instance is removed from the container (and the associated data is removed from the data source). This method can throw the `javax.ejb.RemoveException` exception if removal of an enterprise bean instance is not permitted.
- **setEntityContext**--This method is invoked by the container to pass a reference to the `javax.ejb.EntityContext` interface to an enterprise bean instance. If an enterprise bean instance needs to use this context at any time during its life cycle, the enterprise bean class must contain an instance variable to store this value. This method must contain any code required to store a reference to a context.
- **ejbStore**--This method is invoked by the container when the container needs to synchronize the data in the data source with the values of the container-managed fields in an enterprise bean instance. (That is, the values of the variables in the enterprise bean instance are copied to the data source, overwriting the previous values.) This method must contain any code that you want to execute when the data in the data source is overwritten with the corresponding values in the enterprise bean instance.
- **unsetEntityContext**--This method is invoked by the container, before an enterprise bean instance is removed, to free up any resources associated with the enterprise bean instance. This is the last method called prior to removing an enterprise bean instance.

In entity beans with CMP, the container handles the required data source interaction for these methods. In entity beans with BMP, these methods must directly handle the required data source interaction. For more information on entity beans with BMP, see [More-advanced programming concepts for enterprise beans](#).

These methods have several possible uses, including the following:

- They can contain audit or debugging code.
- They can contain code for allocating and deallocating additional resources used by the bean instance (for example, an SNA connection to a mainframe).

As shown in [Figure 12](#), except for the `setEntityContext` and `unsetEntityContext` methods, all of these methods are empty in the `AccountBean` class because no additional action is required by the bean for the particular life cycle states associated with these methods. The `setEntityContext` and `unsetEntityContext` methods are used in a conventional way to set the value of the *entityContext* variable.

Figure 12. Code example: Implementing the EntityBean interface in the AccountBean class

```

...
public class AccountBean implements EntityBean {

```

```

private EntityContext entityContext = null;
...
public void ejbActivate() throws EJBException { }
...
public void ejbLoad () throws EJBException { }
...
public void ejbPassivate() throws EJBException { }
...
public void ejbRemove() throws EJBException { }
...
public void ejbStore () throws EJBException { }
...
public void setEntityContext(EntityContext ctx) throws EJBException {
    entityContext = ctx;
}
...
public void unsetEntityContext() throws EJBException {
    entityContext = null;
}
}

```

Writing the home interface (entity with CMP)

An entity bean's home interface defines the methods used by clients to create new instances of the bean, find and remove existing instances, and obtain metadata about an instance. The home interface is defined by the enterprise bean developer and implemented in the EJB home class created by the container during enterprise bean deployment.

The container makes the home interface accessible to enterprise bean clients through the Java Naming and Directory Interface (JNDI). JNDI is independent of any specific naming and directory service and allows Java-based applications to access any naming and directory service in a standard way.

By convention, the home interface is named *NameHome*, where *Name* is the name you assign to the enterprise bean. For example, the Account enterprise bean's home interface is named AccountHome. Every home interface must meet the following requirements:

- It must extend the `javax.ejb.EJBHome` interface. The home interface inherits several methods from the `javax.ejb.EJBHome` interface. See [The javax.ejb.EJBHome interface](#) for information on these methods.
- Each method in the interface must be either a create method that corresponds to a set of `ejbCreate` and `ejbPostCreate` methods in the EJB object class, or a finder method. For more information, see [Defining create methods](#) and [Defining finder methods](#).
- The parameters and return value of each method defined in the home interface must be valid for Java RMI. For more information, see [The java.io.Serializable and java.rmi.Remote interfaces](#). In addition, each method's throws clause must include the `java.rmi.RemoteException` exception class.

[Figure 13](#) shows the relevant parts of the definition of the home interface (AccountHome) for the example Account bean. This interface defines two abstract create methods: the first creates an Account object by using an associated AccountKey object, the second creates an Account object by using an associated AccountKey object and specifying an account type and an initial balance. The interface defines the required `findByPrimaryKey` method and a `findLargeAccounts` method, which returns a collection of accounts containing balances greater than a specified amount.

Figure 13. Code example: The AccountHome home interface

```

...
import java.rmi.*;
import java.util.*;
import javax.ejb.*;
public interface AccountHome extends EJBHome {
    ...
    Account create (AccountKey id) throws CreateException, RemoteException;
    ...
    Account create(AccountKey id, int type, float initialBalance)
        throws CreateException, RemoteException;
    ...
    Account findByPrimaryKey (AccountKey id)
        RemoteException, FinderException;
    ...
    Enumeration findLargeAccounts(float amount)
        throws RemoteException, FinderException;
}

```


Defining create methods

A create method is used by a client to create an enterprise bean instance and insert the data associated with that instance into the data source. Each create method must be named `create` and it must have the same number and types of arguments as a corresponding `ejbCreate` method in the enterprise bean class. (The `ejbCreate` method must itself have a corresponding `ejbPostCreate` method.)

Each create method must meet the following requirements:

- It must be named `create`.
- It must return the type of the enterprise bean's remote interface. For example, the return type for the create methods in the `AccountHome` interface is `Account` (as shown in [Figure 13](#)).
- It must have a throws clause that includes the `java.rmi.RemoteException` exception, the `javax.ejb.CreateException` exception, and all of the application exceptions defined in the throws clause of the corresponding `ejbCreate` and `ejbPostCreate` methods.

Defining finder methods

A finder method is used to find one or more existing entity EJB objects. Each finder method must be named `findName`, where *Name* further describes the finder method's purpose.

At minimum, each home interface must define the `findByPrimaryKey` method that enables a client to locate an EJB object by using the primary key only. The `findByPrimaryKey` method has one argument, an object of the bean's primary key class, and returns the type of the bean's remote interface.

Every other finder method must meet the following requirements:

- It must return the type of the enterprise bean's remote interface, the `java.util Enumeration` interface, or the `java.util Collection` interface (when a finder method can return more than one EJB object or an EJB collection).
- It must have a throws clause that includes the `java.rmi.RemoteException` and `javax.ejb.FinderException` exception classes.

While every entity bean must contain the default finder method, you can write additional finder methods if needed. For example, the `Account` bean's home interface defines the `findLargeAccounts` method to find objects that encapsulate accounts with balances of more than a specified amount, as shown in [Figure 14](#). Because this finder method can be expected to return a reference to more than one EJB object, its return type is `Enumeration`.

Figure 14. Code example: The `findLargeAccounts` method

```
Enumeration findLargeAccounts(float amount)
    throws RemoteException, FinderException;
```

Every EJB server can implement the `findByPrimaryKey` method. During enterprise bean deployment, the container generates the code required to search the database for the appropriate enterprise bean instance.

However, for each additional finder method that you define in the home interface, the enterprise bean deployer must associate finder logic with that finder method. This logic is used by the EJB server during deployment to generate the code required to implement the finder method.

The EJB Specification does not define the format of the finder logic, so the format can vary according to the EJB server you are using. For more information on creating finder logic, see [Creating finder logic in the EJB server](#).

Writing the remote interface (entity with CMP)

An entity bean's remote interface provides access to the business methods available in the bean class. It also provides methods to remove an EJB object associated with a bean instance and to obtain the bean instance's home interface, object handle, and primary key. The remote interface is defined by the enterprise bean developer and implemented in the EJB object class created by the container during enterprise bean deployment.

By convention, the remote interface is named *Name*, where *Name* is the name you assign to the enterprise bean. For example, the `Account` enterprise bean's remote interface is named `Account`. Every remote interface must meet the following requirements:

- It must extend the `javax.ejb.EJBObject` interface. The enterprise bean's remote interface inherits several methods from the `javax.ejb.EJBObject` interface. See [Methods inherited from javax.ejb.EJBObject](#) for information on these methods.
- You must define a corresponding business method for every business method implemented in the enterprise bean class.
- The parameters and return value of each method defined in the interface must be valid for Java RMI. For more information, see [The java.io.Serializable and java.rmi.Remote interfaces](#).
- Each method's throws clause must include the `java.rmi.RemoteException` exception class.

Figure 15 shows the relevant parts of the definition of the remote interface (Account) for the example Account enterprise bean. This interface defines four methods for displaying and manipulating the account balance that exactly match the business methods implemented in the AccountBean class. All of the business methods in the remote interface throw the `java.rmi.RemoteException` exception class. In addition, the `subtract` method must throw the user-defined exception `com.ibm.ejs.doc.account.InsufficientFundsException` because the corresponding method in the bean class throws this exception. Furthermore, any client that calls this method must either handle the exception or pass it on by throwing it.

Figure 15. Code example: The Account remote interface

```
...
import java.rmi.*;
import javax.ejb.*;
public interface Account extends EJBObject
{
    ...
    float add(float amount) throws RemoteException;
    ...
    float getBalance() throws RemoteException;
    ...
    void setBalance(float amount) throws RemoteException;
    ...
    float subtract(float amount) throws InsufficientFundsException,
        RemoteException;
}
```

Writing the primary key class (entity with CMP)

Within a container, every entity EJB object has a unique identity that is defined by using a combination of the object's home interface name and its primary key, the latter of which is assigned to the object at creation. If two EJB objects have the same identity, they are considered identical.

Primary keys are specified in two ways:

- Simple primary keys, which map to a single field in the entity bean class and are comprised of primitive Java data types (such as integer or long), are specified in the deployment descriptor.
- Composite primary keys, which map to multiple fields in the entity bean class (or to data structures built from the primitive Java data types), must be encapsulated in a *primary key class*. More complicated enterprise beans are likely to have composite primary keys, with multiple instance variables representing the primary key.

The primary key class is used to manage an EJB object's primary key. By convention, the primary key class is named *NameKey*, where *Name* is the name of the enterprise bean. For example, the Account enterprise bean's primary key class is named `AccountKey`. The primary key class must meet the following requirements:

- It must be public and it must be serializable. For more information, see [The java.io.Serializable and java.rmi.Remote interfaces](#).
- Its instance variables must be public, and the variable names must match a subset of the container-managed field names defined in the enterprise bean class.
- It must have a public default constructor, at a minimum.

Note:

The primary key class of a CMP entity bean must override the `equals` method and the `hashCode` method inherited from the `java.lang.Object` class.

Figure 16 shows a composite primary key class for an example enterprise bean, `Item`. In effect, this class acts as a wrapper around the string variables `productId` and `vendorId`. The `hashCode` method for the `ItemKey` class invokes the corresponding `hashCode` method in the `java.lang.String` class after creating a temporary string object by using the value of the `productId` variable. In addition to the default constructor, the `ItemKey` class also defines a constructor that sets the value of the primary key variables to the specified strings.

Figure 16. Code example: The ItemKey primary key class

```
...
import java.io.*;
// Composite primary key class
public class ItemKey implements java.io.Serializable {

    public String productId;
    public String vendorId;
    // Constructors
    public ItemKey() { };
    public ItemKey(String productId, String vendorId) {
```

```

        this.productId = productId;
        this.vendorId = vendorId;
    }

    public String getProductId() {
        return productId;
    }
    public String getVendorId() {
        return vendorId;
    }
    ...
    // EJB server-specific method
    public boolean equals(Object other) {
        if (other instanceof ItemKey) {
            return (productId.equals(((ItemKey)
                other).productId)
                && vendorId.equals(((ItemKey)
                other).vendorId));
        }
        else
            return false;
    }
    ...
    // EJB server-specific method
    public int hashCode() {
        return (new productId.hashCode());
    }
}

```

A primary key class can also be used to encapsulate a primary key that is not known ahead of time -- for instance, if the entity bean is intended to work with several persistent data stores, each of which requires a different primary key structure. The entity bean's primary key type is derived from the primary key type used by the underlying database that stores the entity objects; it does not necessarily have to be known to the enterprise bean developer.

To specify an unknown primary key, do the following:

- Declare the argument of the `findByPrimaryKey` class as `java.lang.Object`.
- Declare the return value of the `ejbCreate` method as `java.lang.Object`
- In the deployment descriptor, specify the primary key class as being of the type `java.lang.Object`.

When the primary key selection is deferred to deployment, client applications cannot use methods that rely on knowledge of the primary key type. In addition, applications cannot always depend on methods that return the type of the primary key (such as the `EntityContext.getPrimaryKey` method) because the return type is determined at deployment.

Interacting with databases

This section contains general information and tips on enterprise beans and database access.

- Although it is not necessary, it is good practice to specify the user ID and password for a data source either in the enterprise bean to be using the data source, or in the container of the bean.
- The container supports Option A and Option C caching. When Option A caching is in use, the application server hosting the enterprise bean container must be the only updater of the data in the persistent store. As such, Option A caching is incompatible with the following:
 - Workload managed servers (such as a cluster of clones)
 - Databases with data being shared among multiple applications

The default caching option is C (multiple entity bean instances, possibly in different servers, can update bean state in the database). The default caching option can be changed from Option C to Option A by selecting "exclusive persistent store" in the administrative console when creating the entity bean.

Shared database access corresponds to Option C caching. Option A and Option C caching are also known as commit option A and commit option C, respectively.

Developing session beans

In their basic makeup, session beans are similar to entity beans. However, their purposes are very different.

From a component perspective, one of the biggest differences between the two types of enterprise beans is that session beans do not have a primary key class and the session bean's home interface does not define finder methods. Session enterprise beans do not require primary keys and finder methods because session EJB objects are created, associated with a specific client, and then removed as needed, whereas entity EJB objects represent permanent data in a data source and can be uniquely identified with a primary key. Because the data for session beans is never permanently stored, the session bean class does not have methods for storing data to and loading data from a data source.

Every session bean must contain the following basic parts:

- The enterprise bean class. For more information, see [Writing the enterprise bean class \(session\)](#).
- The enterprise bean's home interface. For more information, see [Writing the home interface \(session\)](#).
- The enterprise bean's remote interface. For more information, see [Writing the remote interface \(session\)](#).

Writing the enterprise bean class (session)

A session bean class defines and implements the business methods of the enterprise bean, implements the methods used by the container during the creation of enterprise bean instances, and implements the methods used by the container to inform the enterprise bean instance of significant events in the instance's life cycle. By convention, the enterprise bean class is named *NameBean*, where *Name* is the name you assign to the enterprise bean. The enterprise bean class for the example Transfer enterprise bean is named *TransferBean*. Every session bean class must meet the following requirements:

- It must define and implement the business methods that execute the tasks associated with the enterprise bean. For more information, see [Implementing the business methods](#).
- It must define and implement an `ejbCreate` method for each way in which you want it to be able to instantiate the enterprise bean class. For more information, see [Implementing the ejbCreate methods](#).
- It must be public, it must *not* be abstract, and it must implement the `javax.ejb.SessionBean` interface. For more information, see [Implementing the SessionBean interface](#).

Note:

Version 1.0 of the EJB specification allowed the methods in the session bean class to throw the `java.rmi.RemoteException` exception to indicate a non-application exception. This practice is deprecated in version 1.1 of the specification. A session bean compliant with version 1.1 of the specification should throw the `javax.ejb.EJBException` exception (a subclass of the `java.lang.RuntimeException` class) or another `RuntimeException` exception instead. Because the `javax.ejb.EJBException` class is a subclass of the `java.lang.RuntimeException`, `EJBException` exceptions do not need to be explicitly listed in the `throws` clause of methods.

A session bean can be either stateful or stateless. In a stateless session bean, none of the methods depend on the values of variables set by any other method, except for the `ejbCreate` method, which sets the initial (identical) state of each bean instance. In a stateful enterprise bean, one or more methods depend on the values of variables set by some other method. As in entity beans, static variables are not supported in session beans unless they are also final. Stateful session beans possibly need to synchronize their conversational state with the transactional context in which they operate. For example, a stateful session bean possibly needs to reset the value of some of its variables if a transaction is rolled back or it possibly needs to change these variables if a transaction successfully completes.

If a bean needs to synchronize its conversational state with the transactional context, the bean class must implement the `javax.ejb.SessionSynchronization` interface. This interface contains methods to notify the session bean when a transaction begins, when it is about to complete, and when it has completed. The enterprise bean developer can use these methods to synchronize the state of the session enterprise bean instance with ongoing transactions.

The enterprise bean class can implement the enterprise bean's remote interface, but this is not recommended. If the enterprise bean class implements the remote interface, it is possible to inadvertently pass the *this* variable as a method argument.

[Figure 17](#) shows the main parts of the enterprise bean class for the example Transfer bean. The sections that follow discuss these parts in greater detail.

The Transfer bean is stateless. If the Transfer bean's `transferFunds` method were dependent on the value of the *balance* variable returned by the `getBalance` method, the `TransferBean` would be stateful.

Figure 17. Code example: The `TransferBean` class

```
...
import java.rmi.RemoteException;
import java.util.Properties;
import java.util.ResourceBundle;
import java.util.ListResourceBundle;
import javax.ejb.*;
import java.lang.*;
import javax.naming.*;
```

```

import com.ibm.ejs.doc.account.*;
...
public class TransferBean implements SessionBean {
    ...
    private SessionContext mySessionCtx = null;
    private InitialContext initialContext = null;
    private AccountHome accountHome = null;
    private Account fromAccount = null;
    private Account toAccount = null;
    ...
    public void ejbActivate() throws EJBException { }
    ...
    public void ejbCreate() throws EJBException {
        ...
    }
    ...
    public void ejbPassivate() throws EJBException { }
    ...
    public void ejbRemove() throws EJBException { }
    ...
    public float getBalance(long acctId) throws FinderException,
        EJBException {
        ...
    }
    ...
    public void setSessionContext(javax.ejb.SessionContext ctx)
        throws EJBException {
        ...
    }
    ...
    public void transferFunds(long fromAcctId, long toAcctId, float amount)
        throws EJBException {
        ...
    }
}

```

Implementing the business methods

The business methods of a session bean class define the ways in which an EJB client can manipulate the enterprise bean. The business methods implemented in the enterprise bean class cannot be directly invoked by an EJB client. Instead, the EJB client invokes the corresponding methods defined in the enterprise bean's remote interface, by using an EJB object associated with an instance of the enterprise bean, and the container invokes the corresponding methods in the enterprise bean instance.

Therefore, for every business method defined in the enterprise bean's remote interface, a corresponding method must be implemented in the enterprise bean class. The enterprise bean's remote interface is implemented by the container in the EJBObject class when the enterprise bean is deployed.

[Figure 18](#) shows the business methods for the TransferBean class. The getBalance method is used to get the balance for an account. It first locates the appropriate Account EJB object and then calls that object's getBalance method.

The transferFunds method is used to transfer a specified amount between two accounts (encapsulated in two Account entity EJB objects). After locating the appropriate Account EJB objects by using the findByPrimaryKey method, the transferFunds method calls the add method on one account and the subtract method on the other. Like all finder methods, findByPrimaryKey can throw both the FinderException and RemoteException exceptions. The try/catch blocks are set up around invocations of the findByPrimaryKey method to handle the entry of invalid account IDs by users. If the session bean user enters an invalid account ID, the findByPrimaryKey method cannot locate an EJB object, and the finder method throws the FinderException exception. This exception is caught and converted into a new FinderException exception containing information on the invalid account ID.

To call the findByPrimaryKey method, both business methods need to be able to access the EJB home object that implements the AccountHome interface discussed in [Writing the home interface \(entity with CMP\)](#). Obtaining the EJB home object is discussed in [Implementing the ejbCreate methods](#).

Figure 18. Code example: The business methods of the TransferBean class

```

public class TransferBean implements SessionBean {
    ...
    private Account fromAccount = null;
    private Account toAccount = null;

```

```

...
public float getBalance(long acctId) throws FinderException, EJBException {
    AccountKey key = new AccountKey(acctId);
    try {
        fromAccount = accountHome.findByPrimaryKey(key);
    } catch(FinderException ex) {
        throw new FinderException("Account " + acctId
            + " does not exist.");
    } catch(RemoteException ex) {
        throw new FinderException("Account " + acctId
            + " could not be found.");
    }
    return fromAccount.getBalance();
}
...
public void transferFunds(long fromAcctId, long toAcctId, float amount)
    throws EJBException, InsufficientFundsException, FinderException {
    AccountKey fromKey = new AccountKey(fromAcctId);
    AccountKey toKey = new AccountKey(toAcctId);
    try {
        fromAccount = accountHome.findByPrimaryKey(fromKey);
    } catch(FinderException ex) {
        throw new FinderException("Account " + fromAcctId
            + " does not exist.");
    } catch(RemoteException ex) {
        throw new FinderException("Account " + acctId
            + " could not be found.");
    }
    try {
        toAccount = accountHome.findByPrimaryKey(toKey);
    } catch(FinderException ex) {
        throw new FinderException("Account " + toAcctId
            + " does not exist.");
    } catch(RemoteException ex) {
        throw new FinderException("Account " + acctId
            + " could not be found.");
    }
    try {
        toAccount.add(amount);
        fromAccount.subtract(amount);
    } catch(InsufficientFundsException ex) {
        mySessionCtx.setRollbackOnly();
        throw new InsufficientFundsException("Insufficient funds in "
            + fromAcctId);
    }
}
}
}

```

Implementing the ejbCreate methods

You must define and implement an ejbCreate method for each way in which you want an enterprise bean to be instantiated.

Each ejbCreate method must correspond to a create method in the enterprise bean's home interface. (Note that there is no ejbPostCreate method in a session bean as there is in an entity bean.) Unlike the business methods of the enterprise bean class, the ejbCreate methods cannot be invoked directly by the client. Instead, the client invokes the create method in the bean instance's home interface, and the container invokes the ejbCreate method. If an ejbCreate method is executed successfully, an EJB object is created.

An ejbCreate method for a session bean must meet the following requirements:

- The method must be declared as public and cannot be declared as final or static.
- It must return void.
- A stateless session bean must have only one ejbCreate method, which must return void and contain no arguments. A stateful session bean can have multiple ejbCreate methods.

The throws clause can define arbitrary application exceptions. The javax.ejb.EJBException or another runtime exception can be used to indicate non-application exceptions.

An `ejbCreate` method for an entity bean must meet the following requirements:

- The method must be declared as `public` and cannot be declared as `final` or `static`.
- It must return the entity bean's primary key type.
- It must contain code to set the values of any variables needed by the EJB object.

The `throws` clause can define arbitrary application exceptions. The `javax.ejb.EJBException` or another runtime exception can be used to indicate non-application exceptions. [Figure 19](#) shows the `ejbCreate` method required by the example `TransferBean` class. The `Transfer` bean's `ejbCreate` method obtains a reference to the `Account` bean's home object. This reference is required by the `Transfer` bean's business methods. Getting a reference to an enterprise bean's home interface is a two-step process:

1. Construct an `InitialContext` object by setting the required property values. For the example `Transfer` bean, these property values are defined in the environment variables of the `Transfer` bean's deployment descriptor.
2. Use the `InitialContext` object to create and get a reference to the home object. For the example `Transfer` bean, the JNDI name of the `Account` bean is stored in an environment variable in the `Transfer` bean's deployment descriptor.

Creating the `InitialContext` object

When a container invokes the `Transfer` bean's `ejbCreate` method, the enterprise bean's *initialContext* object is constructed by creating a `Properties` variable (*env*) that requires the following values:

- The location of the name service (`javax.naming.Context.PROVIDER_URL`).
- The name of the initial context factory (`javax.naming.Context.INITIAL_CONTEXT_FACTORY`).

The values of these properties are discussed in more detail in [Creating and getting a reference to a bean's EJB object](#).

Figure 19. Code example: Creating the `InitialContext` object in the `ejbCreate` method of the `TransferBean` class

```
...
public class TransferBean implements SessionBean {
    private static final String INITIAL_NAMING_FACTORY_SYSPROP =
        javax.naming.Context.INITIAL_CONTEXT_FACTORY;
    private static final String PROVIDER_URL_SYSPROP =
        javax.naming.Context.PROVIDER_URL;

    ...
    private String nameService = null;
    ...
    private String providerURL = null;
    ...
    private InitialContext initialContext = null;
    ...
    public void ejbCreate() throws EJBException {
        // Get the initial context
        try {
            Properties env = System.getProperties();
            ...
            env.put( PROVIDER_URL_SYSPROP, getProviderUrl() );
            env.put( INITIAL_CONTEXT_FACTORY_SYSPROP, getNamingFactory() );
            initialContext = new InitialContext( env );
        } catch (Exception ex) {
            ...
        }
        ...
        // Look up the home interface using the JNDI name
        ...
    }
}
```

Although the example `Transfer` bean stores some locale specific variables in a resource bundle class, like the example `Account` bean, it also relies on the values of environment variables stored in its deployment descriptor. Each of these `InitialContext` `Properties` values is obtained from an environment variable contained in the `Transfer` bean's deployment descriptor. A private `get` method that corresponds to the property variable is used to get each of the values (`getNamingFactory` and `getProviderURL`); these methods must be written by the enterprise bean developer. The following environment variables must be set to the appropriate values in the deployment descriptor of the `Transfer` bean.

- `javax.naming.Context.INITIAL_CONTEXT_FACTORY`
- `javax.naming.Context.PROVIDER_URL`

[Figure 20](#) illustrates the relevant parts of the `getProviderURL` method that is used to get the `PROVIDER_URL` property value. The `javax.ejb.SessionContext` variable (*mySessionCtx*) is used to get the `Transfer` bean's environment in the deployment descriptor by

invoking the `getEnvironment` method. The object returned by the `getEnvironment` method can then be used to get the value of a specific environment variable by invoking the `getProperty` method.

Figure 20. Code example: The `getProviderURL` method

```
...
public class TransferBean implements SessionBean {
    private SessionContext mySessionCtx = null;
    ...
    private String getProviderURL() throws RemoteException {
        //get the provider URL property either from
        //the EJB properties or, if it isn't there
        //use "iiop:///", which causes a default to the local host
        ...
        String pr = mySessionCtx.getEnvironment().getProperty(
            PROVIDER_URL_SYSPROP);
        if (pr == null)
            pr = "iiop:///";
        return pr;
    }
    ...
}
```

Getting the reference to the home object

An enterprise bean is accessed by looking up the class implementing its home interface by name through JNDI. Methods on the home interface provide access to an instance of the class implementing the remote interface.

After constructing the `InitialContext` object, the `ejbCreate` method performs a JNDI lookup using the JNDI name of the `Account` enterprise bean. Like the `PROVIDER_URL` and `INITIAL_CONTEXT_FACTORY` properties, this name is also retrieved from an environment variable contained in the `Transfer` bean's deployment descriptor (by invoking a private method named `getHomeName`). The lookup method returns an object of type `java.lang.Object`.

The returned object is narrowed by using the static method `javax.rmi.PortableRemoteObject.narrow` to obtain a reference to the EJB home object for the specified enterprise bean. The parameters of the `narrow` method are the object to be narrowed and the class of the object to be created as a result of the narrowing. For a more thorough discussion of the code required to locate an enterprise bean in JNDI and then narrow it to get an EJB home object, see [Creating and getting a reference to a bean's EJB object](#).

Figure 21. Code example: Creating the `AccountHome` object in the `ejbCreate` method of the `TransferBean` class

```
...
public class TransferBean implements SessionBean {
    ...
    private String accountName = null;
    ...
    private InitialContext initialContext = null;
    ...
    public void ejbCreate() throws EJBException {
        // Get the initial context
        ...
        // Look up the home interface using the JNDI name
        try {
            java.lang.Object ejbHome = initialContext.lookup(accountName);
            accountHome = (AccountHome)javax.rmi.PortableRemoteObject.narrow(
                ejbHome, AccountHome.class);
        } catch (NamingException e) { // Error getting the home interface
            ...
        }
        ...
    }
    ...
}
```

Looking up an enterprise bean's environment naming context

The enterprise bean's environment is implemented by the container. It enables the bean's business logic to be customized without the need to access or change the bean's source code. The container provides an implementation of the JNDI naming context that stores the enterprise bean environment. Business methods access the environment by using the JNDI interfaces. The deployment descriptor

provides the environment entries that the enterprise bean expects at runtime.

Each enterprise bean defines its own environment entries, which are shared between all of its instances (that is, all instances with the same home). Environment entries are not shared between enterprise beans.

An enterprise bean's environment entries are stored directly in the environment naming context (or one of its subcontexts). To retrieve its environment naming context, an enterprise bean instance creates an `InitialContext` object by using the constructor with no arguments. It then looks up the environment naming via the `InitialContext` object under the name `java:comp/env`.

The enterprise bean in [Figure 22](#) changes an account number by looking up an environment entry to find the new account number.

Figure 22. Code example: Looking up an enterprise bean's environment naming context

```
public class AccountService implements SessionBean {
    ...
    public void changeAccountNumber(int accountNumber, ... )
        throws InvalidAccountNumberException{
        ....
        // Obtain the bean's environment naming context
        Context initialContext = new InitialContext();
        Context myEnvironment = (Context)initialContext.lookup("java:comp/env");
        ...
        // Obtain new account number from environment
        Integer newNumber = (Integer)myEnvironment.lookup("newAccountNumber");
        ...
    }
}
```

Implementing the SessionBean interface

Every session bean class must implement the methods inherited from the `javax.ejb.SessionBean` interface. The container invokes these methods to inform the enterprise bean instance of significant events in the instance's life cycle. All of these methods must be public, must return void, and can throw the `javax.ejb.EJBException`. (Throwing the `java.rmi.RemoteException` exception is deprecated; see *** for more information.)

- **ejbActivate**--This method is invoked by the container when the container selects an enterprise bean instance from the instance pool and assigns it a specific existing EJB object. This method must contain any code that you want to execute when the enterprise bean instance is activated.
- **ejbPassivate**--This method is invoked by the container when the container disassociates an enterprise bean instance from its EJB object and places the enterprise bean instance in the instance pool. This method must contain any code that you want to execute when the enterprise bean instance is passivated (deactivated).
- **ejbRemove**--This method is invoked by the container when a client invokes the remove method inherited by the enterprise bean's home interface (from the `javax.ejb.EJBHome` interface). This method must contain any code that you want to execute when an enterprise bean instance is removed from the container.
- **setSessionContext**--This method is invoked by the container to pass a reference to the `javax.ejb.SessionContext` interface to a session bean instance. If an enterprise bean instance needs to use this context at any time during its life cycle, the enterprise bean class must contain an instance variable to store this value. This method must contain any code required to store a reference to the context.

A session context can be used to get a handle to a particular instance of a stateful session bean. It can also be used to get a reference to a transaction context object, as described in [Using bean-managed transactions](#).

As shown in [Figure 23](#), except for the `setSessionContext` method, all of these methods in the `TransferBean` class are empty because no additional action is required by the bean for the particular life cycle states associated with these methods. The `setSessionContext` method is used in a conventional way to set the value of the `mySessionCtx` variable.

Figure 23. Code example: Implementing the SessionBean interface in the TransferBean class

```
...
public class TransferBean implements SessionBean {
    private SessionContext mySessionCtx = null;
    ...
    public void ejbActivate() throws EJBException { }
    ...
    public void ejbPassivate() throws EJBException { }
    ...
    public void ejbRemove() throws EJBException { }
    ...
    public void setSessionContext(SessionContext ctx) throwEJBException {
```

```

        mySessionCtx = ctx;
    }
    ...
}

```

Writing the home interface (session)

A session bean's home interface defines the methods used by clients to create and remove instances of the enterprise bean and obtain metadata about an instance. The home interface is defined by the enterprise bean developer and implemented in the EJB home class created by the container during enterprise bean deployment. The container makes the home interface accessible to clients through JNDI.

By convention, the home interface is named *NameHome*, where *Name* is the name you assign to the enterprise bean. For example, the Transfer enterprise bean's home interface is named TransferHome. Every session bean's home interface must meet the following requirements:

- It must extend the `javax.ejb.EJBHome` interface. The home interface inherits several methods from the `javax.ejb.EJBHome` interface. See [The `javax.ejb.EJBHome` interface](#) for information on these methods.
- Each method in the interface must be a create method that corresponds to a `ejbCreate` method in the enterprise bean class. For more information, see [Implementing the `ejbCreate` methods](#). Unlike entity beans, the home interface of a session bean contains no finder methods.
- The parameters and return value of each method defined in the interface must be valid for Java RMI. For more information, see [The `java.io.Serializable` and `java.rmi.Remote` interfaces](#). In addition, each method's throws clause must include the `java.rmi.RemoteException` exception class.

Figure 24 shows the relevant parts of the definition of the home interface (TransferHome) for the example Transfer bean.

Figure 24. Code example: The TransferHome home interface

```

...
import javax.ejb.*;
import java.rmi.*;
public interface TransferHome extends EJBHome {
    Transfer create() throws CreateException, RemoteException;
}

```

A create method is used by a client to create an enterprise bean instance. A stateful session bean can contain multiple create methods; however, a stateless session bean can contain only one create method with no arguments. This restriction on stateless session beans ensures that every instance of a stateless session bean is the same as every other instance of the same type. (For example, every Transfer bean instance is the same as every other Transfer bean instance.)

Each create method must be named `create` and have the same number and types of arguments as a corresponding `ejbCreate` method in the EJB object class. The return types of the create method and its corresponding `ejbCreate` method are always different. Each create method must meet the following requirements:

- It must return the type of the enterprise bean's remote interface. For example, the return type for the create method in the TransferHome interface is Transfer.
- It must have a throws clause that includes the `java.rmi.RemoteException` exception, the `javax.ejb.CreateException` exception class, and all of the exceptions defined in the throws clause of the corresponding `ejbCreate` method.

Writing the remote interface (session)

A session bean's remote interface provides access to the business methods available in the enterprise bean class. It also provides methods to remove an enterprise bean instance and to obtain the enterprise bean's home interface and handle. The remote interface is defined by the enterprise bean developer and implemented in the EJB object class created by the container during enterprise bean deployment.

By convention, the remote interface is named *Name*, where *Name* is the name you assign to the enterprise bean. For example, the Transfer enterprise bean's remote interface is named Transfer. Every remote interface must meet the following requirements:

- It must extend the `javax.ejb.EJBObject` interface. The remote interface inherits several methods from the `EJBObject` interface. See [Methods inherited from `javax.ejb.EJBObject`](#) for information on these methods.
- You must define a corresponding business method for every business method implemented in the enterprise bean class.
- The parameters and return value of each method defined in the interface must be valid for Java RMI. For more information, see [The `java.io.Serializable` and `java.rmi.Remote` interfaces](#).
- Each method's throws clause must include the `java.rmi.RemoteException` exception class.

Figure 25 shows the relevant parts of the definition of the remote interface (Transfer) for the example Transfer bean. This interface defines the methods for transferring funds between two Account bean instances and for getting the balance of an Account bean instance.

Figure 25. Code example: The Transfer remote interface

```
...
import javax.ejb.*;
import java.rmi.*;
import com.ibm.ejs.doc.account.*;
public interface Transfer extends EJBObject {
    ...
    float getBalance(long acctId) throws FinderException, RemoteException;
    ...
    void transferFunds(long fromAcctId, long toAcctId, float amount)
        throws InsufficientFundsException, RemoteException;
}
```

Implementing interfaces common to multiple types of enterprise beans

Enterprise beans must implement the interfaces described here in the appropriate enterprise bean component.

Methods inherited from javax.ejb.EJBObject

The remote interface inherits the following methods from the javax.ejb.EJBObject interface, which are implemented by the container during deployment:

- **getEJBHome**--Returns the enterprise bean's home interface.
- **getHandle**--Returns the handle for the EJB object.
- **getPrimaryKey**--Returns the EJB object's primary key. (For session beans, this cannot be used because session beans do not have a primary key.)
- **isIdentical**--Compares this EJB object with the EJB object argument to determine if they are the same.
- **remove**--Removes this EJB object.

These methods have the following syntax:

```
public abstract EJBHome getEJBHome();
public abstract Handle getHandle();
public abstract Object getPrimaryKey();
public abstract boolean isIdentical(EJBObject obj);
public abstract void remove();
```

These methods are implemented by the container in the EJB object class.

The javax.ejb.EJBHome interface

The home interface inherits two remove methods and the getEJBMetaData method from the javax.ejb.EJBHome interface. Just like the methods defined directly in the home interface, these inherited methods are also implemented in the EJB home class created by the container during deployment.

The remove methods are used to remove an existing EJB object (and its associated data in the database) either by specifying the EJB object's handle or its primary key. (The remove method that takes a *primaryKey* variable can be used only in entity beans.) The getEJBMetaData method is used to obtain metadata about the enterprise bean and is mainly intended for use by development tools.

These methods have the following syntax:

```
public abstract EJBMetaData getEJBMetaData();
public abstract void remove(Handle handle);
public abstract void remove(Object primaryKey);
```

The javax.ejb.EJBHome interface also contains a method to get a handle to the home interface. It has the following syntax:

```
public abstract HomeHandle getHomeHandle();
```

The java.io.Serializable and java.rmi.Remote interfaces

To be valid for use in a remote method invocation (RMI), a method's arguments and return value must be one of the following types:

- A primitive type; for example, an int or a long.
- An object of a class that directly or indirectly implements java.io.Serializable; for example, java.lang.Long.
- An object of a class that directly or indirectly implements java.rmi.Remote.
- An array of valid types or objects.

If you attempt to use a parameter that is not valid, the java.rmi.RemoteException exception is thrown. Note that the following atypical types are *not* valid:

- An object of a class that directly or indirectly implements both Serializable and Remote.
 - An object of a class that directly or indirectly implements Remote, but contains a method that does not throw the RemoteException or an exception that inherits from RemoteException.
-

Using threads and reentrancy in enterprise beans

An enterprise bean must not contain code to start new threads (nor can methods be defined with the keyword synchronized). Session beans can *never* be reentrant; that is, they cannot call another bean that invokes a method on the calling bean. Entity beans can be reentrant, but building reentrant entity beans is not recommended and is not documented here.

The EJB server enforces single-threaded access to all enterprise beans. Illegal callbacks result in a java.rmi.RemoteException exception being thrown to the EJB client.

Creating an EJB module for enterprise beans

There are two tasks involved in preparing an enterprise bean for deployment:

- Making the components of the bean part of the same Java package. For more information, see [Making bean components part of a Java package](#).
- Creating an EJB module and associated deployment descriptor. For more information, see [Creating an EJB module and deployment descriptor](#).

If you develop enterprise beans in an IDE, these tasks are handled from within the tool that you use. If you do not develop enterprise beans in an IDE, you must handle each of these tasks by using tools contained in the Java Software Development Kit (SDK) and WebSphere Application Server. For more information on the tools used to create an EJB module in the EJB server programming environment, see [Tools for developing and deploying enterprise beans](#).

Making bean components part of a Java package

You determine the best way to allocate your enterprise beans to Java packages. A Java package can contain one or more enterprise beans. The example Account and Transfer beans are stored in separate packages. All of the Java source files that make up the Account bean contain the following package statement:

```
package com.ibm.ejs.doc.account;
```

All of the Java source files that make up the Transfer bean contain the following package statement:

```
package com.ibm.ejs.doc.transfer;
```

Creating an EJB module and deployment descriptor

An EJB module contains one or more deployable enterprise beans. It also contains a deployment descriptor that provides information about each enterprise bean and instructions for the container on how to handle all enterprise beans in the module. The deployment descriptor is stored in an XML file.

During creation of the EJB module, you specify the files for each enterprise bean to be included in the module. These files include:

- The class files associated with each component of the enterprise bean.
- Any additional classes and files associated with the enterprise bean; for example: user-defined exception classes, properties files, and resource bundle classes.

You also specify other information about the bean, such as references to other enterprise beans, resource connection factories, and security roles. After defining the enterprise beans to be included in the module, you specify application assembly instructions that apply to the module as a whole. Both bean and module information are used to create a deployment descriptor. See [The deployment descriptor](#) for a list of deployment descriptor settings and attributes.

Developing EJB clients

An enterprise bean can be accessed by all of the following types of EJB clients in both EJB server environments:

- Java servlets. For more information about writing Java servlets that use enterprise beans, see [Developing servlets that use enterprise beans](#).
- Java Server Pages (JSP). For more information about writing JSP, consult a commercially available book.
- Java applications that use remote method invocation (RMI). For more information on writing Java applications, consult a commercially available book.
- Other enterprise beans. For example, the Transfer session bean acts as a client to the Account bean, as described in [Developing enterprise beans](#).

It is recommended that you avoid accessing EJB entity beans from client or servlet code. Instead, wrap and access EJB entity beans from EJB session beans. This improves performance in two ways:

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

Except for the basic programming tasks described in this chapter, creating a Java servlet, JSP, or Java application that is a client to an enterprise bean is not very different from designing standard versions of these types of Java programs. This chapter assumes that you understand the basics of writing a Java servlet, a Java application, or a JSP file.

Except where noted, all of the code described in this chapter is taken from the example Java application named TransferApplication. This Java application and the other EJB clients available with the documentation example code are explained in [Information about the examples described in the documentation](#).

To access and manipulate an enterprise bean in any of the Java-based EJB client types listed previously, the EJB client must do the following:

- Import the Java packages required for naming, remote method invocation (RMI), and enterprise bean interaction.
- Get a reference to an instance of the bean's EJB object by using the Java Naming and Directory Interface (JNDI). For more information, see [Creating and getting a reference to a bean's EJB object](#).
- Handle invalid EJB objects when using session beans. For more information, see [Handling an invalid EJB object for a session bean](#).
- Remove session EJB objects when they are no longer required or remove entity EJB objects when the associated data in the data source must be removed. For more information, see [Removing a bean's EJB object](#).

In addition, an EJB client can participate in the transactions associated with enterprise beans used by the client. For more information, see [Managing transactions in an EJB client](#).

Importing required Java packages

Although the Java packages required for any particular EJB client vary, the following packages are required by all EJB clients:

- java.rmi -- This package contains most of the classes required for remote method invocation (RMI).
- javax.rmi -- This package contains the PortableRemoteObject class required to get a reference to an EJB object.
- java.util -- This package contains various Java utility classes, such as Properties, Hashtable, and Enumeration used in a variety of ways throughout all enterprise beans and EJB clients.
- javax.ejb -- This package contains the classes and interfaces defined in the EJB specification.
- javax.naming -- The package contains the classes and interfaces defined in the Java Naming and Directory Interface (JNDI) specification and is used by clients to get references to EJB objects.

- The package or packages containing the enterprise beans with which the client interacts.

The Java client object request broker (ORB), which is automatically initialized in EJB clients, does not support dynamic download of implementation bytecode from the server to the client. As a result, all classes required by the EJB client at runtime must be available from the files and directories identified in the client's CLASSPATH environment variable. For information on the JAR files required by EJB clients, see [Setting the CLASSPATH environment variable in the EJB server environment](#). You can install needed files on your client machine by doing a WebSphere Application Server installation on the machine. Select the **Developer's Client Files** option. You also need to make sure that the `ioser` and `ioserx` executable files are accessible on your client machine; these files are normally part of the Java install. If you are using a Windows System, make sure that EJB clients can locate the `ioser.dll` library file at run time. [Figure 26](#) shows the import statements for the example Java application `com.ibm.ejs.doc.client.TransferApplication`. In addition to the required Java packages mentioned previously, the example application imports the `com.ibm.ejs.doc.transfer` package because the application communicates with a `Transfer` bean. The example application also imports the `InsufficientFundsException` class contained in the same package as the `Account` bean.

Figure 26. Code example: The import statements for the Java application `TransferApplication`

```
...
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.rmi.*
...
import javax.naming.*;
import javax.ejb.*;
import javax.rmi.PortableRemoteObject;
...
import com.ibm.ejs.doc.account.InsufficientFundsException;
import com.ibm.ejs.doc.transfer.*;
...
public class TransferApplication extends Frame implements
    ActionListener, WindowListener {
    ...
}
```

Creating and getting a reference to a bean's EJB object

To invoke a bean's business methods, a client must create or find an EJB object for that bean. After the client has created or found this object, it can invoke methods on it in the standard way.

To create or find an instance of a bean's EJB object, the client must do the following:

1. Locate and create an EJB home object for that bean. For more information, see [Locating and creating an EJB home object](#).
2. Use the EJB home object to create or (for entity beans only) find an instance of the bean's EJB object. For more information, see [Creating an EJB object](#).

The `TransferApplication` client contains one reference to a `Transfer` EJB object, which the application uses to invoke all of the methods on the `Transfer` bean. When using session beans in Java applications, it is a good idea to make the reference to the EJB object a class-level variable rather than a variable that is local to a method. This allows your EJB client to repeatedly invoke methods on the same EJB object rather than having to create a new object each time the client invokes a session bean method. As discussed in [Threading issues](#), this approach is not recommended for servlets, which must be designed to handle multiple threads.

Locating and creating an EJB home object

JNDI is used to find the name of an EJB home object. The properties that an EJB client uses to initialize JNDI and find an EJB home object vary across EJB server implementations. To make an enterprise bean more portable between EJB server implementations, it is recommended that you externalize these properties in environment variables, properties files, or resource bundles rather than hard code them into your enterprise bean or EJB client code.

The example Transfer bean uses environment variables as discussed in [Implementing the ejbCreate methods](#). The TransferApplication uses a resource bundle contained in the com.ibm.ejs.doc.client.ClientResourceBundle.class file. To initialize a JNDI name service, an EJB client must set the appropriate values for the following JNDI properties:

javax.naming.Context.PROVIDER_URL

This property specifies the host name and port of the name server used by the EJB client. The property value must have the following format: `iiop://hostname:port`, where *hostname* is the IP address or hostname of the machine on which the name server runs and *port* is the port number on which the name server listens.

For example, the property value `iiop://bankserver.mybank.com:9019` directs an EJB client to look for a name server on the host named bankserver.mybank.com listening on port 9019. The property value `iiop://bankserver.mybank.com` directs an EJB client to look for a name server on the host named bankserver.mybank.com at port number 900. The property value `iiop:///` directs an EJB client to look for a name server on the local host listening on port 900. If not specified, this property defaults to the local host and port number 900, which is the same as specifying `iiop:///`. The port number used by the name service can be changed by using the administrative interface.

javax.naming.Context.INITIAL_CONTEXT_FACTORY

This property identifies the actual name service that the EJB client must use. This property must be set to `com.ibm.ejs.ns.jndi.CNInitialContextFactory`.

Locating an EJB home object is a two-step process:

1. Create a `javax.naming.InitialContext` object. For more information, see [Creating an InitialContext object](#).
2. Use the `InitialContext` object to create the EJB home object. For more information, see [Creating EJB home object](#).

Creating an InitialContext object

[Figure 27](#) shows the code required to create the `InitialContext` object. To create this object, construct a `java.util.Properties` object, add values to the `Properties` object, and then pass the object as the argument to the `InitialContext` constructor. In the TransferApplication, the value of each property is obtained from the resource bundle class named `com.ibm.ejs.doc.client.ClientResourceBundle`, which stores all of the locale-specific variables required by the TransferApplication. (This class also stores the variables used by the other EJB clients contained in the documentation example, described in [Information about the examples described in the documentation](#)). The resource bundle class is instantiated by calling the `ResourceBundle.getBundle` method. The values of variables within the resource bundle class are extracted by calling the `getString` method on the *bundle* object.

The `createTransfer` method of the TransferApplication can be called multiple times as explained in [Handling an invalid EJB object for a session bean](#). However, after the `InitialContext` object is created once, it remains good for the life of the client session. Therefore, the code required to create the `InitialContext` object is placed within an if statement that determines if the reference to the `InitialContext` object is null. If the reference is null, the `InitialContext` object is created; otherwise, the reference can be reused on subsequent creations of the EJB object.

Figure 27. Code example: Creating the InitialContext object

```
...
public class TransferApplication extends Frame implements ActionListener,
    WindowListener {
    ...
    private InitialContext ivjInitContext = null;
    private Transfer ivjTransfer = null;
    private ResourceBundle bundle = ResourceBundle.getBundle(
        "com.ibm.ejs.doc.client.ClientResourceBundle");
    ...
    private String nameService = null;
    private String accountName = null;
    private String providerUrl = null;
    ...
    private Transfer createTransfer() {
        TransferHome transferHome = null;
        Transfer transfer = null;
        // Get the initial context
        if (ivjInitContext == null) {
```

```

    try {
        Properties properties = new Properties();
        // Get location of name service
        properties.put(javax.naming.Context.PROVIDER_URL,
            bundle.getString("providerUrl"));
        // Get name of initial context factory
        properties.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
            bundle.getString("nameService"));

        ...
        ivjInitContext = new InitialContext(properties);
    } catch (Exception e) { // Error getting the initial context
        ...
    }
}

...
// Look up the home interface using the JNDI name
...
// Create a new Transfer object to return
...
return transfer;
}

```

Creating EJB home object

After the `InitialContext` object (*ivjInitContext*) is created, the application uses it to create the EJB home object, as shown in [Figure 28](#). This creation is accomplished by invoking the `lookup` method, which takes the JNDI name of the enterprise bean in String form and returns a `java.lang.Object` object. The JNDI name specified in the deployment descriptor is used.

The example `TransferApplication` gets the JNDI name of the `Transfer` bean from the `ClientResourceBundle` class. After an object is returned by the `lookup` method, the static method `javax.rmi.PortableRemoteObject.narrow` is used to obtain an EJB home object for the specified enterprise bean. The `narrow` method takes two parameters: the object to be narrowed and the class of the EJB home object to be returned by the `narrow` method. The object returned by the `javax.rmi.PortableRemoteObject.narrow` method is cast to the class associated with the home interface.

Figure 28. Code example: Creating the EJBHome object

```

private Transfer createTransfer() {
    TransferHome transferHome = null;
    Transfer transfer = null;
    // Get the initial context
    ...
    // Look up the home interface using the JNDI name
    try {
        java.lang.Object homeObject = ivjInitContext.lookup(
            bundle.getString("transferName"));
        transferHome = (TransferHome)javax.rmi.PortableRemoteObject.narrow(
            homeObject, TransferHome.class);
    } catch (Exception e) { // Error getting the home interface
        ...
    }
    ...
    // Create a new Transfer object to return
    ...
    return transfer;
}

```

Creating an EJB object

After the EJB home object is created, it is used to create the EJB object. [Figure 29](#) shows the code required to create the EJB object by using the EJB home object. A `create` method is invoked to create an EJB object or (for entity beans only) a `finder` method is invoked to find an existing EJB object. Because the `Transfer` bean is a stateless session bean, the only choice is the default `create` method.

Figure 29. Code example: Creating the EJB object

```
private Transfer createTransfer() {
    TransferHome transferHome = null;
    Transfer transfer = null;
    // Get the initial context
    ...
    // Look up the home interface using the JNDI name
    ...
    // Create a new Transfer object to return
    try {
        transfer = transferHome.create();
    } catch (Exception e) { // Error creating Transfer object
        ...
    }
    ...
    return transfer;
}
```

Handling an invalid EJB object for a session bean

Because session beans are ephemeral, the client cannot depend on a session bean's EJB object to remain valid. A reference to an EJB object for a session bean can become invalid if the EJB server fails or is restarted or if the session bean times out due to inactivity. (The reference to an entity bean's EJB object is always valid until that object is removed.) Therefore, the client of a session bean must contain code to handle a situation in which the EJB object becomes invalid.

An EJB client can determine if an EJB object is valid by placing all method invocations that use the reference inside of a try/catch block that specifically catches the `java.rmi.NoSuchObjectException`, in addition to any other exceptions that the method needs to handle. The EJB client can then invoke the code to handle this exception.

You determine how to handle an invalid EJB object. The example `TransferApplication` creates a new `Transfer` EJB object if the one it is currently using becomes invalid. The code to create a new EJB object when the old one becomes invalid is the same code used to create the original EJB object and is described in [Creating and getting a reference to a bean's EJB object](#). For the example `TransferApplication` client, this code is contained in the `createTransfer` method.

[Figure 30](#) shows the code used to create the new EJB object in the `getBalance` method of the example `TransferApplication`. The `getBalance` method contains the local boolean variable `sessionGood`, which is used to specify the validity of the EJB object referenced by the variable `ivjTransfer`. The `sessionGood` variable is also used to determine when to break out of the do-while loop. The `sessionGood` variable is initialized to false because the `ivjTransfer` can reference an invalid EJB object when the `getBalance` method is called. If the `ivjTransfer` reference is valid, the `TransferApplication` invokes the `Transfer` bean's `getBalance` method and returns the balance. If the `ivjTransfer` reference is invalid, the `NoSuchObjectException` is caught, the `TransferApplication`'s `createTransfer` method is called to create a new `Transfer` EJB object reference, and the `sessionGood` variable is set to false so that the do-while loop is repeated with the new valid EJB object. To prevent an infinite loop, the `sessionGood` variable is set to true when any other exception is thrown.

Figure 30. Code example: Refreshing the EJB object reference for a session bean

```
private float getBalance(long acctId) throws NumberFormatException, RemoteException,
    FinderException {
    // Assume that the reference to the Transfer session bean is no good
    ...
    boolean sessionGood = false;
    float balance = 0.0f;
    do {
        try {
            // Attempt to get a balance for the specified account
            balance = ivjTransfer.getBalance(acctId);
            sessionGood = true;
            ...
        } catch(NoSuchObjectException ex) {
            createTransfer();
        }
    } while (!sessionGood);
}
```

```

        sessionGood = false;
    } catch (RemoteException ex) {
        // Server or connection problem
        ...
    } catch (NumberFormatException ex) {
        // Invalid account number
        ...
    } catch (FinderException ex) {
        // Invalid account number
        ...
    }
} while (!sessionGood);
return balance;
}

```

Removing a bean's EJB object

When an EJB client no longer needs a stateful session EJB object, the EJB client should remove that object. Instances of stateful session beans have affinity to specific clients. They will remain in the container until they are explicitly removed by the client, or removed by the container when they time out. Meanwhile, the container might need to passivate inactive stateful session beans to disk. This requires overhead for the container and impacts performance of the application. If the passivated session bean is subsequently required by the application, the container activates it by restoring it from disk. By explicitly removing stateful session beans when finished with them, applications can decrease the need for passivation and minimize container overhead.

You remove entity EJB objects *only* when you want to remove the information in the data source with which the entity EJB object is associated.

To remove an EJB object, invoke the remove method on the object. As discussed in [Creating and getting a reference to a bean's EJB object](#), the TransferApplication contains only one reference to a Transfer EJB object that is created when the application is initialized.

[Figure 31](#) shows how the example Transfer EJB object is removed in the TransferApplication in the killApp method. To parallel the creation of the Transfer EJB object when the TransferApplication is initialized, the application removes the final EJB object associated with *ivjTransfer* reference right before closing the application's GUI window. The killApp method closes the window by invoking the dispose method on itself.

Figure 31. Code example: Removing a session EJB object

```

...
private void killApp() {
    try {
        ivjTransfer.remove();
        this.dispose();
        System.exit(0);    } catch (Throwable ivjExc) {
        ...
    }
}

```

Managing transactions in an EJB client

In general, it is practical to design your enterprise beans so that all transaction management is handled at the enterprise bean level. In a strict three-tier, distributed application, this is not always possible or even desirable. However, because the middle tier of an EJB application can include two subcomponents--session beans and entity beans--it is much easier to design the transactional management completely within the application server tier. Of course, the resource manager tier must also be designed to support transactions.

Note:

EJB clients that access entity beans with CMP that use Host On-Demand (HOD) or the External Call Interface (ECI) for CICS or IMS applications must begin a transaction before invoking a method on these entity beans. This restriction

is required because these types of entity beans must use the Mandatory transaction attribute.

Nevertheless, it is still possible to program an EJB client (that is not an enterprise bean) to participate in transactions for those specialized situations that require it. To participate in a transaction, the EJB client must do the following:

1. Obtain a reference to the `javax.transaction.UserTransaction` interface by using JNDI as defined in the Java Transaction Application Programming Interface (JTA).
2. Use the object reference to invoke any of the following methods:
 - `begin`--Begins a transaction. This method takes no arguments and returns `void`.
 - `commit`--Attempts to commit a transaction; assuming that nothing causes the transaction to be rolled back, successful completion of this method commits the transaction. This method takes no arguments and returns `void`.
 - `getStatus`--Returns the status of the referenced transaction. This method takes no arguments and returns `int`; if no transaction is associated with the reference, `STATUS_NO_TRANSACTION` is returned. The following are the valid return values for this method:
 - `STATUS_ACTIVE`--Indicates that transaction processing is still in progress.
 - `STATUS_COMMITTED`--Indicates that a transaction has been committed and the effects of the transaction have been made permanent.
 - `STATUS_COMMITTING`--Indicates that a transaction is in the process of committing (that is, the transaction has started committing but has not completed the process).
 - `STATUS_MARKED_ROLLBACK`--Indicates that a transaction is marked to be rolled back.
 - `STATUS_NO_TRANSACTION`--Indicates that a transaction does not exist in the current transaction context.
 - `STATUS_PREPARED`--Indicates that a transaction has been prepared but not completed.
 - `STATUS_PREPARING`--Indicates that a transaction is in the process of preparing (that is, the transaction has started preparing but has not completed the process).
 - `STATUS_ROLLEDBACK`--Indicates that a transaction has been rolled back.
 - `STATUS_ROLLING_BACK`--Indicates that a transaction is in the process of rolling back (that is, the transaction has started rolling back but has not completed the process).
 - `STATUS_UNKNOWN`--Indicates that the status of a transaction is unknown.
 - `rollback`--Rolls back the referenced transaction. This method takes no arguments and returns `void`.
 - `setRollbackOnly`--Specifies that the only possible outcome of the transaction is for it to be rolled back. This method takes no arguments and returns `void`.
 - `setTransactionTimeout`--Sets the timeout (in seconds) associated with the transaction. If some transaction participant has not specifically set this value, a default timeout is used. This method takes a number of seconds (as type `int`) and returns `void`.

Figure 32 provides an example of an EJB client creating a reference to a `UserTransaction` object and then using that object to set the transaction timeout, begin a transaction, and attempt to commit the transaction. (The source code for this example is *not* available with the example code provided with this document.) Notice that the client does a simple type cast of the lookup result, rather than invoking a narrow method as required with other JNDI lookups. In both EJB server environments, the JNDI name of the `UserTransaction` interface is `java:comp/UserTransaction`.

Figure 32. Code example: Managing transactions in an EJB client

```
...
import javax.transaction.*;
...
// Use JNDI to locate the UserTransaction object
Context initialContext = new InitialContext();
UserTransaction tranContext = (
    UserTransaction)initialContext.lookup("java:comp/UserTransaction");
// Set the transaction timeout to 30 seconds
tranContext.setTransactionTimeout(30);
...
// Begin a transaction
tranContext.begin();
// Perform transaction work invoking methods on enterprise bean references
```

```
...  
// Call for the transaction to commit  
tranContext.commit();
```

Developing servlets that use enterprise beans

A servlet is a Java application that enables users to access Web server functionality. To use servlets, a Web server is required. The WebSphere Application Server plugs into a number of commonly used Web servers. The IBM HTTP Server with the Advanced Application Server. For more information, consult the Advanced Edition InfoCenter.

Java servlets can be combined with enterprise beans to create powerful EJB applications. This chapter describes how to use enterprise beans within a servlet. The example CreateAccount servlet, which uses the example Account bean, is used to illustrate the concepts discussed in this chapter. The example servlet and enterprise bean discussed in this chapter are explained in [Information about the examples described in the documentation](#).

An overview of standard servlet methods

Usually, a servlet is invoked from an HTML form on the user's browser. The first time the servlet is invoked, the servlet's init method is run to perform any initializations required at startup. For the first and all subsequent invocations of the servlet, the doGet method (or, alternatively, the doPost method) is run. Within the doGet method (or the doPost method), the servlet gets the information provided by the user on the HTML form and uses that information to perform work on the server and access server resources.

The servlet then prepares a response and sends the response back to the user. After a servlet is loaded, it can handle multiple simultaneous user requests. Multiple request threads can invoke the doGet (or doPost) method at the same time, so the servlet needs to be made thread safe.

When a servlet shuts down, the destroy method of the servlet is run in order to perform any needed shutdown processing.

Writing an HTML page that embeds a servlet

[Figure 33](#) shows the HTML file (named create.html) used to invoke the CreateAccount servlet. The HTML form is used to specify the account number for the new account, its type (checking or savings), and its initial balance. The request is passed to the doGet method of the servlet, where the servlet is identified with its full Java package name, as shown in the example.

Figure 33. Code example: Content of the create.html file used to access the CreateAccount servlet

```
<html>
<head>
<title>Create a new Account</title>
</head>
<body>
<h1 align="center">Create a new Account</h1>
<form method="get"
action="/servlet/com.ibm.ejs.doc.client.CreateAccount">
<table border align="center">
<!-- specify a new account number -->
<tr bgcolor="#cccccc">
<td align="right">Account Number:</td>
<td colspan="2"><input type="text" name="account" size="20"
maxlength="10">
</tr>
<!-- specify savings or checking account -->
...
<!-- specify account starting balance -->
...
<!-- submit information to servlet -->
...
<input type="submit" name="submit" value="Create">
...
<!-- message area -->
...
```

```
</form>
</body>
</html>
```

The HTML response from the servlet is designed to produce a display identical to create.html, enabling the user to continue creating new accounts. [Figure 34](#) shows what create.html looks like on a browser.

Figure 34. The initial form and output of the CreateAccount servlet



Developing the servlet

This section discusses the basic code required by a servlet that interacts with an enterprise bean. [Figure 35](#) shows the basic outline of the code that makes up the CreateAccount servlet. As shown in the example, the CreateAccount servlet extends the javax.servlet.http.HttpServlet class and implements an init method and a doGet method.

Figure 35. Code example: The CreateAccount class

```
package com.ibm.ejs.doc.client;
// General enterprise bean code.
import java.rmi.RemoteException;
import javax.ejb.DuplicateKeyException;
// Enterprise bean code specific to this servlet.
import com.ibm.ejs.doc.account.AccountHome;
import com.ibm.ejs.doc.account.AccountKey;
import com.ibm.ejs.doc.account.Account;
// Servlet related.
import javax.servlet.*;
import javax.servlet.http.*;
// JNDI (naming).
import javax.naming.*; // for Context, InitialContext, NamingException
// Miscellaneous:
import java.util.*;
```

```

import java.io.*;
...
public class CreateAccount extends HttpServlet {
    // Variables
    ...
    public void init(ServletConfig config) throws ServletException {
        ...
    }
    public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
        // --- Read and validate user input, initialize. ---
        ...
        // --- If input parameters are good, try to create account. ---
        ...
        // --- Prepare message to accompany response. ---
        ...
        // --- Prepare and send HTML response. ---
        ...
    }
}

```

The servlet's instance variables

Figure 36 shows the instance variables used in the CreateAccount servlet. The *nameService*, *accountName*, and *providerUrl* variables are used to specify the property values required during JNDI lookup. These values are obtained from the ClientResourceBundle class as described in [Creating and getting a reference to a bean's EJB object](#).

The CreateAccount class also initializes the string constants that are used to create the HTML response sent back to the user. (Only three of these variables are shown, but there are many of them). The init method in the CreateAccount servlet provides a way to read strings from a resource bundle to override these US English defaults in order to provide a response in a different national language. The instance variable *accountHome* is used by all client requests to create a new Account bean instance. The *accountHome* variable is initialized in the init method as shown in Figure 36.

Figure 36. Code example: The instance variables of the CreateAccount class

```

...
public class CreateAccount extends HttpServlet {
    // Variables for finding the home
    private String nameService = null;
    private String accountName = null;
    private String providerURL = null;
    private ResourceBundle bundle = ResourceBundle.getBundle(
        "com.ibm.ejs.doc.client.ClientResourceBundle");
    // Strings for HTML output - US English defaults shown.
    static String title = "Create a new Account";
    static String number = "Account Number:";
    static String type = "Type:";
    ...
    // Variable for accessing the enterprise bean.
    private AccountHome accountHome = null;
    ...
}

```

The servlet's init method

The init method of the CreateAccount servlet is shown in Figure 37. The init method is run once, the first time a request is processed by the servlet, after the servlet is started. Typically, the init method is used to do any one-time initializations for a servlet. For example, the default US English strings used in preparing the HTML response can be replaced with another national language. The init method is also the best place to initialize the value of references to the home interface of any enterprise beans used by the servlet. In the CreateAccount's init method, the *accountHome* variable is initialized to

reference the EJB home object of the Account bean.

As in other types of EJB clients, the properties required to do a JNDI lookup are specific to the EJB implementation. Therefore, these properties are externalized in a properties file or a resource bundle class. For more information on these properties, see [Creating and getting a reference to a bean's EJB object](#).

Note that in the CreateAccount servlet, a Hashtable object is used to store the properties required to do a JNDI lookup whereas a Properties object is used in the TransferApplication. Both of these classes are valid for storing these properties.

Figure 37. Code example: The init method of the CreateAccount servlet

```
// Variables for finding the EJB home object
private String nameService = null;
private String accountName = null;
private String providerURL = null;
private ResourceBundle bundle = ResourceBundle.getBundle(
    "com.ibm.ejs.doc.client.TransferResourceBundle");

...
public void init(ServletConfig config) throws ServletException {
    super.init(config);

    ...
    try {
        // Get NLS strings from an external resource bundle
        ...
        createTitle = bundle.getString("createTitle");
        number = bundle.getString("number");
        type = bundle.getString("type");
        ...
        //Get values for the naming factory and home name.
        nameService = bundle.getString("nameService");
        accountName = bundle.getString("accountName");
        providerURL = bundle.getString("providerURL");
    }
    catch (Exception e) {
        ...
    }
    // Get home object for access to Account enterprise bean.
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY, nameService);
    try {
        // Create the initial context.
        Context ctx = new InitialContext(env);
        // Get the home object.
        Object homeObject = ctx.lookup(accountName);
        // Get the AccountHome object.
        accountHome = (AccountHome) javax.rmi.PortableRemoteObject.narrow(
            homeObject, AccountHome.class);
    }
    // Determine cause of failure.
    catch (NamingException e) {
        ...
    }
    catch (Exception e) {
        ...
    }
}
```

Note:

Although the init method is a good place to obtain references to EJB home objects, it is not a good place to create enterprise beans or access other beans that might be protected with WebSphere security. Depending upon the authorization policy on the protected objects, creating or accessing these objects from within the init method could fail for authentication or authorization reasons because they were not accessed with the proper security credentials.

Creating or accessing protected objects should be done after the init method, in one of the servlet's doXXX methods.

The servlet's doGet method

The doGet method is invoked for every servlet request. In the CreateAccount servlet, the method does the following tasks to manage user input. These tasks are fairly standard for this method:

- Read the user input from the HTML form and decide if the input is valid--for example, whether the user entered a valid number for an initial balance.
- Perform the initializations required for each request.

Figure 38 shows the parts of the doGet method that handle user input. Note that the *req* variable is used to read the user input from the HTML form. The *req* variable is a `javax.servlet.http.HttpServletRequest` object passed as one of the arguments to the doGet method.

Figure 38. Code example: The doGet method of the CreateAccount servlet

```
public void doGet (HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    // --- Read and validate user input, initialize. ---
    // Error flags.
    boolean accountFlag = true;
    boolean balanceFlag = true;
    boolean inputFlag = false;
    boolean createFlag = true;
    boolean duplicateFlag = false;
    // Datatypes used to create new account bean.
    AccountKey key;
    int typeAcct = 0;
    String typeString = "0";
    float initialBalance = 0;
    // Read input parameters from HTML form.
    String[] accountArray = req.getParameterValues("account");
    String[] typeArray = req.getParameterValues("type");
    String[] balanceArray = req.getParameterValues("balance");
    // Convert input parameters to needed datatypes for new account.
    // (account)
    long accountLong = 0;
    ...
    key = new AccountKey(accountLong);
    // (type)
    if (typeArray[0].equals("1")) {
        typeAcct = 1;           // Savings account.
        typeString = "savings";
    }
    else if (typeArray[0].equals("2")) {
        typeAcct = 2;           // Checking account
        typeString = "checking";
    }
    // (balance)
    try {
        initialBalance = (Float.valueOf(balanceArray[0])).floatValue();
    } catch (Exception e) {
        balanceFlag = false;
    }
    ...
    // --- If input parameters are good, try to create account bean. ---
    ...
    // --- Prepare message to accompany response. ---
    ...
    // --- Prepare and send HTML response. ---
```

```
    ...  
}
```

Creating an enterprise bean

If the user input is valid, the `doGet` method attempts to create a new account based on the user input as shown in [Figure 39](#). Besides the initialization of the home object reference in the `init` method, this is the only other piece of code that is specific to the use of enterprise beans in a servlet.

Figure 39. Code example: Creating an enterprise bean in the `doGet` method

```
public void doGet(HttpServletRequest req, HttpServletResponse res)  
    throws ServletException, IOException {  
    // --- Read and validate user input, initialize ---.  
    ...  
    // --- If input parameters are good, try to create account bean. ---  
    if (accountFlag && balanceFlag) {  
        inputFlag = true;  
        try {  
            // Create the bean.  
            Account account = accountHome.create(key, typeAcct, initialBalance);  
        }  
        // Determine cause of failure.  
        catch (RemoteException e) {  
            ...  
        }  
        catch (DuplicateKeyException e) {  
            ...  
        }  
        catch (Exception e) {  
            ...  
        }  
    }  
    // --- Prepare message to accompany response. ---  
    ...  
    // --- Prepare and send HTML response. ---  
    ...  
}
```

Determining the content of the user response

Next, the `doGet` method prepares a response message to be sent to the user. There are three possible responses:

- The user input was not valid.
- The user input was valid, but the account was not created for some reason.
- The account was created successfully. If the previous two errors do not occur, this response is prepared.

[Figure 40](#) shows the code used by the servlet to determine which response to send to the user. If no errors are encountered, then the response indicates success.

Figure 40. Code example: Determining a user response in the `doGet` method

```
public void doGet(HttpServletRequest req, HttpServletResponse res)  
    throws ServletException, IOException {  
    // --- Read and validate user input, initialize. ---  
    ...  
    // --- If input parameters are good, try to create account bean. ---  
    ...  
    // --- Prepare message to accompany response. ---  
    ...  
    String messageLine = "";
```

```

if (inputFlag) {
    // If you are here, the client input is good.
    if (createFlag) {
        // New account enterprise bean was created.
        messageLine = createdaccount + " " + accountArray[0] + ", " +
            createdtype + " " + typeString + ", " +
            createdbalance + " " + balanceArray[0];
    }
    else if (duplicateFlag) {
        // Account with same key already exists.
        messageLine = failureexists + " " + accountArray[0];
    }
    else {
        // Other reason for failure.
        messageLine = failureinternal + " " + accountArray[0];
    }
}
else {
    // If you are here, something was wrong with the client input.
    String separator = "";
    if (!accountFlag) {
        messageLine = failureaccount + " " + accountArray[0];
        separator = ", ";
    }
    if (!balanceFlag) {
        messageLine = messageLine + separator +
            failurebalance + " " + balanceArray[0];
    }
    // --- Prepare and send HTML response. ---
    ...
}

```

Sending the user response

With the type of response determined, the `doGet` method then prepares the full HTML response and sends it to the user's browser, incorporating the appropriate message. Relevant parts of the full HTML response are shown in [Figure 41](#). The `res` variable is used to pass the response back to the user. This variable is an `HttpServletResponse` object passed as an argument to the `doGet` method. The response code shown here mixes both display (HTML) and content in one servlet. You can separate the display and the content by using JavaServer Pages (JSP). A JSP allows the display and content to be developed and maintained separately.

Figure 41. Code example: Responding to the user in the `doGet` method

```

public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    // --- Read and validate user input, initialize. ---
    ...
    // --- If input parameters are good, try to create account bean. ---
    ...
    // --- Prepare message to accompany response. ---
    ...
    // --- Prepare and send HTML response. ---
    // HTML returned looks like initial HTML that invoked this servlet.
    // Message line says whether servlet was successful or not.
    res.setContentType("text/html");
    res.setHeader("Pragma", "no-cache");
    res.setHeader("Cache-control", "no-cache");
    PrintWriter out = res.getWriter();
    out.println("<html>");
    ...
    out.println("<title> " + createTitle + "</title>");
}

```

```
    ...  
    out.println( " </html>" );  
}
```

Threading issues

Except for the instance variable required to get a reference to the Account bean's home interface and to support multiple languages (which remain unchanged for all user requests), all other variables used in the CreateAccount servlet are local to the doGet method. Each request thread has its own set of local variables, so the servlet can handle simultaneous user requests.

As a result, the CreateAccount servlet is thread safe. By taking a similar approach to servlet design, you can also make your servlets thread safe.

Tools for developing and deploying enterprise beans

There are two basic approaches to developing and deploying enterprise beans:

- You can use one of the available integrated development environments (IDEs) such as IBM VisualAgeTM for Java Enterprise Edition. IDE tools automatically generate significant parts of the enterprise bean code and contain integrated tools for packaging and testing enterprise beans. VisualAge for Java is the recommended development tool. For more information on using VisualAge for Java, see [Using VisualAge for Java](#).
 - You can use the tools available in the Java Software Development Kit (SDK) and the Advanced Application Server. For more information, see [Developing and deploying enterprise beans](#).
-

Using VisualAge for Java

Before you can develop enterprise beans in VisualAge for Java, you must set up the EJB development environment. You need to perform this setup task only once. This setup procedure directs VisualAge for Java to import all of the classes and interfaces required to develop enterprise beans.

After generating an enterprise bean, you complete its development by following these general steps:

1. Implement the enterprise bean class.
2. Create the required abstract methods in the bean's home and remote interfaces by promoting the corresponding methods in the bean class to the appropriate interface.
3. For entity beans, do the following:
 - a. Create any additional finder methods in the home interface by using the appropriate menu items.
 - b. Create a finder helper interface, if required.
4. Create the EJB module and corresponding deployment descriptor.
5. Generate the deployment code for the bean.

VisualAge for Java contains a complete WebSphere Application Server run time environment and a mechanism to generate a test client to test your enterprise beans. For much more detailed information on developing enterprise beans in VisualAge for Java, refer to the VisualAge for Java documentation.

Developing and deploying enterprise beans

If you have decided to develop enterprise beans *without* an IDE, you need at minimum the following tools:

- An ASCII text editor. (You can also use a Java development tool that does not support enterprise bean development.)
- The SDK Java compiler (**javac**) and Java Archiving tool (**jar**).
- The WebSphere Application Assembly Tool and the WebSphere Administrative Console.

This section describes steps you can follow to develop enterprise beans by using these tools. The following tasks are involved in the development of enterprise beans:

1. Ensure that you have installed and configured the prerequisite software to develop, deploy, and run enterprise beans in the EJB server environment. For more information, see [Installing and configuring the software for the EJB server](#).
2. Set the CLASSPATH environment variable required by different components of the EJB server environment. For more information, see [Setting the CLASSPATH environment variable in the EJB](#)

[server environment](#).

3. Write and compile the components of the enterprise bean. For more information, see [Creating the components of an enterprise bean](#).
4. (*Entity beans with CMP only*) Create a finder helper interface for each entity bean with CMP that contains specialized finder methods (other than the `findByPrimaryKey` method). For more information, see [Creating finder logic in the EJB server](#).
5. Create an EJB module and corresponding deployment descriptor by using the Application Assembly Tool. For more information, see [Creating an EJB module](#).
6. (*Entity beans only*) Create a database schema to enable storage of the entity bean's persistent data in a database. For more information, see [Creating a database for use by entity beans](#).
7. Generate deployment code for the EJB module by using the Application Assembly Tool. For more information, see the WebSphere InfoCenter and the online help available with the Application Assembly Tool.
8. Install the EJB module into an EJB server and start the server by using the WebSphere Administrative Console.

Installing and configuring the software for the EJB server

You must ensure that you have installed and configured the following prerequisite software products before you can begin developing enterprise beans and EJB clients with the EJB server:

- WebSphere Application Server Advanced Edition
- One or more of the following databases for use by entity beans with container-managed persistence (CMP):
 - DB2
 - Oracle
 - Sybase
 - Informix
 - Microsoft SQL Server
 - InstantDB
- The Java Software Development Kit (SDK)

For information on the appropriate version numbers of these products and instructions for setting up the environment, see the WebSphere InfoCenter.

Setting the CLASSPATH environment variable in the EJB server environment

In addition to the `classes.zip` file contained in the SDK, the following WebSphere JAR files must be appended to the CLASSPATH environment variable for developing enterprise beans:

- `ejs.jar`
- `ujc.jar`
- `otherDeployedBean.jar` (if the enterprise bean uses another enterprise bean). This is the deployed JAR file containing the enterprise bean being used by this enterprise bean.

For developing and running an EJB client, the following WebSphere JAR files must be appended to the CLASSPATH environment variable:

- `ejs.jar`

- `ujc.jar`
- `servlet.jar` (required by EJB clients that are servlets)
- *otherDeployedBean.jar*. This is the deployed JAR file containing the enterprise bean being used by this EJB client.

Creating the components of an enterprise bean

If you use an ASCII text editor or a Java development tool that does not support enterprise bean development, you must create each of the components that compose the enterprise bean you are creating. You must ensure that these components match the requirements described in [Developing enterprise beans](#).

To manually develop a session bean, you must write the bean class, the bean's home interface, and the bean's remote interface. To manually develop an entity bean, you must write the bean class, the bean's primary key class, the bean's home interface, the bean's remote interface, and if necessary, the bean's finderHelper interface. After you have properly coded these components, use the Java compiler to create the corresponding Java class files. For example, because the components of the example Account bean are stored in a specific directory, the bean components can be compiled by issuing the following command:

```
C:\MYBEANS\COM\IBM\EJS\DOC\ACCOUNT> javac *.java
```

This command assumes that the CLASSPATH environment variable contains all of the packages used by the Account bean.

Creating finder logic in the EJB server

For the EJB server environment, the following finder logic is required for each finder method (other than the `findByPrimaryKey` method) contained in the home interface of an entity bean with CMP:

- The logic must be defined in a public interface named *NameBeanFinderHelper*, where *Name* is the name of the enterprise bean (for example, `AccountBeanFinderHelper`).
- The logic must be contained in a String constant named *findMethodNameWhereClause*, where *findMethodName* is the name of the finder method. The String constant can contain zero or more question marks (?) that are replaced from left to right with the value of the finder method's arguments when that method is invoked.

Note:

Encapsulating the logic in a String constant named *findMethodNameQueryString* has been deprecated.

If you define the `findLargeAccounts` method shown in [Figure 14](#), you must also create the `AccountBeanFinderHelper` interface shown in [Figure 7](#).

Figure 7. Code example: AccountBeanFinderHelper interface for the EJB server

```
...
public interface AccountBeanFinderHelper{
    String findLargeAccountsWhereClause = "balance > ?";
}
```

Creating an EJB module

The WebSphere Application Server Application Assembly Tool can be used to create an EJB module. An EJB module can contain one or more enterprise beans. The tool automatically creates the required deployment descriptor for the module based on information specified by the user.

Using the Application Assembly Tool

To create an EJB module and corresponding deployment descriptor, use the Create EJB Module wizard in the Application Assembly Tool. This wizard prompts you to specify the following information for each enterprise bean to be included in the module:

- The enterprise bean class, home interface class, and remote interface class.
- The bean type (entity or session), and associated attributes (such as persistence management type and primary key class for entity beans).
- References to another enterprise bean's home interface and to resource connection factories.
- References to security roles for the enterprise bean.
- CMP fields, if applicable.
- Transaction isolation level attributes for enterprise bean methods.

The wizard also prompts you to specify the following application assembly information for the module itself:

- General properties of the EJB module, such as the location of class files needed for a client program to access the enterprise beans in the module and the icons to be associated with the module.
- The deployable enterprise beans that the module will contain.
- Security roles used to access resources in the module.
- Transaction attributes for the enterprise bean methods.

Both bean and module information are used to create the deployment descriptor. See the WebSphere InfoCenter and the online help for details on how to use the Application Assembly Tool.

Creating a database for use by entity beans

For entity beans with *container-managed persistence (CMP)*, you must store the bean's persistent data in one of the supported databases. The Application Assembly Tool automatically generates SQL code for creating database tables for CMP entity beans. The tool names the database schema and table `ejb.beanNamebeantbl`, where *beanName* is the name of the enterprise bean (for example, `ejb.accountbeantbl`). If your CMP entity beans require complex database mappings, it is recommended that you use VisualAge for Java to generate code for the database tables. At run time, the WebSphere Administrative Console displays a prompt asking whether you want to execute the generated SQL code that creates the database table.

For entity beans with *bean-managed persistence (BMP)*, you can create the database and database table by using the database tools or use an existing database and database table. Because entity beans with BMP handle the database interaction, any database or database table name is acceptable.

For more information on creating databases and database tables, consult your database documentation and the online help for the WebSphere Administrative Console.

Appendix A. Changes for version 1.1 of the EJB specification

WebSphere Application Server supports version 1.1 of the EJB specification. This appendix describes features that are new or have changed in version 1.1 and discusses migration issues for enterprise beans written to version 1.0 of the EJB specification.

New and updated features

The following enterprise bean features are new or have changed for version 1.1.

- Environmental dependencies for enterprise beans are now specified using entries in a JNDI naming context. An instance of an enterprise bean creates a `javax.naming.InitialContext` object by invoking the constructor with no arguments specified. It looks up the environment naming context by using the `InitialContext` object under the name `java:comp/env`.
 - Primary keys are handled differently in version 1.1 of the EJB specification. Entity bean providers are not required to specify the primary key class for entity beans with container-managed persistence (CMP), enabling the deployer to select the primary key fields when the bean is deployed into a container.
 - The deployment descriptor has enhanced support for application assembly.
-

Migrating from version 1.0 to version 1.1

From the client's perspective, enterprise beans written to version 1.1 of the EJB specification appear nearly identical to enterprise beans written to version 1.0 of the specification. However, the following EJB 1.1 changes do affect clients:

- Enterprise beans written to version 1.1 of the EJB specification are registered in a different part of the JNDI namespace. For example, a client can look up the initial context of a version 1.0 enterprise bean in JNDI by using the **`initialContext.lookup`** method as follows:

```
initialContext.lookup( "com/ibm/Hello" )
```

The JNDI lookup for the equivalent version 1.1 enterprise bean is:

```
initialContext.lookup( "java:comp/env/ejb/Hello" )
```

- The `UserTransaction` object is obtained differently for enterprise beans written to version 1.1 of the EJB specification. Under version 1.0, it was obtained as:

```
initialContext.lookup( "jta/UserTransaction" )
```

Under version 1.1, it is obtained as:

```
initialContext.lookup( "java:comp/UserTransaction" )
```

- Because entity beans written to version 1.1 of the EJB specification now support primitive primary keys (instead of having to encapsulate them in a primary key class), the client needs to look up these primitive keys directly. For example, a client can look up a primitive key of the type `java.lang.Integer` as follows:

```
accountHome.findByPrimaryKey(new Integer(5))
```

Primary key classes are still supported, although their use for primitive data types is deprecated.

From the application developer's perspective, the following changes need to be made to make enterprise beans written to version 1.0 of the EJB specification compatible with version 1.1 of the specification.

- All deployment descriptors must be converted to the XML format specified in version 1.1 of the EJB specification.
- In general, enterprise beans written to version 1.0 of the EJB specification are compatible with version 1.1. However, you need to modify or recompile enterprise bean code in the following cases:
 - The return value of the `ejbCreate` method must be modified for all entity beans with CMP. The `ejbCreate` method is now required to return the same type as the primary key; the actual value returned must be null. These beans also must be recompiled. For more information, see [Implementing the `ejbCreate` and `ejbPostCreate` methods](#)
 - If the `javax.jts.UserTransaction` interface is used. This interface has been renamed to `javax.transaction.UserTransaction`. Enterprise beans that use this interface must be modified to use the new interface name. There have also been minor changes to the exceptions thrown by this interface.
 - If the `getCallerIdentity` or `isCallerInRole` methods of the `javax.ejb.EJBContext` interface are used. These methods were deprecated because the `javax.security.Identity` class is deprecated under the Java 2 platform.
 - If an entity bean uses the `UserTransaction` interface, which is not permitted under version 1.1 of the EJB specification.
 - If an entity bean whose finder methods do not define the `FinderException` in the methods' throws clauses. Under version 1.1, the finder methods of entity beans must define this exception.
 - If an entity bean uses the `UserTransaction` interface and implements the `SessionSynchronization` interface. Entity beans can neither use the `UserTransaction` interface nor implement the `SessionSynchronization` interface under version 1.1.
 - If a stateless session bean implements the `SessionSynchronization` interface. Stateless session beans should not implement the `SessionSynchronization` interface under version 1.1.
 - If an enterprise bean violates any of the new semantic restrictions defined in version 1.1 of the EJB specification.
 - Throwing the `javax.ejb.RemoteException` exception from the bean implementations is deprecated in version 1.1. This exception should be replaced by the `javax.ejb.EJBException` or a more specific exception such as the `javax.ejb.CreateException`. The `javax.ejb.EJBException` inherits from the `javax.ejb.RuntimeException` and does not need to be explicitly declared in throws clauses.

Declare the `javax.ejb.RemoteException` exception in the remote and home interfaces, as required by RMI. Throwing this exception directly by the bean implementation is deprecated. However, it can be thrown by the container due to a system exception or by mapping an exception thrown by the bean implementation.

Appendix B. Example code provided with WebSphere Application Server

This appendix contains information on the example code provided with the WebSphere Application Server.

Information about the examples described in the documentation

The example code discussed throughout this document is taken from a set of examples provided with the product. This set of examples is composed of the following main components:

- The Account entity bean, which models either a checking or savings bank account and maintains the balance in each account. An account ID is used to uniquely identify each instance of the bean class and to act as the primary key. The persistent data in this bean is container managed and consists of the following variables:
 - *accountId*--The account ID that uniquely identifies the account. This variable is of type long.
 - *type*--An integer that identifies the account as either a savings account (1) or a checking account (2). This variable is of type int.
 - *balance*--The current balance of the account. This variable is of type float.

The major components of this bean are discussed in [Developing entity beans with CMP](#).

- The AccountBM entity bean, which is nearly identical to the Account entity bean; however, the AccountBM bean implements bean-managed persistence. This bean is not used by any other enterprise bean, application, or servlet contained in the documentation example set. The major components of this bean are discussed in [Developing entity beans with BMP](#).
- The Transfer session bean, which models a funds transfer session that involves moving a specified amount between two instances of an Account bean. The bean contains two methods: the transferFunds method transfers funds between two accounts, the getBalance method retrieves the balance for a specified account. The bean is stateless. The major components of this bean are discussed in [Developing session beans](#).
- The CreateAccount servlet, which can be used to easily create new bank accounts (and corresponding Account bean instances) with the specified account ID, account type, and initial balance. Although this servlet is designed to make it easy for you to create accounts and demonstrate the other components in the example set, it also illustrates servlet interaction with an entity bean. This servlet is discussed in [Developing servlets that use enterprise beans](#).
- The TransferApplication Java application, which provides a graphical user interface that was built with the abstract windowing toolkit (AWT). The application creates an instance of the Transfer session bean, which is then manipulated to transfer funds between two selected accounts or to get the balance for a specified account. The TransferApplication code implements many of the requirements for using enterprise beans in an EJB client. The parts of this application that are relevant to interacting with an enterprise bean are discussed in [Developing EJB clients](#).
- The TransferFunds servlet, which is a servlet version of the TransferApplication Java application. This servlet is provided so that you can compare the use of enterprise beans between a Java application and a Java servlet that basically are doing the same tasks. This document does not discuss this servlet in any detail.

Note:

The example code in the documentation was written to be as simple as possible. The goal of these examples is to provide code that teaches the fundamental concepts of enterprise bean and EJB client development. It is not meant to provide an example of how a bank (or any similar company) possibly

approaches the creation of a banking application. For example, the Account bean contains a *balance* variable that has a type of float. In a real banking application, you must not use a float type to keep records of money; however, using a class like `java.math.BigDecimal` or a currency-handling class within the examples would complicate them unnecessarily. Remember this as you examine these examples.

Information about other examples

[Table 2](#) provides a summary of the enterprise bean-specific examples provided with the EJB server

Table 2. Examples available with the EJB server

Name	Bean types	EJB client types	Additional information
Hello	Stateless session	Java servlet	Very simple example of a session bean.
Increment	CMP entity	Java servlet	Very simple example of an entity bean.

Appendix C. Extensions to the EJB Specification

This appendix briefly discusses functional extensions to the EJB Specification that are available in the EJB server environments contained in WebSphere Application Server. These extensions are specific to WebSphere Application Server and use of these features is supported only with VisualAge for Java, Enterprise Edition. For information on implementing these features, consult your VisualAge for Java documentation.

Access beans

Access beans are Java components that adhere to the Sun Microsystems JavaBeans^(TM) Specification and are meant to simplify development of EJB clients. An access bean adapts an enterprise bean to the JavaBeans programming model by hiding the home and remote interfaces from the access bean user (that is, an EJB client developer).

There are three types of access beans, which are listed in ascending order of complexity:

- **Java bean wrapper**--Of the three types of access beans, a Java bean wrapper is the simplest to create. It is designed to allow either a session or entity enterprise bean to be used like a standard Java bean and it hides the enterprise bean home and remote interfaces from you. Each Java bean wrapper that you create extends the `com.ibm.ivj.ejb.access.AccessBean` class.
- **Copy helper**--A copy helper access bean has all of the characteristics of a Java bean wrapper, but it also incorporates a single copy helper object that contains a local copy of attributes from a remote entity bean. A user program can retrieve the entity bean attributes from the local copy helper object that resides in the access bean, which eliminates the need to access the attributes from the remote entity bean.
- **Rowset**--A rowset access bean has all of characteristics of both the Java bean wrapper and copy helper access beans. However, instead of a single copy helper object, it contains multiple copy helper objects. Each copy helper object corresponds to a single enterprise bean instance.

VisualAge for Java provides a SmartGuide to assist you in creating or editing access beans.

Associations between enterprise beans

In the EJB server environment, an association is a relationship that exists between two CMP entity beans. There are three types of associations: one-to-one and one-to-many. In a one-to-one association, a CMP entity bean is associated with a single instance of another CMP entity bean. For example, an Employee bean could be associated with only a single instance of a Department bean, because an employee generally belongs only to a single department.

In a one-to-many association, a CMP entity bean is associated with multiple instances of another CMP entity bean. For example, a Department bean could be associated with multiple instances of an Employee bean, because most departments are made up of multiple employees.

The Association Editor is used to create or edit associations between CMP entity beans in VisualAge for Java.

Inheritance in enterprise beans

In Java, *inheritance* is the creation of a new class from an existing class or a new interface from an existing interface. The EJB server environment permits two forms of inheritance: standard class inheritance and EJB inheritance. In standard class inheritance, the home interface, remote interface, or enterprise bean class inherits properties and methods from base classes that are not themselves enterprise bean classes or interfaces.

In enterprise bean inheritance, by comparison, an enterprise bean inherits properties (such as CMP fields and association ends), methods, and method-level control descriptor attributes from another enterprise bean that resides in the same group.

VisualAge for Java provides a SmartGuide to assist you in implementing inheritance in enterprise beans.

4.4: Personalizing applications

Personalization describes a range of features that enable applications to treat visitors as particular individuals. For a really simple example, consider a site that issues the message "Hello, John Smith" when the customer John Smith logs onto the site.

Personalized service can give your Web site a competitive edge, much like a good customer service team can add value to human-to-human interactions at your physical site and keep customers coming back. Personalization can also increase the chance that your Web site presents a user with content that is of particular interest to that person.

For an e-business site, personalization can be fairly necessary, even if it does not go so far as to call customers by name. For example, suppose several Web site visitors are performing various transactions concurrently. Applications need some way to group each user's transactions into a unit that is separate from the transactions of other users. *Session tracking* provides such functionality.

See articles 0.11 and 0.12 to learn about two complementary personalization approaches supported by IBM WebSphere Application Server -- tracking user sessions and maintaining user profiles.

If you are already familiar with the concepts, skip ahead to 4.4.1 and 4.4.2 for programming details. See 6.6.11 and 6.6.12 to take a look at the administrative aspects.

For additional capability offered by the IBM WebSphere Personalization product, visit the following Web site:

<http://www.ibm.com/software/webservers/personalization/>

4.4.1: Tracking sessions

IBM WebSphere Application Server provides a service for tracking user sessions -- the Session Manager. The service is provided in the form of IBM classes and packages.

The key activities for session tracking are summarized.

1. Become familiar with the programming model for accessing session support from servlets. See article 4.4.1.1 for an overview with links to details about security, clustering, limitations, and other topics.
2. Create or modify your own servlets to use session support to maintain sessions on behalf of Web applications.

Follow the model outlined in the previous step.

3. Ensure the administrator appropriately configures Session Managers in the administrative domain. See [article 6.6.11](#).
4. Adjust configuration settings and perform other tuning activities for optimal use of sessions in your environment. See [article 4.4.1.1.7](#).

4.4.1.1: 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 [SessionSample.java](#), provides a programming model for implementing sessions in your own servlets.

It is also recommended that you read the topics listed in the related information. They can influence how you implement sessions in your own servlets.

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 2.2 API.

When you first obtain the `HttpSession` object, the Session Manager uses one of three ways to establish tracking of the session: cookies, URL rewriting, or SSL information. See [section 4.4.1.1.1](#) for a discussion to help you decide which is more appropriate for your situation.

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 2.2 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/or 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.

In order 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, including permanently saving to a database data changes made during the session.

5. **End the session.**

You can end a session:

- 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.

4.4.1.1.1: Deciding between session tracking approaches

Suppose a servlet implementing sessions is receiving requests from three different users. For each user request, the servlet must be able to figure out the session to which the user request pertains. Each user request belongs to just one of the three user sessions being tracked by the servlet. Currently, the product offers three ways to address the problem.

Cookies provide a fairly simple approach to tracking sessions. Because cookies do not work in all situations, URL rewriting provides an alternative. IBM WebSphere Application Server also provides a more secure mechanism for tracking sessions, through the use of a session ID that is derived from a unique identifier in SSL information. When deciding whether to use URL rewriting or SSL information, carefully review the coding requirements it imposes on applications that require session support.

Cookies

When session management is enabled and a client makes a request, the `HttpSession` object is created and the session ID is sent to the browser as a cookie. On subsequent requests, the browser sends the session ID back as a cookie and the Session Manager uses the cookie to find the `HttpSession` associated with the user.

URL rewriting

There are situations in which cookies will 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 longer than what this example shows.

To use URL rewriting, applications must follow certain coding guidelines. Also, special preparation is required. See the related information for details.

Secure Socket Layer (SSL)

For requests over SSL, SSL information is used to track session requests. When SSL session tracking is enabled for requests over SSL, SSL information is used as the session ID. SSL information takes precedence over cookies and URL rewriting. SSL session information cannot be shared between SSL and non-SSL requests.

4.4.1.1.1.1: Using cookies to track sessions

No special programming is required to track sessions with cookies. Follow the programming model and example described in [section 4.4.1.1](#).

4.4.1.1.2: Using URL rewriting to track sessions

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 `encodeURL()` or `encodeRedirectURL()` 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. For details, see the Related information.

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 that 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 such plain HTML files in your application (or Web application) and portions of the site that the user might access during the session, convert them 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 will be lost if one or more servlets in an application do not call `theencodeURL(String url)` or `encodeRedirectURL(String url)` methods.

4.4.1.1.1.3: Using SSL information to track sessions

No special programming is required to track sessions with SSL information. Follow the programming model and example described in [section 4.4.1.1](#).

To use SSL information, turn on **Enable SSL Tracking** in the Session Manager property sheet. Because the SSL session ID is negotiated between the Web browser and HTTP server, it cannot survive an HTTP server failure. However, the failure of an application server does not affect the SSL session ID. (Of course, if session persistence is not configured, the session itself is lost.) In environments that use WebSphere Edge Server with multiple HTTP servers, an affinity mechanism must be used when the SSL session ID is to be used as the session tracking mechanism.

SSL tracking is supported only for the IBM HTTP Server and iPlanet Web servers. The lifetime of an SSL session ID can be controlled by configuration options in the Web server. For example, in the IBM HTTP Server, the configuration variable SSLV3TIMEOUT must be set to allow for an adequate lifetime for the SSL session ID. Too short an interval could result in 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 might not leave the SSL session ID active long enough to be useful as a mechanism for session tracking.

4.4.1.1.2: Controlling write operations to persistent store

You can manually control when modified session data can be persisted to the datastore by using the `sync()` method in the interface `com.ibm.websphere.servlet.session.IBMSession`, which extends the `javax.servlet.http.HttpSession` interface.

By calling `sync()` from the `service()` method of a servlet, you send any changes in the session to the database.

If neither the manual update nor the time-based write option is enabled, the `sync()` call performs no updates. It merely returns.

Ideally, call `sync()` after all updates have been made to the session and the session will not be accessed any more. In other words, wait until the end of the servlet `service()` method to call `sync()`.

4.4.1.1.3: Securing sessions

HTTP sessions and security can be integrated in IBM WebSphere Application Server. 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. Security integration in Session Manager is not supported in form-based log-in unless LPTA is used.

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

IBM WebSphere Application Server 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." IBM WebSphere Application Server includes the `com.ibm.websphere.servlet.session.UnauthorizedSessionRequestException` interface, which is used when a session is requested without the necessary credentials.

The Session Manager uses the WebSphere security infrastructure to determine the authenticated identity associated with a client HTTP request that either retrieves or creates a session. WebSphere security determines identity using certificates, LPTA, 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 whose outcomes depend on whether the HTTP request was authenticated and whether a valid session ID and user name was passed to the Session Manager.

	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. The user name is "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. TheSession 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. UnauthorizedSessionRequest Exception 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*

* com.ibm.websphere.servlet.session.UnauthorizedSessionRequestException is thrown to the servlet.

4.4.1.1.4: Deciding between single-row and multirow schema for sessions

Using the single-row schema, each user session maps to a single database row. Using the multirow schema, each user session maps to multiple database rows. (In a multirow schema, each session attribute maps to a database row.)

In addition to allowing larger session records, using multirow schema can yield performance benefits, as discussed in [article 4.4.1.1.7.3](#). However, it requires a little work to switch from single-row to multirow schema, as shown in the instructions below.

Switching from single-row to multirow schema

To switch from single-row to multirow schema for sessions:

1. Modify the Session Manager properties to switch from single to multirow schema.
2. Manually drop the database table or delete all the rows in the database table that the product uses to maintain HttpSession objects.

To drop the table:

1. Determine which data source configuration the Session Manager is using.
 2. In the data source configuration, look up the database name.
 3. Use the database facilities to connect to the database.
 4. Drop the SESSIONS table.
3. Restart the Session Manager.

Coding considerations and test environment

Consider configuring direct single-row usage to one database and multirow usage to another database while you verify which option suits your application's specific needs. (Do this in code by switching the data source used; then monitor performance.)

Programming issue	Application scenario
Reasons to use single-row	<ul style="list-style-type: none">● You can read or write all values with just one record read/write.● This takes up less space in a database, because you are guaranteed that each session is only one record long.
Reasons not to use single-row	2-megabyte limit of stored data per session.
Reasons to use multirow	<ul style="list-style-type: none">● The application can store an unlimited amount of data; that is, you are limited only by the size of the database and a 2-megabyte-per-record limit.● The application can read individual fields instead of the whole record. When large amounts of data are stored in the session but only small amounts are specifically accessed during a given servlet's processing of an HTTP request, multirow sessions can improve performance by avoiding unneeded Java object serialization.
Reasons not to use multirow	If data is small in size, you probably do not want the extra overhead of multiple row reads when everything could be stored in one row.

In the case of multirow usage, design your application data objects not to have references to each other, to

prevent circular references. For example, suppose you are storing two objects A and B in the session using `HttpSession.put(..)` , and A contains a reference to B. In the multirow case, because objects are stored in different rows of the database, when objects A and B are retrieved later, the object graph between A and B is different than stored. A and B behave as independent objects.

4.4.1.1.7: Tuning session support

IBM WebSphere Application Server session support has features for tuning session performance and operating characteristics, particularly when sessions are persisted in a database. These options allow the administrator flexibility in determining the performance and failover characteristics for their environment.

The table summarizes the features, including whether they apply to sessions tracked in memory, in a database, or either. Click a feature for details about the feature. Some features are easily manipulated using administrative settings; others require code or database changes.

Feature or option	Goal	Applies to sessions in memory or database?
Write frequency	Minimize database write operations.	Database
Multirow schema	Fully utilize database capacities.	Database
Base in-memory session pool size	Fully utilize system capacity without overburdening system.	Either
Write contents	Allow flexibility in determining what session data to write	Database
Scheduled invalidation	Minimize contention between session requests and invalidation of sessions by Session Manager. Minimize write operations to database for updates to last access time only.	Database
Tablespace and row size	Increase efficiency of write operations to database.	Database

4.4.1.1.7.1: Tuning session support: Session persistence

IBM WebSphere Application Server avoids using the database to read in or access the session when it is determined that the entry in the session cache is still the most recently updated copy. To tune the cache, set the [base in-memory session pool size](#) and allow overflow.

In addition to the cache table itself, the product maintains a list of the most recently used sessions in memory, ordered from least to most recently used. Whenever a session is accessed, it is added to the most-recently-used end of the list. When the cache table becomes full and a session that is not in the cache is accessed, the least recently used session is removed from the cache (but not from the database; the session is still valid until explicitly invalidated or timed out) to make room for the new entry.

This removal occurs whether or not overflow is enabled. However, under heavy-concurrent-access scenarios, multiple new sessions might compete for the space vacated by the single, least recently used entry.

- When overflow is disabled, only one new session is placed in the cache; the others must be reread from the database. To optimize performance, the product does not retry to add the next new session by removing the next least recently used entry.
- When overflow is enabled, one new session is added to the base table, and the rest reside in memory in the overflow table. Analysis and customer experience show that the size of this table remains relatively small compared to the base in-memory session pool size.

It is also important to establish session affinity so that the caching can be most effective. See the [Related information](#) for details.

4.4.1.1.7.3: Tuning session support: Multirow schema

By default, a single session maps to a single row in the database table used to hold sessions. With this setup, there are **hard limits** to the amount of user-defined, application-specific data that WebSphere Application Server can access.

IBM WebSphere Application Server supports the use of a multirow schema option in which each piece of application specific data is stored in a separate row of the database. With this setup, the total amount that can be placed in a session is now bound only by the database capacities. The only practical limit that remains is the size of a session attribute object itself.

The multirow schema potentially has performance benefits in certain usage scenarios, such as when larger amounts of data are stored in the session but only small amounts are specifically accessed during a given servlet's processing of a http request. In such a scenario, avoiding unneeded Java object serialization is beneficial to performance.

It should be stressed that switching between multirow and single row is not a trivial proposition. See the Related information for details.

4.4.1.1.7.4: Tuning session support: Write frequency

In the Session Manager, you can configure the frequency for writing session data to the database. This flexibility enables you to weigh session performance gains against varying degrees of failover support. The following options are available in Session Manager for tuning write frequency:

- End of service method (the default) - Write session data at the end of the servlet's `service()` method call.
- [Manual update](#) - Write session data when the servlet calls the `IBMSession.sync()` method.
- [Time-based write](#) - Write session data every so many seconds (called the *write interval*).

When a session is first created, session information is always written to the database at the end of the `service()` call.

End of service method

By default, IBM WebSphere Application Server updates the database with any changes made to the session during the servlet processing of an HTTP request (for example, during the execution of the `service()` method). These updates minimally include the last access time of the session and typically also include changes affected by the servlet, such as updating or removing application data. Exactly how much is written back can be configured with the [write contents](#) option.

Manual update

With manual updates, the servlet using a session determines when to write session information to the database. Switching to manual updates improves performance when the number of times an HTTP request's processing leads to changing a session (typically its application data) is typically less than the number of times the session is accessed or read in.

When manual update is set, the product session support no longer automatically updates the database at the end of a servlet's `service()` method. (However, when an `HttpSession` object is first created, session information is written to the database as part of postprocessing for the servlet request in which the session was created.) The last update times are cached and updated asynchronously prior to checks for session invalidation.

For any permanent changes to the session as part of servlet processing, the servlet code must specifically call the `sync()` method of the `com.ibm.websphere.servlet.session.IBMSession` interface.

Programming issue	Application scenario
Reasons to use manual update	<ul style="list-style-type: none">● You want direct control over when session information is persisted to the database.● The servlets of the application typically read in the session data but do not write it back as much.
Reasons not to use manual update	<ul style="list-style-type: none">● You do not want to control persistence of session information by using the <code>IBMSession</code> object, or you prefer that WebSphere explicitly control persistence to the database.● The servlets of the application are writing session information frequently.● Your code must comply completely with the Servlet 2.2 specification. The <code>sync()</code> method is not part of the Servlet specification; it is an IBM extension.

Time-based write

With time-based write, session data is written back to the database every time the write interval expires. Expiration of the write interval does not force a write operation; data is written only if the session has been retrieved or modified.

For example, suppose that a web application updates a session object every five seconds.

- If the Write Frequency option is set to **End of service method**, session information gets written every five seconds.
- If the Write Frequency option is set to **Manual update**, session information gets written whenever the application code calls `IBMSession.sync()`.
- If the Write Frequency option is set to **Time based write** and the write interval is set to 120 seconds, session information gets written no more frequently than every 120 seconds.

Using the time based write setting requires that the session invalidation time be at least twice as large as the write interval. This is needed to ensure that a session is not prematurely invalidated.

4.4.1.1.7.5: Tuning session support: Base in-memory session pool size

The base in-memory session pool size 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.
- When sessions are being stored in a database, it also specifies the cache size and the number of last access time updates that are saved in manual update mode.

For persistent sessions, when the session cache has reached its maximum size and a new session is requested, Session Manager removes the least recently used session from the cache to make room for the new one.

General memory requirements for the hardware system, and the usage characteristics of the e-business site, will determine the optimum value.

Note that increasing the base in-memory session pools size can necessitate increasing the heap sizes of the Java processes for the corresponding WebSphere application servers.

Overflow in non-persistent sessions

By default, the number of sessions maintained in memory is specified by Base in-memory session pool size. If you do not wish to place a limit on the number of sessions maintained in memory and allow overflow, set *overflow* to *true*.

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 in any fashion.

With the WebSphere extension to `HttpSession`, `com.ibm.websphere.servlet.session.IBMSession`, an `isOverflow()` method returns *true* if the session is such an invalid session. An application could check this and react accordingly.

4.4.1.1.7.6: Tuning session support: Write contents

In Session Manager, you can configure which session data is written to the database. The following options are available in Session Manager for tuning what is to be written back to the database:

- Write changed (the default) - Write only session data properties that have been updated through `setAttribute()` and `removeAttribute()` method calls.
- Write all - Write all session data properties.

The **Write all** setting might benefit servlet and JSP writers who change Java objects that reside as attributes in an `HttpSession` instance. Previously, every programmatic change in an attribute would require a `setAttribute()` call to make sure changes were reflected in the database in a timely manner. In most cases, the use of this setting eliminates the need for all but the `initialsetAttribute()` call to bind the object to the session.

However, the use of **Write all** could result in more being written back to the database than is necessary. If this situation applies to you, consider combining the use of **Write all** with **Time-based write** to boost performance overall. As always, be sure to evaluate the advantages and disadvantages for your installation.

With either Write Contents setting, when a session is first created, complete session information is written to the database, including all of the objects bound to the session. In subsequent session requests, what is written to the database depends on whether a single-row or **multirow** schema has been set for the session database, as follows:

Write Contents setting	Behavior with single-row schema	Behavior with multirow schema
Write changed	If any session attribute is updated, all objects bound to the session are written.	Only the session data modified through <code>setAttribute()</code> or <code>removeAttribute()</code> calls is written.
Write all	All bound session attributes are written.	All session attributes that currently reside in the cache are written. If the session has never left the cache, all session attributes are written.

4.4.1.1.7.7: Tuning session support: Scheduled invalidation

You can set specific times for the Session Manager to scan for invalidated sessions. When used with persistent sessions, this feature has the following benefits:

- The scan for invalidated sessions can be scheduled for times of low application server activity, avoiding database contention between invalidation scans and read/write operations to service HTTP session requests.
- There may be significantly fewer database write operations when running with the End of Service Method" write mode, because the session's last access time need not be written out after each HTTP session request. (Manual Update and Time Based Write options already minimize the writing of the last access time.)

Usage considerations

- With scheduled invalidation configured, HttpSession time-outs 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 HttpSession.invalidate() is explicitly called.
- The HttpSession.invalidate() method immediately invalidates the session from both the session cache and the database.

4.4.1.1.7.8: Tuning session support: Tablespace and page sizes for DB2 session databases

If you are using DB2 for session persistence, you can increase the page size to optimize performance for the writing of large amounts of data to the database. In versions earlier than 4.0, IBM WebSphere Application Server supported only a 4K page size. Page sizes of 8K, 16K, or 32K are supported in Version 4.0.

To use a page size other than the default (4K), do the following:

- If the SESSIONS table already exists, drop it from the DB2 database.
- Create a new DB2 buffer pool and tablespace, specifying the same page size (8K, 16K or 32K) for both, and assign the new buffer pool to this tablespace. A simple example follows:

```
DB2 Connect to sessionDB2 CREATE BUFFERPOOL sessionBP SIZE 1000 PAGESIZE 8K DB2 Connect reset DB2
Connect to sessionDB2 CREATE TABLESPACE sessionTS PAGESIZE 8K MANAGED BY SYSTEM USING
('D:\DB2\NODE0000\SQL00005\sessionTS.0') BUFFERPOOL sessionBP DB2 Connect reset
```

Refer to DB2 product documentation for details.

- Configure the correct tablespace name and page size in the Session Manager. (Page size is referred to as *row size* in the Session Manager console.)

When the product is restarted, the Session Manager creates a new SESSIONS table in the specified tablespace based on the page size specified.

4.4.1.1.8: Best practices for session programming

When developing new objects to be stored in the HTTP session, make sure to implement the Serializable interface. This enables the object to properly persist session information to the database. An example of this is:

```
public class MyObject implements java.io.Serializable {...}
```

Without this extension, the object will not persist correctly and will throw an error.

When adding Java objects to a session, make sure they are in the correct class path. If Java objects will be added to a session, be sure to place the class files for those objects in the application server class path or in the web application path. In the case of session clustering, this applies to every node in the cluster. Because the HttpSession object is shared among servlets that the user might access, consider adopting a site-wide naming convention to avoid conflicts.

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 to process all of it each time.

Release HttpSession objects when you are finished. HttpSession objects live inside the Web container until:

- 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). When session persistence is used, the application server can maintain only a certain number of HttpSession objects in memory. When this limit is reached, the application server removes the least recently used session entries from the cache to make a room for new ones. If Allow Overflow is enabled, the product also uses an overflow memory table to cache the entries when there is a racing condition for an entry in the cache. The product makes its best effort to keep the cache at base memory size.

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 `req.getSession()`), and a copy of it 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.

You can improve performance by not breaking session affinity. Some suggestions to help avoid breaking session affinity are:

- When using multiframe JSPs, create the session for the frame page but do not create sessions for the pages within the frame. (See discussion later in this topic.)

When applying security to servlets or JSPs that use sessions with security integration enabled, secure all of the pages (not just some). When it comes to security and sessions, it's all or nothing. It does not make sense to protect access to session state only part of the time. When security integration is enabled in Session Manager, all resources from which a session is created or accessed must be either secured or unsecured. You cannot mix secured and unsecured resources.

The problem with securing only a couple of pages is that sessions created in secured pages are created under the identity of the authenticated user. They can be accessed in other secured pages only by the same user. To protect these sessions from use by unauthorized users, they cannot be accessed from an unsecured page. When a request from an unsecured page occurs, access is denied and an `UnauthorizedSessionRequestException` is thrown. (`UnauthorizedSessionRequestException` is a run-time exception; it is logged for you.)

Use manual update and either sync() or time-based write in applications that mostly read session data but update infrequently. When an application is using a session, the `LastAccess` time field is updated any time data is read from or written to that session. If persistent sessions are being used, this produces a new write to the database. This performance hit can be avoided by using manual update and having the record written back to the database only when data values are updated, not on every read or write of the record. To use manual update, you first need to turn it on in the Session Manager. In addition, the application code must use `com.ibm.websphere.servlet.session.IBMSession` instead of the generic `HttpSession` class. Within `IBMSession`, the `sync()` method tells the application server that the data in the session object should be written out to the database. This enables the developer to improve overall performance by having the session information persist only when necessary.

Although manual update gives you the most precise control for sending updates, the use of time-based write and scheduled invalidation options can also help the case in which access is frequent but updating is not.

When using multiframe Java Server Pages (JSP), create the session for the frame page (JSP) but do not create it for the pages (JSPs) within the frame. By default, JSPs create `HttpSession` objects by calling the

`request.getSession(true)` method. By doing this, each page in the browser is requesting a new session, but only one session is used per browser instance. You can use

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

to turn off the automatic session creation. Then if the page needs to access session information, use

```
<% HttpSession session = javax.servlet.http.HttpServletRequest.getSession(false); %>
```

to get the already existing session that was created by the frame JSP. This enables you to not break session affinity on the initial loading of the frame pages.

Implement the following suggestions to achieve high performance:

- Use IBM WebSphere Edge Server, taking advantage of its affinity options.
- If your applications do not change the session data frequently, use manual update and the `sync()` function to efficiently persist session information. As an alternative, consider using the time-based write option.
- Keep the amount of data stored in the session as small as possible. With the ease of using sessions to hold data, sometimes too much data is stored in the session objects. A proper balance of data storage and performance must be determined to effectively use sessions.
- Use a dedicated database for the session database. Do not use the WebSphere repository database or another application's database. This helps to avoid contention for JDBC connections and enables better database performance.

For more information, see the following IBM documents on the Web:

- "WebSphere Application Server: Best Practices using HTTP Sessions," by David Draegar and Jay Toogood. This article is available from the DeveloperWorks site.
- "WebSphere Application Server Development Best Practices for Performance and Scalability," by Harvey W. Gunther. This IBM white paper is available from the Library section of the WebSphere Application Server product site.

4.4.2: Keeping user profiles

IBM WebSphere Application Server provides a service for processing user profiles, called the *User Profile Manager*.

The key activities for implementing user profiles are summarized. For more information about each point, consult the Related information below.

1. Customize the user profile support as necessary. Options include:
 - Using the data representation class with exactly the name/value pairs it currently allows (no action required)
 - Extending the data representation class to allow additional, arbitrary name/value pairs
 - Adding columns to the base user profile representation

Basically, you need to evaluate whether the user profile representation provided by IBM represents the kind of data you would like to keep about your users. You might find it desirable to customize the IBM user profile support in one or more of the above ways.

2. Create or modify servlets to use the User Profile Manager and related user profile support classes to maintain user profiles on behalf of Web applications.
3. Ensure the administrator appropriately configures User Profile Managers in the administrative domain.

If the programmer and administrator are not the same person, the programmer might need to provide settings information to the administrator, based on how the programmer implemented user profiles.

4.4.2.1: Data represented in the base user profile

WebSphere Application Server provides a base implementation for data representation in user profiles through the interface `com.ibm.websphere.userprofile.UserProfile`.

The interface includes these columns corresponding to fields for demographic data on individual users:

- Address (first line)
- Address (second line)
- First Name
- Surname
- Day phone number
- Night phone number
- City
- Nation
- Employer
- Fax number
- Language
- Email address
- State/Province
- Postal code

4.4.2.2: Customizing the base user profile support

The application developer has a few options for customizing the user profile support provided by IBM WebSphere Application Server. The Related information provides instructions and additional details about each option.

Extend the data represented in user profiles

As discussed in [section 4.4.2.1](#), the base implementation allows Web applications to maintain several pieces of data about users. The data representation can be extended to allow the collection of arbitrary name/value pairs.

Adding columns to the base user profile implementation

Application developers can customize user profiles by adding columns to the base user profile implementation. Adding new columns is accomplished by implementing the interface:

```
com.ibm.websphere.userprofile.UserProfileExtender
```

and extending the base class:

```
com.ibm.servlet.personalization.userprofile.UserProfile
```

4.4.2.2.1: Extending data represented in user profiles

Use following interface with `com.ibm.websphere.userprofile.UserProfileExtender` to extend a user profile hash table:

```
com.ibm.websphere.userprofile.UserProfileProperties
```

This enables you to place arbitrary name/value pairs in the user profile. Extending the hash table is similar to using the `java.util.Dictionary` class in the base JDK 1.x or any of the classes that extend it.

4.4.2.2.2: Adding columns to the base user profile implementation

The base implementation of the user profile is contained in the class:

```
com.ibm.servlet.personalization.userprofile.UserProfile
```

It contains the columns discussed in [section 4.4.2.1](#). The application developer can add columns to the base implementation, but cannot delete columns from it.

Adding columns is a two-step process, as follows:

1. Extend the UserProfile class.
2. Modify your existing servlets to use the new columns.

Several examples are available to demonstrate how to extend the base user profile implementation and utilize the extension with a servlet.

Example	Description
UPServletExample.java	Demonstrates how a servlet opens a user profile and prints the fields contained within it
UserProfileExtendedSample.java	Shows how to extend the UserProfile class to add a column to the user profile for a cellular phone number. The WebSphere administrator needs to configure the User Profile Manager to point to the extended class.
UPServletExampleExtended.java	Shows how to modify the UPServletExample servlet to include the cellular phone number in the output
UserProfileExtended.java	Shows how to extend a hash table to place arbitrary name/value pairs into the user profile
UPServletExtended.java	Shows how to extend the servlet. When any of the newly added columns are removed or replaced, look for the table named "USERPROFILE" in the database to which the user profile is configured and drop that table.

The examples are encoded in HTML for viewing in a browser. The documentation directory also contains non-HTML versions (.java files) that are ready for use.

4.4.2.2.3: Extending the User Profile enterprise bean and importing legacy databases

IBM WebSphere Application Server implements user profile support as an entity bean. Although you can extend the base user profile by adding columns to it, you can also extend the user profile enterprise bean itself as an alternative customization approach.

Extending the User Profile enterprise bean is more involved than extending the base user profile. You should be familiar with the concepts that underlie enterprise beans(Java components written to the EJB specification). For detailed information regarding enterprise beans and their implementation in IBM WebSphere Application Server, see [section 4.3](#).

The table below summarizes the procedure for extending the user profile enterprise bean and importing data from an enterprise (legacy) database, with links to example code.

Importing data requires an enterprise bean that maps to the legacy database. For simplicity, the examples assume that the primary keys of the two enterprise beans are identical, although there is no such requirement. The primary key of the legacy enterprise bean does not have to match the primary key of the user profile enterprise bean (the userName column).

Extending the user profile bean to set and get user cell phone data

Task	Examples
<p>1. Start with the base user profile bean and its remote interface.</p> <p>These contain the methods for setting and getting the base user profile fields</p>	<ul style="list-style-type: none">● User profile bean● Remote interface
<p>2. Define a home interface and finder helper for the UPBaseChild bean. Specify the create and finder methods for the bean.</p> <p>Because inheritance between home interfaces is not supported in WebSphere Application Server, you will need to define all of the methods found in the UPBaseHome interface in order to use the managerial functions of the User Profile Manager.</p> <p>You can also add more methods as necessary to further customize the User Profile.</p>	<ul style="list-style-type: none">● For the home interface● For the bean

<p>3. Create read-only and read-write extensions of the UPBaseChild bean.</p> <p>In the base implementation of the User Profile beans provided, read-only and read-write beans are simple extensions of the UPBase bean. You would similarly extend your UPBaseChild bean with read-only and read-write extensions.</p>	<p>Read-only bean extension classes:</p> <ul style="list-style-type: none"> ● Bean ● Remote interface ● Home interface ● Finder class <p>Read-write bean extension classes:</p> <ul style="list-style-type: none"> ● Bean ● Remote interface ● Home interface ● Finder class
<p>4. Define a deployment descriptor for your beans</p> <p>Using the Application Assembly Tool or IBM VisualAge for Java, map both beans to the same table (that is, the same user profile table) and corresponding columns. Then define the deployment descriptor.</p> <p>In the base read-only implementation provided, all of the business methods are marked as read-only, so updating the fields using setter methods on the read-only bean does not update the persistent store.</p>	<p>Mark all the business methods in this example as constant (read-only):</p> <ul style="list-style-type: none"> ● Read only <p>Assuming you are not updating any instance variables in getter methods, you may also mark all your getter methods this example as constant to improve performance:</p> <ul style="list-style-type: none"> ● Read write <p>Mark some or all of the fields (except the remote interface to the legacy bean) as container managed fields. Define other properties to your beans depending on your requirements.</p>
<p>5. Extend the User Profile data wrapper to include the new methods</p> <p>The example shows how to extend the base User Profile data wrapper class to include the methods for setting and getting cell phone information in the User Profile</p>	<p>EJB extension</p>
<p>6. Use a WebSphere Application Server administrative client to configure user profile support. The administrative configuration includes:</p> <ul style="list-style-type: none"> ● The data wrapper class name ● JNDI lookup names ● Home and remote interfaces for the read-only and read-write beans 	

4.4.2.3: Accessing user profiles from a servlet

Servlets and other application building blocks requiring user profile support should make calls to the class:
`com.ibm.websphere.userprofile.UserProfileManager`

The class supports the following functions:

- Creating and deleting user profiles
- Getting and updating (cached and immediate) to and from the database
- Getting user profiles for read-only tasks
- Performing queries on database columns

4.5: Dynamic fragment cache

A WebSphere Application Server performance enhancement is the ability to cache the output of dynamic servlets and JSP files, a technology that improves application performance. This technology, working within an application server's Java Virtual Machine (JVM), intercepts calls to a servlet's service method, and checks whether the invocation can be served from a cache. Because J2EE applications have such high read-write ratios and can tolerate a small degree of latency in the freshness of their data, fragment caching creates an opportunity for significant gains in server response time, throughput, and scalability.

After a servlet is invoked once (generating the output that will be cached), a cache entry is created containing not only the output, but also side effects of the invocation as, for example, calls to other servlets or JSP files, as well as meta data about the entry including timeout and entry priority information.

Unique entries are distinguished by an id string generated from the `HttpServletRequest` object for each invocation of the servlet. Servlet caching can then be based on:

- Request parameters and attributes
- the URI used to invoke the servlet
- Session information
- Other options, including cookies

Since JSP files are compiled by WebSphere Application Server into servlets, the dynamic cache function treats them the same, except in specifically documented situations.

Summary of dynamic fragment caching articles

The dynamic fragment caching documentation describes how to configure dynamic caching in WebSphere Application Server. For both global configuration and the definition of individual cache policies, users can configure dynamic fragment caching through XML files installed on the server, or with graphical user interface (GUI) tools such as the administrative console or the Application Assembly Tool. The XML files are the preferred method for configuring dynamic caching because, in some cases, they are easier to implement and include more function than the GUI configuration.

The dynamic fragment caching articles also discuss advanced features of the cache, such as how to control external caches and how to build user-defined drop in components to customize the cache operation.

The dynamic fragment caching articles are:

- [6.6.0.16: Dynamic fragment cache configuration overview](#)
- [6.6.0.16.1: Global configuration](#)
- [6.6.0.16.2: Policy configuration](#)
- [6.6.0.16.4: Dynamic fragment cache XML examples](#)
- [6.6.0.16.4: Dynamic fragment cache monitor](#)
- [4.5.1: Custom ID and MetaData generators](#)
- [4.5.2: External Cache Adapter building](#)
- [4.5.3: Dynamic fragment cache frequently asked questions](#)

4.5.0: Getting started with Dynamic fragment cache

This article provides you with a quick list of tasks that you must complete to enable dynamic fragment caching with XML. Click any of the links for more information on a specific topic.

1. Create a global configuration file

In the `<product_installation>/properties` directory, locate the `dynacache.sample.xml` file. Copy the `dynacache.sample.xml` file to `dynacache.xml`.

The sample is a valid cache configuration file that will enable caching with default values. With caching enabled, the application server will look for a servlet configuration file in the `/properties` directory that declares which servlets should be cached. The options selected are discussed in detail in article, [Global configuration](#).

2. Create a servlet configuration file

In the `<product_installation>/properties` directory, locate the `servletcache.sample.xml` file. Copy the `servletcache.sample.xml` to `servletcache.xml`.


This sample file configures WebSphere's default server's snoop servlet for caching. Article [Policy configuration](#) describes the different ways for defining how to cache individual servlets or JSP files.

3. Reinitialize the Application Server

Stop and restart the WebSphere Application Server.

4. Verify the cacheable page.

Using a Web Browser, access URI: `/servlet/snoop` to view the snoop servlet in the default application. Invoking and reloading the URI several times should return the same output for the snoop servlet.

 The snoop servlet is now operating incorrectly, since it displays the request information from its first invocation rather than from the current request. You can inspect the entry in the cache with the [Servlet cache monitor](#)

4.5.1: Custom ID and MetaData generators

WebSphere Application Server's dynamic cache technology allows users to replace the standard configuration functions with their own custom configuration classes. The configuration duties managed by cache fall into two categories:

- Generating cache and group ids
- Defining meta data such as timeout, priority, and external caching

Application developers can supply classes to handle either or both of these sets of responsibilities, by implementing `com.ibm.websphere.servlet.cache.IdGenerator` and `com.ibm.websphere.servlet.cache.MetaDataGenerator`.

Overriding the default `MetaDataGenerator` allows users to access configuration information from some other source. or makes timeout, priority, or external cache group a function of a variable rather than a constant. A new `IdGenerator` gives users the ability to determine cache entry ids and their group ids. Both classes can still use the cache policy attributes defined for a servlet (`<timeout>`, `<priority>`, `<request>`, and others) to relay data to their generators using the `com.ibm.websphere.servlet.cache.CacheConfig` class.

Each servlet class has individual `IdGenerator` and `MetaDataGenerator` objects associated with it. So if the same servlet is being executed by WebSphere Application Server in different threads, all threads will use the same pair of generator objects.

Several dynamic caching classes are not described in detail in this article. See the [com.ibm.websphere.servlet.cache package \(Javadoc\)](#) for more information on the classes and interfaces used by the cache function.

Custom Id generators

- Configuring the cache to use a custom `IdGenerator`:

To specify your `IdGenerator` in the XML file, use the `<idgenerator>` tag.

```
<servlet>
  <timeout seconds="0"/>
  <path uri="/servlet/CommandProcessor" />
  <idgenerator class="SampleIdGeneratorImpl" />
</servlet>
```

You can also use the Application Assembly Tool to define the `IdGenerator` class in the cache policy's Advanced tab.

- Building a custom `IdGenerator`:

Your `IdGenerator` must implement the `com.ibm.websphere.servlet.cache.IdGenerator` interface. There are three methods in the `IdGenerator` interface:

1. `public void initialize(CacheConfig cc);`
2. `public String getId(ServletCacheRequest request);`
3. `public int getSharingPolicy(ServletCacheRequest request);`

The `getSharingPolicy` method should return `EntryInfo.NOT_SHARED`

The `initialize` method is called during startup. Normally, the cache processes a servlet's XML configuration and builds a `CacheConfig` object that is made available to the `IdGenerator`. The `initialize` method then builds a list of request and session variables that must be included in the cache ids for the

servlet. Since the "plugged-in" IdGenerator is created with a specific servlet's behavior in mind, working with the CacheConfig is unnecessary; just hard code the configuration into the `getId` method.

The `getId` method returns the unique String cache id when the servlet is invoked. If the servlet is cached, the `getId` method returns null. Typically, an Id will incorporate the following:

- The URI of the servlet
- The character encoding of the request (when the result is not null)
- The names and values of the input variables that determine the servlet's output

See the [coding example](#) for implementation details.

4.5.2: External caching

WebSphere Application Server's dynamic cache has the ability to control external caches on Web servers, such as IBM Edge Server and IBM HTTP Server. When external caching is enabled, the cache matches pages with their URIs and pushes matching pages to the external cache. The entries can then be served from the external cache instead of the application server. This creates a significant savings in performance.

Only certain fragments are eligible for external caching. Since the external cache must use the full URI as a cache id, the servlet being externally cached uses that URI as its internal cache id as well. Also, because the cache automatically uses the URI to build cache ids, it is illegal to define cache variables (for example, request, session, and cookie variables) in an externally cacheable servlet.

Only full pages are pushed out to external caches, so only externally accessible servlets should be defined as externally cacheable. For example, if page1.jsp includes page2.jsp and page3.jsp, then only page1.jsp should be declared externally cacheable. If page3.jsp is invalidated, then the cache also invalidates the external entry for page1.jsp. Therefore the next request for page1.jsp is sent to WebSphere Application Server.

Servlet and JSP file content that is private, requires authentication, or uses SSL should not be cached externally. The authentication required for those servlet or JSP file fragments cannot be performed on the edge. A suitable timeout value should be specified if the content is likely to become stale.

Enabling external caching of servlets and JSP files with IBM Edge Server

Edge Server users should consult the Edge Server documentation for information on configuring external caching between WebSphere Application Server and Edge Server.

Enabling high speed caching of Servlets and JSP files with IBM HTTP Server for Windows NT/2000

IBM HTTP Server for Windows NT/2000 contains a high speed cache referred to as the *Fast Response Cache Accelerator*, or *Cache Accelerator*. WebSphere Application Server's fragment cache can use IBM HTTP Server as an external cache with appropriate configuration.

After installing WebSphere Application Server and IBM HTTP Server for Windows NT, you must do the following to enable the *Cache Accelerator*:

1. Configure caching on the Web server.

In the IBM HTTP Server `conf` directory, locate the `httpd.conf` configuration file, and add the following two lines at the end of the file:

```
LoadModule afpaplugin_module c:/WebSphere/AppServer/bin/afpaplugin.dll
AfpaPluginHost 127.0.0.1:9081
```

The first line loads the IBM HTTP Server plug-in that connects the *Cache Accelerator* to WebSphere's fragment cache. Ensure the `LoadModule` path points to your Websphere Application Server installation.

The second line defines the application server instances that should be connected to this module. When multiple instances of WebSphere Application Server are defined, repeat the second line for each application server. If the instances are on different machines from the Web server, use the instances' IP address instead of the localhost address defined in this example. In the case of multiple application servers on the same host, choose a different port number for each instance.

2. Configure an external cache group on the application server.

For each application server that uses the *Cache Accelerator*, define an external cache group named *afpa*. Add a member to that group with an adapter bean name of `com.ibm.servlet.dynacache.Afpa`. For the address, enter the assigned port number from the Web server's `httpd.conf` file. An example of this configuration is available in the `dyncache.sample.xml` file located at:

[product_installation/properties/dynacache.sample.xml](#)

See the global configuration section in this file for more information.

3. Configure a cache policy using external cache.

Now you can use the external cache attribute when building your cache policies. See article [Policy configuration](#) for information on building cache policies.

You can test your configuration by doing the following:

- a. Do a CD (change directory) to the [product_installation/properties](#) directory.
- b. Copy the `servletcache.sample.xml` file to `servletcache.xml`.
- c. Update the first entry with the following definitions to enable the external high speed cache:

```
<servlet>
    <timeout seconds="0" />
    <servletimpl class="SnoopServlet.class" />
    <externalcache id="afpa" />
</servlet>
```

You can test your changes using the SnoopServlet sample for external caching. The SnoopServlet.class is located in: [product_installation\installedApps\sampleApp.ear\default_app.war\WEB-INF\classes](#)

The first request for `http://yourhost/servlet/SnoopServlet` results in the response being loaded in the high speed cache. Subsequent requests are served directly from the cache, which significantly enhances performance.

Setting the "timeout seconds" to "0" means the cached entry remains permanently cached. Setting it to a positive non-zero value, as for example, 30, causes the high speed cache entry to be deleted after the specified number of seconds, in this case 30.


To enable JSP files for caching, add the following stanza to the `servletcache.xml` file:

```
<servlet>
    <timeout seconds="10" />
    <path uri="/very_simple.jsp" />
    <externalcache id="afpa" />
</servlet>
```

In this example, you cache the response for 10 seconds when the `http://yourhost/very_simple.jsp` file is requested. The file, `very_simple.jsp` is located in:

[product_installation\installedApps\sampleApp.ear\default_app.war](#)

After 10 seconds, the cache entry is deleted. The cache entry is updated when a new request occurs for the JSP file.

 The IBM HTTP Server *Fast Response Cache Accelerator* is available on both Windows NT/2000 and AIX. However, dynamic caching support is currently only available on Windows NT/2000.

Configuring the *Fast Response Cache Accelerator* cache size.

In the default IBM HTTP Server configuration, the maximum *Cache Accelerator* dynamic cache size is calculated as 1/8 of physical (pin-able) memory. On a machine with 384Meg of RAM, it allows a maximum of approximately 50Meg for the *Cache Accelerator* dynamic cache. When this limit is reached, the *Cache Accelerator* then deletes older entries to cache new ones.

The IBM HTTP Server directive, `AfpaDynaCacheMax`, can be used to fine tune the maximum allowed cache size. This directive must be placed in the global server configuration scope (along with the other default *Cache Accelerator* directives), and *Cache Accelerator* must be enabled.

Update the following directives in the `httpd.conf` file of IBM HTTP Server :

```
AfpaEnable
AfpaCache on
AfpaLogFile "c:/Program Files/IBM HTTP Server/logs/afpalog" V-ECLF
AfpaDynaCacheMax 10
```

The above settings enable the *Cache Accelerator* and limit the dynamic cache size to 10 Meg. If you use these directive to increase the cache size, do not make the cache so large that all physical memory is consumed. You can determine how much memory is available, when all the applications are running, by using the Windows Task Manager.

Assign no more than 50% of available physical memory to the dynamic cache. Specifying too large a cache not only decreases the performance of other applications, but also puts you at risk for completely running out of memory.

The default configuration does not include the `AfpDynaCacheMax` directive where the cache size is automatically calculated as 1/8 of physical memory.

4.5.3: Dynamic fragment cache frequently asked questions

View frequently asked questions related to:

- [General cache information](#)
 - [Cache architecture](#)
 - [Application design](#)
 - [Cache configuration](#)
-

General cache information

- **Question:** What is new in dynamic fragment cache WebSphere Application Server version 4.0?

Answer: Version 4.0 allows you to configure dynamic cache through the GUI. The administrative console offers parallel configuration options to the *dynacache.xml* file. Cache policies can be attached to .war files using the Application Assembly Tool. While XML adds some advanced cache policy options, the AAT makes cache policies portable, by installing them along with the application.

The Servlet Cache Monitor is also new in version 4.0. This Web application installs on an application server, and allows administrators to inspect the contents of the fragment cache.

Finally, the APIs for programmatically manipulating the cache have been enhanced. These APIs are located in the [com.ibm.websphere.servlet.cache package](#), and contain classes and methods to customize cache operations.

- **Question:** What are the security implications of using the fragment cache?

Answer: The cache does all the processing within the Web container after the security processing completes. Within the application server, there are no extra security problems to be considered.

When using an external cache, security risks change dramatically. Caches outside of WebSphere Application Server do not undergo security processing. It is important not to store sensitive data in an external cache.

[Return](#)

Cache architecture

- **Question:** What type of servlets and JSP files does the cache support most effectively, simple presentation JSP files? What about servlets that use enterprise java beans or JDBC? What about personalized fragments?

Answer: While caching a simple presentation JSP file gives moderate performance gains, caching servlets that request information from enterprise java beans or databases saves WebSphere Application Server processing power, and decreases load on the back end. You should cache fragments that pull information from outside WebSphere Application Server since the biggest performance gains come from caching servlets that obtain information from outside the application server.

Personalized information can be cached, though how effective it will be depends on the architecture of the application. See the [Dynamic fragment cache XML examples](#) for a discussion of personalization.

- **Question:** The dynamic cache caches the JSP and servlet output. Does this mean it caches the `HttpServletResponse` object?

Answer: Not quite. It caches the output of the servlet (what is written to the `response.getWriter` method, for example). It also caches side effects of the servlet's execution, like setting cookies and headers, including and forwarding other servlets, and setting content type and character encoding.

- **Question:** Is this an "in memory" cache, or an "on disk" cache? Does it use the java heap?

Answer: The cache resides completely inside the Java heap. This keeps it in memory, but allows it to use virtual memory if necessary.

- **Question:** Does the dynamic cache always need an external cache for caching?

Answer: The dynamic cache does not require an external cache to be present for caching. It will, however, extend the abilities of such static caches to include caching certain servlets and JSP files.

- **Question:** If using an external cache that is associated with the dynamic cache (for private user information), how is the security of the information at the external cache enforced?

Answer: Because private information almost always involves session information, requests for private user information are usually not externally cacheable. However, in the case of a user id being present in the URL, or of pages that should only be accessible over HTTPS and not HTTP, external caching can short circuit the authentication or encryption processes. For sensitive information, external caching is not the best choice unless specific measures are taken, such as encrypting that user id.

[Return](#)

Application design

- **Question:** My application has a control servlet that works as a dispatcher for 10 other services, and we only want to cache the output of one or two services from this servlet. Is this servlet cacheable?

Answer: Definitely. Cache policies can be built to "cache" or "not cache" a servlet depending on whether input variables are present or have a specific value. Therefore, you can "cache" the control servlet when it is performing some read only action, and "not cache" it when it is updating application information. In addition, if the cache policy rules are not specific enough for the caching you want to perform, WebSphere Application Server provides a pluggable interface for writing your own Id Generators. See the [Custom Generators](#) article for more information.

- **Question:** If I have a chain of forwards or includes, can I cache some of the fragments and not others? If one cached fragment in the chain is invalidated, do the rest of them get evicted from the cache, too?

Answer: Any of the servlets or JSP files in the chain can be cached. If some or all of them are cached, and one changes (invalidating only its cache entry) then the output of upstream, cached servlets will reflect this change. Cached servlets have 'place holders' in their output for the results of any includes or forwards that servlets might perform.

[Return](#)

Cache configuration

- **Question:** I have existing XML configuration files from WebSphere Application Server version 3.5.x that I would like to use. Are they still valid?

Answer: Yes. `Dynacache.xml` and `servletcache.xml` files from version 3.5.x will work with version 4.0

- **Question:** How can I tell whether the cache is enabled?

Answer: Check the standard output file for the application server that you configured. You will see a message indicating whether the cache is enabled or not. Alternatively, use the servlet cache monitor to inspect the fragment cache.

- **Question:** Is there a way to know whether a fragment is getting cached?

Answer: After executing the servlet or JSP file, you can check for the presence of an entry in the cache using the servlet cache monitor. See article [Dynamic fragment cache monitor](#) for more information on this tool.

- **Question:** How big should my cache be?

Answer: Cache sizes should typically stay in the 1-10K entries range. Ideally, the cache should be big enough to contain an entry for each invocation of the fragments to be cached. More likely you will make the cache big enough to hold the most expensive and most often served fragments, and use the LRU algorithm to eliminate less useful quotes. The capacity can only be measured in number of entries, not in memory size. Increase the memory available to WebSphere Application Server according to the size and number of cached responses.

- **Question:** Can I use cache policies defined in a `servletcache.xml` file and policies attached to an `.ear` or `.war` file on the same server?

Answer: Yes. In the case of a URI being defined in both, the XML file takes precedence.

- **Question:** I do not understand how the `<invalidateonly/>` tag works. Aren't the invalidate and exclude rules enough to invalidate a cache entry?

Answer: While the methods in the Cache class are sufficient for manipulating cache entries, users are not required to implement any java code to enable caching. The `<invalidate>` and `<invalidateonly>` tags extend the capabilities of rule based XML configuration to more completely cover the operations you can perform when using the API directly.

The `<invalidate>` element allows you to use a fragment invocation to invalidate other cache entries. The classic example is a "buy" servlet that adds objects to a shopping cart. Whenever you perform a buy, you would then invalidate the cache entry for that user's shopping cart.

The `<invalidateonly/>` element exists for efficiency. Some servlets may not be cacheable, but have invalidation side effects. With this tag, WebSphere Application Server performs the invalidations, but does not attempt to cache the servlet. (The caching attempt would have failed, anyway.)

[Return](#)

4.6: Java Technologies

The J2EE (Java TM 2 Platform Enterprise Edition) technologies provide standard architectures for defining and supporting a multi-tiered programming model.

The technologies support all application components, namely:

- Application clients
- Enterprise JavaBeans TM
- Servlets and JavaServer PagesTM
- Applets

The Java technologies are:

- [JavaMail](#)
- [Java Naming and Directory Interface \(JNDI\)](#)
- [Java Message Service \(JMS\)](#)

See the *Related information* links for information on other programming topics.

4.6.1: Using JavaMail

WebSphere Application Server supports JavaMail version 1.1.3 and the JavaBeans Activation Framework (JAF) version 1.0.1.

In WebSphere Application Server, JavaMail is supported in all Web application components, namely:

- servlets
- JSP files
- enterprise beans
- application clients

The JavaMail APIs model a mail system. These APIs provide a platform and protocol independent framework to build Java based, e-mail client applications. The JavaMail APIs only provide general mail facilities for reading and sending mail. These APIs require service providers to implement the protocols.

In addition to service providers, JavaMail requires the JavaBeans Activation Framework or JAF to handle mail content that is not plain text as, for example, MIME (Multipurpose Internet Mail Extensions), URL (Uniform Resource Locator) pages, and file attachments.

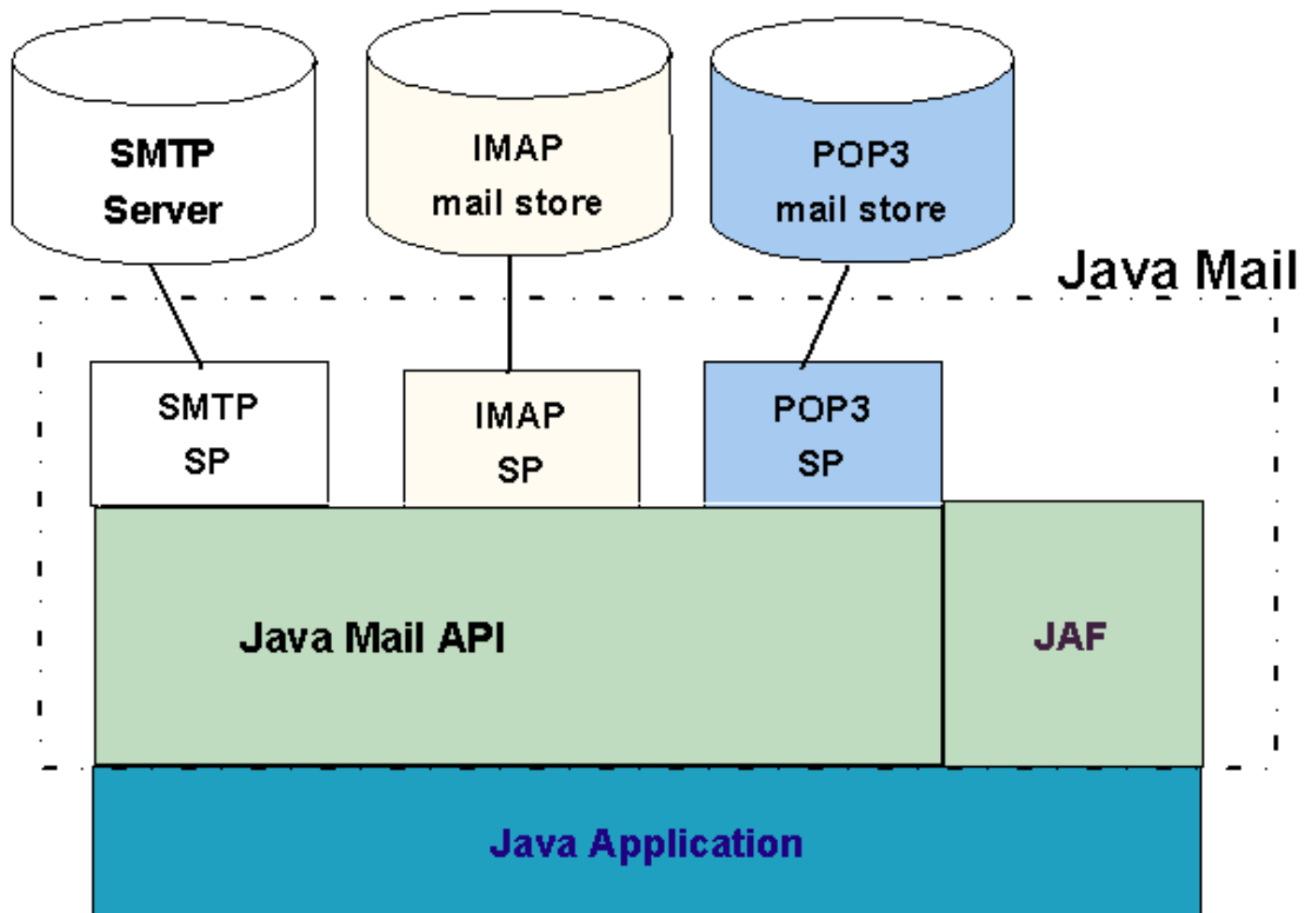
The service providers implement specific protocols. For example, SMTP (or Simple Mail Transfer Protocol), is a transport protocol for sending mail. POP3 or Post Office Protocol 3 is the standard protocol for receiving mail. IMAP or Internet Message Access Protocol is an alternative protocol to POP3.

The following graphic illustrates the relationship among the different JavaMail components:

- [JavaMail APIs](#)
- [JavaBeans Activation Framework](#)
- [Service providers](#)
- [Mail protocols](#)

The dotted line around specific objects represents the grouping that comprises a working JavaMail installation. With the exception of POP3, all the components in the installation view are shipped as part of WebSphere Application Server using the following Sun licensed packages:

- **mail.jar** - contains JavaMail APIs, the SMTP service provider, and the IMAP service provider.
- **activation.jar** - contains the JavaBeans Activation Framework.



4.6.1.1: Writing JavaMail applications

According to the J2EE specifications, each *javax.mail.Session* instance must be treated as a resource factory. Therefore, to use JavaMail, do the following:

1. Declare mail resource references in your application component's deployment descriptors, as described in this 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>
```

2. Configure, during deployment, each referenced mail resource. See article, [4.6.1.2: Configuring JavaMail](#), for a description of the parameters required to configure a mail resource.

3. Locate in your application component, during runtime, each specific JavaMail session using JNDI lookup. An example of the code follows:

```
Session session = (Session)ctx.lookup("java:comp/env/mail/MailSession");
```

Your application component can now use *session* to create messages and get store access.

Coding example for sending and saving a message

The following code segment shows how an application component sends a message and saves it to the mail account's *Sent* folder:

```
javax.naming.InitialContext ctx = new javax.naming.InitialContext();
mail_session = (javax.mail.Session) ctx.lookup("java:comp/env/mail/MailSession");

MimeMessage msg = new MimeMessage(mail_session);
msg.setRecipients(Message.RecipientType.TO, InternetAddress.parse("bob@coldmail.net"));

msg.setFrom(new InternetAddress("alice@mail.eedge.com"));
msg.setSubject("Important message from eEdge.com");
msg.setText(msg_text);
Transport.send(msg);


Store store = mail_session.getStore();
store.connect();
Folder f = store.getFolder("Sent");
if (!f.exists()) f.create(Folder.HOLDS_MESSAGES);
f.appendMessages(new Message[] {msg});
```

See the related information links for the JavaMail APIs.

4.6.1.2: Configuring JavaMail

A mail resource is configured using appropriate system management facilities, as for example, the thin client.


Refer to article [6.6.37: Administering mail providers and mail sessions](#) for detailed configuration instructions.

 The "mail originator" setting (the exact name in the administrative console varies depending on the product edition; see 6.6.37.0.1 for a description) can be overridden for individual messages in your application, using method [*Message.setFrom\(\)*](#)

See the related topics for more JavaMail documentation.

4.6.1.3: Debugging JavaMail

There will be times when you need to debug your JavaMail applications. One option is to turn on JavaMail's debugging feature. With this option on, JavaMail will print to *stdout* its interactions with the mail servers. These interactions are printed in detail, in a step-by-step format.

 With WebSphere Application Server, *stdout* and *stderr* are usually redirected to files. The specific file paths can be set with an application server's *Properties > File* panel. For example, for the Default Server, *stdout* is redirected by default to the file:

`<WAS_HOME>\logs\default_server_stdout.log`

Enable debugging [programmatically](#), or through the [command line](#).

- The easiest way to turn on debugging is to call method `setDebug()` on the mail session after *session* is obtained through a JNDI lookup, as shown below:


```
javax.naming.InitialContext ctx = new javax.naming.InitialContext();
mail_session = (javax.mail.Session) ctx.lookup("java:comp/env/mail/MailSession");
mail_session.setDebug(true);
...
```

This debugging approach requires re-compiling and, very likely, re-loading the application component in which this code is embedded. This approach may be impractical at times.

You can also edit the `startServerBasic.bat` file on Windows NT or `startServerBasic.sh` file on UNIX platforms and add the following flag to the line that starts the Java program:

`-Dmail.debug=true`

- If your JavaMail code is in a Java client which is invoked from the commandline, add the `-Dmail.debug=true` flag to the *java* command, and the debugging output will be displayed in the command window.

 Property *mail.debug*, set with the last two approaches, is shared by all mail session instances within the same JVM process. When debugging is enabled in this manner, JavaMail will print out step-by-step, mail-related interactions to *stdout* for all these mail sessions.

The output in *stdout* looks like the following example:

```
...

DEBUG: getProvider() returning javax.mail.Provider[TRANSPORT,smtp]
DEBUG SMTP: useEhlo true, useAuth false

DEBUG: SMTPTransport trying to connect to host "smtp3.eedge.com", port 25

DEBUG SMTP RCVD: 220 relay14.eedge.com ESMTP Sendmail; Tue, 19 Dec 2000 15:08:42 -0700

DEBUG: SMTPTransport connected to host "smtp3.eedge.com", port: 25

DEBUG SMTP SENT: EHLO y2001
DEBUG SMTP RCVD: 250-relay14.eedge.com Hello testpc.eedge.com, pleased to meet you
250-8BITMIME
250-SIZE 20000000
250-DSN
250-ONEX
250-ETRN
250-XUSR
250 HELP

DEBUG SMTP SENT: MAIL FROM:<alice@mail.eedge.com>
DEBUG SMTP RCVD: 250 <alice@mail.eedge.com>... Sender ok

DEBUG SMTP SENT: RCPT TO:<bob@coldmail.net>
DEBUG SMTP RCVD: 250 <bob@coldmail.net>... Recipient ok

Verified Addresses
bob@coldmail.net
```

DEBUG SMTP SENT: DATA
DEBUG SMTP RCVD: 354 Enter mail, end with "." on a line by itself

DEBUG SMTP SENT:

...

DEBUG SMTP RCVD: 250 PAA125654 Message accepted for delivery

DEBUG SMTP SENT: QUIT

4.6.1.4: Running the JavaMail sample

The JavaMail sample is packaged in an ear file called *jmsample.ear* that is located in directory:

[product_installation_root](#)/installableApps

The JavaMail sample contains three application components:

1. one stateful session bean
2. one servlet
3. one JSP file

Each component, when invoked, gathers data to compose a mail message, and then sends the message. Optionally, the sent message can also be saved in an IMAP account, in a folder named *Sent*. See article [Writing JavaMail applications](#) for the coding example to send a message and create the *Sent* folder.

Complete the following tasks to use *jmsample.ear*:

1. [Set up mail servers](#)
2. [Configure a MailSession resource](#)
3. [Install the *jmsample.ear* components](#)

1. Set up mail servers

To send out email messages, you need a mail transport server. Since SMTP is the most widely used transport protocol, such a mail server is also known as a SMTP server. An alternative to installing and configuring your own SMTP server, is to use an existing implementation. For example, if your Internet mail address is *john_smith@mycompany.com*, then *mycompany.com* could serve as your SMTP server. Ask your company's email administrator for more information.

The components in the *jmsample.ear* file can optionally save a copy of this message into an email account. If you plan to try this capability, you will also need to set up an IMAP mail account.

2. Configure a MailSession resource

See article [Properties related to JavaMail support](#) for information on MailSession resource properties.

One property that requires your attention when you configure the MailSession resource for *jmsample.ear* is *JNDI Name*. You can explicitly define this property or allow System Management to define it for you.

Since for all components in this sample the MailSession resource references have been pre-bound to the JNDI path `mail/DefaultMailSession`, enter this path in the *JNDI Name* property.

If, at this time, you do not define the JNDI Name as shown above, you will have to bind the application's mail resource references to this mail session when you install *jmsample.ear*.

Review the tutorial, [Create a JavaMail session](#), for detailed information on configuring the mail session resource.

3. Install the *jmsample.ear* components

Install the *jmsample.ear* file as an enterprise application. Use the *SEAppInstall* command.

Perform the following steps to install the *jmsample.ear* using the *SEAppInstall* command:

- A. Go to a command prompt.
- B. Go to the `product_installation_root\installableApps` directory, and locate the *jmsample.ear* file.
- C. Enter the following command to install the sample:
 - For Windows, enter:
`SEAppInstall -install jmsample.ear`
 - For UNIX platforms, enter:
`product_installation_root/bin/SEAppInstall.sh -install jmsample.ear`

The *SEAppInstall* command will display several messages indicating its progress. The command will also prompt you for input. In this example of the dialog, the questions asked by the command are in **bold**, and your responses are in *italics*.

```
Do you wish to deploy all of the EJBs in this application ([Y]es/[n]o)?
Yes
(A message should display indicating the EJBs were deployed successfully.)
Which type of database are you using (optional)?
0
What DB Schema name do you want to use for this application (optional)?
Press Enter
Do you wish to precompile all JSPs in this application ([Y]es/[n]o)?
Yes
(At the "Default Datasource JNDI Name (optional) []" prompt, press Enter to accept the default.)
(At the "JNDI Name [ejb/JMSamEJB]:" prompt, press Enter to accept the default.)
(At the "JNDI Name [mail/DefaultMailSession]:" prompt, press Enter to accept the default.)
(At the "JNDI Name [mail/DefaultMailSession]:" prompt, press Enter to accept the default.)
(At the "JavaMailSample WebApp (WebApp_1) [!]" prompt, enter default_host to specify the virtual host.)
(A message should display stating the application installed successfully.)
```

D. Stop and restart WebSphere Application Server with the following commands:

- Go to a command prompt
- On Windows, enter `stopServer`. On UNIX platforms, enter `product_installation_root/bin/stopServer.sh`
- On Windows, enter `startServer`. On UNIX platforms, enter `product_installation_root/bin/startServer.sh`

(Check the `product_installation_root/logs/default_server_stdout.log` file and make sure the "Server Default Server open for e-business" message is displayed.)

After the install, you can invoke the servlet or JSP by using one of the following URLs:

`http://localhost:9080/jmsample/servlet`

`http://localhost:9080/jmsample/Email.jsp`

To test the JavaMail servlet, do the following:

1. Open a browser window
2. Enter `http://localhost:9080/jmsample/servlet`
(The servlet GUI should display.)
3. Enter the following information, replacing variable input as appropriate:
 - **To:** Your e-mail address as for example, *anybody@mycompany.com*
 - **Cc:** Optionally enter another e-mail address here.
 - **From:** *somebody@mycompany.com*
 - **Subject:** *JavaMail Servlet Test*
 - **Message to send:** Any text as for example, *This is a test message that isbeing sent from the JavaMail Mail Servlet.*
 - Check the **Save the sent message into mail store** box.
 - **Send Option:** As is
 - Click the **Send** button.
(If the test number at the top of the page was incremented from **1** to **2**, the message was sent successfully.)
4. Log on to your e-mail account and verify that you received the mail

To test the JavaMail Java Server Page, do the following:

1. Open a browser window
2. Enter `http://localhost:9080/jmsample/Email.jsp`
(The Java Server Page GUI should display.)
3. Enter the following information, replacing variable input as appropriate:
 - **To:** Your e-mail address as for example, *anybody@mycompany.com*
 - **Cc:** Optionally enter another e-mail address here.
 - **From:** *somebody@mycompany.com*
 - **Subject:** *JavaMail JSP Test*
 - **Message to send:** Any text as for example, *This is a test message that isbeing sent from the JavaMail Mail JSP.*
 - Check the **Save the sent message into mail store** box.
 - **Send Option:** As is
 - Click the **Send** button.
(If the test number at the top of the page was incremented from **1** to **2**, the message was sent successfully.)
4. Log on to your e-mail account and verify that you received the mail

Use these instructions to invoke the EJB:

- A. Locate file *deplmtest.jar* in the *product_installation_root/InstalledApps/jmsample.ear* directory.
- B. Copy the *deplmtest.jar* file to the *product_installation_root/classes* directory.
 - The *deplmtest.jar* file contains all the artifacts for the EJB, including the proxies and a simple client.
- C. The client uses the system properties as data input for the message creation. The following example demonstrates how Java system properties are gathered for the EJB client on Windows NT:
 - The line breaks in this example were added to make the information more legible. This information really exists as one line of input.

Change the property values in this example to the ones you defined for your test.

For the Windows platform, specify the following:

```
product_installation_root\java\bin\java
-Djava.ext.dirs=product_installation_root\java\jre\lib\ext;product_installation_root\classes;product_installation_root\lib-Djava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory-Dmailtest.to=bob@mycompany.com
-Dmailtest.cc=alice@mycompany.com-Dmailtest.from=john@mycompany.com -Dmailtest.subj="Important message sent from an
EJB"-Dmailtest.message="As the subject line says, this is a very important message." -Dmailtest.save_msg=off
-Dmailtest.ejbhome=ejb/JMSampEJB mailtest.MailClient
```

For UNIX platforms, specify the following:

- Entering the following command as one, continuous line of input at a command prompt might not work on some UNIX platforms. Use the back slash to indicate the command continues on the next line, or invoke the command from a shell script.

```
product_installation_root/java/bin/java
-Djava.ext.dirs=product_installation_root/java/jre/lib/ext:product_installation_root/classes:product_installation_root/lib-Djava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory-Dmailtest.to=bob@mycompany.com
-Dmailtest.cc=alice@mycompany.com-Dmailtest.from=john@mycompany.com -Dmailtest.subj="Important message sent from an
EJB"-Dmailtest.message="As the subject line says, this is a very important message." -Dmailtest.save_msg=off
-Dmailtest.ejbhome=ejb/JMSampEJB mailtest.MailClient
```

See the related topics for links to Javadoc and the *Create a JavaMail session* tutorial.

4.6.2: JNDI (Java Naming and Directory Interface) overview

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 or JNDI provides a common interface that is used to access the various naming and directory services. See URL 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:

- [Enterprise JavaBeans \(EJB\)](#)
- [JavaMail](#)
- [Java Database Connection Service \(JDBC\)](#)
- [Java Message Service \(JMS\)](#)

4.6.2.1: JNDI implementation in WebSphere Application Server

IBM WebSphere Application Server 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 name server through the JNDI naming interface.

WebSphere Application Server does *not* provide implementations for:

- `javax.naming.directory` or
- `javax.naming.ldap` packages

Also, WebSphere Application Server does *not* support interfaces defined in the

`javax.naming.event` package.

However, to provide access to LDAP servers, the JDK shipped with WebSphere Application Server supports Sun's implementation of:

- `javax.naming.ldap` and
- `com.sun.jndi.ldap.LdapCtxFactory`

WebSphere Application Server's JNDI implementation is based on version 1.2 of the JNDI interface, and was tested with version 1.2.1 of Sun's JNDI SPI (Service Provider Interface).

The default behavior of this JNDI implementation should be adequate for most users. However, users with specific requirements can control certain aspects of the JNDI behavior. See the following section for information on modifying the JNDI behavior:

- **JNDI caching** - Description of the caching feature and properties, and the effects of the different properties on caching behavior.

4.6.2.2: Using JNDI

Refer to these examples to learn how to use JNDI.

- [Get an initial context](#)
 - [Get an initial context using JNDI properties found in the current environment](#)
 - [Get an initial context by explicitly setting JNDI properties](#)
 - [Look up a home for an EJB](#)
 - [Look up a JavaMail session](#)
-

Get an initial context

In general, JNDI clients should assume the correct environment is already configured so there is no need to explicitly set property values and pass them to the **InitialContext** constructor. However, a JNDI client may need to access a name space other than the one identified in its environment. In this event, it is necessary to explicitly set one or more properties used by the **InitialContext** constructor. Any property values passed in directly to the **InitialContext** constructor take precedence over settings of those same properties found elsewhere in the environment.

View the following examples for information on passing property values to the **InitialContext** constructor:

- 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 JNDI client's CLASSPATH. See article [Installing files and setting classpaths](#) for information on defining CLASSPATHs.

```
... import javax.naming.Context;    import javax.naming.InitialContext;    ... Context
initialContext = new InitialContext();    ...
```

- Get an initial context by explicitly setting JNDI properties:

```
... 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:900");    Context initialContext = new
InitialContext(env);    ...
```

- Look up a home for an EJB

The example below shows a lookup of an EJB home. The actual home lookup name is determined by the application's deployment descriptors.

```
// Get the initial context as shown in the previous example    ...    // Look up the home interface
using the JNDI name    try {        java.lang.Object ejbHome =
initialContext.lookup("java:comp/env/comp/mycompany/accounting");        accountHome =
(AccountHome) javax.rmi.PortableRemoteObject.narrow(            (org.omg.CORBA.Object) ejbHome,
AccountHome.class);    }    catch (NamingException e) { // Error getting the home interface        ...
    }
```

- Look up a JavaMail session:

The example below shows a lookup of a JavaMail resource. The actual lookup name is determined by the application's deployment descriptors.

```
// Get the initial context as shown above    ...    Session session = (Session)
initialContext.lookup("java:comp/env/mail/MailSession");
```

4.6.2.3: JNDI caching

To increase the performance of JNDI operations, WebSphere Application Server's JNDI implementation employs caching to reduce the number of remote calls to the name server. For most cases, use the default cache setting.

JNDI context objects employ caching to increase the performance of JNDI lookup operations. Objects bound and looked up are cached in order to speed up subsequent lookups of those objects. Objects are cached as they are bound or initially looked up. Normally, JNDI clients should be able to use the default cache behavior. The following sections describe in detail cache behavior, and how JNDI clients can override default cache behavior if necessary.

- [Cache behavior](#)
- [Cache properties](#)
- [Coding examples](#)

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, a cache name of "iiop://" is used. All instances of `InitialContext` which 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 past the life of the process. A cached object is returned from lookup operations until either the [max cache life](#) for the cache is reached, or the [max entry life](#) for the object's cache entry is reached.

After this time, a lookup on the object will cause the cache entry for the object to be refreshed. If a bind or rebind operation is executed on an object, the change will not be reflected in any caches other than the one associated with the context from which the bind or rebind was issued. This "stale data" scenario is most likely to happen when multiple processes are involved, since different processes do not share the same cache, and `Context` objects in all threads in a process will typically share the same cache instance for a given name service provider.

Usually, cached objects are relatively static entities, and objects becoming stale should not be a problem. However, timeout values can be set on cache entries or on a cache itself so that cache contents are periodically refreshed.

Cache properties

JNDI clients can use several properties to control cache behavior. These properties can be set in the environment Hashtable passed to the `InitialContext` constructor.

You can set properties:

- [From the command line](#)
- [In a properties file](#)
- [Within a Java program](#)

- 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
```

- To set properties in a file, create a text file listing the properties, as for example:

```
... com.ibm.websphere.naming.jndicache.cacheobject=none ...
```

- To set properties in a Java program, use the following **PROPS.JNDI_CACHE*** Java constants, defined in *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"](#), cannot be changed. The "max life" cache properties affect the individual cache's behavior. If the cache already exists, cache behavior will be updated according to the new "max life" property settings. If no "max life" properties exist in the environment, the cache will assume default "max life" settings, irrespective 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. Listed below are the valid values for this property and the resulting cache behavior:

- **"populated"** (default): Use a cache with the specified [name](#). If the cache already exists, leave existing cache entries in cache; otherwise, create a new cache.
- **"cleared"**: Use a cache with the specified [name](#). 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](#) 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.

- **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 providerURL 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. Listed below are the valid options for cache names:

- **"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://server1:900" and "iiop://server1:900/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.

- **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 executed.

- **com.ibm.websphere.naming.jndicache.maxentrylife**

By default, cached objects remain in the cache for the life of the processor 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.

Coding examples

```
import java.util.Hashtable; import javax.naming.InitialContext; import javax.naming.Context; /**
Caching discussed in this section pertains to the WebSphere Application Server initial context
factory. Assume the property, java.naming.factory.initial, is set to
"com.ibm.ejs.ns.WsnInitialContextFactory" as a java.lang.System property. *****/
Hashtable env; Context ctx;
// To clear a cache: env = new Hashtable(); env.put(Props.JNDI_CACHE_OBJECT,
Props.JNDI_CACHE_OBJECT_CLEARED); ctx = new InitialContext(env);
// To set a cache's maximum cache lifetime to 60 minutes: env = new Hashtable(); env.put(Props.JNDI_CACHE_MAX_LIFE, "60"); ctx = new
InitialContext(env);
// To turn caching off: env = new Hashtable(); env.put(Props.JNDI_CACHE_OBJECT,
Props.JNDI_CACHE_OBJECT_NONE); ctx = new InitialContext(env);
// To use caching and no caching: 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); Object o; // Use caching to look up home, since the home should
rarely change. o = ctx.lookup("com/mycom/MyEJBHome"); // Narrow, etc. ... // Do not use cache if data
is volatile. o = noCacheCtx.lookup("com/mycom/VolatileObject"); // ...
```

4.6.2.4: JNDI helpers and utilities

Refer to the [Sun JNDI specification](#) for information on the base JNDI APIs. IBM WebSphere Application Server provides the following JNDI extension and utility to help you implement and debug JNDI.

View the specific files for details.

- [JNDI helper class](#)
- [Name Space Dump utility](#)

4.6.2.4.1: JNDI helper class

The class `com.ibm.websphere.naming.JndiHelpers` contains static methods to simplify common tasks. Refer to the [API documentation](#) for more information.

JNDI helper methods provide assistance with:

- *Recursively creating subcontexts.*

```
[...] import com.ibm.websphere.naming.JndiHelper;    [...] try {        Context
startingContext = new InitialContext();        startingContext =
startingContext.lookup("com/mycompany");        // Creates each intermediate subcontext, if
necessary, as well as leaf context.        // AlreadyBoundException is not thrown.
JndiHelper.recursiveCreateSubcontext(startingContext, "apps/accounting");    } catch
(NamingException e)        // Handle error.    }    [...]
```

- *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
error.    }    [...]
```

- *Binding objects and throwing a `NameAlreadyBoundException` if the object is already bound.*

There are two versions of this JndiHelper method:

```
public static void recursiveBind(Context startingContext, Name name, Object obj)    public
static void recursiveBind(Context startingContext, String name, Object obj)

[...] import com.ibm.websphere.naming.JndiHelper;    [...] try {        Context
startingContext = new InitialContext();        // Creates each intermediate subcontext, if necessary,
and binds object.        JndiHelper.recursiveBind(startingContext, "com/mycompany/apps/accounting",
someObject);    } catch (NamingException e)        // Handle error.    } catch
(Exception e)        // Handle other errors.    }    [...]
```

4.6.2.4.2: JNDI Name Space Dump utility

The name space stored by a given name server can be dumped with the name space dump utility that is shipped with WebSphere Application Server. This utility can be invoked from the command line or from a Java program. The naming service for the WebSphere Application Server host must be active when this utility is invoked.

To invoke this utility using the `class com.ibm.websphere.naming.DumpNameSpace` API, see the [API documentation](#).

To invoke the utility through the command line, enter the following command from the `AppServer/bin` directory:

UNIX: `dumpNameSpace.sh [[-keyword value]...]`

Windows NT: `dumpNameSpace [[-keyword value]...]`

The keywords and associated values for the `dumpNameSpace` utility are:

-host *myhost.austin.ibm.com*

Represents the bootstrap host or the WebSphere Application Server host whose name space you want to dump. The value defaults to *localhost*.

-port *nnn*

Represents the bootstrap port which, if not specified, defaults to **900**.

-factory *com.ibm.websphere.naming.WsnInitialContextFactory*

Indicates the initial context factory to be used to get the JNDI initial context. The value defaults to:

com.ibm.websphere.naming.WsnInitialContextFactory

The default value generally does not need to be changed.

-startAt *some/subcontext/in/the/tree*

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, that is, the bootstrap host root context.

-format {**jndi** | **ins**}

jndi Displays name components as atomic strings.



The default format is **jndi**.

ins Displays name components parsed per INS rules (id.kind).

-report {**short** | **long**}

short Dumps the binding name and bound object type. This output is also provided by JNDI Context.list().



The default report option is **short**.

long 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).

For objects of user-defined classes to display correctly with the long report option, it may be necessary to add their containing directories to the list of directories searched. This can be done by setting the environment variable **WAS_USER_DIRS**. The value can include one or more directories, as for example:

UNIX:

```
WAS_USER_DIRS=/usr/classdir1:/usr/classdir2      export WAS_USER_DIRS
```

Windows NT:

```
set WAS_USER_DIRS=c:\classdir1;d:\classdir2
```

All zip, jar, and class files in the specified directories can then be resolved by the class loader when running `dumpNameSpace`

-traceString *"some.package.name.to.trace.*=all=enabled"*

Represents the trace string with the same format as that generated by the servers. The output is sent to file, `DumpNameSpaceTrace.out`.

-help

Provides a description of Name Space Dump utility and command line usage.

Examples of Name Space Dump utility usage and output

- [Invoke the name space dump utility through a Java program.](#)
- [Invoke the name space dump utility through the command line.](#)
- [View the name space dump utility output.](#)

-
- Invoke the name space dump utility by adding the following code to your Java program:

```
{ [...] java.io.PrintStream filePrintStream = ... Context ctx = new InitialContext(); ctx =
(Context) ctx.lookup("ejssadmin/node"); // Starting context for dump DumpNameSpace dumpUtil = new
DumpNameSpace(filePrintStream, DumpNameSpace.SHORT); dumpUtil.generateDump(ctx); [...]}
```

- Invoke the name space dump utility from the command line by entering the following command:

```
dumpNameSpace -host myhost.mycompany.com -port 901
```

- The generated output will look like the following example, which is the **SHORT** dump format:

```
Getting the initial contextGetting the starting
context=====Name
Space Dump Provider URL: iiop://will:901 Context factory:
com.ibm.websphere.naming.WsnInitialContextFactory Starting context: (top)=bootstrap host root
context Formatting rules: jndi Time of dump: Fri Mar 09 15:11:48 CST
2001=====
=====Beginning of
Name Space Dump=====
1 (top) 2 (top)/jta
javax.naming.Context 3 (top)/jta/usertransaction
com.ibm.ejs.jts.jta.UserTransactionImpl 4 (top)/SecurityCurrent
com.ibm.ejs.security.util.SecurityCurrentRef 5 (top)/ContextHome
com.ibm.ejs.ns.CosNaming.EJSRemoteContextHome 6 (top)/PropertyHome
com.ibm.ejs.ns.CosNaming.EJSRemotePropertyHome 7 (top)/BindingHome
com.ibm.ejs.ns.CosNaming.EJSRemoteBindingHome 8 (top)/will
javax.naming.Context 9 (top)/will/resources javax.naming.Context
10 (top)/will/resources/sec javax.naming.Context 11
(top)/will/resources/sec/SecurityServer com.ibm.WebSphereSecurityImpl.SecurityServerImpl
12 (top)/ejssadmin javax.naming.Context 13 (top)/ejssadmin/node
javax.naming.Context 14 (top)/ejssadmin/node/will javax.naming.Context
15 (top)/ejssadmin/node/will/homes javax.naming.Context 16
(top)/ejssadmin/node/will/homes/DeployEJBHome com.ibm.ejs.sm.tasks.EJSRemoteDeployEJBHome 17
(top)/ejssadmin/node/will/homes/ServletEngineHome
com.ibm.ejs.sm.beans.EJSRemoteServletEngineHome[etc.]
=====End of Name
Space Dump=====
```

4.6.3: Java Message Service (JMS) overview

IBM WebSphere Application Server supports messaging as a method of communication based on the Java MessageService programming interface.

Unlike JavaMail that enables communication initiated by people or by software components to people, Java Message Service (or JMS) only provides communication between software components and applications. Communication provided by JMS is *loosely coupled*, which means the sender and receiver do not have to be active or aware of each other. The communication is also *asynchronous*. This means clients do not have to request messages from the JMS provider in order to receive them, and software components can send messages to other components without stopping their processes to wait for a response.

In this peer-to-peer communication system, each client connects to a messaging agent that provides the framework for sending and receiving messages. The client is required to know only the following:

- message format
- destination of the message

There are two approaches to messaging:

- [Point-to-point](#)
- [Publish/subscribe](#)

The *point-to-point* messaging approach uses such facilities as message queues, senders (or message producers), and receivers (or message consumers). Clients send messages that are destined for a specific receiver to a unique queue. When the receiving client extracts a message from the specific queue, it sends an acknowledgement indicating the message was processed. Queues hold all messages until the messages are received or until they expire.

The *publish/subscribe* messaging approach uses the concepts of publishers, subscribers, and topics. Clients send messages to a topic or a content hierarchy. In order to receive the message, the message consumers must subscribe to that topic. So, in this approach, the message producers are known as publishers and the message consumers are known as subscribers. The JMS provider distributes the messages sent from the multiple publishers to the topic, to the multiple subscribers of that topic.

The [MQSeries product](#) is the default JMS provider for WebSphere Application Server. The MQSeries administration tool, **JMSAdmin**, is used to bind JMS objects (connection factories and destinations) into the namespace, and to set their properties.

WebSphere Application Server Enterprise Edition Version 4.0 also provides the *JMS Listener* function. Similar to an event listener, the JMS Listener enables WebSphere Application Server to react to anonymous, incoming JMS messages by invoking an appropriate enterprise java bean. The invoked enterprise bean is a stateless session bean with `anonMessage()` method.

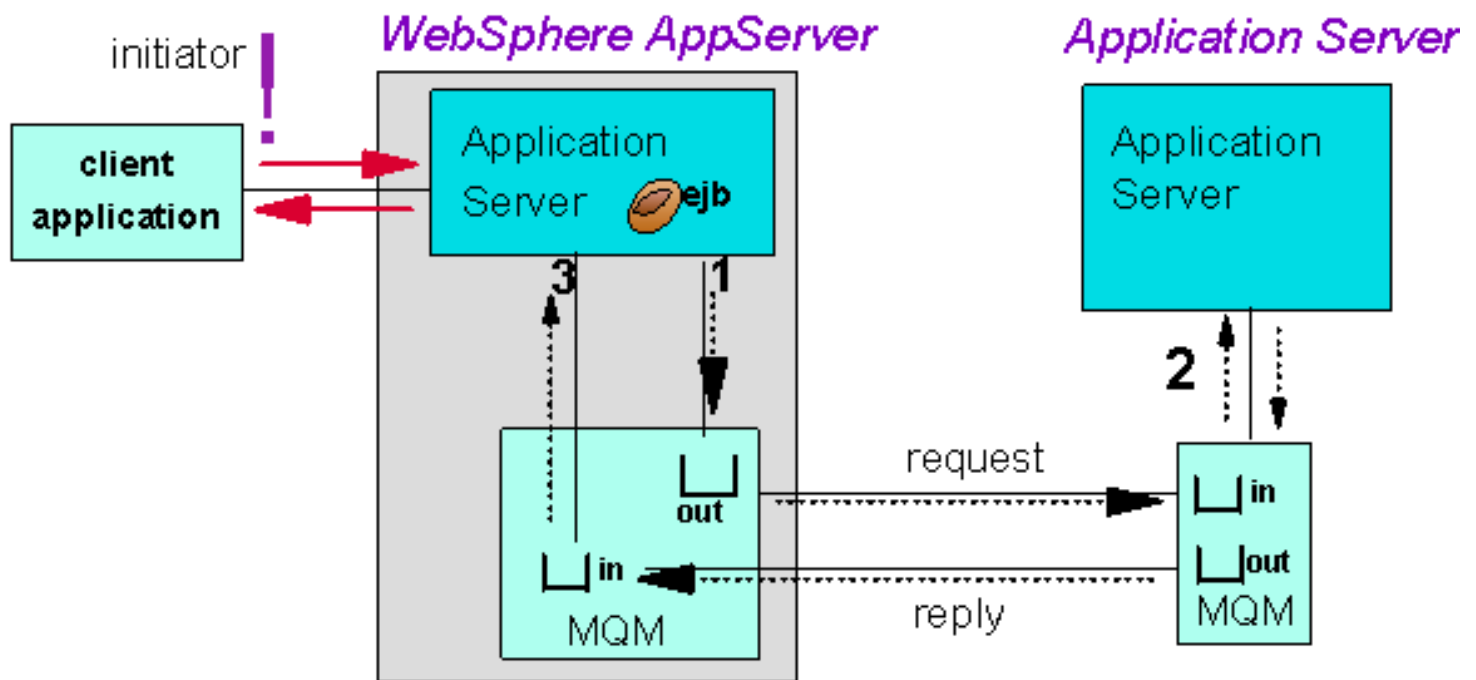
4.6.3.1: Using the JMS point-to-point messaging approach

This article describes the point-to-point messaging approach using WebSphere ApplicationServer's default JMS provider, MQSeries. The MQSeries messaging server implements *point-to-point* communication. To enable the MQSeries point-to-point messaging support, you need:

- MQSeries Version 5.2 or greater
- With MQSeries Version 5.2, you need MQSeries SupportPac MA88

MQSeries can now act as a resource manager in application transactions, and WebSphere Application Server can act as the transaction coordinator. For example, when a client application sends a request, WebSphere Application Server, using MQSeries, puts the message on an *outqueue* and waits for a response to return to the *in* queue. In this scenario, there is no guarantee the message was sent, or that the receiver received the message. These types of messages are known as *non persistent* messages.

The *point-to-point* messaging approach in WebSphere Application Server and MQSeries is illustrated in the following graphic:



Message delivery can be defined as:

- *Persistent* - this is the default mode of delivery. The "message send" is logged into stable storage.
- *Non persistent* - message delivery is not guaranteed. This mode of delivery improves performance and reduces storage overhead.

Message delivery properties can be set:

- On the queue within the queue manager
- On the queue object using the JMSAdmin tool
- On individual messages within your JMS application

To define a queue in the JNDI namespace and to set the persistence properties for the queue, enter the following command in the MQSeries JMSAdmin tool:

```
InitCtx> DEFINE Q(TESTQ) PERSISTENCE(XXX)
```

Where **xxx** is one of the following:

APP	(Default) Persistence is defined by the application
QDEF	Persistence is defined by the queue default (Set in the queue manager)
PERS	Messages are persistent
NON	Messages are non-persistent


If your application sends a message and requires a reply, set a reasonable timeout value in your application to handle a delayed or "no" reply situation. The following application code waits for a maximum of 5000 milliseconds:

```
Message inMessage = queueReceiver.receive(5000);
```

Set a similar timeout in your reply message.

Transactions to MQSeries are boundary transactions not end-to-end transactions. This means that only a *put* to a queue, or a *get* from a queue is part of the transaction. The flow to a remote application is not part of the transaction. In order to guarantee the message is received by the remote application, define that message in the JMSAdmin tool as a *persistent* message.

WebSphere enterprise applications can use the JMS Listener function to automatically receive messages from input queues (JMS destinations) and to coordinate the processing of those messages. This enables automatic asynchronous delivery of messages to an enterprise application, instead of the application having to explicitly poll for messages on the queue. For more information on the JMS Listener function, see [An overview of the JMS Listener](#).

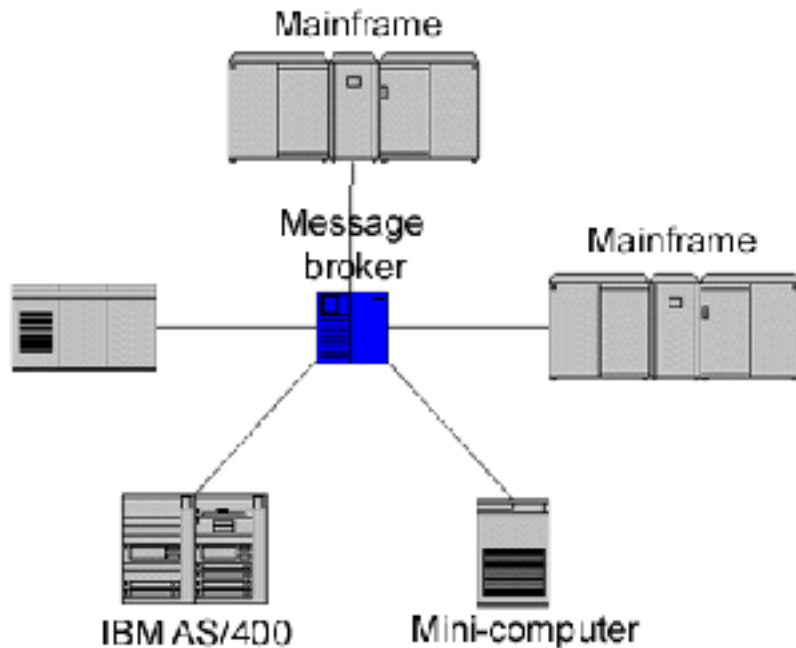
 The link to the JMS Listener documentation will not work unless the WebSphere Application Server Enterprise Edition product extensions are installed on your system.

See the [Support of the MQSeries Java Message Service resources](#) article for configuration information.

4.6.3.2: Using the JMS publish/subscribe messaging approach

This article describes the "publish/subscribe" messaging approach using WebSphere ApplicationServer's default JMS provider, MQSeries. You can implement the "publish/subscribe" messaging approach in MQSeries with the [Pub-Sub SupportPac](#) or with Integrator.

To alleviate the complexity of a multiple queue manager topology, MQSeries introduced the concept of Message Brokers with the MQSeries Integrator product. The following graphic illustrates five queue managers configured to use a Message Broker:



In addition to the Message Broker, the Integrator product also supports a Message Repository Manager, and the *publish and subscribe* messaging approach. With this approach, the Message Broker matches a topic on a published message with a list of clients who have subscribed to that topic. Neither publisher nor subscriber is aware of each other. Publishers only know of the topics they describe for their messages, and subscribers only know of the topics they requested.

In this topology, WebSphere Application Server can be a publisher or subscriber, or both, but requires the configuration and resource support of the MQSeries Integrator product.

Visit the [MQSeries Integrator](#) site for more information.

4.6.3.3: Support of Java Message Service resources

Unlike other J2EE resources that are typically objects that run in and are part of the application server, JMS resources are external to WebSphere Application Server. This means administrators must first use a JMS provider's administration tool to create the connection factories and destinations, and to assign these objects with correct configuration attributes. After this step is completed, administrators can then use WebSphere Application Servers' administrative client to create JMS resource objects to reference the external objects.

In WebSphere Application Server Version 4.0, the MQSeries product is defined as the default JMS provider. However, since MQSeries is not shipped with WebSphere Application Server, this JMS provider is not installed on any node.

Since this provider is predefined, after you install the MQSeries product, you only need to go to the **Nodes** tab of the properties editor for the Provider to install it on the desired nodes.

The following steps describe how to implement JMS support in WebSphere Application Server:

1. Configure a JMS provider. By default this will be the MQSeries product.
2. Create the destination and connection factory with the JMS provider's admin tool.
This will bind references to these objects in the JNDI namespace.
3. Create corresponding JMS resources in WebSphere Application Server, declaring the location where they were bound by the JMS admin tool as an attribute.
The `RepositoryObject` implementation for the resource binds the resource into the WebSphere Application Server namespace.
4. Deploy the application, which resolves the JMS resource references with the JMS objects.
5. Start the application server containing the application.

At this point, the `ResourceBinder` object in the application server binds the JMS resource objects into the namespace.

6. The application code performs a "lookup" on a JMS resource.
The "lookup" finds the `IndirectJNDILookup` bound at the target WebSphere Application Server location, and uses it to perform a subsequent lookup of the actual resource in the provider's namespace.

4.6.3.4: Support for the use of MQSeries Java Message Service resources

WebSphere Application Server Enterprise JavaBeans support the transactional use of MQSeries Java Message Service (JMS) resources.

To use this feature, install MQSeries version 5.2 and the MQSeries classes for Java and JMS. Only MQSeries V5.2 provides this support; earlier versions will not work.


To configure JMS resources for use with WebSphere Application Server:

1. Download the MQSeries Java and JMS classes, or the pub-sub package from one of the following URLs:
 - <http://www.ibm.com/software/ts/mqseries/api/mqjava.html>
 - <http://www.ibm.com/software/ts/mqseries/txppacs/ma0c.html>
2. Review the [MQSeries Using Java](#) book for a description of the parameters required for WebSphere Application Server.


 The instructions in this book refer to a WebSphere Application Server Version 3.5.3 environment, and are not valid for WebSphere Application Server Version 4.0. For example, the following content in the book is invalid for Version 4.0:

- Names of jar files
- Dependent classpath
- Adminserver classpath

3. Do the following to configure WebSphere Application Server and MQSeries for JMS support:
 - a. Modify the `JMSAdmin.bat` file to include the option `-java.ext.dirs=<WS AE>\lib` when running the MQSeries administration tool, [JMSAdmin](#).
 - b. Modify the `JMSAdmin.config` file by uncommenting the following lines:
`INITIAL_CONTEXT_FACTORY=com.ibm.ejs.ns.jndi.CNInitialContextFactory`
`PROVIDER_URL=iiop://localhost/ (or iiop://host-name)`
 - c. *Comment out* the following lines in the `JMSAdmin.config` file:
`INITIAL_CONTEXT_FACTORY=com.ibm.sun.jndi.fscontext.RefFSContextFactory`
`PROVIDER_URL=file:/C:/JNDI-Directory/`
 - d. Add the following to the application server classpath:

 You can add these classes in the console's default JVM settings, or by editing WebSphere Application Server's `admin.config` file.

- `<MQ JMS>\lib` directory
- `com.ibm.mq.jar` file
- `com.ibm.mqjms.jar` file

 An alternative way to set up the configuration is to use the administrative console. In the Resources, JMS Providers folder, specify the ContextFactory, the provider URL, and the path to the MQSeries JMS .jar files. See the article about administering JMS support resources for more information.

4. Bind the classes provided by the new function into the JNDI namespace using the MQSeries administration tool, [JMSAdmin](#). The JMSAdmin tool provides for two, new WebSphere Application Server JMS connection factories:
 - **WSQCF** - a new type of queue connection factory
 - **WSCTF** - a new type of topic connection factory

WebSphere Application Server connection factory objects

The following calls can be used either in a global transaction or in an unspecified transaction context:

- `QueueSender.send`
- `MessageConsumer.receive`
- `MessageConsumer.receiveNoWait`
- `TopicPublisher.publish`

If another resource manager, as for example JDBC, is involved in a global transaction, the MQSeries JMS resources are involved in a 2-phase commit. The 1-phase commit occurs if only the JMS resources are involved in a global transaction. This is a feature of the Transaction Manager optimization.

In a global transaction, messages sent with `QueueSender.send` or published with `TopicPublisher.publish` do not become visible until the transaction is committed. Messages received by `MessageConsumer.receive` or `MessageConsumer.receiveNoWait` are queued if the transaction is rolled back. Both bean-managed transaction demarcation and container-managed transaction demarcation are supported.

If no global transaction is active, then an "unspecified transaction context" situation occurs. The following circumstances cause an "unspecified transaction context":

- EJB methods when a global transaction cannot occur (for example, `ejbCreate`)
- Bean Managed Transaction methods where the bean writer chose not to begin a transaction
- Container Managed Transaction `NOT_SUPPORTED` or `NEVER` methods
- Container Managed Transaction `SUPPORTS` methods when no transaction exists

In an unspecified transaction context, the transactional behavior is specified in the `transacted` flag that is passed when the session is created. If the `transacted` flag is set to *false*, the messaging operations occur immediately. This is also known as the 0-phase commit. If the flag is set to *true*, the send, receive, and publish operations occur on the commit of the session, or also known as the 1-phase commit.

A summary of the transactional behavior for objects created on WSQCF or WSTCF is described in the following table:

	Global transaction context	Unspecified transaction context
<code>transacted=false</code>	2-phase commit	0-phase commit
<code>transacted=true</code>	2-phase commit	1-phase commit


To commit or to roll back the messaging work done on a transacted session, call method `session.commit()` or `session.rollback()`. First check whether `session.getTransacted()` returns **true** before committing the session. `Session.getTransacted()` returns **true** if:

- The user passed in **true** as the `transacted` parameter when the session was created, and
- No global transaction is active at the moment of the call.

If both tests are met, you can commit the session. Trying to commit a session when a global transaction is active will result in the JMS exception, `IllegalStateException`, being thrown.

JMS XA support in WebSphere Application Server is integrated with local transactions. For container managed transactions, an "unspecified transaction context" causes WebSphere Application Server to start a local transaction. In Version 4.0, the scope of the local transaction is the EJB method. The action taken at the end of the EJB method (commit or rollback of the local transaction) depends on the information contained in the deployment descriptor. This information is a WebSphere Application Server extension. The transaction manager will commit or rollback any outstanding, uncommitted work done within the local transaction without any user intervention. The default is to roll back.

Any work performed on a JMS session in an unspecified transaction context, will be rolled back or committed if the corresponding local transaction is rolled back or committed.

 Requestors are only used with non-transacted sessions. Therefore, *QueueRequestor* and *TopicRequestor* cannot be used with sessions created by WebSphere Application Server JMS connection factories.

Unsupported interfaces and methods

The following JMS interfaces are not designed for application use and, therefore, cannot be invoked:

Unsupported interfaces


javax.jms.ServerSession
javax.jms.ServerSessionPool
javax.jms.ConnectionConsumer
all the *javax.jms.XA* interfaces

The following JMS methods are inappropriate in this environment and interfere with connection management by the container. Therefore, these methods cannot be used:

Unsupported methods


javax.jms.Connection.setExceptionListener
javax.jms.Connection.stop
javax.jms.Connection.setClientID
javax.jms.Connection.setMessageListener
javax.jms.Session.getMesssageListener
javax.jms.QueueConnection.createConnectionConsumer
javax.jms.TopicConnection.createConnectionConsumer
javax.jms.TopicConnection.createDurableConnectionConsumer
javax.jms.MessageConsumer.setMessageListener

All the above methods throw the JMS exception, *IllegalStateException*, when invoked.

 You cannot register a *MessageListener* with a *QueueReceiver* or *TopicSubscriber*.

The following methods throw the JMS exception, *IllegalStateException*, if used within a global transaction:

javax.jms.Session.commit
javax.jms.Session.rollback
javax.jms.Session.recover

 With the Enterprise JavaBeans programming model, you must ensure all JMS resources are closed correctly. Since JMS resources never time-out, JMS resources that are not closed correctly will continue to consume MQSeries resources. The MQSeries resources also persist until the application server or MQSeries Queue manager is restarted.

Restrictions

The following restrictions exist regarding the use of JMS XA support in WebSphere Application Server:

- A subscriber can only be used in the same type of transactional context (for example, a global transaction or an unspecified transaction context) as the one that existed when the subscriber was created.

If this restriction is not respected, the JMS exception, *subscriber restriction*, is thrown.

If a global transaction is active at the creation of the subscriber, that subscriber can be used to receive messages in different global transactions, but not in an unspecified transaction context.

If an unspecified transaction context is active when the subscriber is created, that subscriber cannot be used with a global transaction.

- The use of JMS sessions across methods with different transactional attributes is restricted.

If a session was used within a global transaction, it cannot be reused in a different global transaction or in an unspecified context until the first transaction commits. Similarly, if there is work outstanding in a local transaction then the session cannot be used in a global transaction until the local transaction has finished. Session use, in this case, refers to the send, receive, and publish operations using the message producers or consumers that were created on the session.

4.7: Java Clients

In a traditional client server environment, the client requests a service and the server fulfills the request. A single server is used by multiple clients. Clients can also access several different servers. This model persists for Java clients except now these requests make use of a client's runtime environment.

Prior to *J2EE* (Java™ 2 Platform Enterprise Edition), a typical Web-based client application consisted of the following model:

browser (HTML file) -> servlet -> EJB

In this model, the client application requires a servlet to communicate with the Enterprise Java Bean (EJB), and the servlet must reside on the same machine as WebSphere Application Server.

With version 4.0, Java application clients can now consist of the following models:

- [Applet client](#)
- [J2EE application client](#)
- [Java thin application client](#)

In the ***Applet client*** model, a Java applet is embedded in a HyperText Markup Language (HTML) document residing on a client machine that is remote from WebSphere Application Server. With this type of client, the user accesses an EJB in WebSphere Application Server through the Java applet in the HTML document.

The ***J2EE application client*** is a Java application program that accesses EJBs, JDBC databases, and Java Message Service message queues. The *J2EE application client* program runs on client machines. This program follows the same Java programming model as other Java programs; however, the *J2EE application client* depends on the application client runtime to configure its execution environment, and it uses the JNDI name space to access resources.

The ***Java thin application client*** provides a light-weight Java client programming model. This client is best suited for use in situations where a Java client application exists but the application must be enhanced to make use of EJBs, or where the client application requires a thinner, more light-weight environment than the one offered by the J2EE application client.

4.7.1: Applet client programming model

The Java Applet client provides a browser-based Java runtime that is capable of interacting with EJBs directly instead of indirectly through a servlet.

This client is designed to support those users who want a browser-based Java client application programming environment that provides a richer and more robust environment than the one offered by the `Applet`→`Servlet`→`EJB` model.

The programming model for this client is a cross between the Java application thin client and a servlet client. When accessing EJBs from this client, the EJB object references can be considered CORBA object references by the applet.

There is no tooling support for this client for developing, assembling or deploying the applet. You are responsible for developing the applet, generating the necessary client bindings for the EJBs and CORBA objects, and bundling these pieces together to be installed on or downloaded to the client machine. The Java applet client provides the necessary runtime to support communication between the client and the server.

Client side bindings are generated during the deployment phase of J2EE development using the Application Assembly Tool. An applet can utilize these bindings, or you can generate client side bindings using the `rmic` command that is part of the IBM JDK installed with the WebSphere Application Server.

See article [Packaging and distributing Java clients](#) for more information.

The Applet client makes use of the RMI-IIOP protocol. The use of this protocol enables the applet to access EJB references and CORBA object references, but it is restricted in the usage of some supported CORBA services. If you combine the EJB and CORBA environments in one applet, you must understand the differences between the two programming models, and you must use and manage each appropriately.

The Applet client provides the runtime to support the J2EE Applet client. The J2EE Applet client does not have any tooling support for developing, assembling or deploying the Applet. The applet client runtime is provided through the use of the Java applet browser plug-in that is installed on the client machine using the WebSphere Application Server Client CD.

Because the Applet client does not provide for a deployment descriptor, the Applet code cannot make use of the JNDI `java: /comp` lookup. The Applet must know the fully qualified location of the EJB in the JNDI namespace. For example, the JNDI `java: /comp` allows lookup of enterprise java beans using a short name or a nickname such as:

```
java.lang.Object ejbHome = initialContext.lookup("java:/comp/env/ejb/MyEJBHome"); MyEJBHome =  
(MyEJBHome) javax.rmi.PortableRemoteObject.narrow(ejbHome, MyEJBHome.class);
```

But the code in an applet client must be more explicit:

```
java.lang.Object ejbHome =  
initialContext.lookup("the/fully/qualified/path/to/actual/home/in/namespace/MyEJBHome"); MyEJBHome =  
(MyEJBHome) javax.rmi.PortableRemoteObject.narrow(ejbHome, MyEJBHome.class);
```

The Applet environment restricts accessing external resources from the browser runtime environment. Some of these resources can be made available to the Applet by setting the correct security policy settings in the JRE `java.policy` file. If given the correct set of permissions, the Applet client must explicitly create the connection to the resource using the appropriate API (JDBC, JMS, and others). This client does not perform any initialization of any of the services that the client applet may need. For instance, the client application is responsible for the initialization of the naming service, either through `CosNaming` or `JNDI`.

The following table describes the advantages and disadvantages of the *Applet client*:

Advantages	Disadvantages
<ul style="list-style-type: none">● Light-weight client suitable for download.● Provides access to JNDI interfaces for EJB object resolution.● No distribution of the applet to the client machine required (performed through the browser)	<ul style="list-style-type: none">● Designed for use in an intranet environment.● Lack of client runtime initialization of environment and services.● Lack of built-in support for local resource resolution and configuration.● Does not promote portability of client application code.● Requires a browser to be installed on the client machine.

4.7.1.1: Developing an Applet client

Unlike typical applets that reside on either Web servers or WebSphere ApplicationServers and can only communicate using the HTTP protocol, the WebSphere Applet clients are capable of communicating over the HTTP protocol and the RMI-IIOPprotocol. This additional capability gives the Applet direct access to enterprise java beans. As such, Applet clients have the following setup requirements:

- These clients are currently available on the Windows NT or Windows 2000 platforms.



Support for additional platforms will be added in the near future.

Check the [prerequisites page](#) for information on new platform support.

- They require one of these browsers:
 - Internet Explorer version 5.0+
 - Netscape Navigator 4.7+
- The browser must be installed before the client code is installed.
- The Applet client is installed from the *WebSphere Clients for Windows* CD by selecting option, "Java Application/Applet Thin Client."
- You must install the WebSphere Application Server Plug-in for the browser.Select option, "Java Application/Applet Thin Client," from the *WebSphere Clients for Windows* CD.
- From the WebSphere Application Server Java Plug-in Control, enter:

`-Djava.ext.dirs=\WebSphere\AppClient\lib`



1. The **Java Run Time Parameters** field is similar to the command prompt when using command line options. Therefore, most options available from the command prompt (for example, -cp, classpath, and others), can be entered in this field as well.
2. The control panel can be accessed from the **Start** menu.
Click start > control panel > WebSphere Java Plug-in Control.
3. The applet container is the Web browser and the Java Plug-in combination. You must first install the WebSphere Application Server Applet client so that the browser recognizes the IBM Java Plug-in.

Tag requirements

Standard applets require the HTML <APPLET> tag to identify the applet to the browser. The <APPLET> tag invokes the browser's Java Virtual Machine (JVM). So an applet running on Internet Explorer will use Microsoft's JVM.

For applets to communicate with EJBs in the WebSphere Application Server environment, the <APPLET> tag must be replaced with these two new tags:

<OBJECT>

<EMBED>

Additionally, the `classid` and `type` attributes **cannot** be modified, and must be entered as described in the [applet client](#) example. Finally, the `codebase` attribute on the <OBJECT> tag must be excluded.



Do not confuse the `codebase` attribute on the <OBJECT> tag with the `codebase` attribute on the <PARAM> tag. Although both are called `codebase`, they are separate entities.

The following code snippet illustrates the applet code. In this example, *MyApplet.class* is the applet code, *applet.jar* is the file that contains the applet code, and *EJB.jar* is the file that contains the EJB code:


```

<OBJECT classid="clsid:8AE2D840-EC04-11D4-AC77-006094334AA9"
        width="600" height="500">
  <PARAM NAME=CODE VALUE=MyAppletClass.class>
  <PARAM NAME="archive" VALUE='Applet.jar, EJB.jar'>
  <PARAM TYPE="application/x-java-applet;version=1.3">
  <PARAM NAME="scriptable" VALUE="false">
  <PARAM NAME="cache-option" VALUE="Plugin">
  <PARAM NAME="cache-archive" VALUE="Applet.jar, EJB.jar">
  <COMMENT>
  <EMBED type="application/x-websphere-client" CODE=MyAppletClass.class
        ARCHIVE="Applet.jar, EJB.jar" WIDTH="600" HEIGHT="500"
        scriptable="false">
  <NOEMBED>
  </COMMENT>
</NOEMBED>WebSphere Java Application/Applet Thin Client for
        Windows is required.
</EMBED>
</OBJECT>

```



The value of the type attribute on the <EMBED> tag can also be:

```
<EMBED type="application/x-websphere-client, version=4.0" ...
```

Code requirements

The code used by an applet to talk to an EJB is the same as that used by a standaloneJava program or a servlet, except for one additional property called *java.naming.applet*. This property informs the InitialContext and the ORB that this client is an applet rather than a standalone Java application or servlet.

When initializing an instance of the InitialContext class, the first two lines in this code snippet illustrate what both a standalone Java program and a servlet issue to specify the computer name, domain, and port. In this example, <yourserver.yourdomain.com> is the computer name and domain where WebSphere Application Server resides, and 900 is the configured port. After the bootstrap values (<yourserver.yourdomain.com>:900) are defined, the client to server communications occur within the underlying infrastructure. In addition to the first two lines, for applets, you must add the highlighted third line to your code. That line identifies this program as an applet:

```

prop.put(Context.INITIAL_CONTEXT_FACTORY,
"com.ibm.websphere.naming.WsnInitialContextFactory");
prop.put(Context.PROVIDER_URL, "iiop://<yourserver.yourdomain.com>:900");
prop.put("java.naming.applet", this);

```

Security requirements


When code is loaded, it is assigned "permissions" based on the security policy in effect. This policy, specifying which permissions are available for code from various locations, can be initialized from an external policy file. For each client, the *java.policy* file should be updated with the classes that the applet client needs to access, and with the ports on the host machines where it needs different permissions.

The following lines of code must be added to existing *java.policy* files. This code allows access to the required ports so that the applet client can communicate with an EJB.

In the example below, the `java.net.SocketPermission "localhost:1024--", "listen"` entry grants permission for Applets to open sockets for listening on the localhost for any port from 1024 to 65535. Port can be specified as a range of port numbers or a specific port. A port range specification of the form "N-", where N is a port number, signifies all ports numbered N and above. A specification of the form "-N" indicates all ports numbered N and below.

The first SocketPermission statement grants permission to the client to have ports opened for listening. The second grants permission to open a port and make a connection to a host machine, which is your WebSphere Application Server. In this example, *yourserver.yourcompany.com* is the complete hostname of your WebSphere Application Server:


```
permission java.util.PropertyPermission "server.root", "read";
permission java.util.PropertyPermission "*", "read,write";
permission java.io.FilePermission "traceSettingsFile", "read,write";
permission java.util.PropertyPermission "traceSettingsFile", "read,write";
permission java.lang.RuntimePermission "modifyThread";
permission java.net.SocketPermission "localhost:1024-", "listen";
permission java.net.SocketPermission "yourserver.yourcompany.com", connect";
```

 For more information on security relating to user authentication and signed jars, read the official documentation for [Java security architecture](#)

Learn more about the WebSphere Applet client by running the Applet sample. You can install the Applet client sample from the *WebSphere Application Client* CD. This sample is called HelloEJB and is installed in the [product_installation](#)/WSsamples/Clientsubdirectory on the client machine.

4.7.2: J2EE application client programming model

The J2EE application client programming model provides the benefits of the J2EE platform for the Java client application. The J2EE platform offers the ability to seamlessly develop, assemble, deploy and launch a client application. The tooling provided with the WebSphere platform supports the seamless integration of these stages to help the developer create a client application from start to finish.

When a client application is developed using and adhering to the J2EE platform, the client application code is portable from one J2EE platform implementation to another. The client application package might require redeployment using each J2EE platform's deployment tool, but the code that comprises the client application will not change.

The J2EE application client runtime supplies a container that provides access to system services for the client application code. The client application code must contain a main method. The application client runtime invokes this main method after the environment is initialized and runs until the Java virtual machine is terminated.

The J2EE platform allows the J2EE application client to make use of "nicknames" or "short names," defined within the client application deployment descriptor. These deployment descriptors identify EJBs or local resources (JDBC, JMS, JavaMail and URL) for simplified resolution through the use of JNDI. This simplified resolution to the EJB reference and local resource reference also eliminates changes to the client application code when the underlying object or resource either changes or moves to a different server. Should these changes occur, the J2EE application client might require redeployment.

The J2EE application client also provides for initialization of the runtime environment for the client application. This initialization is unique for each client application and is defined by the deployment descriptor. In addition, the J2EE application client runtime provides support for security authentication to the EJBs and local resources.

The J2EE application client makes use of the RMI-IIOP protocol. The use of this protocol enables the client application to access EJB references and to make use of CORBA services that are provided by the J2EE platform implementation. Use of the RMI-IIOP protocol and the accessibility of CORBA services assist users in developing a client application that requires access to both EJB references and CORBA object references. When users combine the J2EE and CORBA environments or programming models in one client application, they need to understand the differences between the two programming models, and they must use and manage each appropriately.

The following table describes the advantages and disadvantages of the J2EE application client.

Advantages	Disadvantages
<ul style="list-style-type: none">● Provides the user with the benefits of the J2EE platform.● Allows the use of nicknames in the Deployment Descriptor for reference identity● Client application code is portable across J2EE compliant platforms (may need to be redeployed for each distinct J2EE platform).● Supports CORBA services (usage of CORBA services may render the client application code non-portable).	<ul style="list-style-type: none">● A heavier client than the Java application thin client, and is not suited for downloading.● Designed for use in an intranet environment.● Requires distribution of the application to the client machine.

4.7.2.1: Resources referenced by a J2EE application client

J2EE application clients access resources by performing lookup operations in the JNDI name space. The application client runtime then provides a mapping of the resource names, used and configured by client application programs, to the actual resource objects. This allows client application programs to access different resources, such as test or production databases, without the need for updates or recompiles.

To provide this service, the application client runtime requires information about the names and types of resources used by the client application program.

The three types of resources a *J2EE application client* can reference are:

1. **EJB references** - are references to Enterprise Java Beans (EJBs) running on WebSphere Application Server.
2. **Resource references** - are references to other types of resources, such as:
 - JDBC databases
 - URLs
 - Java Message Service message queues
 - Java Mail
3. **Environment entries** - are references to simple data types that you would not want to code in your application program, such as timeout values or SQL query strings. The following Java basic data types are supported:

<code>java.lang.Boolean</code>	<code>java.lang.String</code>	<code>java.lang.Integer</code>	<code>java.lang.Double</code>
<code>java.lang.Byte</code>	<code>java.lang.Short</code>	<code>java.lang.Long</code>	<code>java.lang.Float</code>

The resource information is defined and configured using these two WebSphere Application Server tools:

- Application Assembly Tool (AAT) (used for the definition)
- Application Client Resource Configuration Tool (used for the configuration)

The **Application Assembly Tool** manages:

- EJB references
- non-client specific Resource references
- all Environment entries

The **Application Client Resource Configuration Tool** manages:

- client specific Resource references such as a JDBC Datasource name
This information is stored with the client application program in an Enterprise Archive File (.ear file).

4.7.2.2: Developing a J2EE application client

From an application developer's point of view, creating a *J2EE application client* program involves these steps:

1. [Writing the client application program](#)
 2. [Assembling the application client \(using the Application Assembly Tool\)](#)
 3. [Assembling the Enterprise Archive \(EAR\) file](#)
 4. [Distributing the EAR file](#)
 5. [Configuring the application client resources](#)
 6. [Launching the application client](#)
-

1. Writing the client application program

Write the J2EE application client program on any development machine. At this stage, you do not require access to WebSphere Application Server.

A *J2EE application client* program is similar to a standard Java program in that it runs in its own Java virtual machine and is invoked at its main method. The J2EE application client program differs from a standard Java program because it uses the JNDI name space to access resources. In a standard Java program, the resource information is coded in the program.

Storing the resource information separately from the client application program makes the client application program portable and more flexible.

Using the `javax.naming.InitialContext` class, the client application program uses the lookup operation to access the JNDI name space. The `InitialContext` class provides the lookup method to locate resources. For more information on JNDI, see file, [JNDI overview](#).

The following example illustrates how a client application program uses the `InitialContext` class:

```
import javax.naming.*; public class myAppClient {    public static void main(String argv[])    {
InitialContext ctx = new InitialContext();        Object myObj =
ctx.lookup("java:comp/env/ejb/HelloBean");        HelloHome home
=(HelloHome) javax.rmi.PortableRemoteObject.narrow(myObj, HelloHome.class);
...    } }
```

In this example, the program is looking up an Enterprise Java Bean called *HelloBean*. The *HelloBean* EJB reference is located in the client JNDI name space at `java:comp/env/ejb/HelloBean`. Since the actual Enterprise Java Bean is running on the server, the application client runtime returns a reference to *HelloBean's* home interface.

If the client application program's lookup was for a Resource reference or an Environment entry, then lookup would return an instance of the configured type as defined by the client application's Deployment Descriptor. For example, if the program's lookup was a JDBC datasource, the lookup would return an instance of `javax.sql.DataSource`.

2. Assembling the application client (using the Application Assembly Tool)

The JNDI name space knows what to return on a lookup because of the information that is assembled by the Application Assembly Tool (AAT).

Assemble the J2EE application client on any development machine that has the AAT installed.

When you use the Application Assembly Tool to assemble your application client, you provide the *application client* runtime with the required information to initialize the execution environment for your client application program. Refer to the [Application Assembly Tool](#) description for implementation details.

Here is a list of things to keep in mind when you configure resources used by your client application program:

- When configuring Resource references and EJB references in the Application Assembly Tool, the **General** tab contains a required **Name** field. This field specifies where the application client runtime will bind the reference to the real object in the `java:comp/env` portion of the JNDI namespace. The application client runtime always binds these references relative to `java:comp/env`. So, for the programming example above, you would specify `ejb/HelloBean` in the **Name** field on the **General** tab of the Application Assembly Tool, which would require the program to perform a lookup of `java:comp/env/ejb/HelloBean`. If the **Name** field were set to `myString`, the resulting lookup would be `java:comp/env/myString`.
- When configuring Resource references in the Application Assembly Tool, the **Name** field on the **General** tab is used for:
 - binding a reference of that object type into the JNDI name space.
 - retrieving client specific resource configuration information that was configured using the Application Client Resource Configuration Tool.
- When configuring a Resource reference in the Application Assembly Tool, the value in the **Name** field on the **General** tab must match the value in the **JNDIName** field on the **General** tab for the resource in the Application Client Resource Configuration Tool.
- When configuring URL resources using the Client Resource Configuration Tool, the URL provider panel allows you to specify a protocol and a class that handles that protocol. If you want to use the default protocols, such as HTTP, you can leave those fields blank.
- When configuring Resource references using the Application Assembly Tool, the **General** tab contains a field called **Authorization**. This field can be set to either **Container** or **Application**. If the field is set to **Container**, then the application client runtime will use authorization information configured in the Application Client Resource Configuration tool for the resource. If the field is set to **Application**, then the Application Client runtime expects the user application to provide authorization information for the resource. The application client runtime will ignore any authorization information configured with the Application Client Resource Configuration tool for that resource.

3. Assembling the Enterprise Archive (EAR) file

The application is contained in an Enterprise Archive or `.ear` file. The `.ear` file is composed of:

- EJB, Application Client, and user-defined modules or `.jar` files
- Web applications or `.war` files
- Metadata describing the applications or application `.xml` files

You must assemble the `.ear` file on the server machine.

4. Distributing the EAR file

The `.ear` file must be made accessible to those client machines that are configured to run this client.

If all the machines in your environment share the same image and platform, run the Application Client Resource Configuration Tool (ACRCT) on one machine to configure the external resources, and then distribute the configured `.ear` file to the other machines.

If your environment is set up with a variety of client installations and platforms, you must run the ACRCT for each unique configuration.

The `.ear` files can either be distributed to the correct client machines, or made available on a network drive.

See article [Packaging and distributing Java clients](#) for more information.

Distributing the `.ear` files is the responsibility of the system and network administrator.

5. Configuring the application client resources

If local resources are defined for use by the client application, run the ACRCT on the local machine to reconfigure the `.ear` file. Use the ACRCT to change the configuration. The ACRCT is the *Application Client Resource Configuration Tool* described in the previous steps. For example, the `.ear` file may contain a DB2 resource, which is configured as `C:\DB2`. If, however, the user has DB2 installed in directory `D:\Program Files\DB2`, that user must use the ACRCT to create a local version of the `.ear` file.

6. Launching the application client

Using the fully assembled and configured `.ear` file, issue the `launchClient` command to launch the J2EE application client on the client machine.

Note: Learn more about the WebSphere J2EE client by running the client sample. You can install the client sample from the *WebSphere Application Client* CD. On a server machine, the J2EE client sample is part of the samples gallery. See the "Application Client" section of `Samples.ear`. This sample is called `HelloEJB` and is installed in the `product_installation/WSsamples/Client` subdirectory on the client machine.

4.7.2.3: Troubleshooting guide for the J2EE application client

This section provides some debugging tips for resolving common J2EE application client problems. To use this troubleshooting guide, review the trace entries for one of the J2EE application client exceptions, and then locate the exception in the guide.

- [java.lang.NoClassDefFoundError](#)
 - [com.ibm.websphere.naming.CannotInstantiateObjectException](#)
 - [javax.naming.ServiceUnavailableException](#)
 - [javax.naming.CommunicationException](#)
 - [javax.naming.NameNotFoundException](#)
 - [java.lang.ClassCastException](#)
 - [com.ibm.etools.archive.exception.OpenFailureException](#) (message numbers WSCL0206E and WSCL0100E)
-

Error: `java.lang.NoClassDefFoundError`

Explanation: This exception is thrown when Java cannot load the specified class.

Possible causes:

- Invalid or non-existent class
- Classpath problem
- Manifest problem

User response: First check to determine if the specified class exists in a jar file within your ear file. If it does, make sure the path for the class is correct. For example, if you get the exception:

```
java.lang.NoClassDefFoundError: WebSphereSamples.HelloEJB.HelloHome
```

ensure the class `HelloHome` exists in one of the jar files in your ear file. If it exists, ensure the path for the class is `WebSphereSamples.HelloEJB`.

If both the class and path are correct, then it is a classpath issue. Most likely, you do not have the failing class's jar file specified in the client jar file's manifest. To check this, open your ear file with the [Application Assembly Tool](#) and click on the Application Client. Add the names of the other jar files in the ear file to the `Classpath` field. This exception is generally caused by a missing EJB module name from the `Classpath` field.

If you have multiple jars to enter in the `Classpath` field, be sure to separate the jar names with spaces.

If you still have the problem, you have a situation where a class is being loaded from the hard drive instead of the ear file. This is a very difficult situation to debug because the offending class is not the one specified in the exception. Instead, another class is loaded from the hard drive before the one specified in the exception. To correct this, review the classpath specified with the `-CCclasspath` option and the classpaths configured with the [Application Client Resource Configuration Tool](#). Look for classes that also exist in the ear file. You must resolve the situation where one of the classes is found on the hard drive instead of in the .ear file. You do this by removing entries from the classpaths or by including the .jar files and classes in the .ear file instead of referencing them from the hard drive.

If you are using the `-CCclasspath` parameter or `resourceclasspaths` in the Application Client Resource Configuration Tool, and you have configured multiple jars or classes, verify they are separated with the correct character for your operating system. Unlike the classpath field in the Application Assembly Tool, these classpath fields use platform-specific separator characters, usually a colon (on UNIX platforms) or a semi-colon (on Windows).

Note: The system classpath is not used by the Application Client runtime if you use the `launchClient` batch or shell files. In this case, the system classpath would not cause this problem.

However, if you load the `launchClient` class directly, you do have to search through the system classpath as well.

[Return](#)

Error: `com.ibm.websphere.naming.CannotInstantiateObjectException: Exception occurred while attempting to get an instance of the object for the specified reference object. [Root exception is javax.naming.NameNotFoundException: xxxxxxxxxxxx]`

Explanation: This exception occurs when you perform a lookup on an object that is not installed on the host server. Your program can look up the name in the local client JNDI name space, but received a `NameNotFoundException` exception because it is not located on the host server. One typical example is looking up an EJB that is not installed on the host server that you access. This exception might also occur if the JNDI name you configured in your `ApplicationClient` module does not match the actual JNDI name of the resource on the host server.

Possible causes:

- Incorrect host server invoked
- Resource is not defined
- Resource is not installed
- Application server is not started
- Invalid JNDI configuration

User response: If you are accessing the wrong host server, run the `launchClient` command again with the `-CCBootstrapHost` parameter specifying the correct host server name. If you are accessing the correct host server, use the WebSphere `dumpnamespace` command line tool to see a listing of the host server's JNDI name space. If you do not see the failing object's name, the resource is either not installed on the host server or the appropriate application server is not started. If you determine the resource is already installed and started, your JNDI name in your client application does not match the global JNDI name on the host server. Use the [Application Assembly Tool](#) to compare the JNDI bindings value of the failing object's name in the client application to the JNDI bindings value of the object in the host server application. They must match.

[Return](#)

Error: `javax.naming.ServiceUnavailableException: Caught exception when resolving initial reference=NameService. Root exception is org.omg.CORBA.INTERNAL: JORB00105: In Profile.getIPAddress(), InetAddress.getByName(invalidhostname) threw an UnknownHostException minor code: 0 completed: No`

Explanation: This exception occurs when you specify an invalid host server name.

Possible causes:

- Incorrect host server invoked
- Invalid host server name

User response: Run the `launchClient` command again and specify the correct name of your host server with the `-CCBootstrapHost` parameter.

[Return](#)

Error: javax.naming.CommunicationException: Caught CORBA.COMM_FAILURE when resolving initial reference=WsnNameService. Root exception is org.omg.CORBA.COMM_FAILURE: minor code: 3 completed: No

Explanation: This exception occurs when you run `launchClient` to a host server that does not have the Application Server started. You also receive this exception when you specify an invalid host server name. This might happen if you do not specify a host server name when you run `launchClient`. The default behavior is for `launchClient` to run to `localhost`, because WebSphere Application Server does not know the name of your host server. This default behavior only works when you are running the client on the same computer as where WebSphere Application Server is installed.

Possible causes:

- Incorrect host server invoked
- Invalid host server name
- Invalid reference to `localhost`
- Application server is not started
- Invalid bootstrap port

User Response: If you are not running to the correct host server, run the `launchClient` command again and specify the name of your host server with the `-CCBootstrapHost` parameter. Otherwise, start the Application Server on the host server and run the `launchClient` command again.

[Return](#)

Error: javax.naming.NameNotFoundException: Name comp/env/ejb not found in context "java:"

Explanation: This exception is thrown when Java cannot locate the specified name in the local JNDI name space.

Possible causes:

- No binding information for the specified name
- Binding information for the specified name is incorrect
- Wrong class loader was used to load one of the program's classes

User Response: Open the ear file with the [Application Assembly Tool](#) and check the bindings for the failing name. Ensure this information is correct. If it is correct, you could have a [class loader issue](#).

[Return](#)

Error: java.lang.ClassCastException: Unable to load class: org.omg.stub.WebSphereSamples.HelloEJB._HelloHome_Stubat com.ibm.rmi.javax.rmi.PortableRemoteObject.narrow(portableRemoteObject.java:269)

Explanation: This exception occurs when the application program attempts to narrow to the EJB's home class and the classloaders cannot find the EJB's client side bindings.

Note: The `HelloHome_Stub` reference in the **Error** description, is a sample

Possible causes:

- The files, `*_Stub.class` and `_Tie.class`, are not in the EJB .jar file
- Classloader could not find the classes

User Response: Look at the EJB .jar file located in the .ear and verify the class contains the EJB client side

bindings. These are class files whose names end in `_Stub` and `_Tie`. If these files are not present, then use the [Application Assembly Tool](#) to generate the binding classes. For more information, see article [Generating deployment code for modules](#). If the binding classes are in the `EJB.jar` file, then you might have a classloader issue.

[Return](#)

Error: WSCL0206E: File [EAR file name] is not a valid Enterprise Archive file.
WSCL0100E: Exception received:
`com.ibm.etools.archive.exception.OpenFailureException`

Explanation: This error occurs when the Application Client runtime cannot read the Enterprise Archive file.

Possible cause: The most likely cause of this error is the EAR file cannot be found in the path specified on the `launchClient` command.


User Response: Verify that the path and filename specified on the `launchClient` command are correct. If you are running on Windows NT and the path and file name are correct, use a short version of the path and file name (8 character file name and 3 character extension).

[Return](#)

4.7.2.4: J2EE application client classloading overview

When you run your J2EE client application using the WebSphere Application Server [launchClient](#) command, a hierarchy of classloaders is created to load classes used by your application. The parent classloader is used to load the WebSphere Application Client runtime and any classes placed in the WebSphere Application Client user directories. The directories used by this classloader are defined by the `WS_EXT_DIRS` System property in the [product_installation/bin/setupcmdline](#) command shell.

As the J2EE Application Client runtime initializes, additional classloaders are created as children of this parent classloader. If your client application uses resources such as JDBC, JMS, or URLs, a different classloader is created to load each of those resources. Finally, a classloader is created to load classes within the `.ear` file. Before invoking your client application's main method, this classloader is set as the thread's context classloader.

 The system classpath is never used and is not part of the hierarchy of classloaders.

In order to package your client application correctly, you must understand which classloader loads your classes. When Java loads a class, the classloader used to load that class is assigned to it. Any classes subsequently loaded by that class will use that classloader or any of its parents, but it will not use children classloaders.

Unfortunately, Java does not provide a good way for determining which classloader loaded your classes. This makes it difficult to debug classloading problems. See the [Configuring the classpath fields](#) section for more information on configuring the classpath fields in your application.

In some cases the WebSphere Application Client runtime can detect when your client application class is loaded by a different classloader from the one created for it by the WebSphere Application Client runtime. When that occurs, you will see message:

WSCL0205W: The incorrect class loader was used to load {0}.

This message occurs when your client application class is loaded by one of the parent classloaders in the hierarchy. This is typically caused by having the same classes in the `.ear` file and on the hard drive. If one of the parent classloaders locates a class, that classloader will load it before the Application Client runtime classloader. In some cases, your client application will still function correctly. In most cases, however, you will receive "*class not found*" exceptions.

Configuring the classpath fields

When packaging your J2EE client application, you must configure various classpath fields. Ideally, you should package everything required by your application into your `.ear` file. This is the easiest way to distribute your J2EE client application to your clients. However, you should not package such resources as JDBC, JMS, or URLs. In the case of these resources, you want to use classpath references to access those classes on the hard drive. You might also have other classes installed on your client machines that you do not need to redistribute. In that case, you also want to use classpath references to access the classes on the hard drive, as described below.

Referencing classes within the EAR file

WebSphere J2EE applications do not use the system path. Instead, use the MANIFEST Class-Path entries to refer to other classes within the `.ear` file. Configure these values using the module Classpath fields in the [Application Assembly Tool](#). For example, if your client application needs to access an Enterprise Java Bean, you would add the deployed EJB module's name to your application client's Classpath field in the [Application Assembly Tool](#). The format of the Classpath field for each of the different modules (Application Client, EJB, Web) is the same:

- The values must refer to `.jar` and `.class` files that are contained within the `.ear` file.

- The values must be relative to the root of the `.ear` file.
- The values cannot refer to absolute paths in the file systems.
- Multiple values must be separated by spaces, not colons or semi-colons.

Note: This is Java's method for allowing applications to be platform-independent.

Typically, you add modules (`.jar` files) to the root of the `.ear` file. In this case, you only need to specify the name of the module (`.jar` file) in the Classpath field. If you choose to add a module with a path, you need to specify the path relative to the root of the `.ear` file.

For referencing `.class` files, you must specify the directory relative to the root of the `.ear` file. While the Application Assembly Tool allows you to add individual class files to the `.ear` file, it is recommended that these additional class files are packaged in a `.jar` file. That `.jar` file should then be added to the module Classpath fields. If you add `.class` files to the root of the `.ear` file, add `"/"` to the module Classpath fields.

Consider the following example directory structure in which the file `myapp.ear` contains an application client JAR file named `client.jar` and an EJB module called `mybeans.jar`. Additional classes reside in `class1.jar` and `utility/class2.zip`. A class named `xyz.class` is not packaged in a JAR file but is in the root of the EAR file.

Specify `"./ mybeans.jar utility/class2.zip class1.jar"` as the value of the Classpath property.

The search order is:

```
myapp.ear/client.jar
myapp.ear/mybeans.jar
myapp.ear/class1.jar
myapp.ear/utility/class2.zip
myapp.ear/xyz.class
```

View article the 6.4.1: Setting classpaths for more information.

Referencing classes that are not in the EAR file

You have two options to reference classes that are not contained in the `.ear` file. Which option you choose depends on the relationship of the external classes and the classes internal to the `.ear` file. You might use a combination of both options. Your options are:

1. Use the [product_installation/app](#) directory.

Use this option when your external classes do **not** reference classes within the `.ear` file. One example would be stand-alone utility classes. To use this option, add your `.jar` files to the [product_installation/app](#) directory. For `.class` files, add them to this directory in subdirectories that correspond to the package names.

2. If the external classes reference classes within the `.ear` file, the first option will not work because of the hierarchy of WebSphere classloaders. In this case, you can do one of the following:
 - Package the external classes in the `.ear` file.
 - Use the `launchClient -CCclasspath` parameter.

This parameter is specified at run-time and takes platform-specific classpath values, which means multiple values are separated by semi-colons or colons.

Refer to article 6.4.1 about installing application files into the environment, and setting classpaths, for a description of the WebSphere Application Server classloaders. There are many similarities between the client and the server in this respect.

Resource classpaths

When you configure resources used by your client application using the [Application Client Resource Configuration Tool](#), you can specify classpaths that are required by the resource. For example, if your application is using JDBC to a DB2 database, you want to add `db2java.zip` to the classpath field of the database provider. These classpath values are platform-specific and require semi-colons or colons to separate multiple values.

Using the launchClient API

If you use the [launchClient](#) shell/bat command, the WebSphere classloaderhierarchy is created for you. However, if you use the launchClient API, you must perform this setup yourself. You should mimic the `launchClient` shell command in defining the Java system properties.

4.7.3: Java thin application client programming model

The Java application thin client provides the user a light weight, downloadable Java application runtime that is capable of interacting with Enterprise Java Beans. This client is designed to support those users who want a light weight Java client application programming environment without the overhead of the J2EE platform on the client machine. The programming model for this client is heavily influenced by the CORBA programming model but supports access to Enterprise Java Beans. When accessing Enterprise Java Beans from this client, the EJB object references can be considered CORBA object references by the client application.

There is no tooling support for this client for developing, assembling or deploying the client application. The user is responsible for developing the client application, generating the necessary client bindings for the EJB and CORBA objects, and bundling these pieces together to be installed on the client machine.

Client side bindings for Enterprise Java Beans are generated during the deployment phase of J2EE development using the [Application Assembly Tool](#). A Java application can utilize these bindings or you can generate client side bindings using the *rmic* command that is part of the IBM JDK installed with WebSphere Application Server. See article [Packaging and distributing Java clients](#) for more information.

The Java application thin client provides the necessary runtime to support the communication needs between the client and the server.

The Java application thin client makes use of the RMI-IIOP protocol. The use of this protocol enables the client application to access not only EJB references and CORBA object references, but it also allows the client application to make use of any supported CORBA services. Use of the RMI-IIOP protocol and the accessibility of CORBA services can assist a user in developing a client application that needs to access both EJB references and CORBA object references. When users combine the J2EE and CORBA environments in one client application, they need to understand the differences between the two programming models, and they must use and manage each appropriately.

The Java application thin client runtime provides the necessary support for the client application for object resolution, security, RAS and other services. However, it does not support a container that provides ease of use to these services. For instance, there is no support for the use of "nicknames" for EJB or local resource resolution. When resolving to an EJB (using either JNDI or CosNaming) the client application must know the location of the name server and the fully qualified name that was used when the reference was bound into the namespace. When resolving to a local resource, the client application cannot resolve to the resource through a JNDI lookup. Instead the client application must explicitly create the connection to the resource using the appropriate API (JDBC, JMS, etc.). This client does not perform any initialization of any of the services that the client application might require. For instance, the client application is responsible for the initialization of the naming service, either through CosNaming or JNDI.

The following table describes the advantages and disadvantages of the Java thin application client:

Advantages	Disadvantages
<ul style="list-style-type: none">● A light-weight client suitable for download● Requires access to CosNaming or JNDI interfaces for EJB or CORBA object resolution	<ul style="list-style-type: none">● Designed for use in an intranet environment● Lack of client runtime initialization of environment and services● Lack of built-in support for local resource resolution and configuration● Does not promote portability of client application code● Requires distribution of the application to the client machine.

4.7.3.1: Developing a Java application thin client

Install the Java application thin client from the *WebSphere Application Client* CD by selecting option "Java Application Thin Client" or "Java Application/Applet Thin Client."

The Java application thin client offers access to most of the client services that are available in the J2EE application client; however, these services are not as easily accessed in the thin client as they are in the J2EE application client. The J2EE client has the advantage of performing a simple JNDI namespace lookup to access the desired service or resource. The thin client must code explicitly for each resource in the client application. For example, looking up an EJB Home requires the following code in a J2EE application client:

```
java.lang.Object ejbHome = initialContext.lookup("java:/comp/env/ejb/MyEJBHome");
MyEJBHome = (MyEJBHome) javax.rmi.PortableRemoteObject.narrow(ejbHome, MyEJBHome.class);
```

However, the code in a Java thin application client must be more explicit:

```
java.lang.Object ejbHome =
initialContext.lookup("the/fully/qualified/path/to/actual/home/in/namespace/MyEJBHome");
MyEJBHome = (MyEJBHome) javax.rmi.PortableRemoteObject.narrow(ejbHome, MyEJBHome.class);
```

In this example, the J2EE application client accesses a logical name from the `java:/comp` namespace. The J2EE client runtime resolves that name to the physical location, and returns the reference to the client application. The thin client, on the other hand, must know the fully qualified physical location of the EJB Home in the namespace. If this location changes, the thin client application also must change the value placed on the `lookup()` statement. In the J2EE client, the client application is protected from these changes because it makes use of the logical name. A change might require a re-deploy of the EAR file, but the actual client application code remains the same.

The Java thin application client is a traditional Java application that contains a "main" function. WebSphere's Java thin application client provides runtime support for accessing remote EJBs, and provides the implementation for various services (security, WLM, and others). This client can also access CORBA objects and CORBA based services. When using both environments in one client application, the user is responsible for understanding the differences between the EJB and CORBA programming models and for managing both environments.

For instance, the CORBA programming model requires using the CORBA CosNaming name service for object resolution in a namespace while the EJB programming model requires using the JNDI name service. The client application must initialize and properly manage the use of these two naming services.

Another difference applies to the EJB model. The ORB is initialized using JNDI implementation in the EJB model, and the client application is unaware that an ORB is present. The CORBA model, however, requires the client application to explicitly initialize the ORB through the `ORB.init()` static method.

The Java application thin client provides a batch command that you can use to set the `CLASSPATH` and `JAVA_HOME` environment variables to enable the Java application thin client runtime.

Set the Java application thin client environment by using the *setupClient* shell, located in:

```
product_installation\AppletClient\bin\setupClient.bat (on Windows)
product_installation/AppletClient/bin/setupClient.sh (on UNIX platforms)
```

After setting the environment variables, add your specific Java client application JAR files to the `CLASSPATH` and start your Java client application from this environment.

See article [Packaging and distributing Java clients](#) for more information.

4.7.3.2: Java thin application client code example

The code required by a Java application thin client to communicate with an enterprise java bean is similar to servlet code that communicates with enterprise java beans.

The following code example illustrates how a Java application thin client uses the `InitialContext` to do the following:

- Perform a lookup
- Narrow the returned object into the `EJBHome` object
- Invoke the `create` method.

Click a link to view the referenced line of code in the example. Each line in the code snippet is described in this next section.

1. The first three lines in the [try](#) section of the code example show how to:

- [Create a properties class](#)
- [Set the initial context factory](#)
- [Define the provider URL used during the lookup operation](#)

2. The fields in the [provider URL](#) represent:

`iiop://myComputer.myDomain.com:900`

iiop://	myComputer	myDomain.com	900
protocol	name of the server where WebSphere Application Server is installed	name of the domain for the server where WebSphere Application Server is installed	configured port  Since port 900 is the default port value, this may be omitted.

3. This line in the example shows how to:

[create an InitialContext class passing the Properties file](#)

4. Now do a [lookup the EJB Home on the server](#)

For more information on JNDI, see article [4.6.1: JNDI overview](#).

5. The narrow operation in this line:

[safely casts the object into an instance of HelloHome](#)

6. Finally, [call the create method on the HelloHome object to create a Hello object](#).

You can also use `findByPrimary` key instead of `create`. Use the `findByPrimaryKey` method to find an existing Hello object.

Code example

```

import javax.naming.*;
import javax.rmi.*;
import java.rmi.*;
import java.util.*;
import javax.ejb.*;
import WebSphereSample.HelloEJB.*; //package for HelloEJB beans
public class HelloClient
{
    public static void main(String argv[])
    {
        try
        {
            Properties props = new Properties();
            props.put(Context.INITIAL_CONTEXT_FACTORY,
"com.ibm.websphere.naming.WsnInitialContextFactory");
            props.put(Context.PROVIDER_URL,
"iiop://myComputer.myDomain.com:900");
            InitialContext ctx = new InitialContext(props);
            Object myObj = ctx.lookup("WSSamples/HelloEJBHome");
            HelloHome myHome = (HelloHome)
            javax.rmi.PortableRemoteObject.narrow(obj, HelloHome.class);
            Hello hello = myHome.create();
        }
        catch(NamingException e)
        {
            ....
        }
        catch(RemoteException e)
        {
            ....
        }
        catch(CreateException e)
        {
            ....
        }
    }
}

```

Learn more about the WebSphere Java application thin client by running the client sample. You can install the client sample from the *WebSphere Application Client* CD. This sample is called HelloEJB and is installed in the [product_installation](#)/WSSamples/Clients subdirectory on the client machine.

4.7.4: Quick reference to Java client functions

Use the following table to identify the available functions in the different types of Java client.

Available functions	Applet client	J2EE Application client	Java thin application client
Provides all the benefits of a J2EE platform	No	Yes	No
Portable across all J2EE platforms	No	Yes	No
Provides the necessary runtime to support communication between client and server	Yes	Yes	Yes
Allows the use of nicknames in the deployment descriptors	No	Yes	No
Supports use of the RMI-IIOP protocol	Yes	Yes	Yes
Supports use of the HTTP protocol	Yes	No	No
Enables development of client applications that can access EJB references and CORBA object references	Yes	Yes	Yes
Enables the initialization of the client application's runtime environment	No	Yes	No
Supports security authentication to Enterprise Java Beans	No	Yes	Yes
Supports security authentication to local resources	No	Yes	No
Requires distribution of application to client machines	No	Yes	Yes

4.7.5: Quick reference to Java client topics

Use the following table to locate additional Java client topics.

Click a *Topics* entry to view a description. Click a *References* entry for additional information on that topic.

Topics	References
Java clients overview	<ul style="list-style-type: none">● J2EE application model● J2EE architecture● Quick reference to Java client functions
Applet clients	<ul style="list-style-type: none">● Developing an Applet client● Packaging, distributing, and installing● Tracing and logging
J2EE application clients	<ul style="list-style-type: none">● Resources referenced by a J2EE application client● Developing a J2EE application client● Troubleshooting guide for a J2EE application client● J2EE application client client class loading overview● Packaging, distributing, and installing● Tracing and logging● Configuring application client resources● Launching Java application clients in the J2EE application client container● Assembling modules and setting properties for applications● Assembling J2EE application modules (.ear files) with the application assembly tool
Java thin application clients	<ul style="list-style-type: none">● Developing a Java thin application client● Java thin application client code example● Packaging, distributing, and installing● Tracing and logging

4.7.6: Packaging and distributing Java client applications

After a client application has been developed the next step in the process is packaging and distributing the client application for use on client machines. Packaging consists of pulling together the various artifacts that the client application requires. Distributing applies to making the client application available on the target client machines. Each of the WebSphere Java clients differ slightly from each other in the packaging and distributing phases of the development process. These differences are described below.

Application client type	Packaging	Distribution
J2EE Application Client	<p>Packaging of the WebSphere J2EE Application Client is accomplished through the Application Assembly Tool (AAT). This tool generates an Enterprise Archive (.ear) file as the output from the the assembly and deployment process. The .ear file contains all of the class files that are required by the client application to run.</p> <p>The .ear file must be deployed. The deployment can be done through the Application Assembly Tool or when the EJB modules within the .ear file are installed in WebSphere Application Server. The deployment phase generates the client bindings needed by the client application.</p>	<p>Distributing the J2EE Application Client .ear file to the target client machine that has WebSphere J2EE Application Client installed, is a manual process.</p> <p>WebSphere Application Server does not provide any tooling to assist in the distribution of the J2EE Application Client .ear files.</p> <p>The J2EE Application Client might require further configuration if the application makes use of external resources (such as: JDBC, JavaMail, JMS or URL). Perform this configuration with the Application Client Resource Configuration Tool.</p> <p>When the resource configuration is complete, the application can be started using the launchClient command.</p>
Java Application Thin Client	<p>The Java application thin client is packaged by manually collecting appropriate JAR files and Java classes to support the client application.</p> <p>The Enterprise Java Beans that the client application uses, require that client side bindings are available on the client target machine. The client side bindings are available from the deployed EJB JAR files.</p> <p>The .jar files, containing the Enterprise Java Beans, are invoked by the application. These JAR files are located in directory: product_installation\InstalledApps\<YourEJBapplication.ear>\</p>	<p>Distributing the Java application thin client JAR files to the target client machine where WebSphere Java Thin Application Client is installed, is a manual process.</p> <p>WebSphere Application Server does not provide any tooling to assist in the distribution of the JAR files.</p> <p>When the client application files are present on the target client machine, you must set up the environment. WebSphere Application Server provides a command that assists users in setting up the environment by defining several environment variables. Use the setupClient command located in the product_installation\bin directory. After running this command, add your client application JAR files to the CLASSPATH or use the -classpath parameter on the java command.</p>

Applet Client	<p>The Applet client is packaged manually by collecting the appropriate JAR files, Java classes, and HTML files to support the Applet.</p> <p>The Enterprise Java Beans that the Applet uses, require client side bindings. The client side bindings are available from the deployed EJB JAR files. What . jar files are used depends on the Enterprise Java Beans invoked. These JAR files are located in the product_installation\InstalledApps\<YourEJBapplication.ear>\ directory.</p>	<p>Distributing the Applet client to the target client machine that has the Applet client installed or to the target WebServer machine (if you want to make the Applet available for download), is a manual process.</p> <p>WebSphere Application Server does not provide any tooling to assist in the distribution of the JAR files.</p>
----------------------	--	---

4.7.7: Tracing and logging for the Java clients

Tracing and logging functions are available for the WebSphere client runtime. How this support is enabled and the level of support provided, differs for each client model.

• Applet client

You enable the tracing and logging functions for ORB level tracing only, by specifying the following system properties in the Java *Runtime parameters* field of the WebSphere Application Server Java Plug-in Control Panel:

```
-Dcom.ibm.CORBA.CommTrace=true  
-Dcom.ibm.CORBA.Debug=true
```

All verbose, trace, and debug messages are sent to the Java console window on the browser. Applets restrict using files for trace output.

• J2EE Application Client


You enable the tracing and logging functions by specifying one of the following flags on the [launchClient](#) command when starting the J2EE client application:

- [CCtrace](#)
- [CCtracefile](#)

CCtrace flag

The `-CCtrace` flag enables trace. You can trace all or specific components:

- `-CCtrace=true`
(This flag enables trace for all components and all events.)
- `-CCtrace=com.ibm.<component>=<entryexit | debug | event | all>=enabled`
(This flag enables trace for specific components. For example,
`-CCtrace=com.ibm.ws.client.*=all=enabled` enables trace for all loggers with names starting with `com.ibm.ws.client`.)

 If the `-CCtrace` flag is not specified, trace is disabled.

CCtracefile flag


Use the `-CCtracefile` flag to send the trace output to a specific file:

```
-CCtracefile=<fully_qualified_output_filename>
```

(For example,

```
-CCtracefile=c:\MyTraceFile.log
```

 directs the trace output to file, `c:\MyTraceFile.log`.)

 If the `-CCtracefile` flag is not set, all output is directed to stdout.

• Java thin application client

You enable the tracing and logging functions by specifying the following system property on the `java` command when starting the client application:

```
-DtraceSettingsFile=<filename>
```

(Filename is the name of a properties file that must be placed in the classpath accessible by the application.)

The properties file is used for specifying the output file and the components to enable for trace. When you install WebSphere Application Server, a sample trace settings properties file is provided in:


```
<product_installation>/properties/TraceSettings.properties
```

The `TraceSettings.properties` file looks like the following example:

```

# property to specify the fully qualified file name for the tracefile
traceFileName=c:\\MyTraceFile.log
# Specify trace strings here. Trace strings take the form of:
# logger={level}={type} where:
#     level = entryexit || debug || event || all
#     type =  enabled || disabled
# examples:
# com.ibm.ejs.ras.SharedLogBase=all=enabled enables all tracing for the single logger
#     created in class com.ibm.ejs.ras.SharedLogBase.
# com.ibm.ejs.*=debug=enabled enables debug tracing for all loggers with names starting
#     with com.ibm.ejs.
## Multiple trace strings can be specified, one per line.
com.ibm.ejs.ras.*=all=enabled

```

 If you specify a filename but no trace string, only message events are written to the specified file. If you specify a filename and a trace string, both message events and diagnostic trace entries are written to the specified file. If you do not specify a filename for the trace file, all output is directed to stdout.

4.8: Web services - an overview

Web services are self-contained, modular applications that can be described, published, located, and invoked over a network. Web services could be weather reports or stock quotes. Transaction Web services, supporting business-to-business (B2B) or business-to-client (B2C) operations, could be airline reservations or purchase orders.

Web services reflect a new "service-oriented" approach to programming, based on the idea of building applications by discovering and implementing network-available services, or by invoking available applications to accomplish some task. This "service-oriented" approach is independent of specific programming languages or operating systems. Instead, Web services rely on pre-existing transport technologies (such as HTTP) and standard data encoding techniques (such as XML) for their implementation.

The Web services architecture describes three roles:

1. [Service provider](#)
2. [Service requester](#)
3. [Service broker](#)

Web services components provide three basic operations:

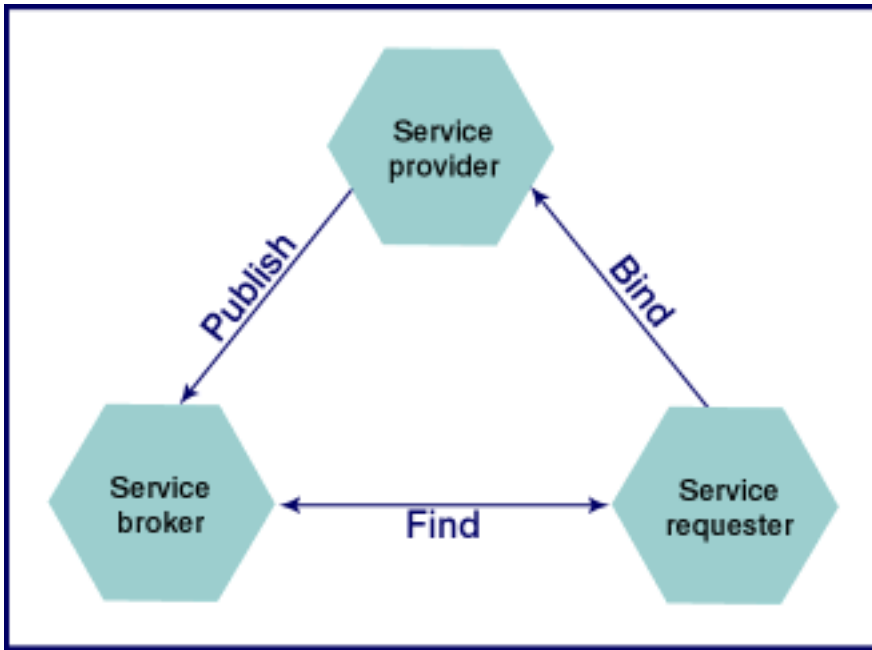
1. [Publish](#)
2. [Find](#)
3. [Bind](#)

In order for some component to become a Web service, it must be:

- Created, and its interfaces and invocation methods must be defined
- Published to some repository
- Easy to locate by potential users
- Invoked and implemented by users
- Unpublished when it is no longer available

As illustrated in the graphic,

- Web service descriptions can be created and published by service providers who create on-line resources for personal and business use.
- Web services can be categorized and searched by specific broker services.
- Web services can be located and invoked by requesters of the services.



With Web services, programming complexity is reduced because application designers do not have to worry about implementing the services they are invoking. Interactions in Web services are bound dynamically at runtime. A service requester describes the features of the required service and uses the service broker to find an appropriate service.

WebSphere Application Server supports making the following artifacts into Web services:

- Java beans
- Enterprise Java Beans
- BSF supported scripts
- DB2 stored procedures

See article [Web services components](#) for a description of the key components that comprise a Web service.

Visit URL, www.alphaworks.ibm.com/tech/webservicestoolkit, to access the Web services toolkit on Alphaworks. This site provides tools for creating WSDL files and SOAP clients, and describes working examples.

Learn more about Web services. Register for the [Web services tutorial](#) on Alphaworks.

4.8.1: Web services components

These are the key components of a Web service:

- [SOAP](#) (simple object access protocol)
- [WSDL](#) (Web Services Description Language)
- [UDDI](#) (Universal Discovery , Description and Integration Protocol)
- [UDDI4J](#) (client version of UDDI)

• SOAP or Simple Object Access Protocol

is a new protocol created by IBM, Microsoft, Userland, and DevelopMentor to support remote procedure calls and other requests over HTTP. Built on HTTP and XML, SOAP attempts to convert application servers into object servers.


See the [W3C SOAP protocol site](#) for more information on SOAP messages, supported datatypes, and attributes. For SOAP implementation guidelines, visit the [Apache site](#).

SOAP requests and the responses are XML based. The following examples illustrate a SOAP request and response:

Sample SOAP Request	
Sample SOAP Request	POST /Supplier HTTP/1.1 Host: www.somesupplier.com Content-Type: text/xml; charset="utf-8" Content-Length: nnnn SOAPAction: "Some-URI" <SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"><SOAP-ENV:Body><m:OrderItem xmlns:m="Some-URI"><RetailerID>557010</RetailerID><ItemNumber>1050420459</ItemNumber><ItemName>AMF Night Hawk Pearl M2</ItemName><ItemDesc>Bowling Ball</ItemDesc><OrderQuantity>100</OrderQuantity><WholesalePrice>130.95</WholesalePrice><OrderDateTime>2000-06-19 10:09:56</OrderDateTime></m:OrderItem></SOAP-ENV:Body></SOAP-ENV:Envelope>

The SOAP request indicates that the OrderItem method, from the "Some-URI" namespace, should be invoked from <http://www.somesupplier.com/Supplier>. Upon receiving this request, the supplier application at www.somesupplier.com executes the business logic that corresponds to OrderItem.

The SOAP protocol does not specify how to process the order. The supplier could run a CGI script, invoke a servlet, or perform any other process that generates the appropriate response.

 See article [SOAP support](#) for the list of artifacts that WebSphere Application Server supports as Web services.

In this example, the SOAP Envelope element is the top element of the XML document that represents the SOAP message. The reference to the XML namespace (xmlns:m="Some-URI") specifies the namespace to use for the SOAP identifiers. This request is asking the application to place an order for the item identified by the elements:

- RetailerID
- ItemNumber
- ItemName
- ItemDesc
- OrderQuantity
- WholesalePrice
- OrderDateTime

The response comes in the form of an XML document that contains the results of the processing, in this case, the order number for the order placed by the retailer. The response is sent by the service provider located at <http://www.somesupplier.com/Supplier>.

Sample SOAP Response	
HTTP/1.1 200 OK Content-Type: text/xml; charset="utf-8" Content-Length: nnnn	<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"><SOAP-ENV:Body><m:OrderItemResponse xmlns:m="Some-URI"><OrderNumber>561381</OrderNumber></m:OrderItemResponse></SOAP-ENV:Body></SOAP-ENV:Envelope>

The response does not include a SOAP-specified header. The results are placed in an element whose name matches the method name (*OrderItem*) with the suffix, "Response" as in *OrderItemResponse*.


Although Apache SOAP allows for SOAP over SMTP, WebSphere Application Server only supports SOAP over HTTP.

The SOAP Javadoc is shipped with WebSphere Application Server.

Review [WebSphere Application Server's Javadoc](#) for SOAP implementation details.

• WSDL or Web Services Description Language

is an XML-based interface definition language that provides operational information about a service, such as the service interface, implementation details, access protocol, and contact endpoints. Compliant server applications must support these interfaces, and client users can learn from the document how a service should be accessed.

 WebSphere Application Server does not provide tools for generating WSDL files.

View a WSDL representation in the [AddressBook2 sample](#).

See article [UDDI4J samples](#) for more information.

Review the WSDL specifications at [W3C WSDL protocol site](#).

• UDDI or Universal Discovery Description and Integration (Project)

is a comprehensive, open industry initiative enabling businesses to:

1. Discover each other
2. Define how they interact over the Internet, and share information in a global registry architecture.

WebSphere Application Server does not provide a private UDDI directory. IBM, among others, provides public UDDI registries. For more information about UDDI, see www.uddi.org. Also visit [Alphaworks](#) for the Web services toolkit, which includes an IBM implementation of a private UDDI registry.

UDDI is the building block which enables businesses to quickly, easily, and dynamically find and transact with one another by means of their preferred applications.

As described in the [Web services overview](#), UDDI provides the three basic Web services functions: publish, find, and bind.

• UDDI4J

is an open-source Java implementation of the Universal Discovery, Description, and Integration protocol (UDDI). UDDI4J contains an implementation of the client side of UDDI (everything your application needs to publish, find, and bind a Web service). It also includes the source code, and the complete Javadoc for the APIs. For more information, visit the UDDI4J open source site at oss.software.ibm.com/developerworks/projects/uddi4j.

Review [IBM's Javadoc](#) for UDDI4J implementation details.

4.8.1.1: UDDI4J Overview

UDDI4J is a Java class library that provides an API that is used to interact with a UDDI registry. This class library generates and parses messages sent to and received from a UDDI server.

The central class in this set of APIs is:

```
com.ibm.uddi.client.UDDIProxy
```

This class is a proxy for the UDDI server that is accessed from the client code. Its methods map to the [UDDI Programmer's API Specification](#). Review [IBM's Javadoc](#) for additional implementation details.

The classes within `com.ibm.uddi.datatype` represent data objects that send or receive UDDI information. In the business and service model, the data objects are also known as subpackages.

The subpackage `com.ibm.uddi.request` contains messages sent to the server. Generally, these classes are not used directly; rather, they are invoked by the `UDDIProxy` class.

Similarly, the subpackage `com.ibm.uddi.response` represents response messages from a UDDI server.

UDDI4J error handling

When invoking `UDDIProxy` inquiry methods, `UDDIException` is thrown when errors are received from the UDDI proxy. `UDDIException` can contain a `DispositionReport` with information regarding the error.

APIs that do not return a data object, provide the disposition report.

`SOAPException` is thrown if a communication error occurs or if the resulting data cannot be parsed as a valid SOAP message.

View the file [4.8.1.1.1: UDDI4J Samples](#) for API usage examples.

For more information, visit the UDDI4J open source site at oss.software.ibm.com/developerworks/projects/uddi4j.

4.8.1.1.1: UDDI4J samples

A set of samples is provided to illustrate using the inquiry and publish APIs, and to demonstrate error handling.

Note: WebSphere Application Server does *not* provide a UDDI registry. The IBM UDDI test registry is located at www.ibm.com/services/uddi/

Any sample that requires you to "publish," "save," or "delete" requires a userid and password. You can only invoke the "find" sample without a userid and password.

To get a userid and password:

1. Access the UDDI test registry
2. Register for your userid and password


The registration process requires you to activate your id before attempting to use the publish or delete examples.

Note: If the registry is not operational, keep trying. This is a test registry and at times it is not available.

3. Use your registered userid and password when running the *SaveBusinessExample* and *DeleteBusinessExample* samples.

Your samples consist of:

- *FindExample* - is the "Hello world" of UDDI programs. It is the simplest sample of the UDDI API.
- *SaveBusinessExample* - is an example of using the publish API. It logs into the server using the `get_authToken` method; then attempts to save a business.
- *DeleteBusinessExample* - searches for a particular business using the inquiry API, finds the associated businesskey, logs into the server, and then attempts to delete the business it found.

 When running *DeleteBusinessExample*, you might receive the following error messages:

```
Get authTokenReturned authToken:ADA3DC40-2531-11D5-9EB0-832611502FD0Search for 'Sample business' to
deleteFound business key:D3DD4036-00E4-F124-050B-C6113996AA77Errno:10140  ErrCode:E_userMismatch
ErrText:E_userMismatch (10140)  Cannot change data that is controlled by another party.
businessEntity = D3DD4036-00E4-F124-050B-C6113996AA77Found business
key:61AE2CC0-0F2C-11D5-BC1E-B763254A2930Errno:10140  ErrCode:E_userMismatch
ErrText:E_userMismatch (10140)  Cannot change data that is controlled by another party.
businessEntity = 61AE2CC0-0F2C-11D5-BC1E-B763254A2930Found business
key:3BB274CF-00E3-FA94-9B72-C6113996AA77
```

This is not a problem with the sample. *DeleteBusinessExample* issues a query for the business name specified in the code and receives a list of entries with that name. The sample then tries to delete each entry in the list. These error messages occur when the sample tries to delete entries that you do not own.

Accessing the samples

To access these samples, you can either install the `soapsamples.ear`, or you can expand the `soapsamples.ear` using the `EarExpander` tool.

These are the steps to access the samples:

1. Create a directory to hold the expanded `soapsamples.ear` contents.
2. From the `product_installation_root\bin` directory, enter the following commands:

```
EarExpander -ear ..\installableApps\soapsamples.ear-expandDir ..\temp\soapsamples -operationexpand
-expansionFlags war
```
3. Issue the `cd` command to change to the `installedApps/soapsamples.ear` or to the target directory specified in the `expandDir` argument
4. Issue the `cd` command to change to `UDDISamples` directory. The source for the samples is included in the `src` directory.


The samples require several pieces of information. The sample source files can be edited and these values substituted. The required values are:

- **InquiryURL:** The URL of the UDDI server against which to run inquiries.
- **PublishURL:** The URL of the UDDI server to run publish requests. Typically, this is a SSL connection.
- **UserId:** When publishing, a userid is required for authentication.
- **Password:** This is the password for the referenced userid. Password is referred to as a credential in UDDI terminology.

Running the samples

WebSphere Application Server provides a number of UNIX scripts and DOS .bat files to run the samples. These scripts (or .bat files) add the required jar files to the classpath. Use a text editor (such as Notepad on Windows NT or VI or E3 on UNIX) to view the scripts (or .bat files). They describe the resources that you need to run the samples.

The scripts are located in directory `UDDISamples/unix_scripts`. On Windows NT, the .bat files are located in directory `UDDISamples\nt_bat`.

 The scripts are put in this location as a result of running the `EarExpander` command.

All the scripts (or .bat files) are named after the samples they run. So, for example, to invoke the *FindExample* sample, you would run the *FindExample.sh* script.

A UDDI registry might limit the number of business entities that you publish. The IBM Test registry limits you to one business entity. This means, for example, that after running the *SaveBusinessExample*, you must run the *DeleteBusinessExample* before attempting to publish another business entity.

See the related information links for an enablement scenario.

4.8.1.2: SOAP support

Version 2.2 of the Apache SOAP implementation is integrated into WebSphere Application Server Version 4.0. Apache SOAP Version 2.2 is a Java-based implementation of the SOAP 1.1 specification with support for SOAP with attachments.

WebSphere Application Server Version 4.0 allows you to expose the following artifacts as SOAP services:

- Standard Java classes
- Enterprise beans
- Bean Scripting Framework (BSF) supported scripts
- DB2 stored procedures

Tools are provided to assist you with deploying these artifacts as SOAP services. See article [Deploying a programming artifact as a SOAP accessible Web service](#) for more information.

As part of deploying your services, you can choose to enable the [XML-SOAP Admin tool](#), which allows you to manage your SOAP-enabled services.

WebSphere Application Server also contains an implementation of the security extensions for SOAP. These security extensions provide secure connections and enable digitally signed messages. See article [Securing SOAP services](#) for more information.

See the related information links for an enablement scenario.


4.8.1.2.1: SOAP samples

WebSphere Application Server 4.0 provides sample services and clients that demonstrate how to access SOAP services. The SOAP samples code is based on the the Apache SOAP 2.2 samples. These samples are contained in the `soapsamples.ear` that is located in the `installableApps` directory. The source for the sample services is located in the `soapsamples.ear`.

See article [DB2 Stored procedure sample setup](#) for information on configuring a datasource to set the `db2-userid` and `db2-password` entries.

Perform the following steps to install the samples in your server:

- 1. In a single-server configuration, do the following:
- 2. Change the directory to:
`product_installation_root/bin`
- 3. Install the EAR file by entering the following data at a command prompt:


 The line breaks in this example are added to make the information legible. This information really exists as one line of unformatted data.

```
Seappinstall -install ..\installableApps\soapsamples.ear -ejbdeploy false
-interactive false
```

- 4. To access the sample services from an external Web server, run the file `GenPluginCfg.sh` on UNIX or `GenPluginCfg.bat` on Windows NT. This file makes the Web server aware of the SOAP samples.
- 5. [Start the product](#).
- 6. Check on the availability of the sample services using the [XML-SOAP Admin tool](#):
 - a. From a browser, go to URL
`http://localhost/soapsamples/admin/index.html`
 - b. At this site, you can:
 - List available services
 - View the Apache SOAP descriptors
 - Stop and start sample services

Running the sample clients



Sample clients are provided to demonstrate how to access the installed SOAP services. These scripts require you to specify the server that will handle the request.

 If you run the script with **no** arguments, as for example *StockQuoteSample*, you will be provided with help on how to use the sample, and you will receive a description of the command line arguments that the script requires.

To access the samples, change the directory to the following on Windows NT:
`product_installation_root\installedApps\soapsamples.ear\ClientCode\nt_bat`

On UNIX platforms, the samples directory is:
`product_installation_root/installedApps/soapsamples.ear/ClientCode/unix_scripts`

 Issue the `chmod 755 *.sh` command to restore the execution permissions of the UNIX scripts.

Sample	Command (entered on a single line)
Stock quote (requires Internet access)	<code>stockquotesample localhost IBM</code>  If the request appears to hang, and then you receive an "Operation timed out" error, the service was unable to reach a server on the Internet to obtain the stock quote information. You may need a direct connection to the Internet.
Address book	<code>AddressBookSample GET localhost "John B. Good" AddressBookSample ALL localhost AddressBookSample PUT localhost "Herman Munster" 1313 "Mockingbird Lane" Salem MA 10013 111 222 3434</code>
Address book example 2	<code>Addressbook2sample localhost</code>
EJB	<code>EJBAdderSample localhost</code> On UNIX platforms, enter: <code>EJBAdderSample.sh localhost</code>
Send Message	<code>sendMessageSample localhost ..\data\msg1.xml</code>
Calculator Sample	<code>CalculatorSample localhost</code>  Unlike the other SOAP samples, which are either java or enterprise beans, the Calculator Sample is a JavaScript sample. The actual calculator processing is performed by the Web service.
Mime Client sample	<code>MimeClientSample localhost ..\data\foo.txt</code>

DB2SPSample sample	DB2SPSample localhost On UNIX platforms, enter: DB2SPSample.sh localhost
--------------------	--

Troubleshooting SOAP sample problems

If you cannot run the SOAP samples, check for the following problems:

- Can you run any of the samples, such as `http://localhost/servlet/snoop`? If not, make sure the Web server is running.

If you can run the `snoop` sample, try accessing one of the SOAP samples again, but this time specify the port number 9080 in addition to the host name, as for example:

```
MimeClientSample localhost:9080 ...data\foo.txt
```

If adding the port number resolves the problem, you need to update the plugin configuration by running the `GenPluginCfg.bat` file on the Windows platform, or the `GenPluginCfg.sh` file on UNIX platforms.

- If the `stockquote` sample fails but the other samples work, you are having problems accessing the external Internet.

See the Related topics section for links to an enablement tutorial.

4.8.1.2.2: Building a SOAP client

Creating clients to access the SOAP services published in WebSphere Application Server is a straightforward process. The Apache SOAP implementation, integrated with WebSphere Application Server, contains a client API to assist in SOAP client application development.

The SOAP API documentation is available in [WebSphere Application Server's javadoc](#).

These are the steps for creating a client that interacts with a SOAP RPC service:

1. **Obtain the interface description of the SOAP service**

This provides you with the signatures of the methods that you wish to invoke. You can either look at a WSDL file for the service, or view the service itself to see its implementation.

2. **Create the "Call" object**

The SOAP "Call" object is the main interface to the underlying SOAP RPC code.

3. **Set the target URI (Uniform Resource Identifier) in the "Call" object using the `setTargetObjectURI()` method.**

Pass the URN (Uniform Resource Name, a type of URI), that the service uses for its identifier, in the deployment descriptor.

4. **Set the method name that you want to invoke in the "Call" object using the `setMethodName()` method**

This method must be one of the methods exposed by the service located at the URN from the previous step.

5. **Create the necessary "Parameter" objects for the RPC call and then set them in the "Call" object using the `setParams()` method.**

Ensure you have the same number and same type of parameters as those required by the service.

6. **Execute the "Call" object's `invoke()` method and retrieve the "Response" object**

Remember the RPC call is synchronous, so it may take some time to complete.

7. **Check the response for a fault using the `getFault()` method, and then extract any results or returned parameters**

While most of the providers only return a result, the DB2 stored procedure provider can also return output parameters.

Interacting with a "document-oriented" SOAP service requires you to use lower-level Apache SOAP API calls. You must first construct an "Envelope" object which contains the contents of the message (including the body and any headers) that you wish to send. Then create a "Message" object where you invoke the `send()` method to perform the actual transmission.

To create a secure SOAP service, do the following:

1. Create a simple object
2. Define an envelope editor
3. Specify a pluggable envelope editor
4. Define the transports

Your code may look like the following example:

```
EnvelopeEditor editor=new PluggableEnvelopeEditor(new InputSource(conf), home);SOAPTransport
transport =new FilterTransport(editor, new SOAPHTTPConnection());call.setSOAPTransport(transport);
```

The characteristics of the secure session are specified by the configuration file, "conf."

See article [Securing SOAP services](#) for more information on creating secure Web services.

See article [4.8.1.2.2.1: Accessing enterprise beans through SOAP](#) for information on calling an EJB service.

Since the SOAP API is a standard for Web services, any clients that you create to access the WebSphere Application Server SOAP services can also run in different implementations.

See the related information links for an enablement scenario.

4.8.1.2.2.1: Accessing enterprise beans through SOAP

Calling enterprise beans through SOAP is handled in the same manner as calling Java bean methods through SOAP. The SOAP runtime handles the bean cases for you, such as calling an enterprise bean's create method if the create was not called previously.

A Web service can be a simple stateless session bean that performs number processing and returns a data value. When the client code makes a call to the data processing method of this service and an instance of the stateless session is not available, the SOAP runtime does the following:

- Calls the EJB create method to obtain a stateless session
- Calls the requested method

At times the client code must do additional work to use enterprise beans through SOAP. For example, if a Web application intends to use stateful or entity beans that persist data between calls, the client requires a reference to identify the bean instance that must be accessed in subsequent calls to methods. This reference/key can be obtained from the response object that the client receives on the initial call to the bean.

Response objects are created:

- When the client explicitly calls a create method
- From a `findByPrimaryKey()` Entity Bean method call
- From a regular bean method call

The following code example demonstrates calling a bean's create method with parameters:

```
/*This code snippet is from a simple MessageBoard bean that stores strings sent to it for retrieval at a later date.*/
...
/*Call create with \"This is a test\" to initialize the EJB*/
call = new Call();
call.setTargetObjectURI("urn:messageboard");

/*Note, you can explicitly call a create. Parameters for the bean's create can be passed like parameters to any SOAP RPC call.*/
call.setMethodName("create");
call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
params = new Vector();
params.addElement(new Parameter("msg", String.class, "This is a test", null));
call.setParams(params);

    System.out.println("Calling create with \"This is a test\"");
    resp = call.invoke(url, "");

/*Now use the same instance of the bean that you just 'created' and initialized. Obtain the reference from the response object through the method getFullTargetObjectURI()*/
    ejbKeyURI = resp.getFullTargetObjectURI();

/*Subsequent calls to this bean can now be made by using the obtained ejb key.*/
/*Call getMessage using the handle from the create*/
call = new Call();
call.setFullTargetObjectURI(ejbKeyURI);
call.setMethodName("getMessage");
call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
System.out.println("Calling getMessage:");
resp = call.invoke(url, "");
...

```

4.8.1.2.3: Deploying a programming artifact as a SOAP accessible Web service

Complete these steps to deploy a SOAP accessible Web service in WebSphere Application Server:

1. Create or locate the software resource to be exposed as a service

To deploy a service, create a programming artifact, one of the supported types, or locate an existing piece of code of the supported type.

2. Assemble an Enterprise Archive (EAR) file

Package the code artifact into an Enterprise Archive (EAR). This step is a deployment packaging requirement of WebSphere Application Server. Use the Application Assembly Tool (AAT) to package the artifact. See article [Application Assembly Tool](#) for information on using the tool.

3. Create the Apache SOAP deployment descriptor for the desired service

In order to deploy an artifact as a SOAP service, create a Apache SOAP deployment descriptor that describes the service you are creating. This step exposes the programming artifact as a "service." The descriptor describes and defines the parts of the code that will be invoked with the SOAP calls.


The information contained in the deployment descriptor varies, depending on the type of artifact you are exposing. For example, the following deployment descriptor might be used with the *StockQuoteSample*:

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment" id="urn:service-urn"
  [type="message"] >
  <isd:provider type="java" scope="Request | Session |
Application" methods="exposed-methods" >
  <isd:java class="implementing-class"
[static="true|false"] />
  </isd:provider>
  <isd:faultListener>org.apache.soap.server.DOMFaultListener</isd:faultListener>
</isd:service>
```

View the [Apache SOAP deployment descriptor documentation](#) for more information.

4. Execute the SoapEarEnabler tool to enable your Web service

As mentioned above, your code artifact must first be packaged into an Enterprise Archive (EAR). Next, using the deployment descriptor as input, add the necessary pieces to the EAR file to enable the artifact as a Web service. To facilitate this process, use the Java based tool called SoapEarEnabler. Depending on whether you secure the Web service, this tool will add two Web modules: soap.war and soap-sec.war to the EAR file. These Web modules include the SOAP deployment descriptors plus the necessary parts to deploy the service into the WebSphere Application Server runtime.

 The service does not become available until the soap-enabled EAR file is installed, and the server is restarted.

View the [SoapEarEnabler tool documentation](#) for more information on SoapEarEnabler.

5. Install the service-enabled EAR file

Take the modified EAR file, created in the previous step, and install it in WebSphere Application Server.

View article [Installing applications with the application installer command line](#) for information on installing EAR files.

6. Update the Web server plugin configuration

Run the GenPluginCfg.bat file on Windows NT or the GenPluginCfg.sh script on UNIX to regenerate the plugin configuration.

7. Restart the application server

See the related information links for an enablement scenario.

4.8.2: Apache SOAP deployment descriptors

Apache SOAP utilizes XML documents called "deployment descriptors" to provide the SOAP runtime with information on client services.

Deployment descriptors provide an array of information such as the:

- Service's URN (Uniform Resource Name)(which is used to route the request when it arrives)
- Method and class details, if the service is being provided by a Java class
- User ID and password information, if the service provider must connect to a database

The contents of the deployment descriptor vary, depending on the type of artifact that is being exposed using SOAP.

4.8.2.1: SOAP deployment descriptors in WebSphere Application Server

This article describes the different types of deployment descriptors that can be used in WebSphere Application Server. Deployment descriptors for each of the soap samples are included in the `soapsamples.ear` file in the `ServerSamplesCode` directory (for example, `<product_installation>/installedApps/soapsamples.ear/ServerSampleCode/src/addressbook/DeploymentDescriptor`)

Standard Java class deployment descriptor

A deployment descriptor which exposes a service that is implemented with a standard Java class (including a normal java bean) looks like this example:

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment" id="urn:service-urn"
[type="message"]>
  <isd:provider type="java" scope="Request | Session | Application" methods="exposed-methods">
    <isd:java class="implementing-class"
[static="true|false"]/>
    </isd:provider>
    <isd:faultListener>org.apache.soap.server.DOMFaultListener</isd:faultListener>
</isd:service>
```

- where:
- **service-urn** is the URN that you give to a service. (All services deployed within a single EAR file must have URNs that are unique within that EAR file.)
 - **exposed-methods** is a list of methods, separated by spaces, which are being exposed
 - **implementing-class** is a fully qualified class name (that is, a `packagename.classname`) that provides the methods that are being exposed.

On the `<service>` element, there is an optional attribute called **type** which is set to the value "message" if the service is document-oriented instead of RPC-invoked.

On the `<java>` element, there is an optional attribute called **static**, which may be set to either "true" or "false", depending on whether the methods are exposed or not exposed. If exposed, this attribute indicates whether the method is static or not static.

On the `<provider>` element, there is a **scope** attribute which indicates the lifetime of the instantiation of the implementing class.

- "Request" indicates the object is removed after the request completes.
- "Session" indicates the object lasts for the current lifetime of the HTTP session.
- "Application" indicates the object lasts until the servlet that is servicing the requests, is terminated.

EJB deployment descriptor

A deployment descriptor that exposes a service which is implemented with an Enterprise Java Bean looks like this next example:

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment" id="urn:service-urn">
<isd:provider type="provider-class" scope="Application"
methods="exposed-methods">
  <isd:option key="JNDIName" value="jndi-name"/>
<isd:option key="FullHomeInterfaceName" value="home-name" />
</isd:provider>
<isd:faultListener>org.apache.soap.server.DOMFaultListener</isd:faultListener>
</isd:service>
```

 The default values for the `iiop` URL and context provider keys are:

```
<isd:option key="ContextProviderURL" value="iiop://localhost:900" />
<isd:option key="FullContextFactoryName" value="com.ibm.websphere.naming.WsnInitialContextFactory" />
```

To use your own values, you must specify:

```
<isd:option key="ContextProviderURL" value="<URL to the JNDI provider>" />
<isd:option key="FullContextFactoryName" value="<Context factory full class name>" />
```

A description of the keys and variables follows:

- **service-urn** and **exposed-methods** have the same meaning as in the standard Java class deployment descriptor
- **provider-class** is one of the following depending on the implementation of the bean:

Provider class	Bean implementation
com.ibm.soap.providers.WASStatelessEJBProvider	stateless session bean
com.ibm.soap.providers.WASStatefulEJBProvider	stateful session bean
com.ibm.soap.providers.WASEntityEJBProvider	entity bean

- **jndi-name** is the registered JNDI name of the EJB
- **home-name** is the fully qualified class name of the EJB's home.

Bean Scripting Framework (BSF) script deployment descriptor

A deployment descriptor that exposes a service which is implemented with a BSF script looks like the following example:

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment" id="urn:service-urn">
<isd:provider type="script" scope="Request | Session | Application"
methods="exposed-methods">
  <isd:script language="language-name"
[source="source-filename"]>[script-body]
</isd:script>
</isd:provider>
```

```
<isd:faultListener>org.apache.soap.server.DOMFaultListener</isd:faultListener>
</isd:service>
```

where:

- **service-urn**, **exposed-methods**, and **scope** have the same meaning as in the standardJava class deployment descriptor
- **language-name** is the name of the BSF-supported language that is used to write the script.

The deployment descriptor must also have a **source** attribute on the <script> element, or a **script-body** attribute. The **script-body** attribute contains the actual script that is used to provide the service. If the deployment descriptor has the **source** attribute, then **source-filename** refers to the file which contains the service implementation.


DB2 stored procedure deployment descriptor

A deployment descriptor which exposes one or more DB2 stored procedures as a service looks like the following example:


```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment" id="urn:service-urn">
<isd:provider type="com.ibm.soap.providers.WASDB2SPPProvider" scope="Application"
methods="*" | exposed-methods">
  <isd:option key="userID" value="db-userid"/>
  <isd:option key="password" value="db-password"/>
  <isd:option key="fullContextFactoryName" value="context-factory"/>
  <isd:option key="datasourceJNDI" value="jndi-name"/>
  <isd:option key="dbDriver" value="db-driver"/>
  <isd:option key="dbURL" value="db-url"/>
</isd:provider>
<isd:faultListener>org.apache.soap.server.DOMFaultListener</isd:faultListener>
</isd:service>
```

where:

- **service-urn** and **exposed-methods** have the same meaning as in the standardJava class deployment descriptor.
- **db-userid** is a valid user ID used to access the database where the stored procedures reside.
- **db-password** is a valid password for the specified user ID

 The **db-userid** and **db-password** entries are optional. These entries can be set in the datasource. In WebSphere Application Server, the preferred way for administering the **db-userid** and **db-password** entries is with a datasource. Changing the user ID and password is easier when the information is located in a datasource rather than in a separate deployment descriptor file. See article [DB2 Stored procedure sample setup](#) for more information.

- **context-factory** is the name of the context factory used to access the database
- **jndi-name** is the datasource used to access the database
- **db-driver** is the database driver used to access the database.

 A **db-driver** is not required if a datasource JNDI name is specified.

- **db-url** is the URL that specifies the database to access

The **methods** attribute on the <provider> element can contain a list of space separated procedure names to expose, or an "*" (asterisk). An asterisk indicates all available stored procedures should be exposed.

See the related topics section for links to an enablement scenario.

4.8.3: Quick reference of Web services resources

Use the following table to link directly to Web services descriptions, and additional resources.

Click on any heading in the *Topic* category for a description of that topic.

Click on any heading in the *Resources* category for links to external sites that provide sample scenarios, toolkits, tutorials, and additional information.

Reference the *Related topics* section for links to the SOAP EAR enabler tool and to a Web services enablement tutorial.

Topic	Resources
Web services overview	<ul style="list-style-type: none">Web services topics and development environmentWeb services wizardWeb services toolkitWeb services tutorial
<ul style="list-style-type: none">SOAP overviewSOAP support in WebSphere Application ServerSOAP samplesBuilding a SOAP clientDeploying a programming artifact as a SOAP accessible Web service	Apache SOAP implementation
<ul style="list-style-type: none">UDDI overviewIBM's UDDI test registry	IBM's UDDI registry implementation
<ul style="list-style-type: none">UDDI4J overviewUDDI4J support in WebSphere Application ServerUDDI4J samplesIBM's Javadoc	UDDI4J topics
WSDL overview	WSDL topics

See the Related topics section for links to an enablement tutorial.

4.8.4: Securing SOAP services

Since the SOAP specification left security issues open, several proposals evolved to bridge the security gaps. Recently the SOAP Security Extension [[SOAP-SEC](#)] was published as a W3C Note, specifically addressing the [XML Digital Signature](#).

The SOAP security extension, included with WebSphere Application Server Version 4.0, is a security architecture based on the SOAP security specification, and widely-accepted security technologies such as [Secure Sockets Layer](#) or SSL.

There are three options for security when using HTTP as the transport protocol.

- [HTTP basic authentication](#)
- [SSL \(HTTPS\)](#)
- [SOAP signature](#)

Application developers are free to combine these security options according to their security requirements. The following scenarios describe the implementation of the security options.

HTTP basic authentication

Many applications require users to provide identifying information. You cannot provide access control for individual services. You can only provide access control for the router servlets (as for example the `rpcrouter` servlet URI). If a user can get to a servlet, he can access any of the Web services served through the servlet. Therefore, if you have a set of "secure" services and "unprotected" services, you have to partition them differently so that "secure" services are accessed through an URI that is secured (for example, `/secureRPCRouter`) and the unprotected services are open for everyone to access (for example, `/unprotectedRCPRouter`).

Using the Application Assembly tool, you can set authorization levels by assigning roles to HTTP methods and by assigning users to roles. You can then *authenticate* users, verifying they are authorized to view specific information. There are many ways to prompt users for authentication data. See articles [Overview: Using programmatic and custom login](#) and [The WebSphere authorization model](#) for more information on different authentication methods, and on role-based authorization scenarios.

SOAP on SSL with HTTP basic authentication

To make a request over HTTPS, using the SSL support of Apache SOAP, you need a separate [Java Secure Socket Extension](#) (JSSE) provider.

WebSphere Application Server includes the `ibmjsse.jar` in the JDK extensions.

The "SOAP on SSL" scenario is useful for many *business-to-business* (B2B) applications because:

- The data in transit is protected from eavesdropping or forgery by SSL.
- The client identity is authenticated through user ID and password, which are encrypted by the SSL transport.

For example, if an inventory application is configured as a Web service, the service provider has the following two SOAP service entries:

- `https://foo.com/inventory/inquiry`
- `https://foo.com/inventory/update`

Each SOAP service entry should be deployed as a separate enterprise application (EAR) because each service has a different access control policy, which is: anyone can inquire about the inventory but only the inventory clerks can update the contents.

The SOAP enablement model limits you to one context root for the unsecured services and another for the secured services. In this example, you want to make the inquiry service unsecured and the update service secured. If you want different levels of security for a "secured" service, then you must deploy the entries in the "secured" service as separate EAR files.

Do the following to enable the "SOAP on SSL" scenario:

- Configure the web server (httpd.conf) so that it only allows SSL access to these servlets.
- Configure the security role for the `RPCRouterServlet` in the inquiry services EAR so that the `RPCRouterServlet` for the 'inquiry' service is accessible by everyone, while the `RPCRouterServlet` for the 'update' service requires authentication based on the HTTP basic authentication (userid/password).

In this case, the 'update' application does not know the identity of the requester; it only knows that access is granted. In other words, the "update" application is not concerned with the identity of the user because it knows WebSphere Application Server is ensuring that only authenticated users have access.

SOAP on SSL with SOAP Signature

Applications might need non-repudiable proof of exchanged messages. One example is a web service that accepts part orders. The business partners establish a form of trust relationship based on public keys. This can be done using the public key infrastructure (PKI) through a third party certificate authority (CA), or by exchanging public keys with a secure channel. The following service is deployed with a *signature verification* function:

`https://foo.com/partorder`

Configure *signature verification* with the following information:

- Scope of signature (indicates the portion of the SOAP envelope that must be authenticated. The default is the content of *SOAP-ENV:Body*).
- Trusted keys or trusted root keys.
- Default key to verify signature if no `KeyInfo` is specified.
- Other policies regarding signature validation.
- Behavior when signature verification fails.
- Additional requirements on signature (as for example, specific requirements on hash/C14N algorithms to be used, timestamp validity, and so forth).

If the signature is missing or if *signature verification* fails, the signature verification function can be configured so that the servlet returns a SOAP fault.

To send part orders to the `https://foo.com/partorder` service, the service requester should sign his SOAP messages with a signature component. The signature component is initialized using two templates:

1. `<ds:SignedInfo>` template
2. `<ds:KeyInfo>` template

The `<ds:SignedInfo>` template controls the following:

- What parts of the SOAP envelope must be signed
- What algorithms (canonicalization, transformation, digest, sign) should be used

The `<ds:KeyInfo>` template controls the following:

- Whether or not to include the entire certificate chain in `<ds:KeyInfo>`
- Decision to include only certificate and serial number
- Public key value
- Decision to provide no key information (so that the default key must be used for verification).

You can combine the service request with HTTP basic authentication, if necessary.

4.8.4.1: Running the security samples

The process for running the SOAP signed samples is identical to the process for running the non-signed samples. The `soapsamples.ear` must be installed, and the server must be started before these samples are invoked.

See article [SOAP samples](#) for information on installing the SOAP samples.

SOAP Signature

The client samples are included in the `soapsamples.ear` file. Do the following to locate and execute the samples:

1. Change your directory (cd) to


`product_installation_root/installedApps/soapsamples.ear/ClientCode`

A set of batch files or script files (on UNIX platforms) have been included to facilitate running the client samples. These batch or script files are located in the `nt_bat` subdirectory on Windows NT, or in the `unix_script` subdirectory on UNIX platforms. These scripts set the classpath and supply parameters.

2. Invoke the samples using the following scripts:

```
DSigAddressSample localhost "c:\WebSphere\AppServer\installedApps\soapsamples.ear" "John B. Good"
```

```
DSigMessageSample localhost "c:\WebSphere\AppServer\installedApps\soapsamples.ear" "..\data\msg1.xml"
```

 If you run the script with **no** arguments, as for example *DSigAddressSample*, you will be provided with help on how to use the sample, and you will receive a description of the command line arguments that the script requires.

3. View the output.

For each sample, at the server, you should see that the signature of the request is validated. At the client, you should see that the signature of the response is validated.

The validation results for both the client and server are logged to the following files that are created in the `product_installation_root/InstalledApps/soapsamples.ear/soapsec.war/logs` directory

- SOAPVHH-all-cl.log
- SOAPVHH-fail-cl.log
- SOAPVHH-all-sv.log
- SOAPVHH-fail-sv.log

Soap signature with SSL connection

Ensuring that a connection is over SSL is not specific to Web services. You must configure the Web server to ensure that the client to Web server connection is over SSL. You must also configure WebSphere Application Server to ensure that the Web server to WebSphere Application Server connection is over SSL.

Article [Configuring SSL in WebSphere Application Server](#) discusses how to configure SSL in WebSphere. See your Web server documentation for information on configuring the SSL server.

For testing purposes, sample client and server key stores are shipped with the SOAP samples. You must use the IBM Key Management Tool to extract the certificates located in files:

- test
- keystore
- databases

Import the certificates into your key databases. See article, [Tools for managing certificates and keys](#) for more information on the IBM Key Management tool.

The test keystores are described in article [Keystore files](#).

Export the client certificates from the test keystore file

Perform the following steps to export the client certificates:

1. Invoke the Key Management Tool (IKeyman)
2. From the file menu, select open
3. Change directory (CD) to
`product_installation_root/InstalledApps/soapsamples.ear/soapsec.war/key/`
4. Select the SOAPClient keystore file.
(The keystore password is "client".)
5. Change the key database content type to "Signer Certificates".
6. Highlight the **soapca** certificate.

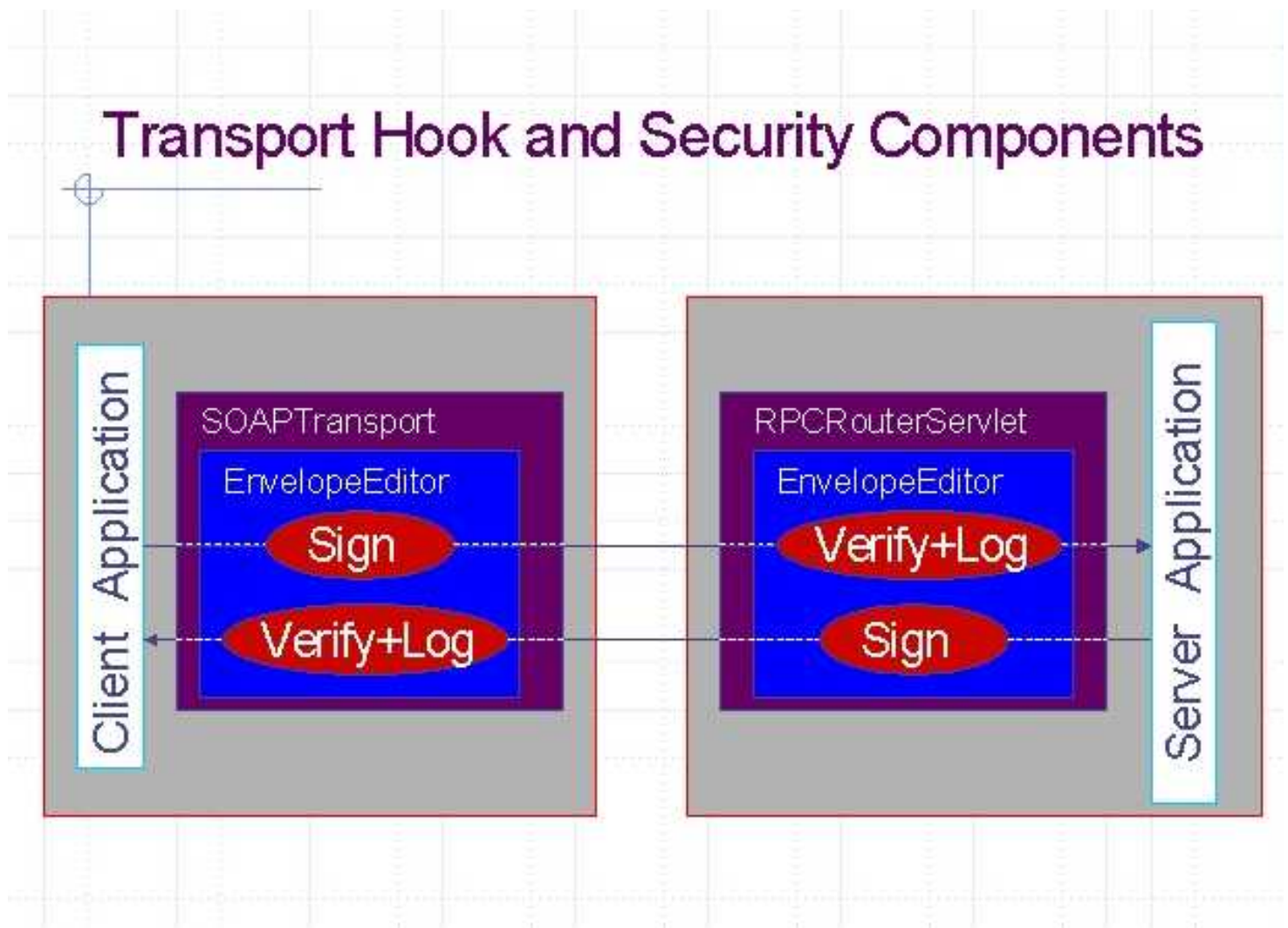
7. Click the **Export** button.
8. Change the exported file name to "soapca.arm".
9. Highlight the "intca1" certificate
10. Click the **Export** button.
11. Change the exported file name to "intca1.arm".

Import the certificates into the web serverkey database

1. Invoke the Key Management Tool (IKeyman)
2. From the file menu, select open (or new if you are creating a new keystore)
3. Change directory (CD) to the directory where the keystore file is located.
4. Select the file.
5. For Signer Certificates, add the "intca1.arm" and the "soapca.arm" you exported in the previous section.
6. For Personal Certificates, click **Import**.
7. Specify a key type of **PKCS12**
8. Browse the *sslserver.p12* file located in:
[*product_installation_root*](#)/InstalledApps/soapsamples.ear/soapsec.war/key/
9. Click OK.
10. Enter "server" when prompted for a password.
11. Select "sslserver" from the key list and press OK.
12. Save the updated keystore file

4.8.4.2: SOAP signature components

An overview of the SOAP signature architecture is illustrated in the figure below.



Using the SOAP transport hook, you can plug-in the security components:

- signer
- verifier with logging capability

The transport hook is called the *EnvelopeEditor*. A *PluggableEnvelopeEditor* is also provided, which allows you to plugin your security components. As illustrated, the *EnvelopeEditor* is encapsulated in the *SOAPTransport* on the client side. On the server side, *EnvelopeEditor* is encapsulated in *RPC/MessageRouterServlet*. This means the same components can be used on either side.

When a client application sends a request, the request is signed and transmitted to the server. At the server side, the request is verified and delivered to a server application or, in the case of a RPC, to a Java object. The response is processed in the same manner. The verifier component also has a logging function to log the verified messages in a file. Signatures and verifier components are configurable. You can specify encryption, digest message algorithm, certificate path policy, and other security technologies.

Signature Components

There are two signature components:

- [Signature Header Handler](#)
- [Verification Header Handler](#)

Signature Header Handler (SHH)

The Signature Header Handler is a XML-based configuration file, which enables:

- Template for <SignedInfo> (for customizing references, sign/hash algorithms, C14N algorithms, optional timestamp)
- Template for <KeyInfo> (for customizing the public key such as X.509 certificate)

Verification Header Handler (VHH)

The Verification Header Handler is a XML-based configuration file, which enables:

- Configurable policy (required scope of signature, trusted root, certstore, certpathchecker) (more sophisticated policy such as timestamp validation may not be included in 2/15 deliverable)
- Exit for Logging (additional application-specific verification) A reference implementation of logging component is also provided.

The digital signature configuration can be changed by editing the configuration for the following components:

- [Envelope Editor](#)
- [Signature Component](#)
- [Verification Component](#)

SOAP Security-related Files

The following table provides an inventory of the SOAP security elements contained in the SOAP security samples module (soapsec.war).a quick reference for SOAP security topics.

Path	Contents	Description
/installedApps/soapsamples.ear/soapsec.war	Web-INF, conf, key, log, etc.	Home of the soap security servlets
/installedApps/soapsamples.ear/soapsec.war/WEB-INF	web.xml	Servlet configuration file for SOAP security samples
/installedApps/soapsamples.ear/soapsec.war/conf	config files	Configuration files for envelope editors and signature components
/installedApps/soapsamples.ear/soapsec.war/key	SOAPclient SOAPserver	See article Keystore files for more information.
/installedApps/soapsamples.ear/soapsec.war/logs	Log files	Logs generated during security exchange
/installedApps/soapsamples.ear/ServerSamplesCode/src/<service_name>	server side samples	Source for both the non-secure and secure samples
/installedApps/soapsamples.ear/ClientCode/nt.bat	scripts to run client samples	Batch files for invoking the client side samples to interact with the server-side services
/installedApps/soapsamples.ear/ClientCode/unix_scripts	scripts to run client samples	Batch files for invoking the client side samples to interact with the server-side services
/installedApps/soapsamples.ear/ClientCode/data	data files used by samples	
/installedApps/soapsamples.ear/ClientCode/src	client side samples source	
/lib	soap.jar, soap-sec.jar, ws-soap-ext.jar	Location of all jar files

Related Documents

- [Simple Object Access Protocol \(SOAP\) 1.1](#) - W3C NOTE.
- [SOAP Security Extensions: Digital Signature](#) - W3C NOTE.
- [XML-Signature Syntax and Processing](#) - W3C CR.
- [XML Security Suite](#) - XML digital signature, encryption, access control.

4.8.4.2.1: Keystore files for testing purposes

Two keystore files, (SOAPserver and SOAPclient), are available for testing purposes. These files are located in directory:

`product_installation_root/installedApps/soapsamples.ear/soapsec.war/key`

This article describes the certificates that are stored in these two keystore files.

File name	Store password	Description
SOAPserver	server	This keystore is used by a service provider.
SOAPclient	client	This keystore is used by a service requester.

Common Certificate Authority certificates

The following three certificates are commonly stored in both keystore files.

Alias	Issuer	Description
soapca	soapca itself	The certificate of the root Certificate Authority (CA) used for testing purposes.
intca1	soapca	The certificate of the CA to issue SSL-related certificates.
intca2	soapca	The certificate of the CA to issue SOAP-DSIG-related certificates.

Certificates for service providers

The following two certificates are stored in the SOAPserver keystore.

Alias	Issuer	Description
sslserver	intca1	This is the certificate of the SSL server. This is also stored in the SOAPclient keystore as a <i>trusted</i> certificate. The PKCS12 file including the corresponding private key for this certificate is <code>sslserver.p12</code> .
soapprovider	intca2	This certificate might be used by a service provider to digitally sign its response message. The key password is "server".

Certificates for service requesters

The following three certificates are stored in the SOAPclient keystore.

Alias	Issuer	Description
sslclient	intca1	This certificate might be used for the SSL client authentication. The key password is "client".
sslserver	intca1	This is the certificate of the <i>trusted</i> SSL server and the same as the one stored in the SOAPserver keystore. The PKCS12 file, including the corresponding private key for this certificate, is <code>sslserver.p12</code> .
soaprequester	intca2	This certificate might be used by a service requester to digitally sign its request message. The key password is "client".

- [IBM HTTP Server documentation on configuring SSL](#)
- [Tools for managing certificates and keys](#)

4.8.4.2.2: Envelope Editor

The Envelope Editor is a component that can be plugged into the Apache SOAP transports. At the server side, it is embedded into the RPC and MessageRouterServlets. At the client side, it is embedded in the FilterTransport, which implements the SOAPTransport interface. WebSphere Application Server provides a *PluggableEnvelopeEditor*, which can be used to plug-in some editing components such as signature and verification.

Enabling Envelope Editor

At the client side, the configuration of the *eEnvelope eEditor* is explicitly programmed. On the server side, the transport hook is enabled automatically in the soapsec.war file when you add the "init" param to the RPC and MessageRouter servlets for the EnvelopeEditorFactory. This entry in the web.xml for the soapsec.war file is added automatically when you "soap enable" an application and indicate the service is secure.

Description of the factory class to instantiate Envelope Editors

A factory class creates *Envelope Editors* at runtime. The factory class is called DSigFactory. The DSigFactory class consumes an editor configuration file, and creates an instance of *Envelope Editor*. The factory class and the configuration file are specified in:

[product_installation_root](#)\installedApps\ear_file_name\soapsec.war\WEB-INF\web.xml

The factory class is described under the <servlet id="Servlet_1"> and <servlet id="Servlet_2"> elements:

```
<display-name>Apache-SOAP-SEC</display-name>          <description>SOAP Security Enablement
WAR</description>      <servlet id="Servlet_1">          <servlet-name>rpcrouter</servlet-name>
<display-name>Apache-SOAP Secure RPC Router</display-name>      <description>no
description</description>
<servlet-class>com.ibm.soap.server.http.WASRPCRouterServlet</servlet-class>      <init-param
id="InitParam_1">      <param-name>faultListener</param-name>
<param-value>org.apache.soap.server.DOMFaultListener</param-value>      </init-param>
<init-param id="InitParam_2">      <param-name>EnvelopeEditorFactory</param-name>
<param-value>com.ibm.soap.dsig.dsigfactory.DSigFactory</param-value>      </init-param>
<init-param id="InitParam_3">      <param-name>SOAPEnvelopeEditorConfigFilePath</param-name>
<param-value>conf/sv-editor-config.xml</param-value>      </init-param>      </servlet>
<servlet id="Servlet_2">      <servlet-name>messagerouter</servlet-name>
<display-name>Apache-SOAP Secure Message Router</display-name>
<servlet-class>com.ibm.soap.server.http.WASMessageRouterServlet</servlet-class>      <init-param
id="InitParam_5">      <param-name>faultListener</param-name>
<param-value>org.apache.soap.server.DOMFaultListener</param-value>      </init-param>
<init-param id="InitParam_6">      <param-name>EnvelopeEditorFactory</param-name>
<param-value>com.ibm.soap.dsig.dsigfactory.DSigFactory</param-value>      </init-param>
<init-param id="InitParam_7">      <param-name>SOAPEnvelopeEditorConfigFilePath</param-name>
<param-value>conf/sv-editor-config.xml</param-value>      </init-param>      </servlet>
```

EnvelopeEditorFactory is a factory class. SOAPEnvelopeEditorConfigFilePath is a configuration file for Envelope Editor.

Configuration file of Envelope Editor

The configuration file, sv-editor-config.xml is located in:

[product_install_root](#)\installedApps\<ear_file_name>\soapsec.war\conf\sv-editor-config.xml

Under the SOAPEnvelopeEditorConfig element, there are two optional elements:

- incoming
- outgoing

The incoming and outgoing element definitions look like the following example:

```
<incoming class="com.ibm.xml.soap.security.dsig.SOAPVerifier">      <init-param>
<param-name>filename</param-name>      <param-value>conf/sv-ver-config.xml</param-value>
</init-param> </incoming> <outgoing class="com.ibm.xml.soap.security.dsig.SOAPSigner">
<init-param>      <param-name>filename</param-name>
<param-value>conf/sv-sig-config.xml</param-value>      </init-param> </outgoing>
```

The incoming element specifies a class which "edits" incoming messages, and a configuration file for the editing class. The outgoing element specifies a class for outgoing message and a configuration file.

Changing the configuration

You do not have a digital signature for response messages if you remove the outgoing element from

[product_installation_root](#)\installedApps\<ear_file_name>\soapsec.war\conf\sv-editor-config.xml

and remove the incoming element from

[product_installation _root](#)\installedApps\<ear_file_name>\soapsec.war\conf\cl-editor-config.xml

4.8.4.2.3: Signature Header Handler

The Signature Header Handler (SHH) inserts a digital signature header into a SOAP envelope. You can customize the SHH configuration with a configuration file. For example, you can specify a signing policy and the key store file.

There are two signature configuration files:

```
product_installation_root\installedApps\<ear_file_name>\soapsec\conf\sv-sign-config.xml  
product_installation_root\installedApps\<ear_file_name>\soapsec\conf\cl-sign-config.xml
```

The `soapsamples.ear` file contains samples of these configuration files.

An explanation of each configuration element in the Signature Header follows:

- **KeyStore**

The `KeyStore` element specifies a keystore file that holds the signing key. In the following example, the attribute "type" indicates a keystore type, and "jks" indicates Java Key Store. "path" is a keystore file, and "storepass" is its store password.

```
<KeyStore type="jks" path="key\SOAPserver" storepass="server" />
```

The Key Management tool (iKeyman) can be used to create a keystore file.

- **Policy**

The `PublicKey` element specifies the information that should be included in the `<ds:KeyInfo>` element. With the current implementation, you must either include the complete certificate chain, or omit the `<ds:KeyInfo>`. When `<ds:KeyInfo>` is omitted, the recipient must know the default key to verify the signature.

- **Template**

The contents of the `Template` element specify all the details related to XML Signature, including signature algorithms, digest algorithms, canonicalization algorithms, transform algorithms, the portion of the SOAP envelope to be signed, and so on.

- **Object**

The template can also have `Object` element(s) for additional authentication information, such as a timestamp.

- **ValueOfTimestamp**

This SHH understands one special element type, `ValueOfTimestamp`, which is replaced with a current time and date before being inserted into the signature.

4.8.4.2.4: Verification Header Handler

The Verification Header Handler (VHH) validates a digital signature header in a SOAP envelope. Its configuration can be customized using a configuration file where you specify the following:

- a verification policy
- the certificate path
- logging files to record verified messages

There are two signature configuration files:

```
product_installation_root\installedApps\<ear_file_name>\soapsec.war\conf\sv-ver-config.xml
product_installation_root\installedApps\<ear_file_name>\soapsec.war\conf\cl-ver-config.xml
```

Samples of these configuration files are provided in the `soapsamples.ear` file.

An explanation of each configuration element in the Verification Header follows:

• AllowedAlgorithms

All the algorithms supported by this VHH must be listed in this element. Algorithms other than these cannot be used in `SOAP-SEC:Signature` header. The current implementation supports all required algorithms in the XML Signature specification, except for SHA1-MAC.

• RequiredAuthenticatedParts

This section specifies what parts of SOAP message need to be authenticated through the `SOAP-SEC:Signature` header. Currently two values are supported for the `part` attribute:

1. When `part="root,"` the whole envelope must be signed through the enveloped-signature transform.
2. When `part="body,"` the `SOAP-ENV:Body` element in the SOAP envelope must be referenced by one of the reference elements in the signature.

`Part=""` allows an attachment to be specified.

If the specified parts are not authenticated through the signature header entry, verification fails.


• DefaultVerificationKeys

When `KeyInfo` is missing in the signature, the content of this element is used as a part of the signature. When communicating parties know the identity of each other, the default `KeyInfo` can be used to reduce the communication data volume.

• Log

Specifies the logging behavior. The following versions of logging exist:

- When `target="all,"` all verification attempts are logged.
- When `target="success,"` only successful verification are logged.
- When `target="fail,"` only unsuccessful verification are logged.

 Multiple `LogFile` elements can be specified.

The following example illustrates how to specify logging:

```
<Log>    <SOAPDSigLogger      class="com.ibm.xml.soap.security.dsig.SOAPDSigLoggerImpl">
<LogFile target="all" path="SOAPVHH-all.log" append="yes"/>    </SOAPDSigLogger>    <SOAPDSigLogger
class="com.ibm.xml.soap.security.dsig.SOAPDSigLoggerImpl">    <LogFile target="fail"
path="SOAPVHH-fail.log" append="yes"/>    </SOAPDSigLogger> </Log>
```

• PKIXParameters

Currently VHH supports X.509/PKIX certificates only (no HMAC, no PGP, and so forth). The policies for PKIX certificate verification are specified in this element. This is a straightforward mapping of Java `CertPath` API. Not all of the entries are meaningful in this initial release.

Current implementation only allows the use of keystore as the means of specifying trusted root.

4.10: Developing custom services

You can write a custom service class that implements the `com.ibm.websphere.runtime.CustomService` interface (shown below). The administrator can then create a custom service configuration for an application server, supplying the class name. When the application server is started, the custom service will be started and initialized.

The properties passed by the application server runtime to the `initialize` method can include one for an external file containing configuration information for the service (retrieved with the `externalConfigURLKey`). In addition, the properties can contain any name-value pairs that are stored for the service, along with the other system administration configuration data for the service. The properties are passed to the `initialize` method of the service as a `Properties` object.

There is a shutdown method for the interface as well. Both methods of the interface declare that they may throw an `Exception`, although no specific exception subclass is defined. If an exception is thrown, the runtime will log it, disable the `CustomService`, and proceed with Server startup.

The interface does not pass in the context for the services to use for registration binding. This is how it differs from the `ServiceInitializer` interface provided in Version 3.5.

Custom service interface

The following code is the complete source for the interface.

```
package com.ibm.websphere.runtime;/** * The CustomService interface must be implemented by all *
WebSphere Custom Service extensions. * The application server runtime will call the initialize
method of * this interface on every enabled Custom Service configured in the server. */public
interface CustomService {    static final java.lang.String externalConfigURLKey =
"com.ibm.websphere.runtime.CustomService.externalConfigURLKey";/** * The initialize method is called
by the application server runtime during * server startup. The Properties object
passed in on this method must * contain all configuration information necessary for this
service to * initialize properly. * * @param configProperties java.util.Properties */
void initialize(java.util.Properties configProperties) throws Exception;/** * The shutdown method is
called by the application server runtime when the * server begins its shutdown
processing. * * @param configProperties java.util.Properties */ void shutdown() throws Exception;}
```

Limitations of the custom service implementation

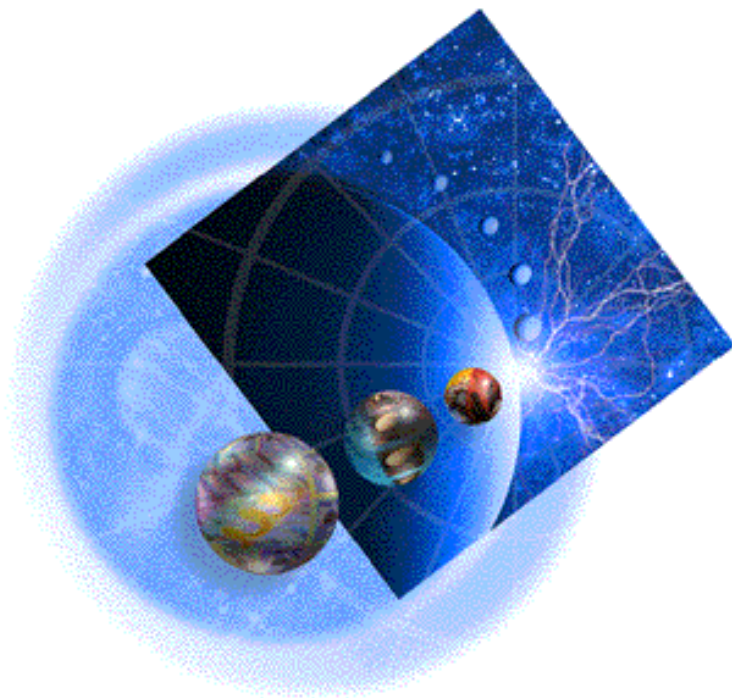
There are restrictions on the functions that can be executed within a custom service. The `initialize` method of all custom services is invoked by the server runtime before most other components have been initialized, including the ORB, Trace, Naming, Transaction Manager, and Connection Manager. This provides the custom service with great flexibility in effecting the server runtime environment, but also prevents the custom service `initialize` method implementation from being able to take advantage of the application server runtime services.

For example, a custom service cannot count on being able to make JNDI lookup method calls in its `initialize` method, since the Naming Service has not been initialized within the server runtime at that point in server process startup. A custom service can execute code in its `initialize` method that reads its configuration values and processes them in whatever manner makes sense for that service. This includes executing code that reads the external configuration file for the custom service, provided that the permissions set for the external config file match those of the server identity under which the custom service code runs. A JDBC Connection can be obtained and used as long as the pooling services of the WebSphere Connection Manager are not expected to be used (since the Connection Manager has not been initialized at the time that the custom service `initialize` method is invoked).

Specifically, within the implementation of a custom service:

- The `initialize` and `shutdown` methods must return control to the runtime.
- No work will be dispatched into the server instance until all custom service `initialize` methods return.
- Custom services are initialized serially, but in no predictable order.
- The `initialize` and `shutdown` methods will be called only once on each service, and once for each operating system process that makes up the server instance. File I/O is supported.
- Initialization of process level "static" data without leaving the process is supported.
- The identity/credential that the custom service code runs under is the server identity.
- Only JDBC RMLT (local tran) operations are supported. All UOW must be completed before the methods return.
- JNDI operations are not supported.
- Creation of threads is not supported.
- Creation of sockets and I/O other than file I/O is not supported.
- Execution of standard J2EE code (client code, servlets, enterprise beans) is not supported.
- The JTA interface is not available. This feature is available in J2EE server processes and distributed generic server processes only.
- While the runtime will make a best effort to call `shutdown`, there is no guarantee that `shutdown` will be called prior to process termination. These restrictions apply to `shutdown` and `init` equally.

WebSphere Application Server Samples



The Samples gallery offers a set of samples that show you how to perform common Web application tasks, provide reusable components and demonstrate handy techniques.

The gallery includes:

- The YourCo intranet Web site, which integrates many of the small samples into one common application.
- The Java Pet Store Application, which demonstrates J2EE technology via an online pet store.
- A Trade application, which demonstrates an online brokerage.

Once installed on your local machine, the Samples are located at

<http://localhost/WSsamples/index.html>

Open the above URL in your Web browser, follow the database configuration instructions, and try the Samples.

On Windows 2000, it has been found that localhost is not always recognized. In such a case, use the actual host name.

The above links will not work if:

- The Samples are not installed on the machine on which you are viewing this documentation ("localhost").
- Your Web server is not running.
- You are viewing this documentation from the IBM Web site instead of viewing locally installed documentation.

If you don't find the Samples on your localhost, confirm their installation. The Samples are an option in the product installation. See [the installation documentation](#) for a variety of case-specific installation steps.