

Programming

Developing application components

[What are enterprise applications?](#)

[What are application servers?](#)

[Server terminology](#)

[Developing applications](#)

[Programming model and environment](#)

[Finding supported specifications](#)

[Supported programming languages](#)

[Support for the newest XML APIs](#)

Developing enterprise beans

[What are enterprise bean containers?](#)

[What are enterprise beans?](#)

[What are models and clones?](#)

[What is workload management?](#)

[Developing enterprise beans](#)

[Late-breaking enterprise beans programming tips](#)

Writing Enterprise Beans book:

[About this book](#)

[An introduction to enterprise beans](#)

[An architectural overview of the EJB programming environment](#)

[WebSphere Programming Model Extensions](#)

[Developing enterprise beans](#)

[More-advanced programming concepts for enterprise beans](#)

[Developing EJB clients](#)

[Enabling transactions and security in enterprise beans](#)

[Developing servlets that use enterprise beans](#)

[Tools for developing and deploying enterprise beans in the EJB server \(CB\) environment](#)

[Tools for developing and deploying enterprise beans in the EJB server \(AE\) environment](#)

[Appendix A. Example code provided with WebSphere Application Server](#)

[Appendix B. Using XML in enterprise beans](#)

[Appendix C. Extensions to the EJB Specification](#)

Developing servlets

[What are servlets?](#)

[What are .servlet configuration files?](#)

[What are page lists?](#)

[Developing servlets](#)

[Servlet lifecycle](#)

[Servlet support and environment in WebSphere](#)

[Features of Java Servlet API 2.1](#)

[IBM extensions to the Servlet API](#)

[Using the WebSphere servlets for a head start](#)

[Avoiding the security risks of invoking servlets by class name](#)

[Servlet content, examples, and samples](#)

[Creating HTTP servlets](#)

[Overriding HttpServlet methods](#)

[Having servlets communicate](#)

[Forwarding and including data \(request and response\)](#)

[Example: Servlet communication by forwarding](#)

[Using page lists to avoid hard coding URLs](#)

[Obtaining and using servlet XML configuration files \(.servlet files\)](#)

[Extending PageListServlet](#)

[Example: Extending PageListServlet](#)

[Using XMLServletConfig to create .servlet configuration files](#)

[XML servlet configuration file syntax \(.servlet syntax\)](#)

[Example: XML servlet configuration file](#)

[Filtering and chaining servlets](#)

[Servlet filtering with MIME types](#)

[Servlet filtering with servlet chains](#)

[Enhancing servlet error reporting](#)

[Public methods of the ServletErrorReport class](#)

[Example: JSP file for handling application errors](#)

[Using your error reporting servlet or JSP file](#)

[Serving servlets by classname](#)

[Serving all files from application servers](#)

[Obtaining the Web application classpath from within a servlet](#)

Developing JSP files

[What are JSP files?](#)

[Developing JSP files](#)

[JavaServer Pages \(JSP\) lifecycle](#)

[JSP access models](#)

[JSP support and environment in WebSphere](#)

[JSP support for separating logic from presentation](#)

[JSP processors](#)

[Tools for developing JSP files](#)

[Batch compiling JSP files](#)

[Overview of JSP file content](#)

[JSP directives](#)

[Class-wide variables and methods](#)

[Inline Java code \(scriptlets\)](#)

[Java expressions](#)

[BEAN tags](#)

[<BEAN> tag syntax](#)

[Accessing bean properties](#)

[Setting bean properties](#)

[Supported NCSA tag reference](#)

[Using the Bean Scripting Framework](#)

[BSF examples and samples](#)

[IBM extensions to JSP 1.0 syntax](#)

[Tags for variable data](#)

[<tsx:getProperty> tag syntax and examples](#)

[<tsx:repeat> tag syntax](#)

[The repeat tag results set and the associated bean](#)

[Tags for database access](#)

[<tsx:dbconnect> tag syntax](#)

[<tsx:userid> and <tsx:passwd> tag syntax](#)

[<tsx:dbquery> tag syntax](#)

[<tsx:dbmodify> tag syntax](#)

Compiling JSP 1.0 files as a batch

JSP 1.0 examples

Example: <tsx:dbquery> tag syntax

Example: <tsx:dbmodify> tag syntax

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

IBM extensions to JSP .91 syntax

Tags for variable data

JSP .91 <INSERT> tag syntax

Alternate syntax for the <INSERT> tag

JSP .91 <REPEAT> tag syntax

<REPEAT> tag results set and the associated bean

JSP tags for database access

<DBCONNECT> tag syntax

<USERID> and <PASSWD> tag syntax

<DBQUERY> tag

<DBMODIFY> tag syntax

Compiling JSP .91 files as a batch

JSP .91 examples

Example: INSERT tag syntax

Example: <DBQUERY> tag syntax

Example: <DBMODIFY> tag syntax

Example: <INSERT> and <REPEAT> tags

Developing XML

What is XML?

XML application model

XML compared to HTML

XML advantages

What are XML Document Type Definitions (DTDs)?

What are well-formed XML documents?

What are valid XML documents?

What are style sheets?

What are the Document Object Model (DOM) and DOM objects?

Incorporating XML

[XML document structure](#)

[Creating or obtaining DTDs](#)

[Quick reference to DTD contents](#)

[Example of XML document and its DTD](#)

[Some existing DTDs](#)

[Associating DTDs with XML documents](#)

[Parsing XML documents](#)

[Creating an XML parser](#)

[Passing a parser name to create an XML parser](#)

[Explicitly instantiating an XML parser](#)

[Enabling XML parsers to handle errors without failing](#)

[Improving XML for Java Version 2.0.x parser processing time](#)

[New methods for the XML for Java API Version 2.0.x parsers](#)

[XML syntax quick reference](#)

[XML document content](#)

[Using document factory methods to generate XML document content](#)

[Example: Using document factory methods to generate XML document content](#)

[Extracting XML document content from a database](#)

[Using TXCatalog and XML Catalog for public identifier resolution](#)

[XML document presentation](#)

[Using XSL to convert XML documents to other formats](#)

[Converting XML documents to HTML at the Web server](#)

[Finding XML APIs, parsers, validators, and other resources](#)

[Using DOM to incorporate XML documents into applications](#)

[Quick reference to DOM object interfaces](#)

[Manually generating an XML element node](#)

[Revalidating a DOM tree after modifying it](#)

[Revalidating a DOM tree after modifying it](#)

[SiteOutliner sample](#)

Combining components and adding special features

[What are Web applications?](#)

[Planning, building, and deploying Web applications](#)

[Putting it all together \(Web applications\)](#)

Using WAR files

[What are WAR files?](#)

[What is WAR file conversion?](#)

[Converting to XML format](#)

[Converting WAR files \(GUI\)](#)

[wartoxmlconfig script \(command line\)](#)

[Converting to webapp format](#)

Accessing data with Web applications

[What is data access?](#)

[What is connection pooling?](#)

[Connection pooling advantages](#)

[How the product manages connection pools](#)

[What is the connection manager?](#)

[What is administrative data?](#)

[What is application data?](#)

[What are data access JavaBeans?](#)

[Features of data access JavaBeans](#)

[Obtaining and using database connections](#)

[Connection pooling with the JDBC 2.0 Standard Extension APIs](#)

[Example: Retrieving an object from a name service](#)

[How the JDBC 1.0 and 2.0 implementations differ in closing connections](#)

[Implementing connection pooling with servlets](#)

[Tips for utilizing connection pooling](#)

[Database access with the JDBC 1.0 reference model](#)

[Using IBM data access JavaBeans to access relational databases](#)

[Example: Servlet using data access JavaBeans](#)

[Database access by servlets and JSP files](#)

Adding personalization to Web applications

[Personalizing applications](#)

[What are sessions and Session Managers](#)

[What are session transactions?](#)

[What is session clustering?](#)

[What are cookies?](#)

[What are user profiles and User Profile Managers?](#)

[Tracking sessions](#)

[Session programming model and environment](#)

[Deciding between session tracking approaches](#)

[Using cookies to track sessions](#)

[Using URL rewriting to track sessions](#)

[Locking and unlocking session transactions](#)

[Securing sessions](#)

[Deciding between single to multirow schema for sessions](#)

[Using sessions in a clustered environment](#)

[Current limitations in session support](#)

[Tuning session support](#)

[Tuning session support: Session caching](#)

[Session support: Establish session affinity](#)

[Tuning session support: Multirow schema](#)

[Tuning session support: Manual update mode](#)

[Tuning session support: Base in-memory session pool size](#)

[Keeping user profiles](#)

[Data represented in the base user profile](#)

[Customizing the base user profile support](#)

[Extending data represented in user profiles](#)

[Adding columns to the base user profile implementation](#)

[Extending the User Profile enterprise bean and importing legacy databases](#)

[Accessing user profiles from a servlet](#)

Providing ways to invoke Web applications

[Providing ways for clients to invoke applications](#)

[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](#)

[Invoking JSP files within other JSP files](#)

[Providing Web clients access to servlets](#)

[Invoking servlets within SERVLET tags](#)

[Invoking servlets within JSP files](#)

Securing Web applications

[Securing Web applications from the inside and outside](#)

Performance considerations

[Programming high performance Web applications](#)

Language encoding considerations

[Setting language encoding in Web applications](#)

Special programming topics

[Employing pervasive computing](#)

[Using applets as clients](#)

[Example: Using applets as clients](#)

Programmatic login

[Programmatic and custom login](#)

[Client-side login](#)

[The TestClient](#)

[LoginHelper](#)

[Server-side login](#)

[The security TestServer](#)

[ServerSideAuthenticator](#)

[Accessing secured resources from Java clients](#)

[Custom login challenges](#)

[AbstractLoginServlet](#)

[CustomLoginServlet](#)

[SSOAuthenticator](#)

Distributing and load balancing applications

[What are models and clones?](#)

[What is workload management?](#)

Distribute and load balance

Managing workloads

Workload management for enterprise beans

Workload management for servlets

Workload management for administrative servers

Using models and clones

Cloning for workload management, failover, and scaling

Modifying models and clones

Advice for cloning

Containment relationships

Considerations for cloning servers

Using workload management - a sample procedure

Migrating applications

Migrating APIs and specifications:

Migrating to supported EJB specification

Migrating to supported Servlet specification and extensions

Example: Migrating `HttpServletResponse.callPage()`

Migrating to supported JSP specification

Updating JSP .91 files for use with Version 3.5

Tips for migrating JSP .91 files to JSP 1.0

Migrating to supported XML specification

Migrating to supported user profile APIs

Migrating to supported session APIs

Migrating from Version 2.0 session support

Migrating to supported security APIs

Migrating to supported database connection APIs (and JDBC)

Modifying import statements

Modifying `init()`

Obtaining connections

Utilizing JDBC data access

Closing connections

Modifying preemption handling

Deprecated connection manager APIs

Migrating to supported transaction support

Migrating to supported XML configuration

Migrating Web application files from Version 2.0x directories

[Optimally migrating Version 2.0x Web application files](#)

[Quickly migrating Version 2.0x Web applications](#)

0.1: What are enterprise applications?



An enterprise application is a combination of resources (building blocks) that work together to perform a business function. The resources can include:

- HTML files
- XML files
- JSP files
- servlets
- enterprise beans
- other elements, such as graphics

In fact, an *enterprise* application usually contains enterprise beans, but does not have to for the purposes of IBM WebSphere Application Server.

Although they can have the same contents, an *enterprise* application differs from a *Web* application, another type of resource in the world of IBM WebSphere Application Server (see the Related information for details). In the context of the product, "application" refers to an enterprise application.

A programmer or team of programmers might create an enterprise application to allow users at Web browsers to query a database. The application could have the following components:

Building block	Role in application
HTML form	Collect user queries
servlet	Prepare queries for processing by an enterprise bean
enterprise bean	Connect to the database and extract information based on queries
Second HTML page	Present the query results

Alternatively, the application programmers could omit the enterprise bean and have the servlet perform the data access. Or a JSP file or two could replace the HTML files and servlet.

After an administrator installs the application files where IBM WebSphere Application Server can find them, the administrator can manage the application as a single, logical unit.

An application server, combined with a Web server, makes the application available to users at Web browsers and processes their requests for the application.

Related information...

- [0.3: What are application servers?](#)
- [0.5: What are enterprise beans?](#)
- [0.8: What are Web applications?](#)
- [0.9: What are servlets?](#)
- [0.10: What are JSP files?](#)

0.3: What are application servers?



Application servers extend a Web server's capabilities to handle application requests. Given an application, the application server makes the following exchange possible:

1. A user at a Web browser on the public Internet visits a company Web site. The user requests to use an application that provides access to data in a database.
2. The user request flows to the Web server.
3. The Web server determines that the request involves an application containing resources not handled directly by the Web server (such as servlets). It forwards the request to IBM WebSphere Application Server.
4. The IBM WebSphere Application Server product forwards the request to one of its application servers on which the application is running.
5. The application processes the user request.

For example, a servlet prepares the user request for processing by an enterprise bean that performs the database access.

The application produces a Web page containing the results of the user query.

6. The application server collaborates with the Web server to return the results to the user at the Web browser.

The IBM WebSphere Application Server product (Advanced Edition) provides multiple application servers that can be either separately configured processes or nearly identical clones.

Related information...

- [0.3.1: Server terminology](#)
- [0.1: What are enterprise applications?](#)
- [0.5: What are enterprise beans?](#)
- [0.8: What are applications?](#)
- [0.9: What are servlets?](#)
- [0.22: What are models and clones?](#)
- [0.28: What are Web server plug-ins?](#)

0.3.1: Server terminology



This section defines a few frequently confused terms:

WebSphere Application Server

An IBM product providing one or more application servers for deploying enterprise applications.

To support the application servers, it provides a complete environment, including an administrative server and clients, tracing and debugging, performance monitoring, and many additional features.

WebSphere application server

One of possibly several application server processes within the WebSphere Application Server product. Its parts include an EJB server-container runtime and servlet engine support. Advanced Edition supports multiple application servers on a machine.

EJB server

A generic term for a server that handles components coded to the Enterprise JavaBeans specification. This functionality comprises part of an Advanced Edition WebSphere application server.

WebSphere administrative server

The server that handles administrative data for the WebSphere Application Server product. It is not the same thing as an application server. Rather, it maintains data about the configurations of application servers and their contents.

Enterprise JavaBeans (EJB) is a trademark of Sun Microsystems, Inc.

Related information...

- [0.4: What are enterprise bean containers?](#)
- [0.7: What are servlet engines?](#)
- [0.3: What are application servers?](#)

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, based on the concept introduced in the Java Servlet 2.1 specification, are groups of one or more servlets, plus static content.

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

See article 4.1 to review the WebSphere application programming model and environment, including how to find the specifications and IBM extensions supported by Version 3.5.

Consult sections 4.2 and 4.3 for a focus on developing Web applications and enterprise beans, respectively. Other Related information links help you bring these building blocks together, adding personalization, pervasive computing support, and other features.

Related information...

- [4.1: Programming model and environment](#)
- [4.2: Planning, building, and deploying Web applications](#)
- [4.3: Writing Enterprise Beans \(EJB\)](#)
- [4.4: Personalizing applications](#)
- [4.5: Employing pervasive computing](#)
- [4.6: Using applets as clients](#)
- [0.1: What are applications?](#)
- [3: Migration overview](#)
- [Index to API documentation \(Javadoc\)](#)

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.

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. Combine them into a Web application with the sought features
5. Create the enterprise beans
6. Combine the Web application and enterprise beans

Application developers might specialize in areas such as data access, Java programming, and Web page design. The layered approach provides a model allowing these programmers to collaborate in designing, implementing, deploying, and maintaining applications with maximum efficiency.

Related information...

- [4.1.1: Finding supported APIs and specifications](#)
- [4.1.1.1: Supported programming languages](#)
- [Index to API documentation \(Javadoc\)](#)
- [4: Developing applications](#)

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, Enterprise JavaBeans (EJB), 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 full InfoCenter:

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

Related information...

- [4.1.1.1: Supported programming languages](#)
- [4.1.1.2: Using the newest XML APIs](#)
- [4.1: Programming model and environment](#)

4.1.1.1: Supported programming languages



The Standard and Advanced Editions of IBM WebSphere Application Server are designed and tested to support applications and clients based on the **Java** programming language and technologies.

Accessing native libraries in C or C++ from Java servlets or Enterprise JavaBeans (Advanced Edition only) deployed on IBM WebSphere Application Server is possible with the right coding and classpath practices.

The IBM WebSphere Application Server Enterprise Edition is the **recommended** solution for environments requiring C and C++ clients. The Enterprise Edition supports CORBA, COM, and DCOM clients in addition to the Java and browser clients supported by the Advanced and Standard Editions.

Related information...

- [Product Web site](#) (for Enterprise Edition information)
- [Index to API documentation \(Javadoc\)](#)
- [4.1.1: Finding supported APIs and specifications](#)

4.1.1.2: Support for the newest XML APIs



IBM WebSphere Application Server Version 3.5 officially supports XML4J API 2.0.15.

Newer XML APIs are available:

- Apache Xerces and Xalan versions of the APIs
- IBM XML4J 3.0.1 and LotusXSL APIs

It is likely the above XML APIs are partially or completely compatible with Version 3.5. They cannot be recommended because they have not been formally tested with Version 3.5.

However, if you use the APIs presented in this documentation, it is likely you will need to migrate your code to the newer APIs in the near future. For example, the TX compatibility classes referenced in article 4.2.3.2.5 are no longer the recommended way to construct document elements in the newer versions of Xerces and XML4J.

Therefore, if you currently have applications using the newer XML APIs, you might want to try them with Version 3.5 in a test environment. If they run successfully, you can avoid using the officially supported XML 2.0.15 APIs, for which migration is imminent.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.1.1: Finding supported APIs and specifications](#)

0.4: What are enterprise bean containers?



The enterprise beans installed in an application server do not communicate directly with the server; instead, an enterprise bean *container* provides an interface between the enterprise beans and the server. Together, the container and server provide the bean runtime environment.

The container provides many low-level services, including threading and transaction support. Perhaps most important from an administrative viewpoint, the container manages data storage and retrieval for the beans within.

One or more JAR files can be installed in a single container.

Related information...

- [0.3: What are application servers?](#)
- [0.5: What are enterprise beans?](#)

0.5: What are enterprise beans?



An enterprise bean is a Java component that can be combined with other resources to create distributed client/server applications.

There are two types of enterprise beans, entity beans and session beans:

- Entity beans require database connections because they store permanent data.
- Session beans do not *require* database access, though they can obtain it indirectly (as needed) by accessing entity beans.

Beans requiring data access use *data sources*, administrative resources defining pools of database connections.

All beans reside in enterprise bean containers, which provide an interface between the beans and the application server on which they reside.

Related information...

- [0.4: What are enterprise bean containers?](#)

0.22: What are models and clones?



A *model* is a template for creating additional, nearly identical copies of an administrative resource and its contents. Usually, the administrative resource is an application server. The resource is referred to as the *original*. The copies are called *clones*.

The clones can be used for workload management purposes. That is, a request for the original resource can be handled by the original or any of its clones. The clones might be distributed (located on different machines). Cloning also supports failover and vertical scaling.

Related information...

- [0.22.1: What is workload management?](#)

0.22.1: What is workload management?



Workload management optimizes the distribution of client processing tasks. Incoming work requests are distributed to the application servers, enterprise beans, servlets, and other objects that can most effectively process the requests. Workload management also provides failover when servers are not available, improving application availability.

In the WebSphere Application Server environment, workload management is implemented by using modeling and cloning; servlet redirection; and workload management-enabled Java Archive (JAR) files for enterprise beans. Administrative servers can also participate in workload management.

Related information...

- [0.22: What are models and clones?](#)

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 Version 3.5. It discusses both the Advanced Edition and Enterprise Edition of the product.

Format

[PDF](#)

[HTML](#)

See section 4.3.1 for additional information that could not be added to the book in time for this product release.

Related information...

- [4.3.1: Late-breaking enterprise beans programming tips](#)
- [4.3.2: JNDI caching](#)
- [0.5: What are enterprise beans?](#)
- [Product Web site](#) (for Enterprise Edition information)
- [Index to API documentation \(Javadoc\)](#)
- [4: Developing applications](#)

4.3.1: Late-breaking enterprise beans programming tips



This article provides programming tips and considerations to supplement the Writing Enterprise Beans book. Also see the product Release Notes.

Inheritance by remote objects

An enterprise bean or other remote object cannot inherit from two interfaces that have methods with the same name, even if those methods have different signatures, due to the Java-IDL mapping specification.

Java programmers accustomed to the usual Java inheritance model should take care to note this limitation of the specification. By the Enterprise JavaBeans (EJB) specification, enterprise beans should not be written to inherit from two interfaces as described above. If they do, they will encounter errors when deployed.

Option A caching incompatible with clusters and shared data

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:

- Workload managed servers (such as a cluster of clones)
- Database with data being shared among multiple applications

Shared database access corresponds to Option C caching. See the EJB specification for further details.

Option A and Option C caching are also known as commit option A and commit option C, respectively.

Best practice for data source ID and password

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 the container of the bean.

Addition to book: Developer's Client Files

The English version of the book already has this addition, but the national language versions do not. To the following paragraph in "Developing EJB clients":

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 (AE) environment or Setting the CLASSPATH environment variable in the EJB server (CB) environment.

Add the text:

You can install needed files on your client machine by doing a WebSphere Application Server install on the client machine, selecting the "Developer's Client Files" option. You also need to make sure that the ioser/ioserx executable files are accessible on your client machine (these files are normally part of the Java 1.2.x install).

Committing transactions based on EJB 1.1 specification

According to the EJB 1.0 and 1.1 specifications, if an enterprise bean container catches an exception from the business method of an enterprise bean, and the method is running within a container managed transaction, the container should rollback the transaction before passing the exception on to the client.

However, if the business method is throwing an Application exception as defined in Chapter 12 of EJB 1.1 specification, then the normal behavior for the container in this case is to COMMIT the transaction. Even though IBM WebSphere Application Server Version 3.5 does not officially support the EJB 1.1. specification level, in such a case it behaves as determined by the 1.1. specification. If a business method throws an exception, the container will commit the transaction before re-throwing the exception.

EJB clients need ioser library to run

If using Windows NT, ensure that EJB clients can locate the following library file at their run time: ioser.dll

Related information...

- [Release notes \(Version 3.5\)](#)
- [Index to API documentation \(Javadoc\)](#)
- [4: Developing applications](#)

About this book

This document focuses on the development of enterprise beans written to the Sun Microsystems Enterprise JavaBeans^(TM) Specification in the WebSphere 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^(R) 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 in the EJB server \(AE\) environment](#) explains how to set up and use the tools contained in the EJB server (AE) environment. It also discusses the major steps in developing and deploying enterprise beans in that environment. The EJB server (AE) is the EJB server implementation available with the WebSphere Application Server Advanced Edition.
- [Tools for developing and deploying enterprise beans in the EJB server \(CB\) environment](#) explains how to set up and use the tools contained in the EJB server (CB) environment. It also discusses the major steps in developing and deploying enterprise beans in that environment. The EJB server (CB) is the EJB server implementation available with Component Broker as part of the WebSphere Application Server Enterprise Edition.
- [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, 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 B, Using XML in enterprise beans](#) describes the extensible markup language (XML) that can be used to create deployment descriptors for use with enterprise beans in WebSphere.
 - [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:

- Building Business Solutions with WebSphere
 - Component Broker Problem Determination Guide
 - Component Broker System Administration Guide
 - Getting Started with TXSeries
 - Getting Started with Advanced Edition
 - Getting Started with Component Broker
 - Component Broker Release Notes
-

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 waseedoc@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).

Copyright [IBM Corporation 1999](#). All Rights Reserved

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

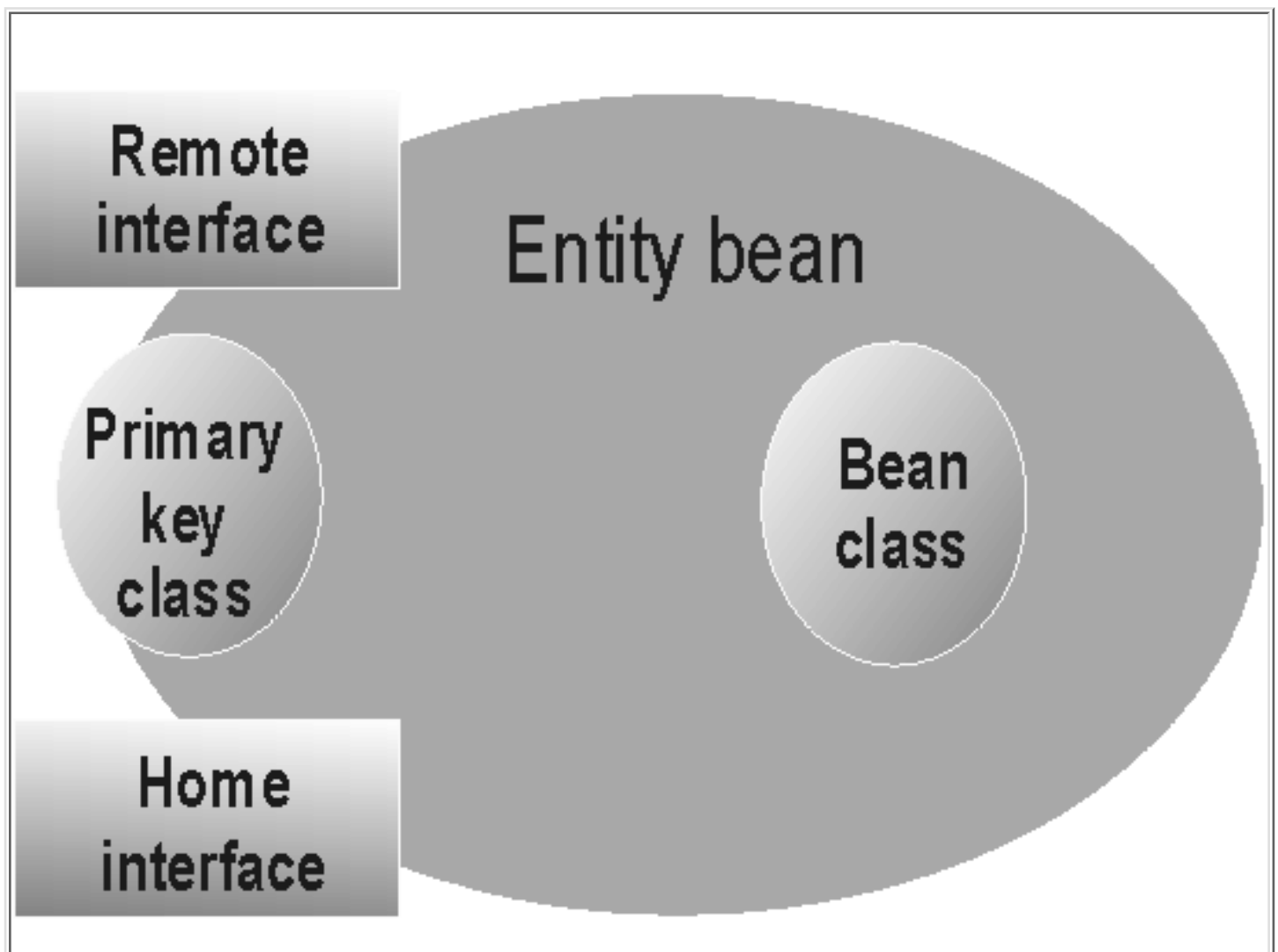
Every entity bean must have the following components, which are illustrated in [Figure 3](#):

- *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 class*--This class encapsulates one or more variables that uniquely identify a specific entity bean instance. It 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. You must use entity beans with BMP if you want to use a data source that is not supported by the EJB server. 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.

Session beans

This section discusses the basics of session beans.

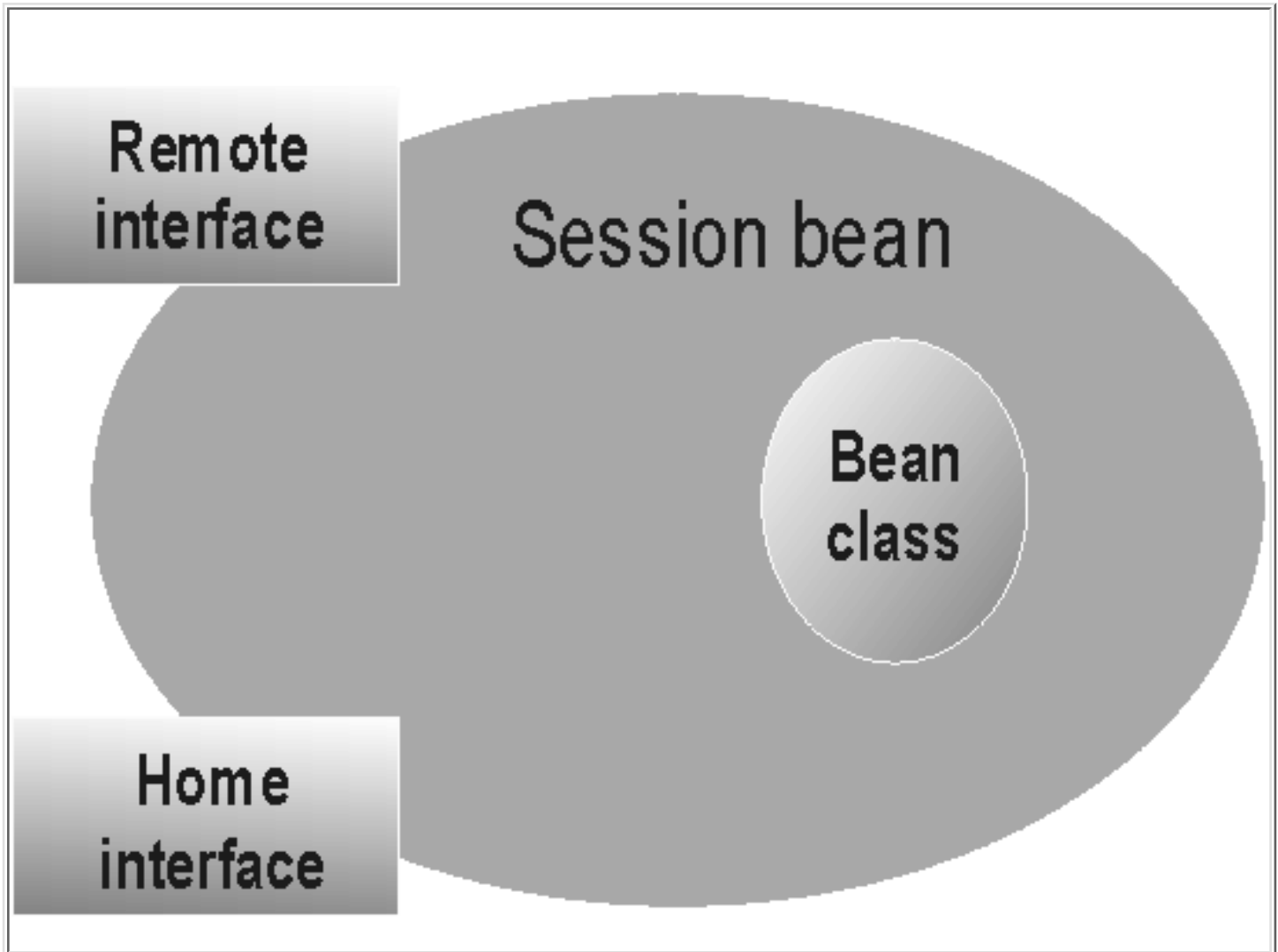
Basic components of a session bean

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, a session bean represents 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.
-

Packaging enterprise beans

The last step in the development of an enterprise bean is the creation of the deployment descriptor and the EJB JAR file. After the EJB JAR file is created, the enterprise bean can be deployed into the container of an EJB server.

The deployment descriptor

The *deployment descriptor* contains attribute and environment settings that define how the container invokes enterprise bean functionality. Every enterprise bean (both session and entity) must have a deployment descriptor that contains settings for the following attributes; these 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.

- *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. The values for this attribute are described in [Creating and getting a reference to a bean's EJB object](#).
- *Transaction* attribute--Defines the transactional manner in which the container invokes a 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. The values for this attribute are described in [Enabling transactions and security in enterprise beans](#).
- *Access control* attribute--Defines an access control entry that identifies users or roles that are permitted to access the methods in the enterprise bean. This value is not used by the WebSphere EJB servers.
- *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. The *RunAsMode* attribute is used by the WebSphere EJB

servers; the *RunAsIdentity* attribute is not. The values for the *RunAsMode* attribute are described in [Enabling transactions and security in enterprise beans](#).

The deployment descriptor for an entity bean must also contain settings for the following attributes. These attributes can be set on the bean only; they cannot be set on a per-method level.

- *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 \(entity with BMP\)](#).
- *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](#).

The deployment descriptor for a session bean must also contain settings for the following attributes. These attributes can be set on the bean only; they cannot be set on a per-method level.

- *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.

Deployment descriptors can be created by using the tools within an integrated development environment (IDE) such as IBM VisualAge^(R) for Java Enterprise Edition or by using the stand-alone tools contained in Websphere Application Server. For more information, see [Tools for developing and deploying enterprise beans in the EJB server \(AE\) environment](#) or [Tools for developing and deploying enterprise beans in the EJB server \(CB\) environment](#).

The EJB JAR file

The *EJB JAR file* is used to package enterprise beans; this file uses the standard Java archive file format. The EJB JAR file can be used to contain individual enterprise beans, multiple enterprise beans, and entire enterprise bean-based applications. For more information, see [Creating an EJB JAR file](#).

An EJB JAR file can be created by using the tools within an integrated development environment (IDE) like IBM's VisualAge for Java or by using the stand-alone tools contained in Websphere. For more information, see [Tools for developing and deploying enterprise beans in the EJB server \(AE\) environment](#).

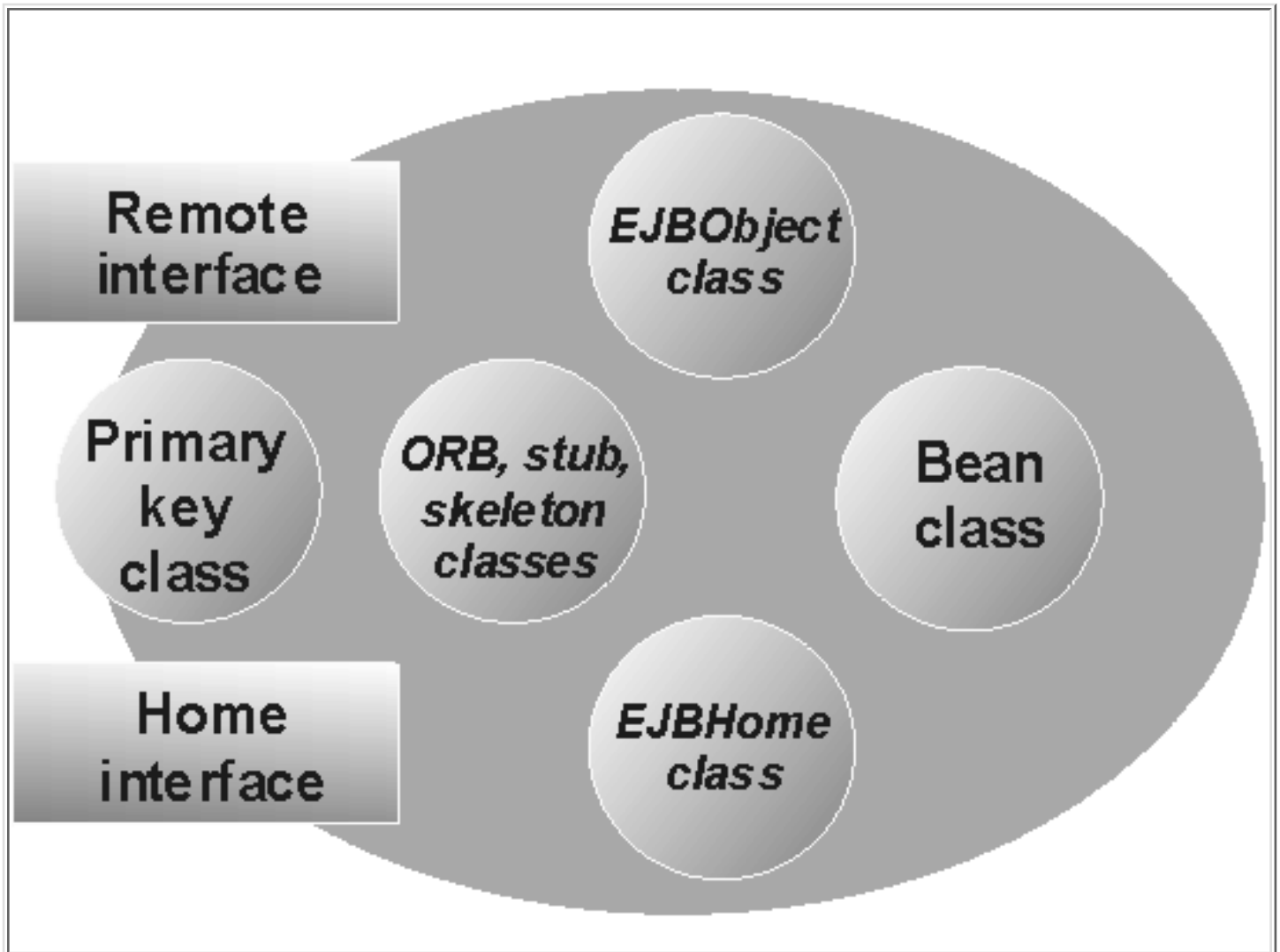
Deploying an enterprise bean

When you deploy an enterprise bean, 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

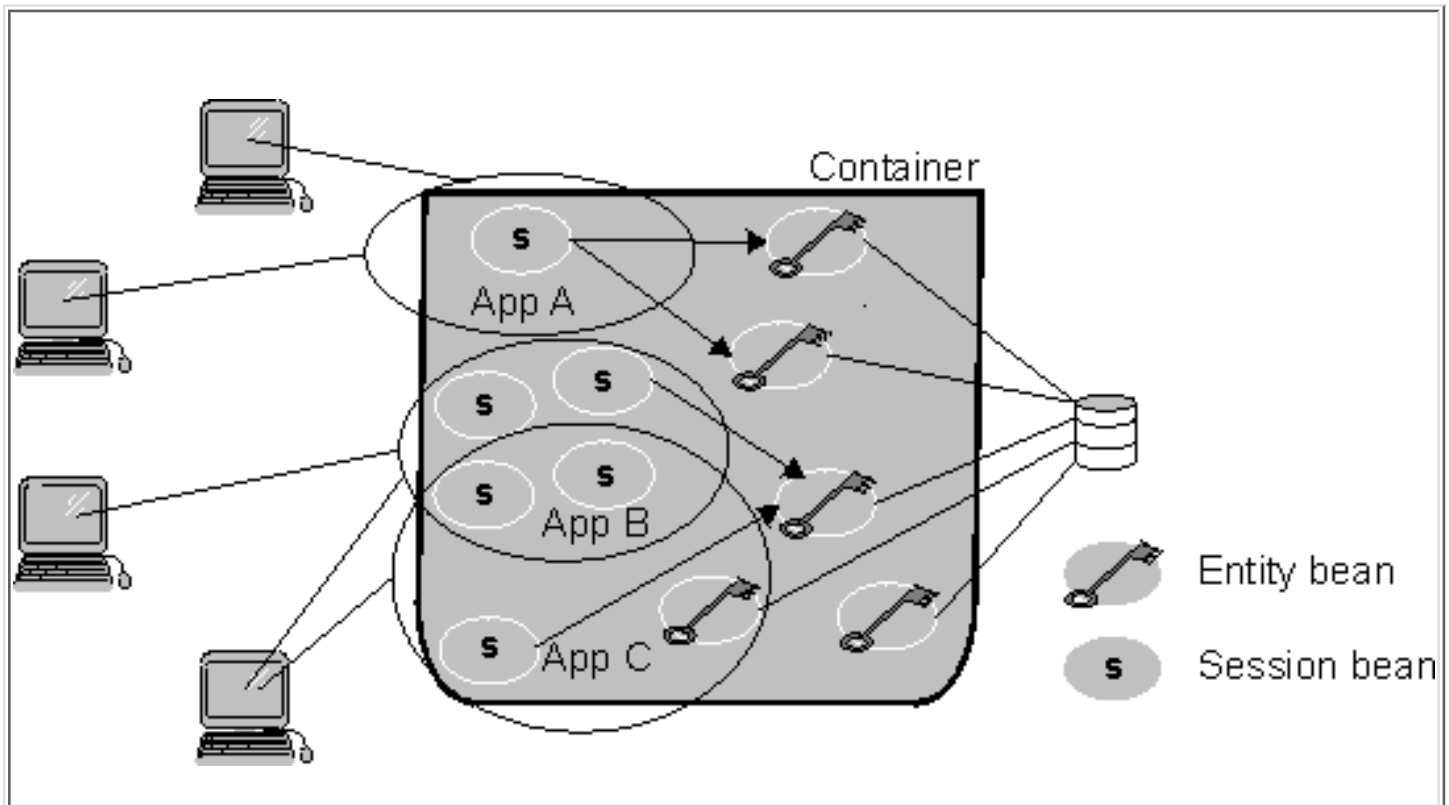


You can deploy an enterprise bean with a variety of different tools. For more information, see [Tools for developing and deploying enterprise beans in the EJB server \(AE\) environment](#) or [Tools for developing and deploying enterprise beans in the EJB server \(CB\) environment](#).

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](#).

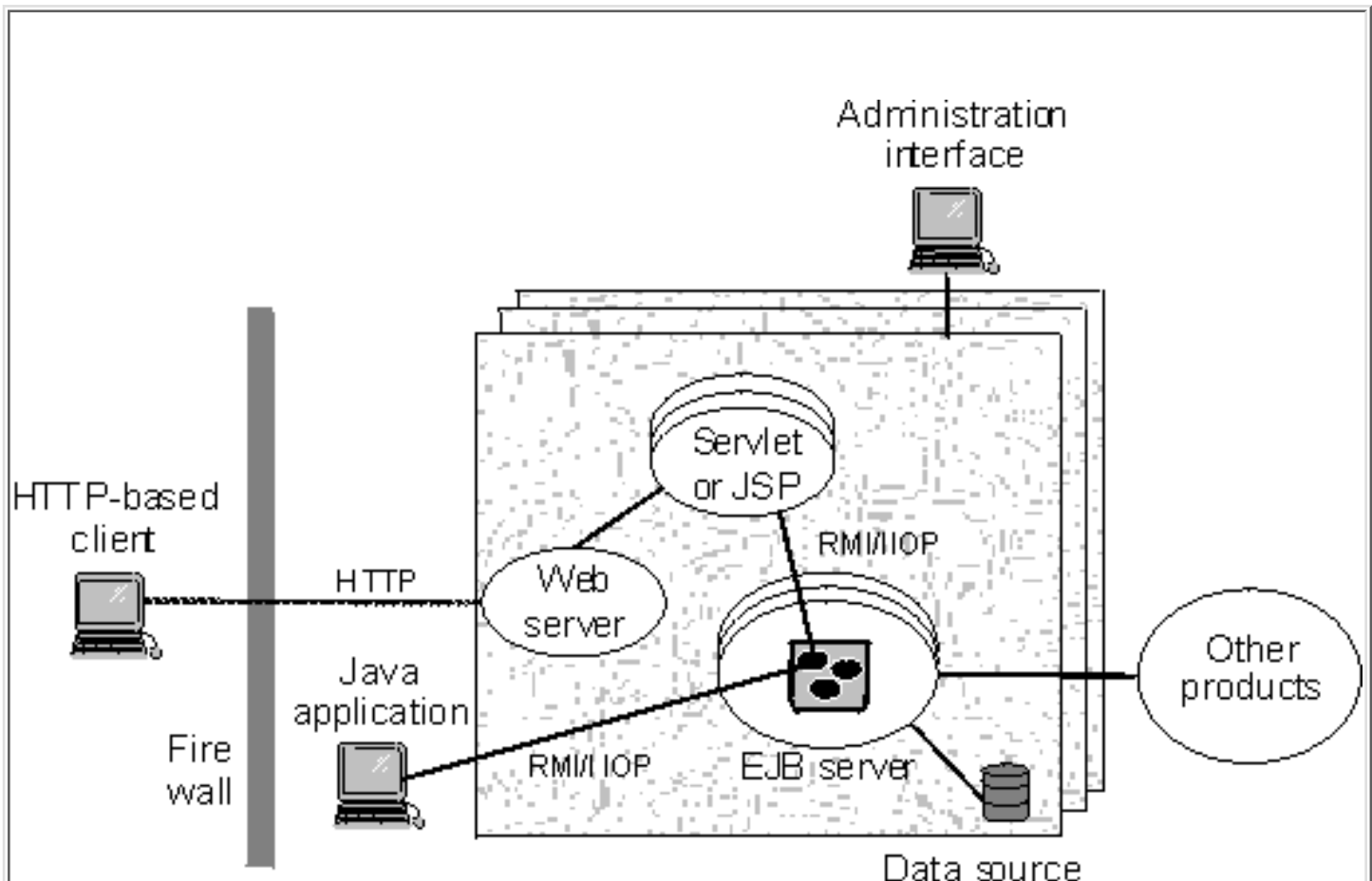
An architectural overview of the EJB programming environment

In the past few years, 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 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^(TM) (EJB) Specification enables users of the WebSphere Application Server Advanced Edition and WebSphere Application Server Enterprise Edition 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 (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, connecting the client tier (Java servlets, applets, applications, and JSP) with the resource management tier (the data source). The WebSphere Application Server contains two types of EJB servers. If you have the Advanced Application Server, you get only one of these EJB servers; if you have the Enterprise Application Server, you get both. When referring generically to EJB servers, this documentation uses the phrase *EJB server*; when the documentation needs to refer specifically to one or the other, it uses the following terms:

- *EJB server (AE)*--The EJB server that comes with the Advanced Application Server. (Because Advanced Application Server is available as a part of Enterprise Application Server, this EJB server is also available with Enterprise Application Server.)
- *EJB server (CB)*--The EJB server that comes only with the Enterprise Application Server and is part of Component Broker (CB).

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.

The EJB server (CB) includes two standard types of containers: entity containers and session containers. As their names imply, these containers are specifically optimized to handle entity beans and session beans, respectively. The EJB server (AE) has one standard container that supports both entity and session beans.

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 a *deployment descriptor* that defines properties associated with the bean; the deployment descriptor and the other enterprise bean components are packaged in a file known as an *EJB JAR file*. For more information on deployment, see [Deploying an enterprise bean](#).
- 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 (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 in the EJB server environment

There are some similarities between the security service in the two EJB server environments. In both EJB server environments, 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 major differences between the two security services are discussed in the following sections.

Security in the EJB server (AE) environment

In the EJB server (AE) environment, 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 in the EJB server (AE) environment, see the online help available with the WebSphere Administrative Console.

Security in the EJB server (CB) environment

In the EJB server (CB) environment, you must secure all the Component Broker name servers and applications servers in the network. Securing the name server on each server host prevents unauthorized access to the system objects (including name contexts used in the Component Broker namespace) in that server. Securing an application server prevents unauthorized access to the business objects for applications in that server.

To secure your name servers and application servers, you must do the following:

- Install and configure the Distributed Computing Environment (DCE) to provide authentication services to the servers. This allows secure access between servers.
- Configure key rings for clients and servers to provide authentication services to Java-based SSL clients.
- Configure authorization for access to business objects in the application service.
- Create a delegation policy to allow the application server to pass the requesting client principal to other servers.
- Configure credential mapping to provide access to any third tier system.
- Configure the qualities of protection to be used to protect messages that flow between clients and the application server.

The Component Broker System Administration Guide provides more detail about each of these tasks.

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 the EJB server (AE) environment and from within the Systems Management End User Interface for the EJB server (CB) environment. For more information on workload management, consult 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.

In the EJB server (AE) environment, 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.

In the EJB server (CB) environment, the persistence service uses the following components to accomplish its task:

- The X/Open XA interface, which gives entity beans a standard interface to relational databases.
- The Object Management Group's (OMG) Object Transaction Service (OTS), which is also discussed in [Using transactions in the EJB server environment](#).

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.

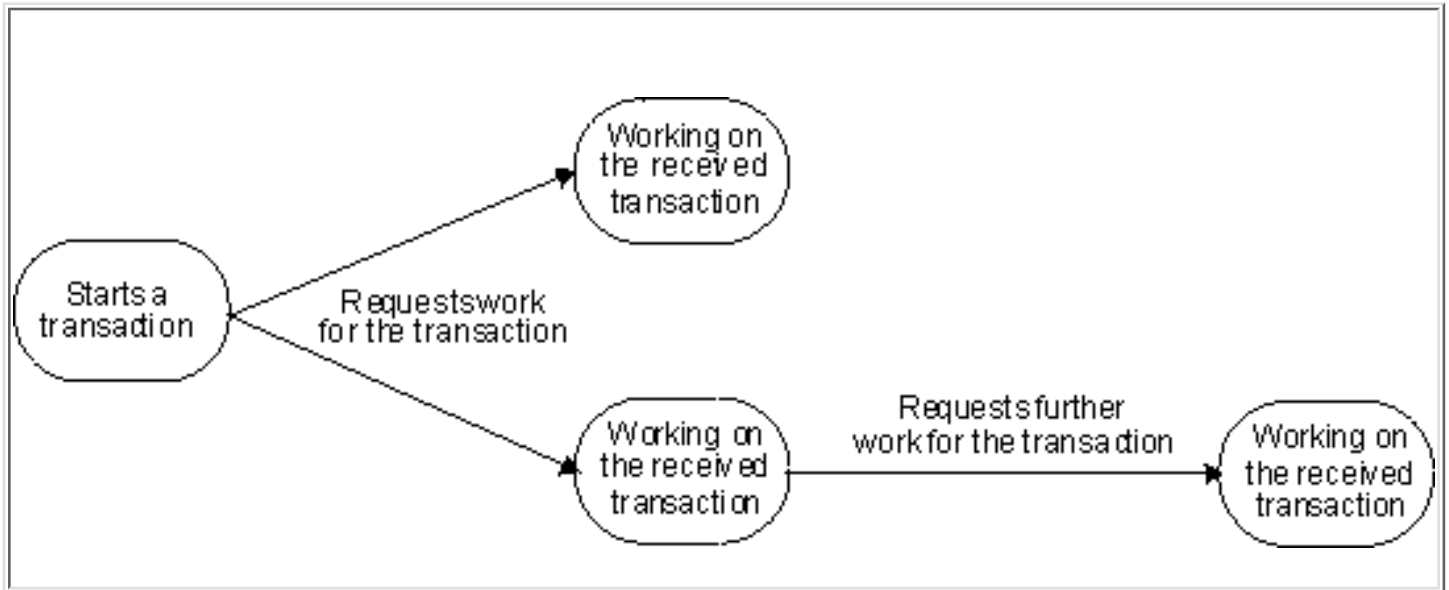
Withdrawing the money from one account and depositing it in the other account are two parts of an *atomic* transaction: if both cannot be completed, neither must happen. If multiple requests are processed against an account at the same time, they must be *isolated* so that only a single transaction can 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:

- The EJB server (AE) supports DB2^(R), Oracle, Sybase, and InstantDB.
- The EJB server (CB) supports DB2, Oracle, CICS^(R), IMS^(TM), and MQSeries^(R).

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. For the EJB server (CB), a Java applet can be used to directly interact with enterprise beans. For the EJB server (AE), a Java applet can be used only in combination with a servlet.

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 (AE) 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.

- All EJB clients that access an EJB server (CB) and EJB clients that access an EJB server (AE) by using RMI (for example, Java applications) encounter the second security layer only. Like HTTP-based EJB clients, these EJB clients must authenticate with the security service.

For more information, see [The security service](#).

- *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](#).

In the EJB server (CB) environment, an enterprise bean can also be accessed by EJB clients that use Microsoft ActiveXTM, CORBA-based Java, and to a limited degree, CORBA-based C++.

[More information on EJB clients specific to the EJB server \(CB\)](#) provides additional information.

Note: In the EJB server (AE) environment, ActiveX and CORBA-based access to enterprise beans is not supported.

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 (CB) and EJB server (AE) each have their own administration tools:

- The EJB server (AE) uses the WebSphere Administrative Console. For more information on this interface, consult the online help available within the WebSphere Administrative Console.
- The EJB server (CB) uses the System Management End User Interface (SM EUI). For more information on this interface, see the Component Broker System Administration

Guide.

You can also administer the EJB server (AE) using the **wscp** command-line tool. For more information, see the Advanced Edition Information Center.

Copyright [IBM Corporation 1999](#). All Rights Reserved

WebSphere Programming Model Extensions

This section discusses the two 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](#).

These packages are available as part of WebSphere Application Server Advanced Edition and Enterprise Edition. They are general-purpose utilities, designed to provide common functions in a reusable way. Although these two 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*.

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 73](#) 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 73. 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 resourceName,
                               String resourceKey,
                               Object[] formatArguments,
                               String defaultText)
    {...}
    public DistributedException(String resourceName,
                               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 73](#)). The constructors can simply pass arguments to the constructor in the DistributedException class by using super methods, as illustrated in [Figure 74](#).

Figure 74. 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 75](#) 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 75. 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 73](#)). 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 76](#) shows standard implementations for the six constructors.

Figure 76. 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 resourceName, String resourceKey,
                           Object[] formatArguments, String defaultText)
    {
        super ();
        exceptionInfo = new DistributedExceptionInfo(resourceName,
                                                    resourceKey, formatArguments, defaultText, this);
    }
    AccountCreateException(String resourceName, String resourceKey,
                           Object[] formatArguments, String defaultText,
                           Throwable exception)
    {
        super ();
        exceptionInfo = new DistributedExceptionInfo(resourceName,
                                                    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 77](#) 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 77. 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 78](#) shows the use of the `instanceof` method to test for exception chaining.

Figure 78. 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 79](#) shows the use of the `MyDistributedException(Throwable)` constructor.

Figure 79. 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 80](#) illustrates calling the `getPreviousException` and `getOriginalException` methods.

Figure 80. 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();
            ...
        }
    }
}
...

```

The command package

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 81](#) shows the structure of a command interface for a targetable command.

Figure 81. 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 82](#) shows the structure of a command interface for a targetable, compensable command.

Figure 82. 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 83](#) shows the structure of an implementation class for the interface in [Figure 82](#).

Figure 83. 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 84](#) shows the structure of the remote interface and enterprise bean class for an entity bean that implements the `CommandTarget` interface.

Figure 84. 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 methods 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 85](#) shows the interface for the `ModifyCheckingAccountCmd` command. This command is both targetable and compensable, so the interface extends both `TargetableCommand` and `CompensableCommand` interfaces.

Figure 85. 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 86](#) shows the structure of the `ModifyCheckingAccountCmdImpl` class.

Figure 86. 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 87](#) shows the variables used by the `ModifyCheckingAccountCmd` command.

Figure 87. 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 88](#) 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 97](#)). 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 88. 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 89](#) 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 89. 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 90](#) 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 90. 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 91](#) 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 91. 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 92](#) shows the implementation for the ModifyCheckingAccountCmd command. The implementation simply returns an instance of the ModifyCheckingAccountCompensatorCmd command associated with the current command.

Figure 92. 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 93](#) 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 93. 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 94 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 94. 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 95](#) 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 95. 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 96](#) 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 96. 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());
}
...
}

```

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 97](#) shows the `CheckingAccount` interface.

Figure 97. 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 amout) throws RemoteException;
    float deposit (int amout) throws RemoteException;
    String getAccountName() throws RemoteException;

    float getBalance() throws RemoteException;
    float setBalance(float amount) throws RemoteException;

    float withdrawal (float amout) throws RemoteException, Exception;
    float withdrawal (int amout) 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 98](#) shows the relevant parts of the CheckingAccountBean class.

Figure 98. 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) {
                RemoteException 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 99](#) shows the relevant variables and the methods in the `TargetPolicyDefault` class.

Figure 99. 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 setDefaultTargetName(String defaultTargetName) {
        ... }
}
```

Setting the command target

The `ModifyCheckingAccountImpl` class provides two command constructors (see [Figure 88](#)). 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 95](#) 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 100](#) uses the same constructor to set the target explicitly. This example differs from [Figure 95](#) 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 100. 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 101](#) illustrates this technique. This example compares with [Figure 95](#) 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 101. 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 102](#) 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 95](#), 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 102. 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 103](#) shows a simple custom policy that sets the target of every command to `MySessionBean`.

Figure 103. 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 88](#)). 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 104](#) 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 104. 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 105](#) shows the structure of the client-side class; it implements the CommandTarget interface by implementing the executeCommand method.

Figure 105. 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 106](#).

Figure 106. 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 107](#) 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 107. 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");
        URLConnection httpURLConnection =
            (URLConnection) 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 108](#). 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 108. 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 use 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.

Copyright [IBM Corporation 1999](#). All Rights Reserved

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.

Note: In the EJB server (CB) environment, do not duplicate unqualified interface and exception names in enterprise beans. For example, the `com.ibm.ejs.doc.account.Account` interface must not be reused in a package named `com.ibm.ejs.doc.bank.Account`. This restriction is necessary because the EJB server (CB) tools generate enterprise bean support files that use the unqualified name only.

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.

[Figure 18](#) 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 18. Code example: The AccountBean class

```
...
import java.rmi.RemoteException;
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.

Note: In the EJB server (CB) environment, container-managed fields in entity beans must be valid for use in CORBA IDL files. Specifically, the variable names must use ISO Latin-1 characters, they must *not* begin with an underscore character (`_`), they must *not* contain the dollar character (`$`), and they must *not* be CORBA keywords. Variables that have the same name but different cases are not allowed. (For example, you cannot use the following variables in the same class: *accountId* and *AccountId*. For more information on CORBA IDL, consult a CORBA programming guide.

Also, container-managed fields in entity beans must be valid Java types, but they *cannot* be of type `ejb.javax.Handle` or an array of type `EJBObject` or `EJBHome`. Furthermore, container-managed fields of type `String` (or arrays of type `String`) *cannot* be set to null at run time because these types map to CORBA IDL type `string` or `wstring`, which are prohibited by CORBA from having null values.

The `AccountBean` class contains three container-managed fields (shown in [Figure 19](#)):

- *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 19. 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 20](#) 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 25](#). 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: In the EJB server (CB) environment, use of underscores (`_`) in the names of user-defined interfaces and exception classes is discouraged.

Figure 20. 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;
    }
    ...
}

```

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 void.
- 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 `java.rmi.RemoteException` exception and the `javax.ejb.CreateException` exception.

[Figure 21](#) 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 21. 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 void ejbCreate(AccountKey key) {
        ejbCreate(key, 1, 0.0f);
    }
    ...
    public void ejbCreate(AccountKey key, int type, float initialBalance)
    throws RemoteException {
        accountId = key.accountId;
        type = type;
        balance = initialBalance;
    }
    ...
    public void ejbPostCreate(AccountKey key) throws RemoteException {
        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, and they can throw the `java.rmi.RemoteException` exception.

- `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 22](#), 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 the these methods. The `setEntityContext` and `unsetEntityContext` methods are used in a conventional way to set the value of the `entityContext` variable.

Figure 22. Code example: Implementing the `EntityBean` interface in the `AccountBean` class

```

...
public class AccountBean implements EntityBean {
    private EntityContext entityContext = null;
    ...
    public void ejbActivate() throws RemoteException { }
    ...
    public void ejbLoad () throws RemoteException { }
    ...
    public void ejbPassivate() throws RemoteException { }
    ...
    public void ejbRemove() throws RemoteException { }
    ...
    public void ejbStore () throws RemoteException { }
    ...
    public void setEntityContext(EntityContext ctx) throws RemoteException {
        entityContext = ctx;
    }
    ...
    public void unsetEntityContext() throws RemoteException {
        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 23](#) 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 23. 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 23](#)).
- 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 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 or the `java.util.Enumeration` interface (when a finder method can return more than one EJB object).
- 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 24](#). Because this finder method can be expected to return a reference to more than one EJB object, its return type is `Enumeration`.

Figure 24. 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 \(AE\)](#) or [Creating finder logic in the EJB server \(CB\)](#).

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.

Note: In the EJB server (CB) environment, do not use method names in the remote interface that match method names in the Component Broker Managed Object Framework (that is, methods in the `IManagedServer::IManagedObjectWithCachedDataObject`, `CosStream::Streamable`, `CosLifeCycle::LifeCycleObject`, and `CosObjectIdentity::IdentifiableObject` interfaces). For more information on the Managed Object Framework, see the Component Broker Programming Guide. In addition, do not use underscores (`_`) at the end of property or method names; this restriction prevents name collision with queryable attributes in business object interfaces that correspond to container-managed fields.

[Figure 25](#) 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 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 25. 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.

The primary key class is used to encapsulate 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`.

A primary key class is used to create and manage the primary key for an EJB object. 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: For the EJB server (AE) environment, 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 26](#) shows the primary key class for the Account enterprise bean. In effect, this class acts as a wrapper around the primitive long variable *accountId*. The `hashCode` method for the `AccountKey` class simply invokes the corresponding `hashCode` method in the `java.lang.Long` class after creating a temporary `Long` object by using the value of the *accountId* variable. In addition to the default constructor, the `AccountKey` class also defines a constructor that sets the value of the primary key variable to a specified long.

More complicated enterprise beans are likely to have composite primary keys, with multiple instance variables representing the primary key.

Figure 26. Code example: The AccountKey primary key class

```

...
import java.io.*;
public class AccountKey implements Serializable {
    public long accountId;
    ...
    public AccountKey() {
        super();
    }
    ...
    public AccountKey(long accountId) {
        this.accountId = accountId;
    }
    ...
    // EJB server (AE)-specific method
    public boolean equals(Object o) {
        if (o instanceof AccountKey) {
            AccountKey otherKey = (AccountKey) o;
            return ((accountId == otherKey.accountId));
        }
        else {
            return false;
        }
    }
    ...
    // EJB server (AE)-specific method
    public int hashCode() {
        return ((new Long(accountId).hashCode()));
    }
}

```

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](#).

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.

Note: The `SessionSynchronization` interface is *not* supported in the EJB server (CB) environment.

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 27](#) 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 27. 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 RemoteException { }
    ...
    public void ejbCreate() throws RemoteException {
        ...
    }
    ...
    public void ejbPassivate() throws RemoteException { }
    ...
    public void ejbRemove() throws RemoteException { }
    ...
}
```

```

    public float getBalance(long acctId) throws FinderException,
        RemoteException {
        ...
    }
    ...
    public void setSessionContext(javax.ejb.SessionContext ctx)
        throws java.rmi.RemoteException {
        ...
    }
    ...
    public void transferFunds(long fromAcctId, long toAcctId, float amount)
        throws java.rmi.RemoteException {
        ...
    }
}

```

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 28](#) 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 28. 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, RemoteException {
        AccountKey key = new AccountKey(acctId);
        try {
            fromAccount = accountHome.findByPrimaryKey(key);
        } catch(FinderException ex) {
            throw new FinderException("Account " + acctId + " does not exist.");
        }
        return fromAccount.getBalance();
    }
}

```

```

}
...
public void transferFunds(long fromAcctId, long toAcctId, float amount)
    throws RemoteException, 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.");
    }
    try {
        toAccount = accountHome.findByPrimaryKey(toKey);
    } catch(FinderException ex) {
        throw new FinderException("Account " + toAcctId
            + " does not exist.");
    }
    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. 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.

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.) Like 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.

Each `ejbCreate` method must meet the following requirements:

- It must return `void`.
- It must contain code to set the values of any variables needed by the EJB object.

[Figure 29](#) 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 29. 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 RemoteException {
        // 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

([Setting environment variables for an enterprise bean](#) shows an example of the **jetace** page required to set these variables.)

[Figure 30](#) 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 30. 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

After constructing the InitialContext object (*initialContext*), 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 31. 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 RemoteException {
        // 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(
                (org.omg.CORBA.Object) ejbHome, AccountHome.class);
        } catch (NamingException e) { // Error getting the home interface
            ...
        }
        ...
    }
    ...
}

```

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, return void, and throw the `java.rmi.RemoteException` exception.

- `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](#).

Note: In the EJB server (CB) environment, the `isCallerInRole` and `getCallerIdentity` methods inherited from the `javax.ejb.EJBContext` interface are not supported.

As shown in [Figure 32](#), 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 32. Code example: Implementing the `SessionBean` interface in the `TransferBean` class

```
...
public class TransferBean implements SessionBean {
    private SessionContext mySessionCtx = null;
    ...
    public void ejbActivate() throws RemoteException { }
    ...
    public void ejbPassivate() throws RemoteException { }
    ...
    public void ejbRemove() throws RemoteException { }
    ...
    public void setSessionContext(SessionContext ctx) throws RemoteException {
        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 33](#) shows the relevant parts of the definition of the home interface (TransferHome) for the example Transfer bean.

Figure 33. 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 34](#) 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 34. 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 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 (AE) enforces single-threaded access to all enterprise beans. Illegal callbacks result in a java.rmi.RemoteException exception being thrown to the EJB client.

The EJB server (CB) enforces single-threaded access to enterprise beans only if their transaction attribute is set to either TX_NOT_SUPPORTED or TX_BEAN_MANAGED. For other enterprise beans, access from different transactions is serialized, but serialized access from different threads running under the same transaction is not enforced. For enterprise beans deployed with the transaction attribute value of TX_NOT_SUPPORTED or TX_BEAN_MANAGED, illegal callbacks result in a RemoteException exception being thrown to the EJB client.

Packaging enterprise beans

There are three tasks involved in packaging an enterprise bean:

- 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 a deployment descriptor for the bean. For more information, see [Creating the deployment descriptor file](#).
- Creating an EJB JAR file. For more information, see [Creating an EJB JAR file](#).

If you develop enterprise beans in an IDE, these packaging 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 package beans in the EJB server (AE) programming environment, see [Tools for developing and deploying enterprise beans in the EJB server \(AE\) environment](#).
- For more information on the tools used to package beans in the EJB server (CB) programming environment, see [Tools for developing and deploying enterprise beans in the EJB server \(CB\) environment](#).

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 the deployment descriptor file

The deployment descriptor file provides instructions for the container on how to handle a particular bean. A standard deployment descriptor must support the attributes described in [The deployment descriptor](#). The deployment descriptor is stored in a special file that contains a serialized instance of a `javax.ejb.deployment.EntityDescriptor` object for an entity bean or a `javax.ejb.deployment.SessionDescriptor` object for a session bean.

To create a deployment descriptor, you can use either the **jetace** tool provided with WebSphere Application Server or the mechanism built into an integrated development environment (IDE) that supports enterprise bean development (for example, VisualAge for Java). You can also create a deployment descriptor programmatically, though this approach is not discussed in this documentation.

Creating an EJB JAR file

A JAR file for an enterprise bean, known as an EJB JAR file, must contain the following components:

- 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.
- The deployment descriptor file for the enterprise bean.
- The manifest file that describes the content of the EJB JAR file.

Manifest files are organized into sections that are separated by blank lines; each section corresponds to a file stored in the JAR file. The manifest file must be named `META-INF/MANIFEST.MF`. Each section contains one or more tag-value pairs with the syntax `tag:value`. The section corresponding to the deployment descriptor file for each enterprise bean in an EJB JAR file must contain the following headers:

```
Name: deploymentDescriptorFile
Enterprise-Bean: True
```

[Figure 35](#) shows the first two sections of the manifest file for the Account bean's EJB JAR file. Although not shown in the example, the manifest file contains a section for each of the class files--`Account.class`, `AccountBean.class`, `AccountKey.class`, and `AccountBeanFinderHelper.class`--and any other files in the EJB JAR file. However, only the section associated with the deployment descriptor file contains the enterprise bean-specific headers.

Figure 35. Code example: Fragment of the manifest file for the Account EJB JAR file

```
Manifest-Version: 1.0
Name: com/ibm/ejs/doc/account/Account.ser
Enterprise-Bean: true
Digest-Algorithms: SHA MD5
SHA-Digest: xhCUGthNU+Kds5X3xol5q7Mz9JI=
MD5-Digest: t40IcinyAjss0hrM1dpY0A==
Name: com/ibm/ejs/doc/account/AccountHome.class
Digest-Algorithms: SHA MD5
SHA-Digest: 02pfJv1buUu0FqpFwwERUstsNIg=
MD5-Digest: a8auOXob9ryPgbcnpFvzpQ==
...
```

To build an EJB JAR file, you can use either the **jetace** tool provided with WebSphere Application Server or the mechanism

built into an integrated development environment (IDE) that supports enterprise bean development (for example, VisualAge for Java). Both of these tools automatically can create the deployment descriptor, appropriately format a manifest file, and create an EJB JAR file to contain one or more enterprise beans.

Copyright [IBM Corporation 1999](#). All Rights Reserved

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 57](#) shows the import statements and class declaration for the example AccountBM enterprise bean.

Figure 57. 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 58. 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 connections in the EJB server \(CB\) environment](#) or [Managing database connections in the EJB server \(AE\) environment](#).

For more information on how to set an enterprise bean's environment variables, refer to [Setting environment variables for an enterprise 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.

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 20](#).

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 59](#) 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 59. 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.

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 60](#) 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 60](#). 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 60. Code example: The `ejbFindByPrimaryKey` method of the `AccountBMBean` class

```
public AccountBMKey ejbFindByPrimaryKey (AccountBMKey key) throws FinderException,
    RemoteException {
    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 61 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 61. Code example: The `ejbFindLargeAccounts` method of the `AccountBMBean` class

```

public Enumeration ejbFindLargeAccounts(float amount) throws RemoteException {
    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 62](#) 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 62. 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 23](#)).
- 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 or the `java.util.Enumeration` interface (when a finder method can return more than one EJB object).
- 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 62](#). 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 63](#) 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 63. 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 CMP, you must write a distinct primary key class. In an entity bean with BMP, 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 26](#), with the exception that the key class is named AccountBMKey.

Note: For the EJB server (AE) environment, the primary key class of an entity bean with BMP does not need to implement the hashCode and equals method. However, the primary key class can contain these methods if you require their functionality. 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 code required to manage database connections varies across the EJB server implementations:

- The EJB server (CB) uses JDBC 1.0 to manage database connections directly. For more information on the EJB server (CB), see [Managing connections in the EJB server \(CB\) environment](#).
- The EJB server (AE) 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 on the EJB server (AE), see [Managing database connections in the EJB server \(AE\) 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. If you are using the EJB server (CB), which does not contain a connection manager, this approach is probably the best one.
- 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. If you are using the EJB server (AE), which contains a connection manager, 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`. These methods must be coded differently based on which EJB server environment you use:

- The code required to make the `AccountBM` bean work with the connection manager in the EJB server (CB) is shown in [Managing connections in the EJB server \(CB\) environment](#).
- The code required to make the `AccountBM` bean work with the connection manager in the EJB server (AE) is shown in [Managing database connections in the EJB server \(AE\) environment](#).

The code required to manipulate data in a database is identical for both EJB server environments. For more information, see [Manipulating data in a database](#).

Managing connections in the EJB server (CB) environment

In the EJB server (CB) environment, the standard `java.sql.DriverManager` interface is used to load and register a database driver and to get and release connections to the database.

Loading and registering a data source

The example `AccountBM` bean uses an IBM DB2 relational database to store its persistent data. To interact with DB2, the example bean must load one of the available JDBC drivers. [Figure 64](#) shows the code required to load the driver class. The value of the `driverName` variable is obtained by the `getEnvProps` method, which accesses a corresponding environment variable in the deployed enterprise bean.

The `Class.forName` method loads and registers the driver class. The `AccountBM` bean loads the driver in its `setEntityContext` method, ensuring that every instance of the bean has immediate access to the driver after creating the bean instance and establishing the bean's context.

Note: In the EJB server (CB) environment, entity beans with BMP that use JDBC to access a database cannot participate in distributed transactions because the environment does not support XA-enabled JDBC. In addition, a BMP entity bean that uses JDBC to access a DB2 database must not be run in the same server process as a CMP entity bean that uses DB2 or in the same server process as an ordinary CB business object that uses DB2. Similarly, a BMP entity bean that uses JDBC to access an Oracle database must not be run in the same server process as a CMP entity bean (or other CB business object) that uses Oracle.

Figure 64. Code example: Loading and registering a JDBC driver in the `setEntityContext` method

```
public void setEntityContext(EntityContext ctx)
    throws RemoteException {
    entityContext = ctx;
    try {
        getEnvProps();
        // Load the applet driver for DB2
        Class.forName(driverName);
    } catch (Exception e) {
        ...
    }
}
```

Creating and closing a connection to a database

After loading and registering a database driver, the BMP entity bean must get a connection to the database. When it no longer needs that connection, the BMP entity bean must close the connection.

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.

The `makeConnection` method is invoked when a new database connection is required. It invokes the static method `java.sql.DriverManager.getConnection` and passes the DB2 URL value defined in the `jdbcUrl` variable (and described in [Defining instance variables](#)). The `getConnection` method is overloaded; the method shown here only uses the database URL, other versions require the URL and the database user ID or the URL, database user ID, and the user password.

Figure 65. Code example: The `checkConnection` and `makeConnection` methods of the `AccountBMBean` class

```
# import java.sql.*;
...
private void checkConnection() throws RemoteException {
    if (jdbcConn == null) {
        makeConnection();
    }
    return;
}
...
private void makeConnection() throws RemoteException {
    ...
    try {
        // Open database connection
        jdbcConn = DriverManager.getConnection(jdbcUrl);
    } catch (Exception e) { // Could not get database connection
        ...
    }
}
```

Entity beans with BMP must also drop database connections when a particular bean instance no longer requires it. The `AccountBMBean` class contains a `dropConnection` method to handle this task. To drop the database connection, the `dropConnection` method does the following:

1. Invokes the `commit` method on the connection object (`jdbcConn`), to drop any locks held on the database.
2. Invokes the `close` method on the connection object to close the connection.
3. Sets the connection object reference to null.

Figure 66. Code example: The `dropConnection` method of the `AccountBMBean` class

```
private void dropConnection() {
    try {
        // Close and delete jdbcConn
        jdbcConn.commit();
    } catch (Exception e) {
        // Could not commit transactions to database
        ...
    } finally {
        jdbcConn.close();
        jdbcConn = null;
    }
}
```

Managing database connections in the EJB server (AE) environment

In the EJB server (AE) 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.stdext.javax.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 (AE). The connection manager creates a pool of database connections, which are allocated and deallocated to individual entity beans as needed.

Note: The example code contained in this section cannot be found in the `AccountBMBean`, which manages database connections by using the `DriverManager` interface described in [Managing connections in the EJB server \(CB\) environment](#). This section shows the code that is required if the `AccountBM` bean were rewritten to use the `DataSource` interface.

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 67](#) 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 67. 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 RemoteException {
    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 68](#)).

Figure 68. Code example: The checkConnection and makeConnection methods of the AccountBMBean class (rewritten to use DataSource)

```
private void checkConnection() throws RemoteException {
    if (jdbcConn == null) {
        makeConnection();
    }
    return;
}
...
private void makeConnection() throws RemoteException {
    ...
    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 69](#)):

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 69. 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 70](#), 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 70. Code example: Constructing and executing an SQL update call in an `ejbCreate` method

```
private void ejbCreate(AccountBMKey key, int type, float initialBalance)
    throws CreateException, RemoteException {
    // 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. [Figure 71](#) provides an example of how the data in a `ResultSet` is manipulated in the `ejbLoad` method of the `AccountBMBean` class.

Figure 71. Code example: Manipulating a `ResultSet` object in the `ejbLoad` method

```
public void ejbLoad () throws RemoteException {
    // 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 TX_BEAN_MANAGED, you indicate to the container that the bean is an active participant in transactions.

Note: In the EJB server (AE) environment, the value TX_BEAN_MANAGED 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.)

Note: In the EJB server (CB) environment, a stateful session bean that implements the TX_BEAN_MANAGED attribute must begin and complete a transaction within the scope of a single method.

[Figure 72](#) shows the standard code required to obtain an object encapsulating the transaction context. There are three basic 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 72. 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 RemoteException {
        mySessionCtx = ctx;
    }
    ...
    public float doSomething(long arg1) throws FinderException, RemoteException {
        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.

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](#).

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](#).

Note: In the EJB server (CB) environment, an enterprise bean can also be accessed by a Java applet, an ActiveX client, a CORBA-based Java client, and to a limited degree, by a C++ CORBA client. The Travel example briefly described in [Information about the examples described in the documentation](#) illustrates some of these types of clients. [More information on EJB clients specific to the EJB server \(CB\)](#) provides additional information about EJB clients that use ActiveX and CORBA-based Java and C++.

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 \(AE\) environment](#) or

[Setting the CLASSPATH environment variable in the EJB server \(CB\) environment.](#) You can install needed files on your client machine by doing a WebSphere Application Server installation on the machine, selecting 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 1.2.x install.

[Figure 36](#) 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 36. 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:///`. In the EJB server (AE), 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.

- In the EJB server (AE) environment, this property must be set to `com.ibm.ejs.ns.jndi.CNInitialContextFactory`.
- In the EJB server (CB) environment, this property must be set to `com.ibm.ejb.cb.runtime.CBCtxFactory`. When using this context factory, the `javax.naming.Context.list` and `javax.naming.Context.listBindings` methods can return no more than 1000 elements in the `javax.naming.NamingEnumeration` object.

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 37](#) 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 37. 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 38](#). 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:

- When performing a JNDI lookup on an enterprise bean deployed in an EJB server (AE), only the JNDI name specified in the deployment descriptor is used.
- When performing a JNDI lookup on an enterprise bean deployed in an EJB server (CB), the JNDI home name passed to the lookup method is the JNDI name specified in the enterprise bean's deployment descriptor with a CB-specific prefix attached. The content of this prefix depends on where in the Component Broker namespace the system administrator bound the EJB home (by using the **ejbbind** tool).

If the system administrator binds the EJB home in the host name tree of a specific bootstrap host, then the JNDI name prefix will be `host/resources/factories/EJBHomes`. If the system administrator binds the EJB home in a workgroup name tree, then the JNDI name prefix will be `workgroup/resources/factories/EJBHomes`, and the EJB client must belong to the same preferred workgroup. If the system administrator binds the EJB home in the cell name tree, then the JNDI name prefix is `cell/resources/factories/EJBHomes`.

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 38. 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(
            (org.omg.CORBA.Object) homeObject, TransferHome.class);
    } catch (Exception e) { // Error getting the home interface
        ...
    }
    ...
    // Create a new Transfer object to return
    ...
    return transfer;
}

```

Migration considerations for creating an EJB home object

If you are migrating existing applications from WebSphere Application Server 2.x to 3.x, you will need to make a change in how home objects are narrowed. In the 2.x environment, each class has its own Helper whose name follows the ClassNameHelper format. In the 3.x environment, however, the PortableRemoteObject class must be used for narrowing all classes, instead of a specific Helper for each class.

For example, a 2.x version of the example TransferApplication would contain lines similar to those in [Figure 39](#), where the narrowing is done using the TransferHomeHelper class that is specific for TransferHome.

Figure 39. Code example: home object narrowing in WebSphere Application Server 2.x

```

. . .
java.lang.Object homeObject = ivjInitContext.lookup(
    bundle.getString("transferName"));
transferHome = TransferHomeHelper.narrow(
    (org.omg.CORBA.Object)homeObject);
. . .

```

In order to migrate to a 3.x version of the example TransferApplication, change the second line to use the narrow method of the PortableRemoteObject class, supplying as parameters the object to be narrowed and its class. This is shown in the code sample in [Figure 40](#). (The complete example code for creating an EJB home object is discussed in [Creating EJB home object](#).)

Figure 40. Code example: home object narrowing in WebSphere Application Server 3.x

```

. . .
java.lang.Object homeObject = ivjInitContext.lookup(
    bundle.getString("transferName"));
transferHome = (TransferHome)javax.rmi.PortableRemoteObject.narrow(
    (org.omg.CORBA.Object)homeObject, TransferHome.class);
. . .

```

Creating an EJB object

After the EJB home object is created, it is used to create the EJB object. [Figure 41](#) 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 41. 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 42](#) 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 42. 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();
            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 session EJB object, the EJB client must remove that object. Removing unneeded session EJB objects can prevent memory leaks in the EJB server. 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 43](#) 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 43. 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 TX_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 44](#) 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 `jta/usertransaction`.

Figure 44. Code example: managing transactions in an EJB client


```
...
javax.transaction.*;
...
// Use JNDI to locate the UserTransaction object
Context initialContext = new InitialContext();
UserTransaction tranContext = (
    UserTransaction)initialContext.lookup("jta/usertransaction");
// Set the transaction timeout to 30 seconds
tranContext.setTimeout(30);
...
// Begin a transaction
tranContext.begin();
// Perform transaction work invoking methods on enterprise bean references
...
// Call for the transaction to commit
tranContext.commit();
```

More information on EJB clients specific to the EJB server (CB)

When developing EJB clients for the EJB server (CB) environment, you can develop the following types of clients:

- Microsoft ActiveX clients. For some general information, see [EJB clients that use ActiveX](#).
- CORBA-based Java and C++ clients. For some general information, see [C++ and Java EJB clients that use a CORBA interface](#).
- Clients using the Component Broker Session Service. For some general information, see [Clients using the Component Broker Session Service](#).

For more information on developing these types of clients, see the IBM Redbook entitled *IBM Component Broker Connector Overview*, form number SG24-2022-02.

EJB clients that use ActiveX

If you write your EJB client as a component that adheres to the JavaBeans(TM) Specification, you can use the JavaBeans bridge to run the EJB client as an ActiveX control. An EJB client of this type must provide a no-argument constructor, it must implement the `java.io.Serializable` interface, and it must have a `readObject` and a `writeObject` method, if applicable.

If your EJB client is also an applet, you must not perform your JNDI initialization as part of object construction. Rather, perform JNDI initialization in the applet's start method. The JavaBeans bridge must create an instance of your EJB client so that it can introspect it and make the necessary stubs to create the ActiveX proxy for it. You must delay the JNDI connections until the user can specify the necessary properties by way of the ActiveX property sheet.

C++ and Java EJB clients that use a CORBA interface

Typically, Java EJB clients are written to use the enterprise bean's RMI interface. However, you can also write a Java-based EJB client that uses the enterprise bean's CORBA interface. To generate Java/CORBA bindings for an enterprise bean deployed in a Component Broker server, use the `-idl` option of the Java `rmic` command to generate an interface definition language (IDL) file from the enterprise bean's remote and home interfaces.

Then, use the IDL file as input to Component Broker's IDL compiler (`com.ibm.idltoJava.Compile`). You must not generate Java/CORBA bindings from the IDL file generated by the `cbejb` deployment tool, because this IDL file incorporates special interface-name and method-signature mangling that allows the IDL file to be used to generate C++ bindings for enterprise beans without requiring C++ implementations of the Java `Serializable` types used by the bean.

You can develop EJB clients that use C++ CORBA; however, these clients are restricted to invoking methods that do *not* use parameters that are arrays or that are of the `java.io.Serializable` type or the `java.lang.String` type. This restriction effectively prohibits C++ EJB clients from accessing entity beans directly because primary key classes must be serializable. The `String` and `array` types in the remote or home interface are mapped to IDL value types to allow null values to be passed between a Java EJB client and an enterprise bean. CORBA C++ EJB clients cannot invoke the `javax.ejb.EJBHome.remove` and

javax.ejb.EJBObject.getHandle methods because these methods contain Serializable parameters. EJB clients of this type cannot be built with Microsoft Visual C++.

To generate the CORBA C++ bindings for an enterprise bean, run the Component Broker **idlc** tool on the IDL file generated from the enterprise bean by the **cbejb** tool. Do not generate CORBA C++ bindings by using the IDL file generated by the Java **rmic** command, because this IDL file contains nested modules that can be re-opened, and these are not supported by the Component Broker C++ bindings due to the lack of namespace support in the C++ compiler.

Clients using the Component Broker Session Service

In addition to the Transaction Service, Component Broker also provides a Session Service for the Procedural Application Adaptor (PAA) that enables the use of backend systems such as CICS and IMS. Since the JTA does not have a Session Service, it is not possible to use JNDI to look up a handle to the service in an EJB client. In this case, the EJB client must act as an ordinary CB Java client.

The normal lookup procedure for a CB Java client is to use the CORBA `resolve_initial_references` method. In this case, the CORBA object to look up is named `SessionCurrent`.

Before you can call the `resolve_initial_references` method, the ORB needs to be properly initialized for the CB runtime environment. The initialization method depends on whether or not you are using VisualAge for Java access beans in the CB environment. If you are using access beans, then the ORB must be manually initialized. ORB initialization in access beans is done in a "lazy" fashion. That is, initialization is not done until the first remote method is invoked. However, because a session must be started before that method is called, the ORB initialization must be done manually. The example code in [Figure 45](#) shows this initialization.

Figure 45. Code example: initializing the ORB (if using access beans)

```
String[] CBargs = null;
CBargs = new String[6];
CBargs[0] = "-ORBBootstrapHost";
// substitute your bootstrap host name
CBargs[1] = "cbs3.rchland.ibm.com";
CBargs[2] = "-ORBBootstrapPort";
CBargs[3] = "900";
CBargs[4] = "-ORBClass";
CBargs[5] = "com.ibm.CORBA.iiop.ORB";
com.ibm.CBCUtil.CBSeriesGlobal.Initialize(CBargs);
```

If you are not using access beans, initialization code is not necessary. The ORB is properly initialized during the creation of the `InitialContext` object with the appropriate properties. For example, your client code should already contain lines similar to those in [Figure 46](#). This code is used to find the service, look up the home object, narrow the home object, and create the proxy object (tasks automatically done if an access bean is being used).

Figure 46. Code example: creating the InitialContext object (if not using access beans)

```
Properties properties = new Properties();
properties.put(javax.naming.Context.PROVIDER_URL, "iiop:///");
// CB Factory Name
properties.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
"com.ibm.ejb.cb.runtime.CBCtxFactory");
Context ctx = new InitialContext(properties);
```

After the ORB is initialized (either automatically or manually), you must use CB-specific APIs for creating and using the `sessionCurrent` object. You must include code similar to the example code in [Figure 47](#).

Figure 47. Code example: creating and using the sessionCurrent object

```
org.omg.CORBA.Object orbCurrent = null;
com.ibm.ISessions.Current sessionCurrent = null;
...
orbCurrent =
com.ibm.CBCUtil.CBSeriesGlobal.orb().resolve_initial_references("ISessions::Current");
sessionCurrent = com.ibm.ISessions.CurrentHelper.narrow(orbCurrent);
sessionCurrent.beginSession("myApp");

...

// commit
sessionCurrent.endSession(com.ibm.ISessions.EndMode.EndModeCheckPoint, true);
```

For more information on using the `resolve_initial_references` method, see the [Component Broker Programming Guide](#).

Copyright [IBM Corporation 1999](#). All Rights Reserved

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* TX_BEAN_MANAGED and set the transaction isolation level attribute. To enable bean-managed transactions, you must set the transaction attribute to TX_BEAN_MANAGED 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, only 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 with either the EJB server (AE) or the EJB server (CB). For more information, see [Tools for developing and deploying enterprise beans in the EJB server \(AE\) environment](#) or [Tools for developing and deploying enterprise beans in the EJB server \(CB\) environment](#).

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 each enterprise bean, 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](#).

The EJB specification defines two attributes in a standard deployment descriptor that 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. [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. [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 can be set for the bean as a whole and for individual methods in a bean.

Note: The EJB server (CB) does not support the setting of the transaction attribute for individual enterprise bean methods; the transaction attribute can be set only for the entire bean.

The following are valid values for this attribute in decreasing order of transactional strictness:

TX_BEAN_MANAGED

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](#).

In the EJB server (CB) environment, if a stateful session bean has this attribute value, a method that begins a transaction must also complete that transaction (commit or roll back the transaction). In other words, a transaction cannot span multiple methods in a stateful session bean when used in the EJB server (CB) environment.

TX_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 `TX_BEAN_MANAGED` value or use the `TX_REQUIRED` or `TX_REQUIRES_NEW` 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](#).

In the EJB server (CB) environment, this attribute value must be used in entity beans with container-managed persistence (CMP) that use Host On-Demand (HOD) or the External Call Interface (ECI) to access CICS or IMS applications.

TX_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.

TX_REQUIRES_NEW

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.

The EJB server (CB) does *not* support this attribute value.

TX_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.

In the EJB server (CB) environment, entity beans with CMP must be accessed within a transaction. If an entity bean with CMP uses this transaction attribute, the EJB client must initiate a transaction before invoking a method on the entity bean.

TX_NOT_SUPPORTED

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.

In the EJB server (CB) environment, entity beans with CMP must be accessed within a transaction. Therefore, this attribute value is not supported in entity beans with CMP in the EJB server (CB) environment.

Table 1. Effect of the enterprise bean's transaction attribute on the transaction context

Transaction attribute	Client transaction context	Bean transaction context
TX_MANDATORY	No transaction	Not allowed
	Client transaction	Client transaction
TX_NOT_SUPPORTED	No transaction	No transaction
	Client transaction	No transaction
TX_REQUIRES_NEW	No transaction	New transaction
	Client transaction	New transaction
TX_REQUIRED	No transaction	New transaction
	Client transaction	Client transaction
TX_SUPPORTS	No transaction	No transaction
	Client transaction	Client transaction

Setting the transaction isolation level attribute

Note: The EJB server (CB) does not support the transaction isolation level attribute.

The transaction isolation level determines how strongly one transaction is isolated from another. This attribute can be set for the enterprise bean as a whole and 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:

TRANSACTION_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.

TRANSACTION_REPEATABLE_READ

This level prohibits dirty reads and nonrepeatable reads, but it allows phantom reads.

TRANSACTION_READ_COMMITTED

This level prohibits dirty reads, but allows nonrepeatable reads and phantom reads.

TRANSACTION_READ_UNCOMMITTED

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 `TRANSACTION_SERIALIZABLE` value protects data by eliminating concurrent access to it. Conversely, the `TRANSACTION_READ_UNCOMMITTED` 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.

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:

CLIENT_IDENTITY

The security service makes no changes to the principal's credential properties.

SYSTEM_IDENTITY

The security service alters the principal's credential properties to match the credential properties associated with the EJB server.

SPECIFIED_IDENTITY

The security service attempts to match the principal's credential properties with the identity of any application with which the enterprise bean is associated. If successful, the security service alters the principal's credential properties to match the credential properties of the application.

The *run-as identity* and *access control* attributes are not used in the EJB server environments.

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. In addition, the IBM HTTP Web server is available with both the Advanced Application Server and the Enterprise Application Server. For more information, consult the Getting Started with Advanced Edition document.

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 48](#) 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.

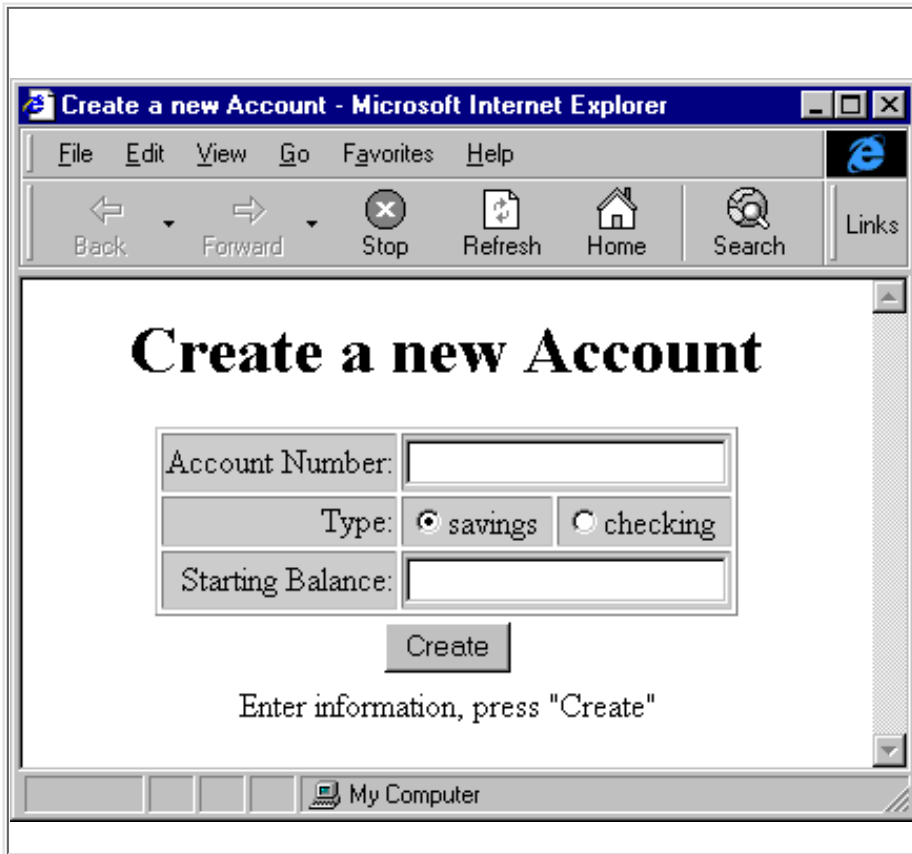
Figure 48. Code example: Content of the create.html file used to access the CreateAccount servlet

```
<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 49](#) shows what create.html looks like on a browser.

Figure 49. 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 50](#) 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 50. 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 51](#) 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 51](#).

Figure 51. 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 52](#). 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 class are valid for storing these properties.

Figure 52. 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(
            (org.omg.CORBA.Object)homeObject, AccountHome.class);
    }
    // Determine cause of failure.
    catch (NamingException e) {
        ...
    }
    catch (Exception e) {
        ...
    }
}

```

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 53](#) 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 53. 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 54](#). 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 54. 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 55](#) 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 55. 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 56](#).

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 56. 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 in the EJB server (CB) environment

The following are the basic approaches to developing and deploying enterprise beans in the EJB server (CB) environment:

- You can use the tools available in the Java Software Development Kit (SDK) and WebSphere Application Server, Enterprise Edition. For more information, see [Developing and deploying enterprise beans with EJB server \(CB\) tools](#).
- You can use one of the available integrated development environments (IDEs) such as IBM VisualAge for Java. IDE tools automatically generate significant parts of the enterprise bean code and contain integrated tools for packaging and testing enterprise beans. For more information, see [Using VisualAge for Java](#).
- You can create an enterprise bean from an existing CICS or Information Management System (IMS) application by using the **PAOToEJB** tool. The application must be mapped into a procedural adapter object (PAO) before this tool is used. For more information, see [Creating an enterprise bean from an existing CICS or IMS application](#).
- You can create an enterprise bean that communicates with IBM MQSeries by using the **mqaajb** tool. For more information, see [Creating an enterprise bean that communicates with MQSeries](#).

Before beginning development of enterprise beans in the EJB server (CB) environment, review the list of development restrictions contained in [Restrictions in the EJB server \(CB\) environment](#).

Note: Deployment and use of enterprise beans for the EJB server (CB) environment must take place on the Microsoft Windows NT operating system, the IBM AIX operating systems, or the Sun Solaris operating system.

For information on developing and deploying enterprise beans in the EJB server (AE) environment, see [Tools for developing and deploying enterprise beans in the EJB server \(AE\) environment](#).

Developing and deploying enterprise beans with EJB server (CB) tools

You need the following tools to develop and deploy enterprise beans for the EJB server (CB):

- 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 following tools available in the WebSphere Application Server, Enterprise Edition:
 - **jetace**, which enables you to create or update an EJB JAR file for one or more enterprise beans; this includes the creation of the enterprise bean's deployment descriptor, which instructs the EJB server on how to properly manage the enterprise bean.
 - Object Builder, which is the recommended tool for deploying enterprise beans. Use of this tool is not documented in this book. For more information on using Object Builder to deploy enterprise beans, see the Component Broker Application Development Tools Guide.
 - **cbejb**, which works with Object Builder to create and compile the necessary files needed by the EJB server (CB) to manage an enterprise bean. The **cbejb** tool looks inside the EJB JAR file to examine the EJB home and EJB object classes and the deployment descriptors. The **cbejb** tool generates a model

that Object Builder uses to create the necessary deployment library files. The output of this process is a set of server-side and client-side JAR and library files.

- **ejbbind**, which binds an enterprise bean's Java Naming and Directory Interface (JNDI) home name (found in its deployment descriptor) to a factory in an EJB server (CB). This tool is deprecated for servers running on the AIX, Windows NT, and Solaris platforms.
- **appbind**, which allows enterprise bean deployers to create an application-specific naming context and associate it with a selected factory finder so that the EJB home lookup operations are resolved with that factory finder. This tool is available only on the AIX, Windows NT, and Solaris platforms and only be applied to servers installed on any of those platforms.

This section describes the steps you must follow to manually develop and deploy enterprise beans by using the EJB server (CB) tools. The following tasks are involved:

1. Ensure that you have the prerequisite software to develop and deploy enterprise beans in the EJB server (CB). For more information, see [Prerequisite software for the EJB server \(CB\)](#).
2. Set the CLASSPATH environment variable required by different components of the EJB server (CB) environment. For more information, see [Setting the CLASSPATH environment variable in the EJB server \(CB\) environment](#).
3. Write and compile the components of the enterprise bean. For more information, see [Creating the components of an enterprise bean](#).
4. Create a finder helper class 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 \(CB\)](#).
5. Use the **jetace** tool to create an EJB JAR file to contain the enterprise bean. For more information, see [Creating a deployment descriptor and an EJB JAR file](#).
6. Deploy the enterprise bean by using the **cbejb** command. For more information, see [Deploying an enterprise bean](#).
7. Build a data object (DO) implementation for use by the enterprise bean by using Object Builder (This step is part of the deployment process). For more information, see [Building a data object during CMP entity bean deployment](#).
8. Install the deployed enterprise bean and configure its EJB server (CB). For more information, see [Installing an enterprise bean and configuring its EJB server \(CB\)](#).
9. Bind the JNDI name of the enterprise bean into the JNDI namespace by using the **ejbbind** tool. (This step is not necessary on the AIX, Windows NT, or Solaris platforms.) For more information, see [Binding the JNDI name of an enterprise bean into the JNDI namespace](#).
10. Start the EJB server (CB). For more information see the Component Broker System Administration Guide.

Prerequisite software for the EJB server (CB)

Note: Any items marked *PAO only* are needed only if you intend to use the **PAOToEJB** tool and need the CICS- or IMS-related support.

You must configure the tools provided with the EJB server (CB) environment; however, before you can configure the tools, you must ensure that you have installed and configured the following prerequisite software products contained in the Enterprise Application Server:

- CB Server
- CB Tools (including the Object Builder, VisualAge Component Development toolkit, samples, the Server SDK, and (*PAO only*) CICS and IMS Application Adapter SDK
- (*PAO only*) CICS/IMS Application run time
- (*PAO only*) CICS/IMS Application client

Setting the CLASSPATH environment variable in the EJB server (CB) environment

To do any of the tasks listed below, make sure that the classes.zip file contained in the Java Development Kit is included in the CLASSPATH environment variable. In addition, make sure that the following files are identified by the CLASSPATH environment variable to perform the associated task:

- Developing an enterprise bean or an EJB client: no additional files.
- Deploying an EJB JAR file:
 - somojor.zip
 - The EJB JAR file being deployed and any JAR or ZIP files on which it depends
- Running an EJB server (CB) managing an enterprise bean named *beanName*. These JAR files are automatically added to the CLASSPATH environment variable.
 - *beanNameS.jar*
 - The EJB JAR file used to create *beanNameS.jar* and any JAR or ZIP files on which it depends
- Running a pure Java EJB client using an enterprise bean named *beanName*:
 - *beanNameC.jar*
 - somojor.zip
- Running an EJB server (CB) that contains an enterprise bean named *clientBeanName* that accesses another enterprise bean named *beanName* as a client. These JAR files are automatically added to the CLASSPATH environment variable.
 - *clientBeanNameS.jar*
 - The EJB JAR file used to create *clientBeanNameS.jar* and any JAR or ZIP files on which it depends
 - *beanNameC.jar*

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 of the EJB specification. These components are 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, and the bean's remote interface.

After you have properly coded these components, use the Java compiler to create the corresponding Java class files. For example, since the components of the example Account bean are stored in a specific directory, you can compile the bean components 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 (CB)

In the EJB server (CB), finder logic is contained in a finder helper class. The enterprise bean deployer must implement the finder helper class before deploying the enterprise bean and then specify the name of the class with the `-finderhelper` option of the `cbejb` tool.

For each specialized finder method in the home interface (other than the `findByPrimaryKey` method), the finder helper class must have a corresponding method with the same name and parameter types. When an EJB client

invokes a specialized finder method, the generated CB home that implements the enterprise bean's home interface invokes the corresponding finder helper method to determine what to return to the EJB client.

The finder helper class must also have a constructor that takes a single argument of type `com.ibm.IManagedClient.IHome`. When the CB home instantiates the finder helper class, the CB home passes a reference to itself to the finder helper constructor. This allows the finder helper to invoke methods on the CB home within the implementation of the finder helper methods, which is particularly useful when the CB home is an `IQueryableIterableHome` because the finder helper can narrow the `IHome` object passed to the constructor and invoke query service methods on the CB home.

The names of the entity bean's container-managed fields are mapped to interface definition language (IDL) attributes of the same name, except that an underscore (`_`) is appended, in the business object (BO) interface, the CB key class, and the CB copy helper class. These names are mapped exactly to IDL attributes in the DO interface. For example, in the `AccountBean` class, the `accountId` variable is mapped to `accountId_` in the BO interface, the CB key class, and the CB copy helper class, but is mapped to `accountId` in the DO interface.

This renaming is necessary, and relevant to finder helper classes implemented by using the Component Broker Query Service, because the entity bean's remote interface can also have a property named `accountId` (of potentially a different type) that must also be exposed through the BO interface. If that is the case, then a query over the BO attribute `accountId` is done in object space, whereas a query over the BO attribute `accountId_` is done directly against the underlying data source, which is typically more efficient.

If a home interface's specialized finder method returns a single entity bean, then the corresponding method in the finder helper class must return the `java.lang.Object` type. When invoked, the finder helper method can return the EJB object, the CB key object, or the entity bean's primary key object. If the finder helper method returns a CB object or a primary key object, the CB home determines the corresponding EJB object to return to the EJB client.

If a home interface's specialized finder method returns a `java.util Enumeration` type, the corresponding finder helper method must also return `java.util Enumeration`. When invoked, the finder helper method can return an `Enumeration` of EJB objects, CB key objects, enterprise bean primary key objects, or a heterogeneous mix of one or more of the three. The CB home then constructs a serializable `Enumeration` object containing the corresponding EJB objects, which is returned to the EJB client.

An optional base class, named `com.ibm.ejb.cb.runtime.FinderHelperBase`, is provided with the EJB server (CB) environment to assist in the development of a finder helper class. This class encapsulates the Component Broker Query Service, so that the deployer does not need to write any CB-specific code. The `FinderHelperBase` base class contains an `evaluate` method, which takes an Object-Oriented Structured Query Language (OOSQL) predicate as a parameter and returns an `Enumeration` of objects that meet the conditions of the query. (The `evaluate` method calls the `IQueryableIterableHome.evaluate` method, then iterates through the returned `Iterator` object to construct an `Enumeration` object containing the search results. This method throws a `javax.ejb.FinderException` if any errors occur; the finder helper class does not need to catch this exception; instead, the class can pass it on to the EJB client.)

A utility class, named `com.ibm.ejb.cb.emit.cb.FinderHelperGenerator` (contained in the `developEJB.jar` file), is also provided to further assist the deployer in the development of a finder helper class. This utility takes the name of an entity bean's home interface and generates a Java source file containing a class that extends `com.ibm.ejb.cb.runtime.FinderHelperBase` and that contains skeleton methods for each specialized finder method in the home interface. In addition, each finder helper method (with a corresponding finder method that returns an `Enumeration` object) contains code to invoke the `evaluate` method.

By using the `FinderHelperGenerator` utility, the deployer can easily implement the finder helper class. You can use a batch file to run the utility. For example, to generate a finder helper class for the example `AccountHome` interface, enter the following command:

```
# ejbfhgen com.ibm.ejs.doc.account.AccountHome
```

This command generates the finder helper class shown in [Figure 16](#).

Figure 16. Code example: Generated AccountFinderHelper class for the EJB server (CB)

```
...
public class AccountFinderHelper extends FinderHelperBase {
    ...
    AccountFinderHelper(IManagedClient.IHome iHome) {
        ...
    }
    public Enumeration findLargeAccounts(float amount) {
        return evaluate("Place SQL String here");
    }
}
```

To enable the helper class for use in a deployed enterprise bean, the deployer makes a few simple edits to the parameters of the evaluate invocations. For example, for the AccountFinderHelper class, the "Place SQL String here" String is replaced with "balance_>" + amount as shown in [Figure 17](#). The generated finder helper class can be used only with an enterprise bean that is deployed to have a queryable home by using the -queryable option of the **cbejb** tool.

Figure 17. Code example: Completed AccountFinderHelper class for the EJB server (CB)

```
...
public class AccountFinderHelper extends FinderHelperBase {
    ...
    AccountFinderHelper(IManagedClient.IHome iHome) {
        ...
    }
    public Enumeration findLargeAccounts(float amount) {
        return evaluate("balance_>" + amount);
    }
}
```

Using lazy enumeration

The Enumeration returned by the evaluate method is called *eager*, because all the enterprise bean references that match the query are brought into memory and stored in the enumeration before being passed from the server to the client. If the number of references returned by the query is large, the deployer can use *lazy* enumeration; that is, it incrementally fetches more enterprise bean references only when the client calls the nextElement method on the Enumeration.

To use lazy enumeration, change the call to the evaluate method in the FinderHelper to a call to the lazyEvaluate method. A transaction must already be started before the home's finder method is called. The caller must not call the nextElement method on the Enumeration after the completion of the transaction.

At configuration time, the System Management End User Interface must be used to enable the settings for lazy Enumerations. Refer to [Configuring systems management to enable lazy enumeration](#)

Creating an EJB JAR file for an enterprise bean

Once the bean components are built, the next step is package them into an EJB JAR file. The **jetace** tool is used to create an EJB JAR file for one or more enterprise beans. For more information on creating an EJB JAR file, see [Creating a deployment descriptor and an EJB JAR file](#).

Deploying an enterprise bean

During deployment, a deployed JAR file is generated from an EJB JAR file. Use the **cbejb** tool to deploy enterprise beans in the EJB server (CB) environment. The deployed JAR file contains classes required by the EJB server. The **cbejb** tool also generates the data definition language (DDL) file used during installation of the enterprise bean into the EJB server (CB).

If you want to use an enterprise bean on a different machine from the one on which it was developed (and on which you ran **cbejb**), follow the guidelines for installing applications in the Component Broker document entitled System Administration Guide. If an enterprise bean uses additional files (such as other JAR files) that need to be copied with the enterprise bean, specify these files in the properties notebook of the application (*not* the family).

The **cbejb** tool has the following syntax:

```
cbejb.ejb-jarFile [-rsp responseFile][-ob projDir] [-nm] [-ng] [-nc] [-cc]
[-bean beanNames] [-platform [NT | AIX | OS390 | Solaris | HP]]
[-guisg] [-usecurdopo] [-nouseraction] [-dllname DLLName beanName]
[-polymorphichome [beanNames]] [-queryable [beanNames]]
[-dbname DBName [beanName]]
[-cacheddb2v52 | -cacheddb2v61 | -db2v61 | -oracle | -informix |
-jdbcaa [beanNames]]
[-hod | -eci | -appc | -exci | -otma | -ccf [beanNames]]
[-family familyName [beanNames]]
[-finderhelper finderHelperClassName [beanNames]]
[-usewstringindo [beanNames]] [-workloadmanaged [beanNames]]
[-clientdep deployed-jarFile [beanNames]]
[-serverdep deployed-jarFile [beanNames]]
[-sentinel [JavaPrimitiveObjectType=]sentinelValue [beanNames[+CMFieldNames]]]
[-strbehavior [strip | corba] [beanNames[+CMFieldNames]]]
```

The *ejb-jarFile* parameter is required; it must be the first argument and it must specify a valid EJB JAR file as described in [Creating an EJB JAR file](#). If the `-ob` option is used, it must come second on the command line. The other options can be specified in any order. The *beanNames* argument is a list of one or more fully qualified enterprise bean names delimited by colons (:) (for example, `com.ibm.ejs.doc.transfer.Transfer:com.ibm.ejs.doc.account.Account`). For the enterprise bean name, specify either the bean's remote interface name or the name of its deployment descriptor. If the *beanNames* argument is not specified for a particular option, then the effect of that option is applied to all enterprise beans in the EJB JAR file for which the option is valid.

Note: The relative file name of the JAR files specified by the *ejb-jarFile* variable and by the two *deployed-jarFile* variables must be different from each other. JAR file names that have the same relative file names but different paths are not valid.

The rest of the command parameters are optional and can be specified in any order. For explanation purposes, the options can be grouped by function into three general categories:

- Deployment options, which govern the generation and compilation of code.
- Storage options, which govern persistent storage.
- Execution options, which govern the run time environment.

The `-rsp` option does not fit into these categories. This option allows you to create a file containing some or all of the other options and their values (except the *ejb-jarFile* parameter). You can then submit the file to the **cbejb** command. This allows the common setting to be saved and makes commands easier to issue.

- Deployment options
 - `-ob projDir` -- Specifies the relative or full path of the project directory in which the generated files are stored. If this option is not specified, the current working directory is used as the project directory.

- Compilation modifiers -- By default, the **cbejb** tool does the following for each enterprise bean contained in the EJB JAR file:
 1. Generate and import XML.
 2. Generate code--Creates a DDL file, makefile, and other source files for each enterprise bean contained in the EJB JAR file. These files are placed in the specified project directory.
 3. Compile and link--Invokes the generated makefile to compile an application. Each application file is placed in the specified project directory. While the Dynamic Link Libraries (DLLs) are being linked, numerous duplicate symbol warnings appear; these warnings are harmless and can be ignored.

The following command options modify the default compilation behavior:

- -nm -- Suppresses the XML-processing step.
- -ng -- Suppresses the code-generation step.
- -nc -- Suppresses the compilation-and-linking step.
- -cc -- Removes previously compiled and linked code by invoking the generated makefile to remove non-source files. This option must be used if you specify either of these combinations:
 - -ng -nc
 - -nm -ng -nc
- -bean *beanNames* -- Identifies the enterprise beans in the EJB JAR file to be deployed. By default, all enterprise beans in the EJB JAR file are deployed. To deploy multiple enterprise beans, delimit the bean names with a : (colon). For example, `Account:Transfer`.
- -platform -- Specifies the platform for which to generate code. This also sets the deployment platform in the Object Builder tool, but it does not set the platform for viewing, generating, or applying development constraints. You must set these manually by using the choices on the **Platform** menu.
- -guisg -- Directs the tool to present the Object Builder graphical user interface (GUI), which enables the tool to collect options from the user rather than from the command line.
- -usecurdopo -- Directs the tool to use the current mapping between the data object and the persistent object in the existing model rather than bringing up the Object Builder interface to build a mapping. Use this option when redeploying beans for which a satisfactory mapping already exists. The deployment will proceed automatically.

When you first deploy CMP entity beans, you must *not* use this option. The tool will then build the default mapping between the data and persistent objects and, if you specify the -guisg option, launch the Object Builder interface.

- -nouseraction -- Directs the tool to use only the information on the command line after building the mapping between data objects and persistent objects. Otherwise, if you have also specified the -guisg option, the tool prompts you for the next action.
- -polymorphichome -- Specifies the beans that use polymorphic home interfaces.
- -queryable -- Directs the tool to generate a queryable CB home object. This option can be used only for entity beans with CMP that store their persistent data in a relational database. This option must be used if the finder helper class, which is used to implement the finder methods in a CMP entity bean, uses the CB query service. This option must *not* be used if an entity bean uses CICS or IMS to store its persistent data.

By default, the interface definition language (IDL) interface of an enterprise bean's CB home extends the `IManagedClient::IHome` class, and the home implementation extends the `IManagedAdvancedServer::ISpecializedHome` class. An IDL interface of a queryable home extends the `IManagedAdvancedClient::IQueryableIterableHome` class, and the home implementation extends the `IManagedAdvancedServer::ISpecializedQueryableIterableHome` class.

In addition, the generated BO interface is marked as queryable. For queryable homes, the EJB client programming model remains unchanged; however, a Common Object Request Broker Architecture

(CORBA) EJB client can treat the EJB home as an `IManagedAdvancedClient::IQueryableIterableHome` object.

For more information on queryable homes, see the Advanced Programming Guide.

- Storage options

- `-dbname DBName` -- Specifies the name of the database for beans with CMP.
- Database choices--The default database for persistent storage of container-managed beans is DB2 version 5.2 with embedded SQL. You can override this default by using:
 - `-cacheddb2v52` -- Identifies entity beans with CMP that require DB2 version 5.2 used with the Cache Service to store persistent data.
 - `-cacheddb2v61` -- Identifies entity beans with CMP that require DB2 version 6.1 used with the Cache Service to store persistent data.
 - `-db2v61` -- Identifies entity beans with CMP that require DB2 version 6.1 used with embedded SQL to store persistent data.
 - `-oracle` -- Identifies entity beans with CMP that require Oracle to store persistent data. If you specify this option, you must also use the `-queryable` option.
 - `-informix` -- Identifies entity beans with CMP that require Informix to store persistent data. A given transaction cannot access more than one Informix database from a CB server. To access two Informix databases in one transaction, you must access each from a different CB server. If you specify this option, you must also use the `-queryable` option.
 - `-jdbcaa` -- Identifies entity beans with BMP that require JDBC to store persistent data. This option enables the beans to join distributed transactions by allowing the bean implementation to connect to the Transaction Service. Beans with BMP that do not use this option will handle transactions in an implementation-dependent manner.
- `-hod` -- Identifies entity beans with CMP that use Host-on Demand (HOD) to store persistent data. These beans will use the Session Service. This option must *not* be used for enterprise beans generated from the **PAOToEJB** tool.
- `-eci` -- Identifies entity beans with CMP that use the external call interface (ECI) to store persistent data. These beans will use the Session Service. This option must *not* be used for enterprise beans generated from the **PAOToEJB** tool.
- `-appc` -- Identifies entity beans with CMP that use advanced program-to-program communications (APPC) to store persistent data. These beans will use the Transaction Service. This option must *not* be used for enterprise beans generated from the **PAOToEJB** tool.
- `-exci` -- Identifies entity beans with CMP that use the EXCI to store persistent data. These beans will use the Transaction Service. This option must *not* be used for enterprise beans generated from the **PAOToEJB** tool.
- `-otma` -- Identifies entity beans with CMP that use the OTMA to store persistent data. These beans will use the Transaction Service. This option must *not* be used for enterprise beans generated from the **PAOToEJB** tool.
- `-ccf` -- Identifies entity beans with CMP that use the SAP interface, which is a common connector framework (CCF) back end. These beans will use the Transaction Service.

- Execution options

- `-family familyName` -- Specifies the application family name to be generated. By default, this name is set to the name of the EJB JAR file with the word Family appended. This option can be specified more than once, as long as the values are unique.
- `-finderHelper finderHelperClassName remoteInterface`-- Specifies the finder helper class name (*finderHelperClassName*) and remote interface name (*remoteInterface*) for entity beans with CMP. If unspecified, it is assumed that no finder helper class is provided by the deployer. This option can be specified more than once, as long as the values are unique. For more information on finder helper classes, see [Defining finder methods](#).

- `-usewstringindo` -- Directs the tool to map the container-managed fields of an entity bean to the `wstring` IDL type (rather than the `string` type) on the DO. It is preferable to map to the `string` IDL type if the data source contains single-byte character data; it is preferable to map to the `wstring` IDL type if the data source contains double-byte or Unicode character data.
- `-workloadmanaged` -- Directs the tool to configure a CMP entity bean or a stateless session bean into a workload managing container and with a workload managed home interface. For a BMP entity bean or a stateful session bean it directs the tool to configure the bean only with a workload managed home interface.
- `-clientdep deployed-jarFile` -- Specifies the name of a dependent JAR required by an EJB client that uses the enterprise bean being deployed. You must specify the full path of the file. To create multiple client JAR files, you must specify this option for each JAR file. This option can be specified more than once, as long as the values are unique.
- `-serverdep deployed-jarFile` -- Specifies the name of a dependent JAR required by the EJB server (CB) that runs the deployed enterprise bean. You must specify the full path of the file. To create multiple dependent JAR files, you must specify this option for each JAR file. This option can also be used to identify existing JAR files that contain classes required by the enterprise bean being deployed; when this is done, the EJB server's `CLASSPATH` environment variable is automatically updated to include this specified JAR file. This option can be specified more than once, as long as the values are unique.
- `-sentinel sentinelValue` -- Specifies an value for a Java type or container-managed field for the deployed beans. If you set a value for a Java type, do not put spaces around the `=` (equals) sign.
- `-strbehavior` -- Specifies how the tool should determine the behavior of the strings for a container-managed string fields in deployed beans. The `corba` value indicates that strings should be handled as CORBA strings; the `strip` value directs the tool to remove trailing spaces from strings.

For session beans or entity beans with BMP, the code generation process runs without additional user intervention. For entity beans with CMP, the Object Builder GUI is displayed during execution of the command, and you must create a DO implementation to manage the entity bean's persistent data. For more information, see [Building a data object during CMP entity bean deployment](#).

The **cbejb** tool deploys enterprise beans by generating extensible markup language (XML) files and importing those files into Object Builder. If the XML import fails, you can view any error messages generated by Object Builder in the `import_model.log` file located in the project directory.

If your `CLASSPATH` environment variable is too long, the **cbejb** command file fails. If this happens, shorten your `CLASSPATH` by removing any unnecessary files.

The **cbejb** tool generates the following files for an EJB JAR file containing an enterprise bean named Account:

- `AccountS.jar` and (*Windows NT*) `AccountS.dll` or (*AIX or Solaris*) `libAccountS.so`--The files required by the EJB server (CB) that contains this enterprise bean. The `AccountS.jar` file contains the code generated from the Account EJB JAR file. The `AccountS.dll` and `libAccountS.so` files contain the required C++ classes.

(*Windows NT*) To run the Account enterprise bean in an EJB server (CB), the `AccountS.jar` file must be defined in the server's `CLASSPATH` environment variable, and the `AccountS.dll` file must be defined in the server's `PATH` environment variable. Typically, the System Management End User Interface (SM EUI) sets these environment variables during installation of the deployed enterprise bean into an EJB server (CB).

(*AIX or Solaris*) To run the Account enterprise bean in an EJB server (CB), the `AccountS.jar` file must be defined in the server's `CLASSPATH` environment variable, and the `libAccountS.so` file must be defined in the server's `LD_LIBRARY_PATH` environment variable. Typically, the SM EUI sets these environment variables during installation of the deployed enterprise bean into an EJB server (CB).

- `AccountC.jar`--The file required by an EJB client, including enterprise beans that access other enterprise beans. This JAR file contains everything in the original EJB JAR file except the enterprise bean implementation class. To use the Account enterprise bean, a Java EJB client must have the `AccountC.jar` and the IBM Java ORB defined in its `CLASSPATH` environment variable.

- (PAO only) *paotoejbName.jar*--This file is created by the **PAOToEJB** tool and is used to wrap an existing procedural adapter object (PAO) in an enterprise bean.
- *EJBAccountFamily.DDL*--This file is used during installation of the Account family into an EJB server (CB) to update the database used by the SM EUJ. Its name is composed of the EJB JAR file name with the string *Family.DDL* appended.

Building a data object during CMP entity bean deployment

When deploying an entity bean with CMP in the EJB server (CB), you must create a DO implementation by using Component Broker's Object Builder. This DO implementation manages the entity bean's persistent data.

To build a DO implementation, you must map the entity bean's container-managed fields to the appropriate data source as described in [Guidelines for mapping the container-managed fields to a data source](#). Then, you must do one of the following:

- Use an existing DB2 or Oracle database to store the bean's persistent data; for more information, see [Using an existing DB2 or Oracle data source to store persistent data](#).
- Use an existing CICS or IMS application to store the bean's persistent data; for more information, see [Using an existing CICS or IMS application to store persistent data](#).
- Define a new DB2 or Oracle database to store the bean's persistent data; for more information, see [Defining a new DB2 or Oracle database to store persistent data](#).

Guidelines for mapping the container-managed fields to a data source

When you deploy enterprise beans with the **cbejb** tool, a Component Broker DO IDL interface is created. The IDL attributes of this interface correspond to the entity bean's container-managed fields. You must then define the DO implementation by using Object Builder to map the DO attributes to the attributes of a Persistent Object (PO) or Procedural Adapter Object (PAO), which correspond to the data types found in the data source.

This section contains information on how the **cbejb** tool maps the container-managed fields of entity beans to DO IDL attributes, and how the enterprise bean deployer maps DO IDL attributes to the entity bean's data source. These guidelines apply whether you are using an existing data source (also known as meet-in-the-middle deployment) or defining a new one (also known as top-down deployment).

- **EJBOject or EJBHome variables**--Objects of classes that implement the **EJBOject** or **EJBHome** interface map to the Object IDL type. At run time, this DO attribute contains the CORBA proxy for the **EJBOject** or **EJBHome** object. The CB EJB run time automatically converts between the **EJBOject** or **EJBHome** object (stored in the bean's container-managed field) and the **CORBA::Object** attribute (stored in the C++ DO). It is possible to deploy container-managed beans that have container-managed fields of the same type, for example, a linked list implementation where each node of the list is a container-managed bean that has a reference to the next node. It is also possible to have circular references in a container-managed field, for example, a container-managed Bean A can have a container-managed field of type Bean B, which in turn has a container-managed field of type Bean A. When defining the DO-to-PO mapping in Object Builder, you can use either a predefined Component Broker mapping of **CORBA::Object** to the data source, or implement a C++ DO-to-PO mapping helper (in the standard Component Broker way) to invoke methods on the C++ proxy to obtain the persistent data. For more information on creating a C++ DO-to-PO mapping, see the Component Broker Programming Guide.

Note: Although Component Broker allows an entity bean's container-managed fields to be **EJBOject** or **EJBHome** objects, the Enterprise JavaBeans 1.0 specification does not.

- **Primary key variables**--Do not map an enterprise bean's primary key variables to the SQL type long varchar in a DB2 or an Oracle database. Instead, use either a varchar or a char type and set the length appropriately.
- **java.lang.String variables**--Objects of this class are mapped to a DO IDL attribute of type string or wstring, depending on the command-line options used when the entity bean was deployed by using the **cbejb** tool (see [Deploying an enterprise bean](#)). By default, a variable of type **java.lang.String** is mapped to a DO IDL attribute

of type string; however, the `-usewstringindo` option of the **cbejb** tool can be used to map `java.lang.String` variables to DO IDL attributes of type `wstring`. (Mapping some of a bean's `String` fields to the IDL `string` type and others to the IDL `wstring` type is not supported.) It is preferable to map to the `string` IDL type if the data source contains single-byte character data; it is preferable to map to the `wstring` IDL type if the data source contains double-byte or Unicode character data.

- `java.io.Serializable` variables--Objects of classes that implement this interface are mapped to a DO IDL attribute of type `ByteString` (which is a typedef for sequence of octet defined in the `IManagedClient.idl` file). The EJB server (CB) automatically converts serializable objects (stored in the entity bean's container-managed fields) to the C++ sequence of octets containing the serialized form of the object (stored in the DO). Use the Component Broker default DO-to-PO mapping for `ByteString` to store the serialized object directly in the data source.

Unless you implement a C++ DO-to-PO mapping helper that passes the C++ `ByteString` to a Java implementation by way of the interlanguage object model (IOM), it is not possible to manipulate the serialized Java object contained in a `ByteString` from within a C++ DO implementation. Therefore, if you are doing top-down enterprise bean development and you don't want to store a serialized Java object in the data source, it is recommended that you avoid defining container-managed fields of type `Serializable`. Instead, make the `Serializable` variable a nonpersistent variable, define primitive type container-managed fields to capture the state of the `Serializable` variable, and convert between the `Serializable` variable and the primitive variable in the `ejbLoad` and `ejbStore` methods of the enterprise bean.

- Array variables--These variables are mapped to a DO IDL sequence of the corresponding type in the same way that the individual types are mapped to DO IDL attributes. For example, an array of the `java.lang.String` class is mapped to a DO IDL attribute that is a sequence of type `string` (or a sequence of type `wstring`, if the `-usewstringindo` option of the **cbejb** tool is used). The EJB server (CB) automatically converts between the array (stored in the entity bean's container-managed fields) and the C++ sequence (stored in the DO). You can store the entire sequence in the data source as a whole, or you can write a C++ DO-to-PO mapping helper (in the standard Component Broker way) to iterate through the sequence and store individual elements in the data source separately. For more information on creating a C++ DO-to-PO mapping, see the Component Broker Programming Guide.
- Date/Time fields--The **cbejb** tool maps container-managed fields of type `java.util.Date` and its subclasses (`java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp` *only*) differently from other `Serializable` fields. The following mapping rules are used:
 - `java.util.Date`: ISO-formatted timestamp string (yyyy-mm-dd-hh.mm.ss.mmmmmm)
 - `java.sql.Date`: ISO-formatted date string (yyyy-mm-dd)
 - `java.sql.Time`: ISO-formatted time string (hh.mm.ss)
 - `java.sql.Timestamp`: ISO-formatted timestamp string (yyyy-mm-dd-hh.mm.ss.mmmmmm)

Therefore a container-managed field of one of the above types should be mapped to either a string or a database-specific date/time field that can take an ISO-formatted string as input. (For example, both DB2 and Oracle `Date/Time/Timestamp` column types can take ISO strings as input values.) If a deployer chooses to map a `Date/Time` container-managed field to something other than the types mentioned above, then a special data mapping code should be written in the DO implementation. The mapping code must be able to convert an ISO-formatted string to a backend-specific type and vice versa.

The `java.sql.Timestamp` class has a precision of nanoseconds, whereas ISO timestamp format has a precision of microseconds. Therefore, precision is compromised (by rounding nanoseconds to nearest microseconds) when a `Timestamp` CMP field is mapped. Users should be particularly aware of this when they use the `java.sql.Timestamp` class as one of the attributes of bean's primary key.

While mapping `java.sql.Date` to ISO Date format, the time field values are ignored. Similarly while mapping `java.sql.Time` to ISO Time, the date field values are ignored.

Note: For DB2 only: If an existing database outputs date/time in a non-ISO format, then the deployer must rebind DB2 packages using the "DATETIME ISO" option.

Using an existing DB2 or Oracle data source to store persistent data

To use an existing DB2 or Oracle database to store a CMP entity bean's persistent data, follow these steps. The end result is a PO with attributes that correspond to the items in the database schema.

1. When Object Builder starts, it presents the Open Project dialog. Choose the location of the project directory for your enterprise bean and click **Finish**.
2. To import an existing relational database schema, click **DBA-Defined Schemas** and right-click the appropriate database type.
 - a. On the pop-up menu, click **Import** and **SQL**.
 - b. On the **Import SQL** dialog box, click **Find** and browse for your SQL file.
 - c. Double-click your SQL file.
 - d. Change the name in the **Database Name** text field from `Database` to the actual name of the database.
 - e. Select the appropriate database type and click **Finish**.
3. To create a persistent object (PO) from the database schema, expand **DBA-Defined Schemas** and expand your group.
 - a. Highlight your schema and then right-click it to display a pop-up menu. Click **Add->Persistent Object**.
 - b. On the **Names and Attributes** dialog box, accept the defaults and click **Finish**.
4. Create a DO implementation as follows:
 - a. Expand the **User-Defined DOs**, expand the **DO File** (for example `CBAccountDO`), expand the **DO Interface** (for example, `com_ibm_ejs_doc_account_AccountDO`), and select the **DO Implementation**.
 - b. On the **DO Implementation** pop-up menu, select **Properties**.
 - c. On the **Name and Platform** page, select the **Deployment Platform** (for example, `NT`, `AIX`, or `Solaris`) and click **Next**.
 - d. On the **Behavior** page, make the appropriate selections and click **Next**:
 - *For DB2*, select `BOIM` with any Key for **Environment**, select `Embedded SQL` for **Form of Persistent Behavior and Implementation**, select `Delegating` for **Data Access Pattern**, and select `Home` name and key for **Handle for Storing Pointers**.
 - *For Oracle*, select `BOIM` with any Key for **Environment**, select `Oracle Caching services` for **Form of Persistent Behavior and Implementation**, select `Delegating` for **Data Access Pattern**, and select `Home` name and key for **Handle for Storing Pointers**.
 - e. On the **Implementation Inheritance** page, make the appropriate selections for the parent class and click **Next**:
 - *For DB2*, select `IRDBIMExtLocalToServer::IDataObject`
 - *For Oracle*, select `IRDBIMExtLocalToServer::ICachingServiceDataObject`
 - f. Accept the defaults for the **Attributes**, **Methods**, and **Key and Copy Helper** pages by clicking **Next** on each page.
 - g. On the **Associated Persistent Objects** page, click **Add Another**. Accept the default for the instance name (iPO) and select the correct type. Click **Next**.
 - h. On the **Attribute Mapping** page, map the container-managed fields of the entity bean to the corresponding items in the database schema. Object Builder creates default mappings for the data object attributes for which it can identify corresponding persistent object attributes. The default mapping is generally suitable for everything except for the primary key variable, which you must map to a `varchar` or `char` type rather than a long `varchar` type. For more information, see [Guidelines for mapping the container-managed fields to a data source](#). After you finish mapping the attributes, click **Finish**.
 - i. *Oracle only*. When mapping an entity bean with CMP to an Oracle database, expand the **Container**

Definition folder and right-click the EJB container. From the pop-up menu, click **Properties**. In the wizard, click **Next** until you reach the **Data Access Patterns**; on that page, check the **Cache Service** checkbox and click **Finish**.

- j. Exit from Object Builder by clicking **File->Exit**; save any changes if prompted.
- k. Create the database specified by the **Database** text field and use the SQL file specified by the **Schema File** text field to create a database table. For more information on creating a database and database table with an SQL file, consult your DB2 or Oracle documentation. The SQL file can be found in the following directory, where *projDir* is the project directory created by the **cbejb** tool:
 - *On Windows NT, projDir\Working\NT*
 - *On AIX, projDir/Working/AIX*
 - *On Solaris, projDir/Working/Solaris*

Using an existing CICS or IMS application to store persistent data

To use CICS or IMS for Persistent Adaptor Object (PAO) storage, following these instructions. Note that if the persistent store uses a CICS or IMS application (by way of a PAO), only application data is used; the methods on the CICS or IMS application are pushdown methods, which run application-specific logic rather than storing and loading data.

The following prerequisites must be met to map an entity bean with CMP to an existing CICS or IMS application:

- The entity bean's transaction attribute must be set to TX_MANDATORY if you want to map the bean to a HOD- or ECI-based application. The transaction attribute must be set to either the TX_MANDATORY or TX_REQUIRED if you want to map it to an APPC-based application.
- The existing CICS or IMS application must be represented as a procedural adapter object (PAO). See the Procedural Application Adaptor Development Guide for more information on creating PAOs.
- The PAO class files must be specified in the CLASSPATH environment variable.
- The entity bean must implement all enterprise bean logic; the only remaining requirement is to map the entity bean's container-managed fields to the PAO. Pushdown methods on the PAO cannot be utilized from the enterprise bean. (PAO pushdown methods can be used from an entity bean with CMP generated by using the **PAOtoEJB** tool as described in [Creating an enterprise bean from an existing CICS or IMS application](#).)
- The **cbejb** tool must be run as follows, where the *ejb-jarFile* is the EJB JAR file containing the entity bean:

```
# cbejb ejb-jarFile [-hod | -eci | -appc [beanNames]]
```

For a description of the **cbejb** tool's syntax, see [Deploying an enterprise bean](#).

If you have met the prerequisites, use Object Builder to create the mapping between the entity bean and the CICS or IMS application:

1. When Object Builder starts, it presents the Open Project dialog. Choose the location of the project directory for your enterprise bean and click **Finish**.
2. From the main menu, click **Platform** and then **Target**. Uncheck the 390 platform.
3. Click **User-Defined PA Schemas** and right-click the selection.
4. From the pop-up menu, click **Import** and then **Bean**. On the **Import Bean** dialog box, type the class name of the PAO bean and click **Next**.
5. Select the appropriate connector type and click **Next**.
6. Select the primary key attribute name from the **Properties** list.
7. Click >> to move the primary key to the **Key Attributes** list and click **Finish**.
8. *For HOD and ECI only*, do the following for both the MO and the HomeMO:
 - a. In the **Tasks and Object** panel, expand the **User-Defined Business Objects**, expand the object, and expand the object's BO. From the MO file's pop-up menu, click **Properties**.

- b. Change the **Service to use** property from `Transaction Service` to `Session Service`.
9. Create a DO implementation as follows:
 - a. On the **Tasks and Object** panel, expand the **User-Defined DOs**, expand the **DO File** from the menu, and click the **DO Interface**.
 - b. On the **DO Interface** pop-up menu, select **Add Implementation**.
 - c. On the **Behavior** page, select `BOIM` with any Key for **Environment**, select `Procedural Adapters` for **Form of Persistent Behavior and Implementation**, select `Delegating` for **Data Access Patterns**, and select `Default` for **Handle for Storing Pointers**. Click **Next**.
 - d. Click **Next** on the **Implementation Inheritance** page, the **Attributes** page, the **Methods** page, and the **Key and Copy Helper** page.
 - e. On the **Associated Persistent Object** page, click **Add Another**, verify that the PO that you previously created is selected, and click **Next**.
 - f. On the **Attribute Mapping** page, designate how the container-managed fields of the entity bean correspond to the items in the existing PAO. This designation is done by defining a mapping between the attributes of the DO (which match the entity bean's container-managed fields) to the attributes of the PO (which match the existing PAO). In the **Attributes** list, there is a DO attribute corresponding to each of the bean's container-managed fields.

For each DO attribute in the **Attributes** list, right-click the attribute and click **Primitive** from the menu. From the **Persistent Object Attribute** drop-down menu, select the PO attribute (the item from the existing database schema) that corresponds to the DO attribute. For more information, see [Guidelines for mapping the container-managed fields to a data source](#). After you have processed all container-managed fields, click **Next**.
 - g. On the **Methods Mapping** page, for each method in the list of **Special Framework Methods**, right-click a method and click **Add Mapping**. From the **Persistent Object Method** drop-down menu, select the PO method with the same name as the selected DO method. If there are more methods than available mappings, map methods to similarly named methods. For example, map `update` to `update()`. After you have processed all of the methods, click **Finish**.
 - h. Expand the **Container Definition** folder and right-click the EJB container. From the pop-up menu, click **Properties**. In the wizard, click **Next** until you reach the **Data Access Patterns** page.
 - i. On the **Data Access Patterns** page, select one of the following items and then click **Next**:
 - *For HOD or ECI*, select `Use PAA Session services`.
 - *For APPC*, select `Use PAA Transaction services`.
 - j. On the **Service Details** page, do the following and then click **Next**:
 - *For HOD or ECI*, select `Throw an exception and abandon the call for Behavior for Methods Called Outside a Transaction`; define a connection name, for example, `MY_PAA_Connection`; select `Host on Demand` or `ECI connection`, respectively, for the **Type of connection**.
 - *For APPC*, select `Throw an exception and abandon the call for enterprise beans with the TX_MANDATORY transaction attribute`, or select `Start a new transaction and complete the call for enterprise beans with the TX_REQUIRED transaction attribute`.
 - k. Select `Caching` for **Business Object**.
 - l. Select `Delegating` for **Data Object**.
 - m. Click **Finish**.
 10. Exit from Object Builder by clicking **File->Exit**; save any changes if prompted.

Defining a new DB2 or Oracle database to store persistent data

When you use a top-down development approach to enterprise bean development, enterprise bean deployment must

occur in three phases:

1. Define the database schema, map the container-managed fields of the entity bean with CMP to the database schema, and generate the code to encapsulate this mapping. For more information, see [Mapping the database schema](#).
2. Create the database and database tables. For more information, see [Creating the database and database table](#).
3. Compile the code generated in phase [1](#); compilation fails if the database and database tables do not exist.

Mapping the database schema

After you have defined the manner in which the entity bean maps to a database, create the mapping by running the **cbejb** tool with the `-nc` option to prevent automatic compilation after code generation. For example, to create a mapping for an Account bean stored in an EJB JAR file named `EJBAccount.jar`, enter the following command:

```
# cbejb EJBAccount.jar -nc -queryable [-oracle | -cacheddb2]
```

Note: If the database being used to store the persistent data is either Oracle or DB2, those options must also be specified.

Creating the database and database table

Follow these instructions to create a database and database table by using the Object Builder GUI:

1. When Object Builder starts, it presents the Open Project dialog. Choose the location of the project directory for your enterprise bean and click **Finish**.
2. Create a DO implementation as follows:
 - a. Expand the **User-Defined DOs**, expand the **DO File** from the menu, and click the **DO Interface**.
 - b. On the **DO Interface** pop-up menu, select **Add Implementation**. If the implementation is already present, you can modify it by selecting the implementation, invoking the pop-up menu, and selecting **Properties**.
 - c. On the **Name and Platform** page, select the platform and click **Next**.
 - d. On the **Behavior** page, make the appropriate selections and click **Next**:
 - *For DB2:* select **BOIM** with any Key for **Environment**, select **Embedded SQL** for **Form of Persistent Behavior and Implementation**, select **Delegating** for **Data Access Pattern**, and select **Home name** and key for **Handle for Storing Pointers**.
 - *For Oracle:* select **BOIM** with any Key for **Environment**, select **Oracle Caching services** for **Form of Persistent Behavior and Implementation**, select **Delegating** for **Data Access Pattern**, and select **Home name** and key for **Handle for Storing Pointers**.
 - e. On the **Implementation Inheritance** page, make the appropriate selections for the parent class and click **Next**:
 - *For DB2,* select `IRDBIMExtLocalToServer::IDataObject`
 - *For Oracle,* select `RDBIMExtLocalToServer::ICachingServiceDataObject`
 - *For CICS or IMS PAO,* select `IRDBIMExtLocalToServer::IDataObject`
 - f. Accept the defaults for the **Attributes**, **Methods**, and **Key and Copy Helper** pages by clicking **Next** on each page.
 - g. On the **Associated Persistent Objects** page, click **Add Another**. Accept the default for the instance name (iPO) and select the correct type. Click **Next**.
 - h. On the **Attribute Mapping** page, map the container-managed fields of the entity bean to the corresponding items in the database schema. The default mapping is generally suitable for everything except for the primary key variable, which you must map to a varchar or char type rather than a long varchar type. Object Builder creates default mappings for the data object attributes for which it can identify corresponding persistent object attributes. For more information, see [Guidelines for mapping](#)

[the container-managed fields to a data source](#). After you finish mapping the attributes, click **Finish**.

- i. *Oracle only*. When mapping an entity bean with CMP to an Oracle database, expand the **Container Definition** folder and right-click the EJB container. From the pop-up menu, click **Properties**. In the wizard, click **Next** until you reach the **Data Access Patterns**; on that page, check the **Cache Service** checkbox and click **Finish**.
- j. Exit from Object Builder by clicking **File->Exit**; save any changes if prompted.
- k. Create the database specified by the **Database** text field and use the SQL file specified by the **Schema File** text field to create a database table. For more information on creating a database and database table with an SQL file, consult your DB2 or Oracle documentation. The SQL file can be found in the following directory, where *projDir* is the project directory created by the **cbejb** tool:
 - *On Windows NT*, *projDir\Working\NT*
 - *On AIX*, *projDir/Working/AIX*
 - *On Solaris*, *projDir/Working/Solaris*

Compiling the generated code

After both the database and database table are created, compile the enterprise bean code by using the following commands:

- **On Windows NT.**

```
# cd projDir\Working\NT
# nmake -f all.mak
```

- **On AIX.**

```
# cd projDir/Working/AIX
# make -f all.mak
```

- **On Solaris.** Transfer the code that was generated on Windows NT or AIX, then use the following commands.

```
# cd projDir/Working/Solaris
# make -f all.mak
```

Installing an enterprise bean and configuring its EJB server (CB)

Follow these steps to install an enterprise bean and configure the resulting EJB server (CB):

1. (*Entity bean with CMP using DB2 only*) Use the bind file, which Object Builder generates as a side effect of using the **cbejb** tool, to bind the enterprise bean to the database (for example, `db2 bind AccountTblPO.bnd`).
2. Using the SM EUI, install the application generated by **cbejb**. In general, this installation is the same as installing a Component Broker application generated by Object Builder:
 - a. Load the application into a host image.
 - b. Add the application to a configuration.
 - c. Associate the EJB application with a server group or server. (If the server group or server does not already exist, you must create it.)
 - d. (*Entity bean with CMP only*) Associate the entity bean's data source (DB2, Oracle, CICS, or IMS PAA) with the EJB application:
 - DB2: associate the DB2 services (`iDB2IMServices`) with the EJB server.
 - Oracle: associate the Oracle services (`iOAAServices`) with the EJB server.
 - CICS or IMS PAA: associate the PAA services (`iPAAServices`) with the EJB server.
 - e. Configure the EJB server (CB) with a host.

- f. Set the ORB request timeout for both clients and servers to 300 seconds.
- g. If the EJB server requires Java Virtual Machine (JVM) properties to be set, edit the JVM properties. Do this in the server model instead of the server image. For instance, if the enterprise bean performs a JNDI lookup to access other enterprise beans, the server hosting the enterprise bean must have its JVM properties set to include values for JNDI properties.
- h. Activate the EJB server configuration.
- i. Start the EJB server.

Binding the JNDI name of an enterprise bean into the JNDI namespace

Note: This section does not apply to servers running on the AIX, Windows NT, or Solaris platforms.

An enterprise bean's JNDI home name is defined within its deployment descriptor as described in [The deployment descriptor](#). This name is used by EJB clients (including other enterprise beans) to find the home interface of an enterprise bean.

The **ejbbind** tool locates the CB home that implements the enterprise bean's EJBHome interface in the Component Broker namespace. It also rebinds the home name into the namespace, using the JNDI home name specified in the enterprise bean's deployment descriptor. This binding enables an EJB client to look up the EJB home by using the JNDI name specified in the bean's deployment descriptor. An enterprise bean can be bound on a different machine from the one on which the bean was deployed.

The subtree of the Component Broker namespace in which the JNDI name is bound can be controlled by the command-line options used with the **ejbbind** tool. The manner in which the name is bound (the subtree chosen) affects the JNDI name that EJB clients must use to look up the enterprise bean's EJB home and also affects the visibility of the enterprise bean's EJB home. Specifically, the JNDI name can be bound in one of the following ways:

- The JNDI name can be bound into the local root. Under this binding approach, EJB clients use the JNDI name in the enterprise bean's deployment descriptor. The approach restricts the visibility of the EJB home to EJB clients using the same name server (the same bootstrap host) and can cause collisions with other names in the tree.
- The JNDI name can be bound into the host name tree (at `host/resources/factories/EJBHomes`). Under this binding approach, EJB clients must prefix the string `host/resources/factories/EJBHomes` to the JNDI name given in the bean's deployment descriptor. This approach minimizes collisions with other names in the tree, but restricts visibility of the enterprise bean home to clients using the same name server.
- The JNDI name can be bound into the workgroup name tree (at `workgroup/resources/factories/EJBHomes`). Under this binding approach, EJB clients must prefix the string `workgroup/resources/factories/EJBHomes` to the JNDI name given in the enterprise bean's deployment descriptor, and the EJB home is visible to all EJB clients using a name server that belongs to the same preferred workgroup.
- The JNDI name can be bound into the cell name tree (at `cell/resources/factories/EJBHomes`). Under this binding approach, EJB clients must prefix `cell/resources/factories/EJBHomes` to the JNDI name in the bean's deployment descriptor, and the EJB home is visible throughout the cell.

Before running the **ejbbind** tool, do the following:

- Deploy your enterprise bean for Component Broker by using the **cbejb** tool. For more information, see [Deploying an enterprise bean](#).
- Install the Component Broker application that **cbejb** tool generates, and configure it on a specific EJB server (CB) by using the SM EUI. For more information, see [Installing an enterprise bean and configuring its EJB server \(CB\)](#).
- Start the CBConnector Service and a name server, if they are not already running. For more information, see the Component Broker System Administration Guide.

- Activate the configuration containing the EJB server (CB) that runs the application.
- Determine the IP address (the bootstrap host name) and port number (the bootstrap port) of the machine running the name server.

Invoke the **ejbbind** command with the following syntax:

```
ejbbind ejb-jarFile [beanParm] [-f]
[-BindLocalRoot ] [-BindHost] [-BindWorkgroup] [-BindCell] [-BindAllTrees]
[-ORBInitialHost hostName] [-ORBInitialPort portNumber]
[-u] [-UnbindLocalRoot] [-UnbindHost] [-UnbindWorkgroup] [-UnbindCell]
[-UnbindAllTrees]
```

The *ejb-jarFile* is the fully-qualified path name of the EJB JAR file containing the enterprise bean to be bound or unbound. The optional beanParm argument is used to bind a single enterprise bean in the EJB JAR file; you can identify this bean by supplying a fully qualified name (for example, com.ibm.ejs.doc.account.Account, where Account is the bean name) or the name of the enterprise bean's deployment descriptor file without the .ser extension. If an enterprise bean has multiple deployment descriptors in the same EJB JAR file, you must supply the deployment descriptor file name rather than the enterprise bean name.

When no options are specified, the JNDI name is bound into the local root's name tree, using the local host and port 900 for the bootstrap host (the name server).

The other options do the following:

- -f -- Force the bind, even if the JNDI name is already bound in the namespace; this option is not valid with the unbind command options.
- -BindLocalRoot -- Bind the JNDI name into the local root's name tree.
- -BindHost -- Bind the JNDI name into the host name tree.
- -BindWorkgroup -- Bind the JNDI name into the workgroup name tree.
- -BindCell -- Bind the JNDI name into the cell name tree.
- -BindAllTrees -- Bind the JNDI name into the host, the workgroup, and the cell name trees.
- -ORBInitialHost hostName -- Identify the bootstrap host (the default is the local host).
- -ORBInitialPort portNumber -- Identify the bootstrap port (the default is port 900).
- -u -- Unbind the JNDI name; this option is not valid with bind command options.
- -UnbindLocalRoot -- Unbind the JNDI name from the local root's name tree.
- -UnbindHost -- Unbind the JNDI name from the host name tree.
- -UnbindWorkgroup -- Unbind the JNDI name from the workgroup name tree.
- -UnbindCell -- Unbind the JNDI name from the cell name tree.
- -UnbindAllTrees -- Unbind the JNDI name from the host, the workgroup, and the cell name trees.

If the command is successful, it issues a message similar to the following:

```
Name AccountHome was bound to CB Home
```

You must run the **ejbbind** tool again if any of the following occurs:

- You modify the JNDI name of an enterprise bean. You can modify the JNDI name by using the **jetace** tool. For more information, see [Creating a deployment descriptor and an EJB JAR file](#).
- You reconfigure Component Broker. In this case, you must rebind every enterprise bean served by this configuration.
- You move the enterprise bean to a different EJB server (CB) or a different machine.

Configuring systems management to enable lazy enumeration

To enable lazy enumeration (see [Creating finder logic in the EJB server \(CB\)](#)), follow these steps:

1. From the System Management End User Interface (SM EUI), go to the View menu, and set the View Level to Control.
2. Expand **Host Images**
3. Expand the name of your host.
4. Expand **Server Images**.
5. Expand the name of your server.
6. Expand **Container Images**.
7. Right-click **iIteratorSysObjsNoPRef**. From the pop-up menu, select **Properties**. Change the following properties:
 - Change the **Default transaction policy** to `throwException`.
 - Change the **Memory management policy** to `passivate at end of transaction`.

The transaction policy ensures that the caller starts a transaction. The memory management policy ensures that the lazy enumerations are passivated when the transaction completes.

Resolving to EJB homes using lifecycle services in CBCConnector

Note: This section applies only to servers running on the AIX, Windows NT, or Solaris platforms.

When an EJB client performs a simple JNDI lookup, a 1-to-1 mapping is made between the name and the particular EJB home instance. In a distributed environment, this model can be limiting. In such an environment, for example, there may be many EJB homes supporting the same type of enterprise bean. It is better to have an approach that does not require an application to request a specific instance of that home. In addition, as changes are made to the system, it is important that applications not have to be changed or redeployed to specify a different instance of an EJB home. The CBCConnector LifeCycle Service provides a level of indirection and abstraction that allows the application to request a home that is within a particular scope of location within the distributed environment, yet be isolated from the specifics of the exact configuration of the environment. For more info on lifecycle factory finders, see the LifeCycle section in the Advanced Programming Guide.

Using CBCConnector, a JNDI context can be associated with a LifeCycle Service factory finder so that the associated factory finder is used to resolve EJB home lookup operations from the context. Contexts such as these enable deployers of EJB applications to take advantage of the power of factory finders in a manner that is transparent to clients of these applications.

To resolve EJB home lookups with factory finders, the application deployer can use pre-defined default application contexts associated with the various CBCConnector-supplied default factory finders or use the **appbind** tool to create application-specific contexts and associate them with any given factory finder. For more information on each approach, see [Default context-to-finder associations](#) and [Application-specific contexts and the appbind tool](#).

Note: Default application contexts and application-specific contexts eliminate the need for the **ejbbind** tool, which creates a simple 1-to-1 mapping of a JNDI name and an EJB home instance. Clients must use one of the default initial context factories or an application-specific context factory generated by the **appbind** tool.

Default context-to-finder associations

There are several default factory finders built into CBCConnector, each of which searches particular scopes of location when finding a factory. When an EJB application is deployed on a CBCConnector server, the EJB homes for the application are bound in the LifeCycle repository using the names for the EJB homes as specified by the deployment descriptors contained in the application's EJB jar file. A factory finder can find any EJB home within the scope of its particular search rules.

An EJB client can use a particular built-in CBCConnector default factory finder simply by using the initial context factory that corresponds to that factory finder. The initial context returned by the context factory will use its corresponding factory finder to resolve EJB home lookup requests.

Contexts returned by the following initial context factories:

1. `com.ibm.ejb.cb.runtime.CBCtxFactoryHostDefault`
2. `com.ibm.ejb.cb.runtime.CBCtxFactoryHostWidenedDefault`
3. `com.ibm.ejb.cb.runtime.CBCtxFactoryHostServerDefault`
4. `com.ibm.ejb.cb.runtime.CBCtxFactoryHostServerWidenedDefault`
5. `com.ibm.ejb.cb.runtime.CBCtxFactoryWorkGroupDefault`
6. `com.ibm.ejb.cb.runtime.CBCtxFactoryWorkGroupWidenedDefault`
7. `com.ibm.ejb.cb.runtime.CBCtxFactoryWorkGroupServerDefault`
8. `com.ibm.ejb.cb.runtime.CBCtxFactoryWorkGroupServerWidenedDefault`
9. `com.ibm.ejb.cb.runtime.CBCtxFactoryCellDefault`
10. `com.ibm.ejb.cb.runtime.CBCtxFactoryCellServerDefault`
11. `com.ibm.ejb.cb.runtime.CBCtxFactoryCellServerWidenedDefault`

resolve EJB home lookup operations with the corresponding factory finders:

1. `host/resources/factory-finders/host-scope`
2. `host/resources/factory-finders/host-scope-widened`
3. `host/resources/factory-finders/server-server-scope`
4. `host/resources/factory-finders/server-server-scope-widened`
5. `workgroup/resources/factory-finders/workgroup-scope`
6. `workgroup/resources/factory-finders/workgroup-scope-widened`
7. `workgroup/resources/factory-finders/server-server-scope`
8. `workgroup/resources/factory-finders/server-server-scope-widened`
9. `cell/resources/factory-finders/host-scope`
10. `cell/resources/factory-finders/server-server-scope`
11. `cell/resources/factory-finders/server-server-scope-widened`

Server-based context factories can only be used by a client that is running as a CBCConnector server, in which case, *server* is the name of the CBCConnector server.

Default context factories can only be used by client applications that issue fully qualified EJB home lookups. If a client traverses to a subcontext and then performs a partially qualified EJB home lookup, you must run the **appbind** tool to create an application-specific context with home subcontexts and to generate an application-specific initial context factory. For more information, see [Application-specific contexts and the appbind tool](#).

Application-specific contexts and the appbind tool

If a CBCConnector-supplied default factory finder is being used to locate an EJB home, CBCConnector supplies a default mapping between application contexts and default factory finders (for more information, see [Default context-to-finder associations](#)). For added flexibility, an enterprise bean deployer can create an application-specific context with optional EJB home subcontexts and associate it with any factory finder. The factory finder association can be changed at a later time if desired. To isolate clients from the actual context name, the enterprise bean deployer generates an initial context factory for the application-specific context by using the **appbind** tool.

The **appbind** tool allows deployers to create an application-specific naming context and associate it with a selected factory finder so that lookup operations are resolved with that factory finder. These application-specific contexts are designed to be initial JNDI contexts for EJB clients so that JNDI lookup calls on EJB homes are transparently resolved with the associated factory finder. The **appbind** tool enables users to create, modify, and delete such application-specific contexts. Note that the application's EJB home instances are not actually bound under the application-specific context. Instead, they are bound to the LifeCycle repository. The associated factory finder will resolve the EJB home lookups using the lifecycle rules defined for it.

All application-specific contexts must have one of the following context name stems:

- host/applications/initial-contexts
- workgroup/applications/initial-contexts
- cell/applications/initial-contexts

depending on whether a scope of host, workgroup, or cell is specified when the context is created.

By default, the factory finder `host/resources/factory-finders/host-scope-widened` is associated with an application-specific context created with the **appbind** tool. However, the user can specify another factory finder. The factory finder can be one of the other default factory finders, one created by an administrator using System Management, or one created by an application program you write. For more information, see the LifeCycle section in the Advanced Programming Guide.

Under an application-specific context, subcontexts for EJB home names optionally can be created. For example, if the name for a home is `com/mycom/myapp/MyHome`, the subcontext `com/mycom/myapp` can be created. These subcontexts provide additional transparency to the client. They allow a client to traverse the JNDI name space from the application-specific context down to any subcontext that corresponds to a non-leaf component of an EJB home name. The factory finder associated with the application-specific context is also used to resolve EJB home lookup operations from these subcontexts. The **appbind** tool creates a subcontext for each home name in the deployment descriptors within a specified EJB JAR file.

The **appbind** tool can optionally create a Java source file for an initial context factory for the application-specific context being created. This initial context factory can be used as the initial context factory by clients. The **appbind** tool also allows the user to override the default bootstrap host to use for ORB initialization.

Invoke the **appbind** tool with the following syntax:

```
appbind [-u] -name contextName [-sc jarFileName] [-host | -workgroup | -cell]
[-factoryfinder factoryFinderPath]
[-genctxfactory factoryClassName [-o targetDir]]
[-boothost bootstrapHostUrl]
```

The context being bound or unbound is specified with the required `-name` option, where `contextName` is the name of the JNDI application-specific context to bind or unbind. All application context names are relative to one of the following context name stems

- host/applications/initial-contexts
- workgroup/applications/initial-contexts
- cell/applications/initial-contexts

depending on whether a scope of host, workgroup, or cell was specified. (See the `-host`, `-workgroup`, and `-cell` options below.)

A bind operation is performed unless the `-u` option is specified, in which case, an unbind operation is performed. If a bind operation is performed on an existing context, the current factory finder association is added or replaced. The context cannot be a child or parent of a context which already has a factory finder association.

The other options do the following:

- `-u`--This flag is used to perform an unbind operation. An unbind operation unbinds the context specified with the `-name` option and the `-sc` option, if specified. If the `-sc` option is specified, only the subcontexts corresponding to the JNDI home names in the JAR's deployment descriptors are removed. If the `-sc` option is not used, the context specified by the `-name` option and all of its subcontexts are unbound. To help keep the name tree manageable, once a context or subcontext is unbound, parent contexts are recursively unbound up to the context name stem (see the `-name` option above) or until a non-empty parent is encountered.
- `-sc`--This option is used to specify subcontexts, where `file jarFileName` is the name of an EJB JAR file that contains deployment descriptors with EJB home names. Each of the EJB home names, not including the leaf-name component, is treated as a subcontext name. For example, if the name for a home is `com/mycom/myapp/MyHome`, the subcontext name is `com/mycom/myapp`.

When binding, the subcontext names are created under the application-specific context specified by the `-name` flag. When unbinding, the contexts which are unbound are restricted to the subcontext names identified by the JAR file. Whether binding or unbinding, other subcontexts are not affected.

- `-host`, `-workgroup`, `-cell`--These flags control the scope of the application context being bound or unbound. Each scope has a corresponding context name stem, as described in the `-name` flag section above. The `-host`, `-workgroup`, and `-cell` flags specify a scope of host, workgroup, or cell, respectively, for the context. The default scope is host scope. Only one scope can be specified per bind or unbind operation.
- `-factoryfinder`--This option is used to specify which factory finder to associate with the application-specific context being bound, where `factoryFinderPath` is the name of the factory finder. The default factory finder is `host/resources/factory-finders/host-scope-widened`.

This option does not apply to unbind operations.

- `-genctxfactory`--Typically, when an application-specific context is bound, it is desirable to have an initial context factory for the application-specific context. This option directs the **appbind** tool to create a Java source file for an initial context factory, where `factoryClassName` is the fully-qualified class name of the context factory. All package prefix subdirectories are created, if necessary. If the source file already exists, it is replaced. The file and its containing subdirectories are created relative to the directory specified with the `-o` option or, by default, relative to the current directory.

This option does not apply to unbind operations.

- `-o`--This option is used to specify the target directory for the initial context factory file (see the `-genctxfactory` option), where `targetDir` is the directory path (not including package prefix directories). The default target directory is the current directory.

This option does not apply to unbind operations.

If the `-o` option is used, use of the `-genctxfactory` flag is required.

- `-bootstrapHostUrl`--This option is used to override the default host and port used for ORB initialization, where `bootstrapHostUrl` is the URL of the bootstrap host. The bootstrap host URL has the form
`iiop:// hostName [: portNumber]`

Creating an enterprise bean from an existing CICS or IMS application

You can create an enterprise bean from an existing CICS or IMS application by using the **PAOToEJB** tool. The application must be mapped into a PAO prior to creating the enterprise bean. For more information on creating PAOs, see the Component Broker document entitled Procedural Application Adaptor Development Guide and the VisualAge for Java, Enterprise Edition documentation.

The **PAOToEJB** tool runs independently of the other tools described in this chapter. To create an enterprise bean from a PAO class, do the following:

1. Change to the directory where your PAO class file exists.
2. Add the PAO class file's directory, or the JAR file containing the class, to your CLASSPATH environment variable.
3. Invoke the **PAOToEJB** command with the following syntax:

```
PAOToEJB -name [ejbName] paoClass -hod | -eci | -appc
```

The *ejbName* argument is optional and specifies the enterprise bean's name (for example, Account). If this name is not supplied, the enterprise bean is named by using the short name of the PAO class. The *paoClass* argument is required and specifies the fully qualified Java name of the PAO class without the .class extension; the PAO class is always a subclass of com.ibm.ivj.eab.paa.EntityProceduralAdapterObject. You must also specify one of the following options:

- -hod --This indicates that the PAO class is for Host On-Demand (HOD). HOD is a browser-based 3270 telnet connection.
- -eci --This indicates that the PAO class is for External Call Interface (ECI). ECI is a proprietary protocol that provides a remote procedure call (RPC)-like interface into CICS.
- -appc --This indicates that the PAO class is for advanced program-to-program communications (APPC), which is the System Network Architecture (SNA) for LU 6.2 communications.

Note: EJB clients that access entity beans with CMP that use HOD or ECI for CICS or IMS applications must begin a transaction before invoking a method on these entity beans. This is necessary because these types of entity beans must use the TX_MANDATORY transaction attribute.

4. If the *paoClass* is part of a Java package, then you must create the corresponding directory structure and move the generated Java files into this directory.
5. Compile the Java source files of the newly created enterprise bean:

```
javac ejbName*.java
```
6. Place the compiled class components of the enterprise bean into a JAR or ZIP file and use the **jetace** tool to create an EJB JAR file for the bean, as described in [Creating a deployment descriptor and an EJB JAR file](#).
7. Deploy the EJB JAR file by using the **cbejb** tool as described in [Deploying an enterprise bean](#).

Creating an enterprise bean that communicates with MQSeries

Component Broker contains tools for developing BOs that send or receive MQSeries messages. It also allows access to MQSeries queues within distributed transactions. The EJB server (CB) builds on this MQSeries support and allows you to create an enterprise bean that wraps an MQSeries-based BO.

The MQSeries EJB support enables an EJB client application to indirectly interact with MQSeries through an EJB client interface. Both the Component Broker support for MQSeries BOs and the EJB support described here require you to modify the DO implementation generated by Object Builder. The main difference between these two

supported approaches is that when Component Broker MQSeries-based BOs are built, the MQSeries message content is specified through Object Builder, whereas the EJB support requires the MQSeries message content to be specified in a Java properties file.

For more information on the MQSeries support in Component Broker, see the MQSeries Application Adaptor Development Guide document.

The **mqaajb** tool generates a session bean that wraps a Component Broker BO based on the MQSeries Application Adaptor. The resulting session bean implementation is specific to the EJB server (CB) and is not portable to other EJB servers. To deploy the generated session bean, use the **cbejb** tool. The **mqaajb** tool runs independently of other EJB server (CB) tools.

To create a session bean for a particular MQSeries queue, do the following:

1. Create a Java properties file that contains these items:

- The message type specification--The property name must be `messageType`, and its value must be either `Inbound`, `Outbound`, or `InOut`. If `InOut` is chosen, a pair of enterprise beans, instead of a single one, are created to accommodate paired inbound and outbound message queues. Here is an example of this specification:

```
messageType=Inbound
```

- A list of message field specifications--For each message field, the property name is the field name, and the property value is the field type. Here is an example of this specification:

```
bankName=java.lang.String
```

```
accountNumber=int
```

Note: Java class names in the type specifications must be the fully qualified package name.

2. Run the **mqaajb** command with the following syntax:

```
# mqaajb -f propertiesFile -n baseBeanName [-p packageName]
  [-i existingInboundBOInterfaceName]
  [-o existingOutboundBOInterfaceName]
  [-c existingOutboundCopyName]
```

The `-f` and `-n` options are required. The *propertiesFile* specifies the name of the properties file created in Step [1](#), and the *baseBeanName* argument specifies the base name of the enterprise bean or beans to be generated. For example, if the base name is `Account` and the properties file specifies that it is for both an inbound and an outbound message, then the **mqaajb** command generates session beans, related interfaces, and artifacts with the following names:

`AccountInboundBean`

`AccountEJBObject`

`AccountInboundEJBHome`

`AccountOutboundBean`

`AccountOutboundEJBObject`

`AccountOutboundEJBHome`

`AccountMsgTemplate`

The `-p` option specifies the package name of the enterprise bean; if not specified, the package name defaults to `mytest.ejb.mqaa`.

Unless the `-i` option or the `-o` and `-c` options are specified, the **mqaajb** command makes a mark for the **cbejb** command; later, when the **cbejb** command is run over the beans, it generates the required backing

message BOs for the session beans. If you have already created and tested MQSeries Application Adaptor-based BOs (following the procedure described in the MQSeries Application Adaptor Development Guide), you now need only wrap them in session beans. You can specify the names of these BOs and the Copy object to the **mqaajb** command. The **mqaajb** command then creates session beans that use the specified BOs. The names of these objects must be fully qualified. For example:

```
mqaajb -f mymsg.properties -n Account -i TextMessage::TMInbound \
      -o TextMessage::TMOutbound -c TextMessageCopy::TMOutboundCopy
```

You still must specify the base bean name with the **-n** option independently of the existing BOs. You also must provide a properties file; the message format specified in this file must be consistent with the existing BOs. The correct mapping between the C++ field types in the BOs and the Java types in the properties file can be established by referring to the IDL C++/Java binding documentation.

The following items are generated in the working directory on successful completion of the **mqaajb** command:

- The Java source files (and the corresponding compiled class files) that compose the enterprise bean in the subdirectory corresponding to the package name.
 - A JAR file containing the Java source files and compiled files that compose the enterprise bean.
 - An XML file containing the enterprise bean's deployment descriptor.
3. Run the **jetace** tool as follows to generate an EJB JAR file for the enterprise bean:

```
# jetace -f beanName.xml
```

4. Run the **cbejb** tool to deploy the enterprise bean contained in the EJB JAR file. For more information, see [Deploying an enterprise bean](#). When the **cbejb** command is complete, unless you are using existing BOs, you possibly need to follow the steps in the MQSeries Application Adaptor Development Guide to modify the DO implantation.

Restrictions in the EJB server (CB) environment

The following restrictions apply when developing enterprise beans for the EJB server (CB) environment:

- Unqualified interface and exception names cannot be duplicated in enterprise beans. For example, the `com.ibm.ejs.doc.account.Account` interface must not be reused in a package named `com.ibm.ejs.doc.bank.Account`. This restriction is necessary because the EJB server (CB) tools generate enterprise bean support files that use the unqualified name only.
- Container-managed fields in entity beans must be valid for use in CORBA IDL files. Specifically, the variable names must use ISO Latin-1 characters; they must *not* begin with an underscore character (`_`), they must *not* contain the dollar character (`$`), and they must *not* be CORBA keywords. Variables that have the same name but different cases are not allowed. (For example, you cannot use the following variables in the same class: `accountId` and `AccountId`. For more information on CORBA IDL, consult a CORBA programming guide.

Also, container-managed fields in entity beans must be valid Java types, but they *cannot* be of type `ejb.javax.Handle` or an array of type `EJBObject` or `EJBHome`. Furthermore, container-managed fields of type `String` (or arrays of type `String`) *cannot* be set to null at run time because these types map to CORBA IDL type `string` or `wstring`, which are prohibited by CORBA from having null values.

- The use of underscores (`_`) in the names of user-defined interfaces and exception classes is discouraged.
- Method names in the remote interface must *not* match method names in the Component Broker Managed Object Framework (that is, methods in the `IManagedServer::IManagedObjectWithCachedDataObject`, `CosStream::Streamable`, `CosLifeCycle::LifeCycleObject`, and `CosObjectIdentity::IdentifiableObject` interfaces). For more information on the Managed Object Framework, see the Component Broker Programming Guide. In addition, do not use underscores (`_`) at the end of property or method names; this restriction prevents name collision with queryable attributes in BO interfaces that correspond to

container-managed fields.

- 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`.
- The `javax.ejb.SessionSynchronization` interface is *not* supported.
- Entity beans with BMP that use Java Database Connectivity (JDBC) to access a database cannot participate in distributed transactions because the environment does not support XA-enabled JDBC. In addition, a BMP entity bean that uses JDBC to access a DB2 database must not be run in the same server process as a CMP entity bean that uses DB2 or in the same server process as an ordinary CB BO that uses DB2. Similarly, a BMP entity bean that uses JDBC to access an Oracle database must not be run in the same server process as a CMP entity bean (or other CB BO) that uses Oracle.
- The variables of the primary key class of a BMP entity bean must be public.
- The *run-as identity* and *access control* deployment descriptor attributes are not used.
- The `remove` method inherited by an enterprise bean's remote interface (from the `javax.ejb.EJBObject` interface) does not throw the `javax.ejb.RemoveException` exception, even if the enterprise bean's corresponding `ejbRemove()` method throws this exception. This restriction is necessary because of the name conflict between the `remove` method and the CORBA `CosLifecycle::LifecycleObject::remove` method, which is inherited by all Component Broker managed objects.
- Single-threaded access to enterprise beans is enforced only if a bean's transaction attribute is set to either `TX_NOT_SUPPORTED` or `TX_BEAN_MANAGED`. For other enterprise beans, access from different transactions is serialized, but serialized access from different threads running under the same transaction is not enforced. Illegal callbacks for enterprise beans deployed with the `TX_NOT_SUPPORTED` or `TX_BEAN_MANAGED` transaction attribute result in a `java.rmi.RemoteException` exception being thrown to the EJB client.
- The session bean timeout attribute is *not* supported.
- The transaction attribute can be set only for the bean as a whole; the transaction attribute cannot be set on individual methods in a bean.
- If a stateful session bean has the `TX_BEAN_MANAGED` transaction attribute value, a method that begins a transaction must also complete that transaction (commit or roll back the transaction). In other words, a transaction cannot span multiple methods in a stateful session bean when used in the EJB server (CB) environment.
- The `TX_MANDATORY` transaction attribute value must be used in entity beans with container-managed persistence (CMP) that use HOD or ECI to access CICS or IMS applications. As a result, EJB clients that access these entity beans must do so within a client-initiated one-phase commit transaction (CB session service).
- The `TX_NOT_SUPPORTED` transaction attribute value is not supported for entity beans with CMP, because these beans must be accessed within a transaction.
- The `TX_REQUIRES_NEW` transaction attribute is *not* supported.
- The `TX_SUPPORTS` transaction attribute *can* be used in entity beans with CMP; however, EJB clients that access these beans must do so within a client-initiated transaction.
- The transaction isolation level attribute is *not* supported.
- When using the `com.ibm.ejb.cb.runtime.CBCtxFactory` context factory, any of the default initial context factories (see [Default context-to-finder associations](#)), or an application-specific initial context factory generated by the **appbind** tool (see [Application-specific contexts and the appbind tool](#)), the `javax.naming.Context.list` and `javax.naming.Context.listBindings` methods can return no more than 1000 elements in the `javax.naming.NamingEnumeration` object.

- C++ CORBA-based EJB clients are restricted to invoking methods that do *not* use parameters that are arrays or that are of the java.io.Serializable type or the java.lang.String type. This restriction effectively prohibits these EJB clients from accessing entity beans directly because primary key classes must be serializable. The String and array types in the remote or home interface are mapped to IDL value types to allow null values to be passed between a Java EJB client and an enterprise bean. CORBA C++ EJB clients cannot invoke the javax.ejb.EJBHome.remove and javax.ejb.EJBObject.getHandle methods because these methods contain Serializable parameters. EJB clients cannot be built with Microsoft Visual C++^(R).
-

Copyright [IBM Corporation 1999](#). All Rights Reserved

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

There are two basic approaches to developing and deploying enterprise beans in the EJB server (AE) environment:

- You can use one of the available integrated development environments (IDEs) such as IBM VisualAge 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. In the EJB server (AE) environment, use of VisualAge for Java is strongly recommended. 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 with EJB server \(AE\) tools](#).

Before beginning development of enterprise beans in the EJB server (AE) environment, review the list of development restrictions contained in [Restrictions in the EJB server \(AE\) environment](#).

Note: Deployment and use of enterprise beans for the EJB server (AE) environment must take place on the Microsoft Windows NT^(R) operating system, the IBM AIX^(TM) operating systems, or the Sun Microsystems Solaris^(R) operating system.

For information on developing enterprise beans in the EJB server (CB) environment, see [Tools for developing and deploying enterprise beans in the EJB server \(CB\) environment](#).

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.

To access the enterprise bean development window, click the **EJB** tab from the Workbench. All enterprise beans must be part of an EJB group that is associated with a VisualAge for Java project. You must create a project and an EJB group before creating an enterprise bean.

After you have created an EJB group, you can add beans to the EJB group. This action brings up the SmartGuide for creating enterprise beans. The SmartGuide prompts you for the information needed to generate all of the components of an enterprise bean and much of the required code.

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 class for the EJB server (CB) environment or a finder helper interface for the EJB server (AE) environment, if required.
4. Define the deployment descriptor for the bean.
5. Package the bean components in an EJB JAR file.
6. Generate the deployment code for the bean.

VisualAge for Java contains a complete WebSphere Application Server runtime 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 with EJB server (AE) tools

If you have decided to develop enterprise beans *without* an IDE, you need at minimum the following low-level tools:

- An ASCII text editor. (You can use 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 Server **jetace** 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 (AE) environment. For more information, see [Installing and configuring the software for the EJB server \(AE\)](#).
2. Set the CLASSPATH environment variable required by different components of the EJB server (AE) environment. For more information, see [Setting the CLASSPATH environment variable in the EJB server \(AE\) 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 \(AE\)](#).
5. Create a deployment descriptor file and an EJB JAR file for the enterprise bean by using the **jetace** tool. For more information, see [Creating a deployment descriptor and an EJB JAR file](#).
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. Deploy the enterprise bean by using the WebSphere Administrative Console. For more information, see the online help available with the WebSphere Administrative Console.
8. Install the enterprise beans into an EJB server (AE) and start the server by using the WebSphere Administrative Console. For more information, see the online help available with the WebSphere Administrative Console.

Installing and configuring the software for the EJB server (AE)

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 (AE):

- 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
 - 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 Advanced application server *Getting Started* document.

Setting the CLASSPATH environment variable in the EJB server (AE) 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 the listed tasks:

- Developing an enterprise bean that does *not* use another enterprise bean:
 - ejs.jar
 - ujc.jar
 - iioptools.jar
- Developing an enterprise bean that *does* use another enterprise bean:
 - ejs.jar
 - ujc.jar
 - iioptools.jar
 - *otherDeployedBean.jar* (the deployed JAR file containing the enterprise bean being used by this enterprise bean)
- Developing an EJB client:
 - ejs.jar
 - ujc.jar
 - iioptools.jar
 - servlet.jar (required by EJB clients that are servlets)
 - *otherDeployedBean.jar* (the deployed JAR file containing the enterprise bean being used by this EJB client)
- Running an EJB client:
 - ejs.jar
 - ujc.jar
 - servlet.jar (required by EJB clients that are servlets)
 - *otherDeployedBean.jar* (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, since 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 (AE)

For the EJB server (AE) 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 *findMethodNameQueryString*, 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.

If you define the `findLargeAccounts` method shown in [Figure 24](#), you must also create the `AccountBeanFinderHelper` interface shown in [Figure 7](#).

Figure 7. Code example: AccountBeanFinderHelper interface for the EJB server (AE)

```
...
public interface AccountBeanFinderHelper{
    String findLargeAccountsQueryString =
        "select * from ejb.accountbeantbl where balance > ?";
}
```

Creating a deployment descriptor and an EJB JAR file

The WebSphere Application Server **jetace** tool can be used to create an EJB JAR file for one or more enterprise beans and generate a deployment descriptor file for each enterprise bean. The resulting EJB JAR file contains each enterprise bean's class files and deployment descriptor and an EJB-compliant manifest file. The **jetace** tool is available in both the EJB server (AE) and the EJB server (CB) environments.

Note: Before using the **jetace** tool in the EJB server (AE) environment, ensure that the `JAVA_HOME` environment variable identifies the path to the SDK installation directory. For example on Windows NT, if your SDK installation directory is `C:\SDK`, set this environment variable as follows:

```
C:\> set JAVA_HOME=C:\SDK
```

Before you create an EJB JAR file for one or more enterprise beans, you must do *one* of the following:

- Place all of the components of each enterprise bean into a single directory.
- Create a standard JAR file that contains the class and interface files of each enterprise bean by using the Java Archiving tool (**jar**). The following command, when run from the root directory of the Account bean's full package name, can be used to create the file `AccountIn.jar` with a default manifest file:

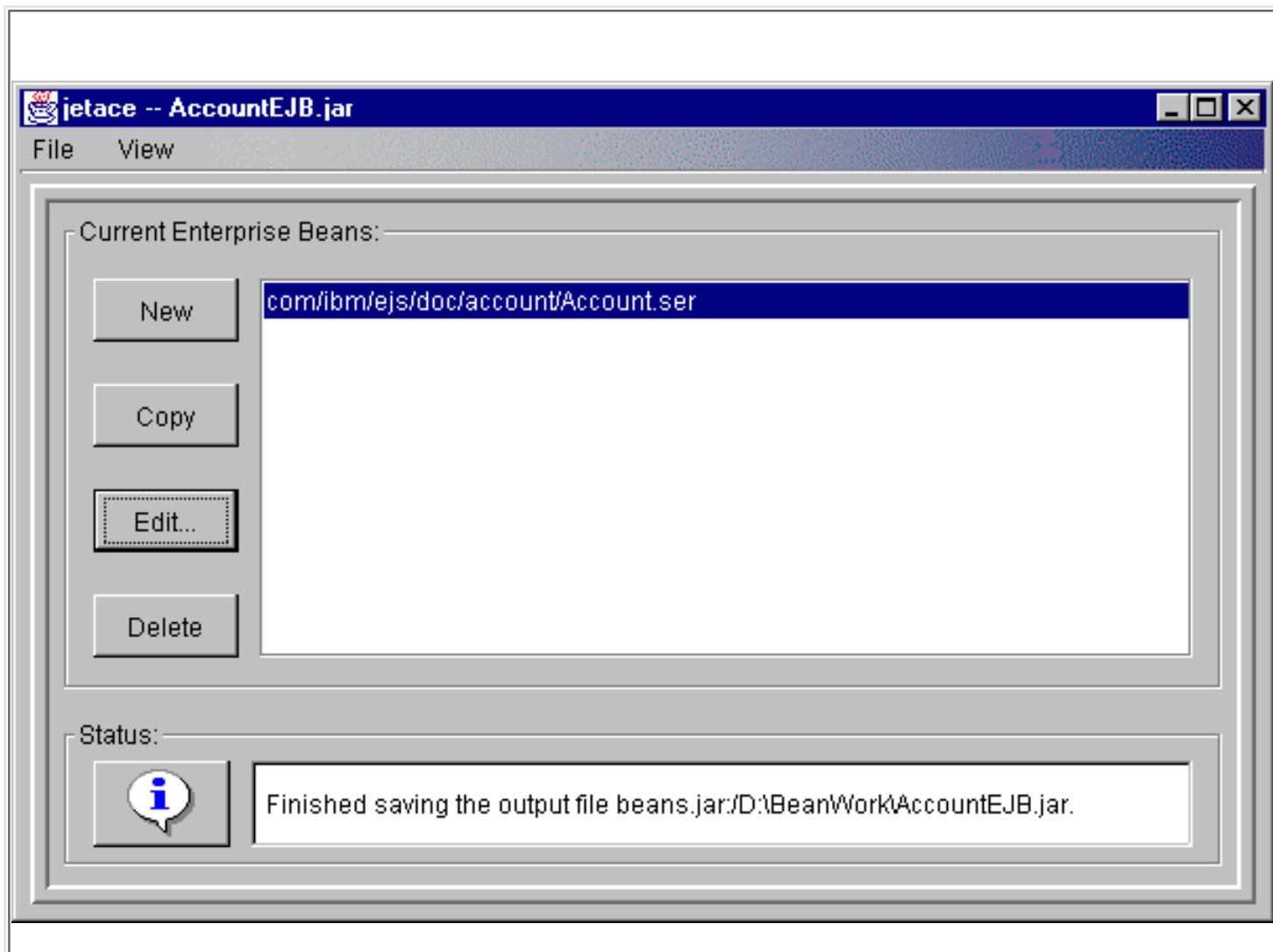
```
C:\MYBEANS> jar cfv AccountIn.jar com\ibm\ejb\doc\account\*.class
```

- Create a standard ZIP file that contains the class and interface files of each enterprise bean by using a tool like WinZip^(R).

Running the jetace tool

To run the **jetace** tool, type **jetace** on the command line. The window shown in [Figure 8](#) is displayed.

Figure 8. The initial window of jetace tool



To generate an EJB JAR file with the **jetace** tool, do the following:

1. Click the **File->Load** item, and select the JAR or ZIP file or the directory containing one or more enterprise beans. Use the **Browse** button to obtain the file or directory.

Note: To specify the current directory as the input source, type an = (equals character) in the **File Name** field of the browser window and click **Open**.

If you are creating a new EJB JAR file, click **New** and a default name for the deployment descriptor (for example, UNAMED_BEAN_1.ser) appears in the **Current Enterprise Beans** list box. (You can edit this name on any of the remaining tabbed pages of the **jetace** GUI by editing the **Deployed Name** field at the top of each tabbed page. This field is described in [Specifying the enterprise bean components and JNDI home name.](#))

If you are editing an existing EJB JAR file, the name of the deployment descriptor for each enterprise bean in the EJB JAR file is displayed in the **Current Enterprise Beans** list box, as shown in [Figure 8](#).

- If you do not want to include a listed enterprise bean in the resulting EJB JAR file, highlight that enterprise bean's deployment descriptor and click **Delete**. This action removes the deployment descriptor from the list box.
 - If you want to create a duplicate of an enterprise bean, highlight its deployment descriptor and click **Copy**. This action adds a new default deployment descriptor to the list box. Copying can be useful if you want to create a deployment descriptor for one enterprise bean that is similar to the deployment descriptor of the copied bean. You must then edit the new deployment descriptor.
2. To create a new deployment descriptor or edit an existing one, highlight the deployment descriptor and press the **Edit** button. This action causes the **Basic** page to display. On this page, set or confirm the names of the deployment descriptor file, the enterprise bean class, the home interface, and the remote interface and specify the JNDI name of the enterprise bean. For information, see [Specifying the enterprise bean components and JNDI home name.](#)
 3. Set the entity bean or session bean attributes for the enterprise bean's deployment descriptor on the **Entity** or **Session** page, respectively. For information on setting deployment descriptor attributes for entity beans, see [Setting the entity bean-specific attributes.](#) For information on setting deployment descriptor attributes for session beans, see [Setting the session bean-specific attributes.](#)
 4. Set the transaction attributes for the enterprise bean's deployment descriptor on the **Transactions** page. For information, see [Setting transaction attributes.](#)
 5. Set the security attributes for the enterprise bean's deployment descriptor on the **Security** page. For information, see [Setting security attributes.](#)
 6. Set any environment variables to be associated with the enterprise bean on the **Environment** page. For information, see [Setting environment variables for an enterprise bean.](#)
 7. Set any class dependencies to be associated with the enterprise bean on the **Dependencies** page. For information, see [Setting class dependencies for an enterprise bean.](#)
 8. After you have set the appropriate deployment descriptor attributes for each enterprise bean, click **File->Save As** to create an EJB JAR file. (If desired, a ZIP file can be created instead of a JAR file.)

The **jetace** tool can also be used to read and generate an XML version of an enterprise bean's deployment descriptor. To read an XML file, click the **File->Read XML** item. To generate an XML file from an existing enterprise bean (after saving the output EJB JAR file) click the **File->Write XML** item.

The **jetace** tool can also be run from the command line to create an EJB JAR file. The syntax of this command follows, where *xmlFile* is the name of an XML file containing the enterprise bean's deployment descriptor:

```
% jetace -f xmlFile
```

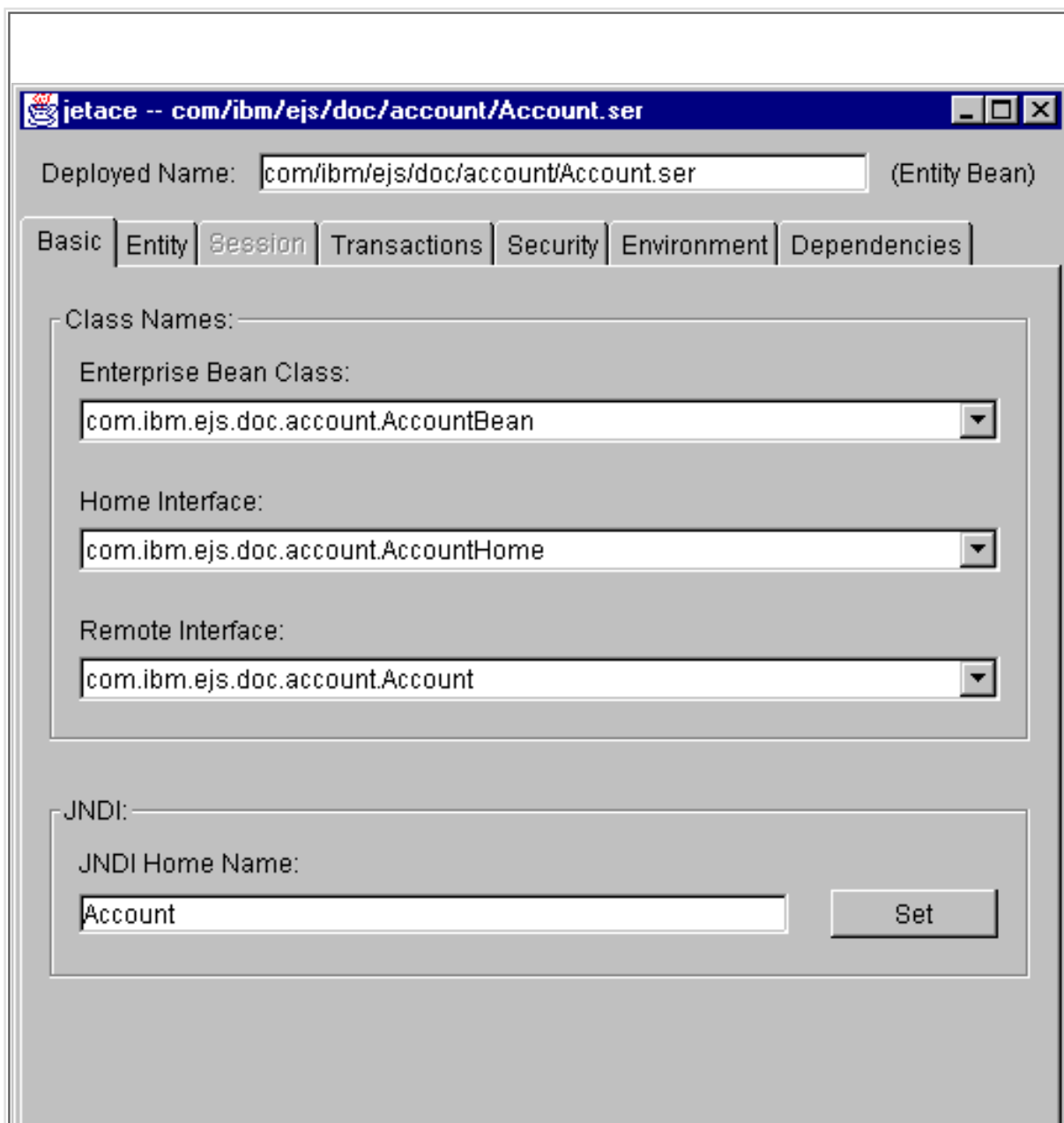
Note: In the EJB server (AE) environment, use of the XML feature provided by the **jetace** tool is not recommended.

For more information on the syntax of the XML file required for this command, see [Appendix B, Using XML in enterprise beans](#).

Specifying the enterprise bean components and JNDI home name

The **Basic** page is used to set the full pathname of the deployment descriptor file and the Java package name of the enterprise bean class, home interface, and remote interface and to set the enterprise bean's JNDI home name. To access this page, which is shown in [Figure 9](#), click the **Basic** tab.

Figure 9. The Basic page of the jetace tool



The screenshot shows a window titled "jetace -- com/ibm/ejs/doc/account/Account.ser". The "Deployed Name" field contains "com/ibm/ejs/doc/account/Account.ser" and is labeled "(Entity Bean)". Below this are several tabs: "Basic", "Entity", "Session", "Transactions", "Security", "Environment", and "Dependencies". The "Basic" tab is selected. Under "Class Names:", there are three dropdown menus: "Enterprise Bean Class" (com.ibm.ejs.doc.account.AccountBean), "Home Interface" (com.ibm.ejs.doc.account.AccountHome), and "Remote Interface" (com.ibm.ejs.doc.account.Account). Under "JNDI:", there is a "JNDI Home Name" field containing "Account" and a "Set" button.

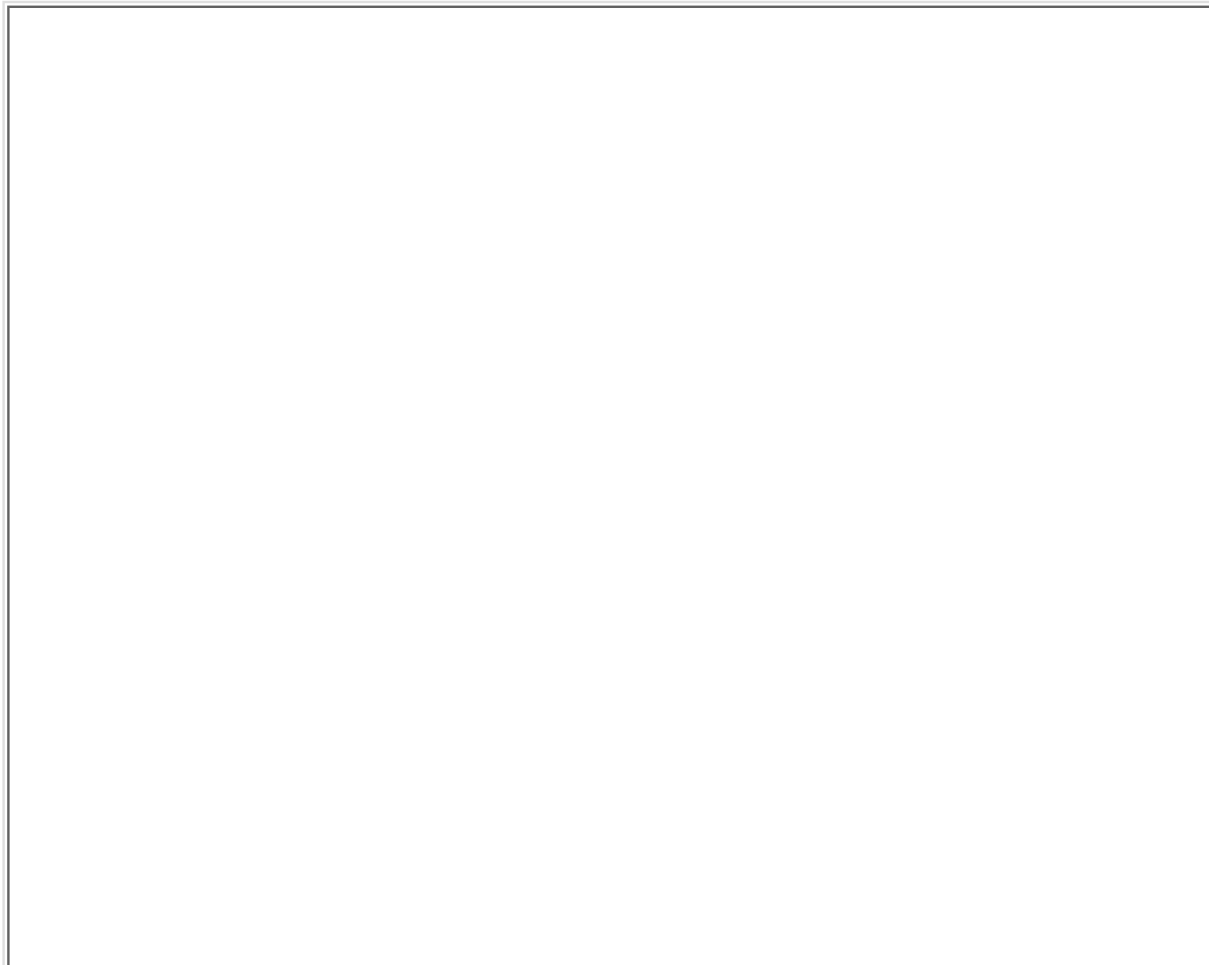
In the Basic page, you must select or confirm values for the following fields:

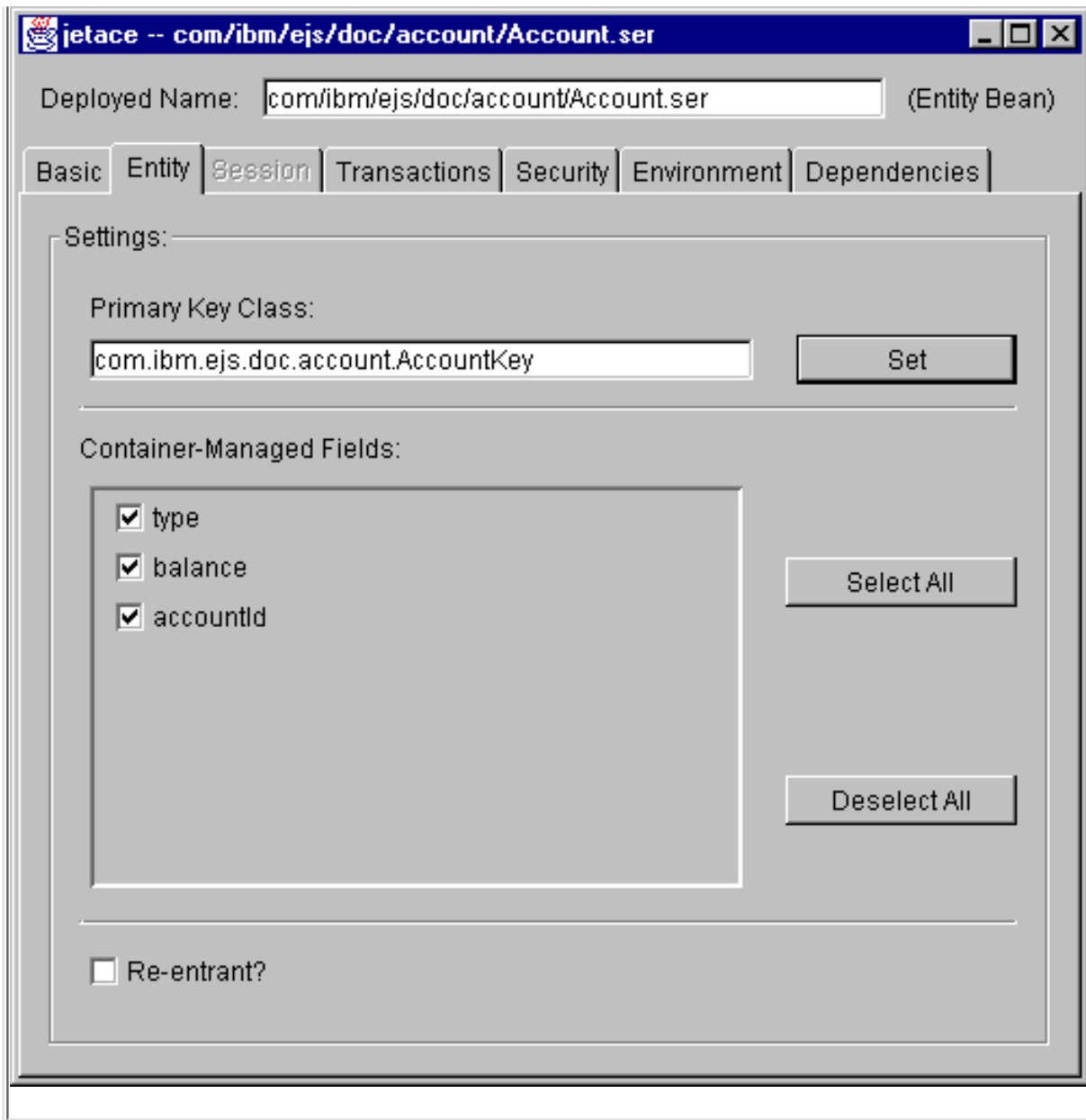
- **Deployed Name**--The pathname of the deployment descriptor file to be created. It is recommended that this directory name match the full package name of the enterprise bean class. For the Account bean, the full name is `com/ibm/ejs/doc/account/Account.ser`.
- **Enterprise Bean Class**--Specify the full package name of the bean class. For the Account bean, the full name is `com.ibm.ejs.doc.account.AccountBean`.
- **Home Interface**--Specify the full package name of the bean's home interface. For the Account bean, the full name is `com.ibm.ejs.doc.account.AccountHome`.
- **Remote Interface**--Specify the full package name of the bean's remote interface. For the Account bean, the full name is `com.ibm.ejs.doc.account.Account`.
- **JNDI Home Name**--Specify the JNDI home name of the bean's home interface. This the name under which the enterprise bean's home interface is registered and therefore is the name that must be specified when an EJB client does a lookup of the home interface. For the Account bean, the JNDI home name is `Account`.

Setting the entity bean-specific attributes

To set the deployment descriptor attributes associated specifically with an entity bean, click the **Entity** tab in the **jetace** tool to display the **Entity** page shown in [Figure 10](#). This tab is disabled if the highlighted enterprise bean in the initial **jetace** window is a session bean.

Figure 10. The Entity page of the jetace tool





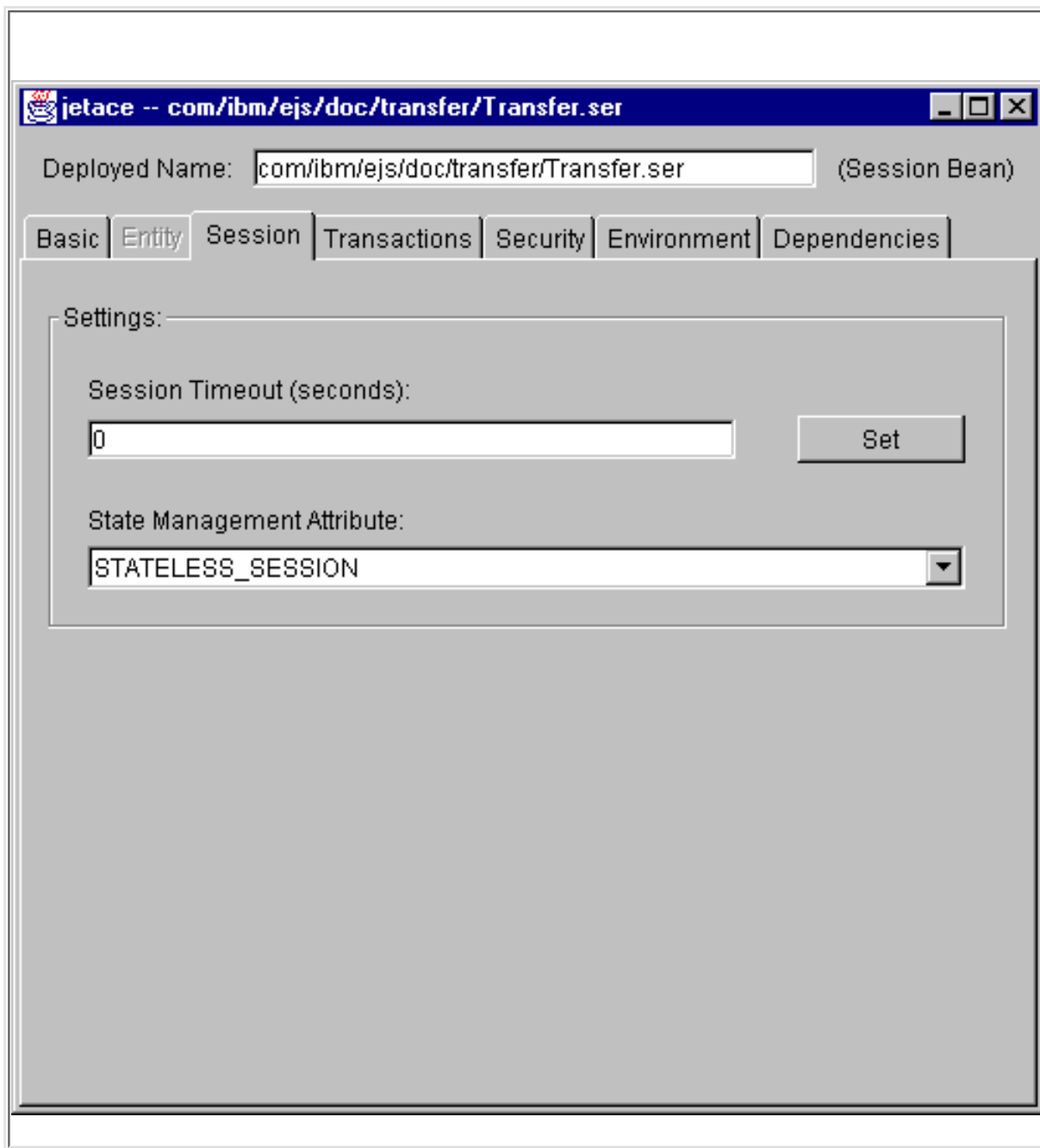
In the **Entity** page, you must select or confirm values for the following fields:

- **Primary Key Class**--Specify the full package name of the bean's primary key class. For the example Account bean, the full name is `com.ibm.ejs.doc.account.AccountKey`.
- **Container-Managed Fields**--Check the check boxes of the variables in the bean class for which the container needs to handle persistence management. This is required for entity beans with CMP only, and must *not* be done for entity beans with BMP. For the Account bean, the `type`, `balance`, and `accountId` variables are container managed, so each box is checked.
- **Re-entrant?**--Check this check box if the bean is reentrant. By default, an entity bean is not reentrant. If an instance of a non-reentrant entity bean is executing a client request in a transaction context and it receives another request using the same transaction context, the EJB container throws the `java.rmi.RemoteException` exception to the second request. Since a container cannot distinguish between a legal loopback call from another bean and an illegal concurrent call from another client or client thread, a client must take care to prevent concurrent calls to a reentrant bean. The example Account bean is *not* reentrant.

Setting the session bean-specific attributes

To set the deployment descriptor attributes associated specifically with a session bean, click the **Session** tab in the **jetace** tool to display the **Session** page shown in [Figure 11](#). This tab is disabled if the highlighted enterprise bean in the initial **jetace** window is an entity bean.

Figure 11. The Session page of the jetace tool



On the **Session** page, you must select or confirm values for the following fields:

- **Session Timeout (seconds)**--Specify the idle timeout value for this bean in seconds; a 0 (zero) indicates that idle bean instances timeout after the maximum allowable timeout period has elapsed (by default in the EJB server (AE), this timeout is 600 seconds or 10 minutes). For the Transfer bean, the value is left at 0 to indicate that the default timeout is used.

Note: In the EJB server (CB) environment, this attribute is not used.

- **State Management Attribute**--Specify whether the bean is stateless or stateful. The example Transfer bean is STATELESS_SESSION. For more information, see [Stateless versus stateful session](#)

[beans.](#)

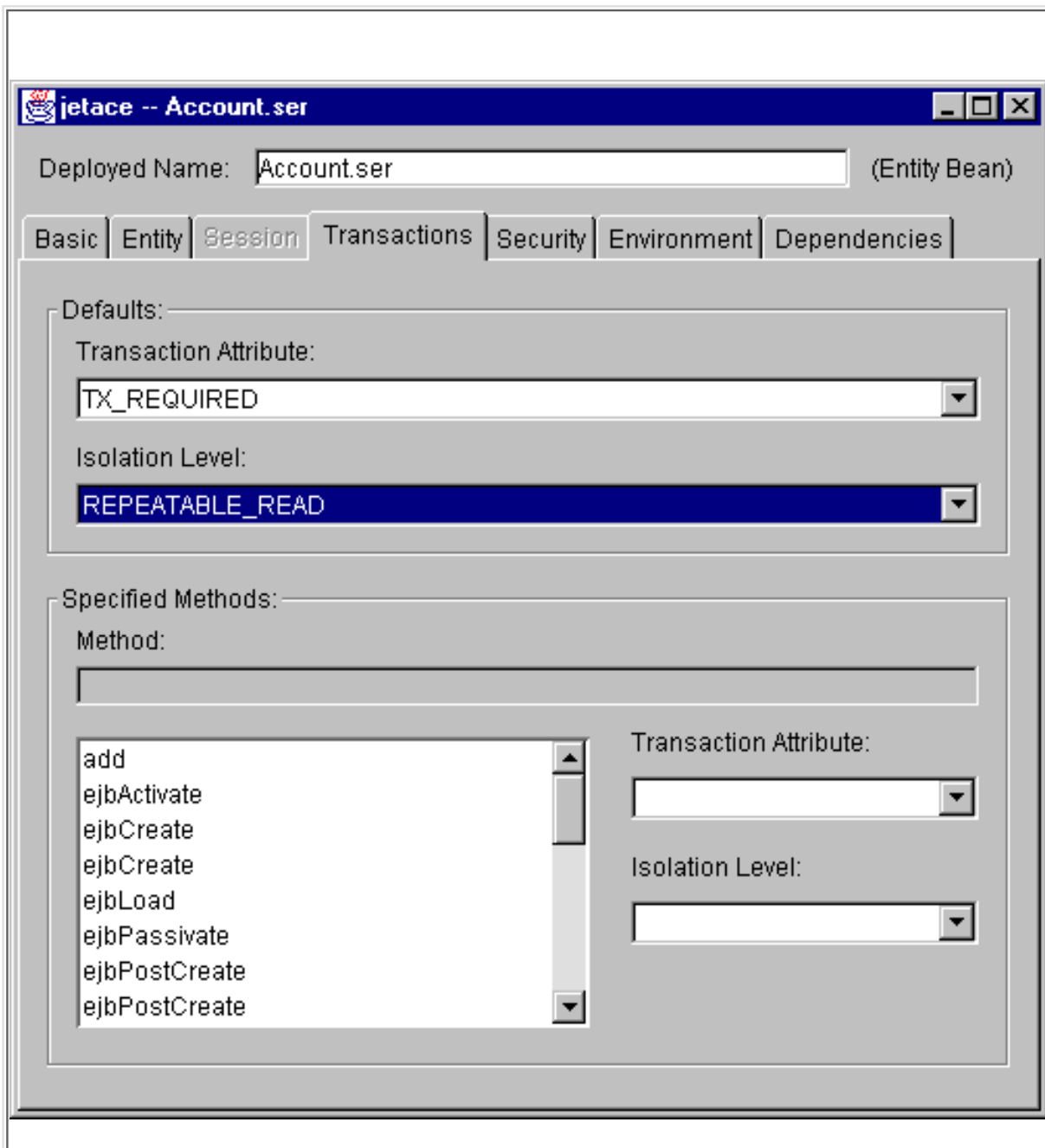
Setting transaction attributes

The **Transactions** page is used to set the transaction and transaction isolation level attributes for all of the methods in an enterprise bean and for individual methods in an enterprise bean. If an attribute is set for an individual method, that attribute overrides the default attribute value set for the enterprise bean as a whole.

Note: In the EJB server (CB), the transactional attribute can be set only for the bean as a whole; the transaction attribute cannot be set on individual methods in a bean.

To access the **Transaction** page, click the **Transactions** tab in the **jetace** tool. [Figure 12](#) shows an example of this page.

Figure 12. The Transactions page of the jetace tool



On the **Transactions** page, you must select or confirm values for the following fields in the **Defaults** group

box:

- **Transaction Attribute**--Set a value for the transaction attribute. The values for this attribute are described in [Enabling transactions and security in enterprise beans](#). For the Account bean, the value TX_MANDATORY is used because the methods in this bean must be associated with an existing transaction when invoked; as a result, the Transfer bean must use the value that begins a new transaction or passes on an existing one.
- **Isolation Level**--Set a value for the transaction isolation level attribute. The values for this attribute are described in [Enabling transactions and security in enterprise beans](#). For the Account bean, the value REPEATABLE_READ is used.

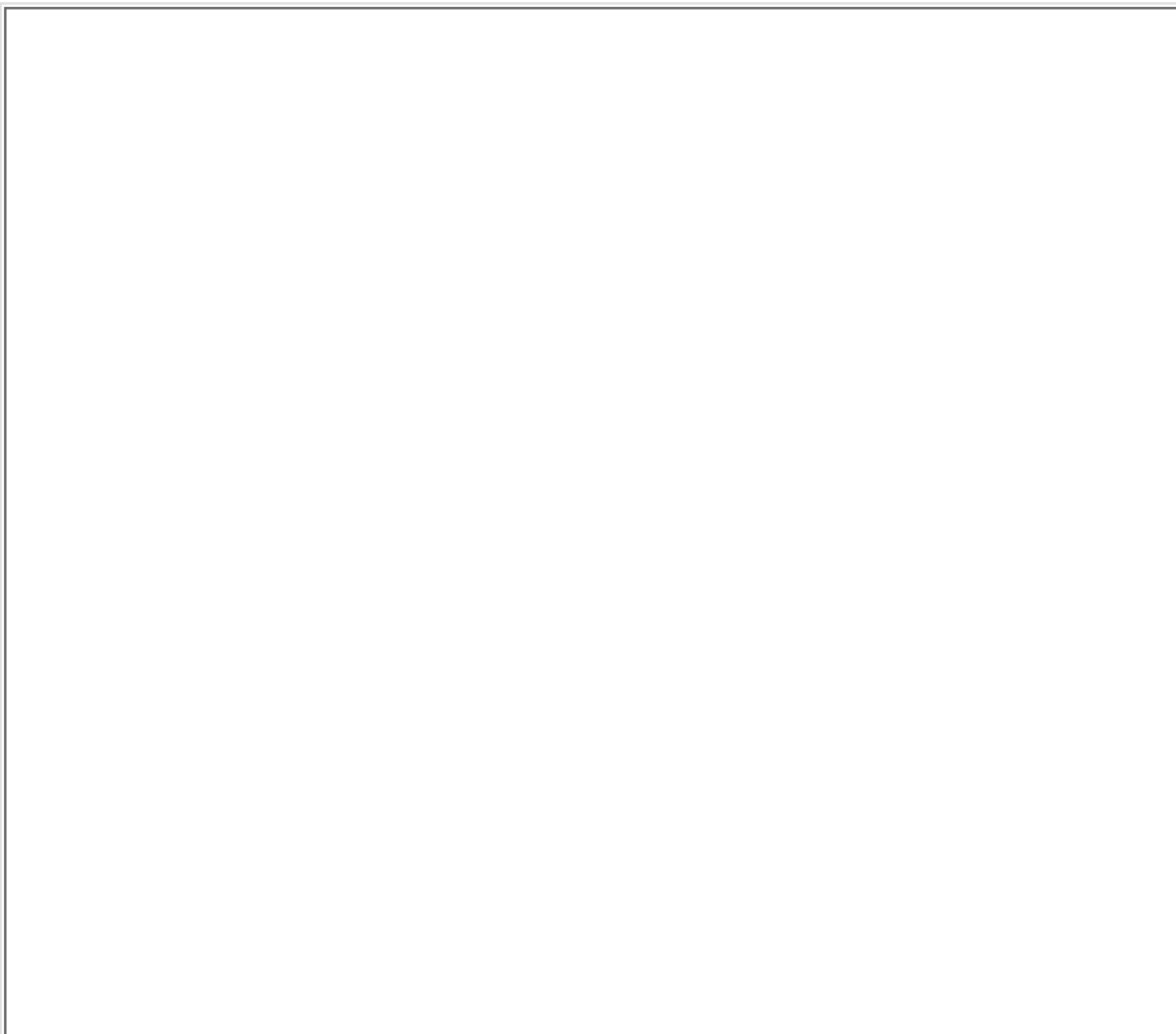
If necessary, you can also set these attributes on individual methods by highlighting the appropriate method and setting one or both of the attributes in the **Specified Methods** group box.

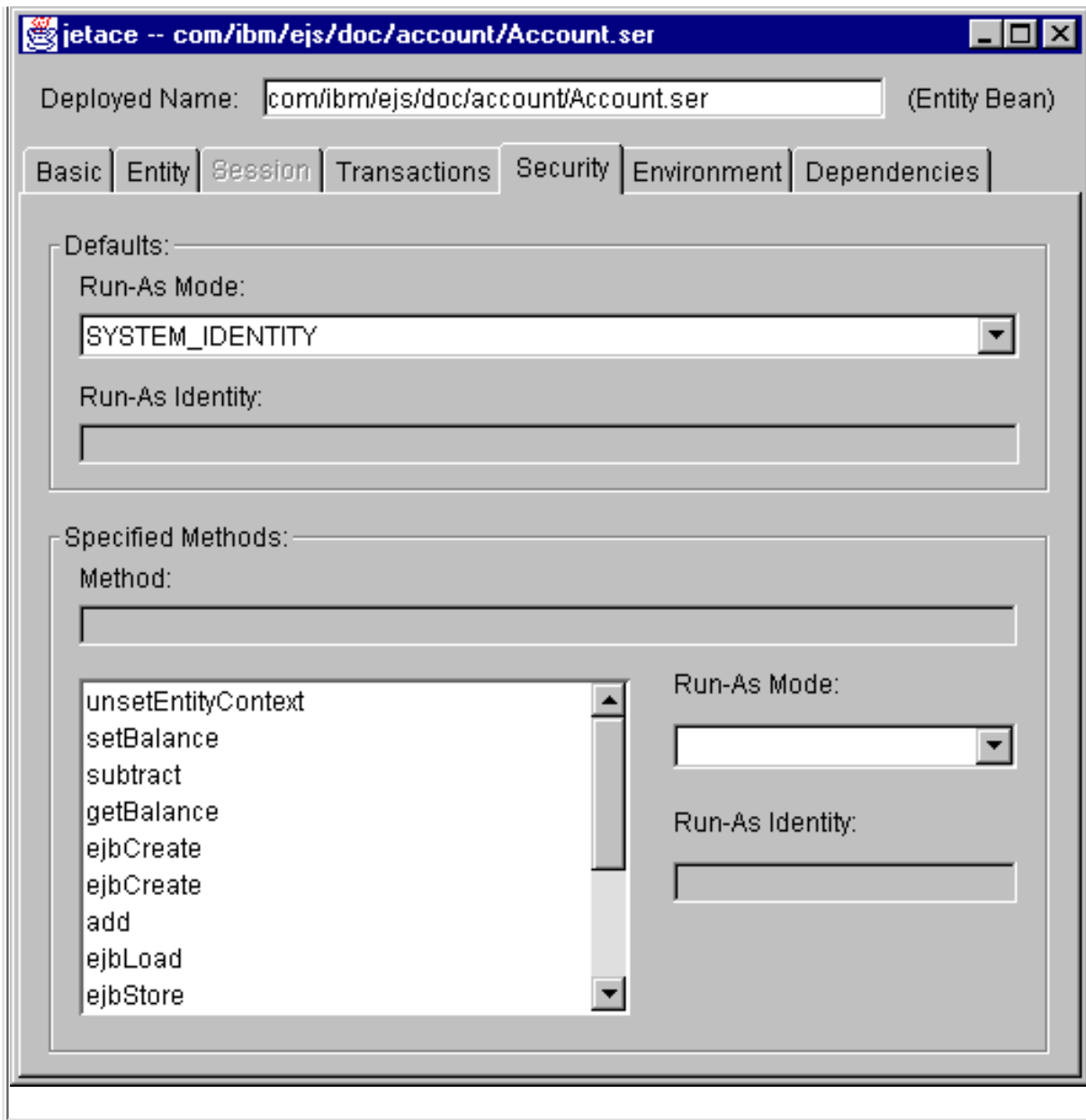
Setting security attributes

The **Security** page is used to set the security attributes for all of the methods in an enterprise bean and for individual methods in an enterprise bean. If an attribute is set for an individual method, that attribute overrides the default attribute value set for the enterprise bean as a whole.

To access the **Security** page, click the **Security** tab in the **jetace** tool. [Figure 13](#) shows an example of this page.

Figure 13. The Security page of the jetace tool





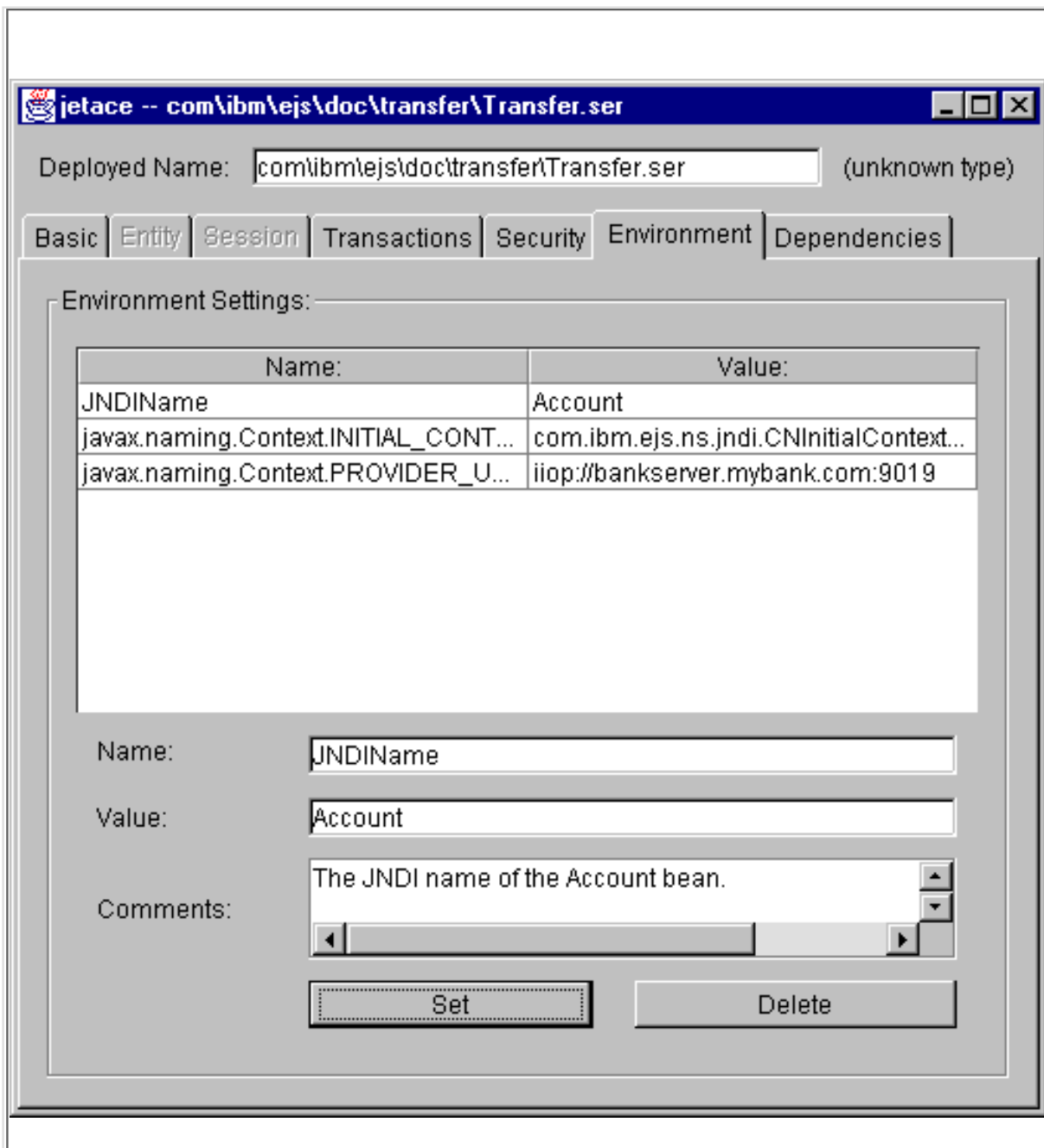
On the **Security** page, you must select or confirm values for the **Run-As Mode** field in the **Defaults** group box. This field must be set to one of the values described in [Setting the security attribute in the deployment descriptor](#). The *run-as identity* attribute is not used by WebSphere EJB servers, so you cannot set the value for the corresponding field in the **jetace** tool.

If necessary, you can also set the *run-as mode* attribute on individual methods by highlighting the appropriate method and setting the attribute in the **Specified Methods** group box.

Setting environment variables for an enterprise bean

The **Environment** page is used to associate environment variables (and their corresponding values) with an enterprise bean. To access the **Environment** page, click the **Environment** tab in the **jetace** tool. [Figure 14](#) shows an example of this page.

Figure 14. The **Environment** page of the **jetace** tool



To set an environment variable to its value, specify the environment variable name in the **Name** field and specify the environment variables value in the **Value** field. If desired, use the **Comment** field to further identify the environment variable. Press the **Set** button to set the value. To delete an environment variable, highlight the variable in the **Environment Settings** window and press the **Delete** button.

For the example Transfer bean, the following environment variables are required:

- JNDIName--The JNDI name of the Account bean, which is accessed by the Transfer bean. For more information, see [Figure 9](#).
- javax.naming.Context.INITIAL_CONTEXT_FACTORY--The name of the initial context factory used by the Transfer bean to look up the JNDI name of the Account bean
- javax.naming.Context.PROVIDER_URL--The location of the naming service used by the Transfer bean to look up the JNDI name of the Account bean.

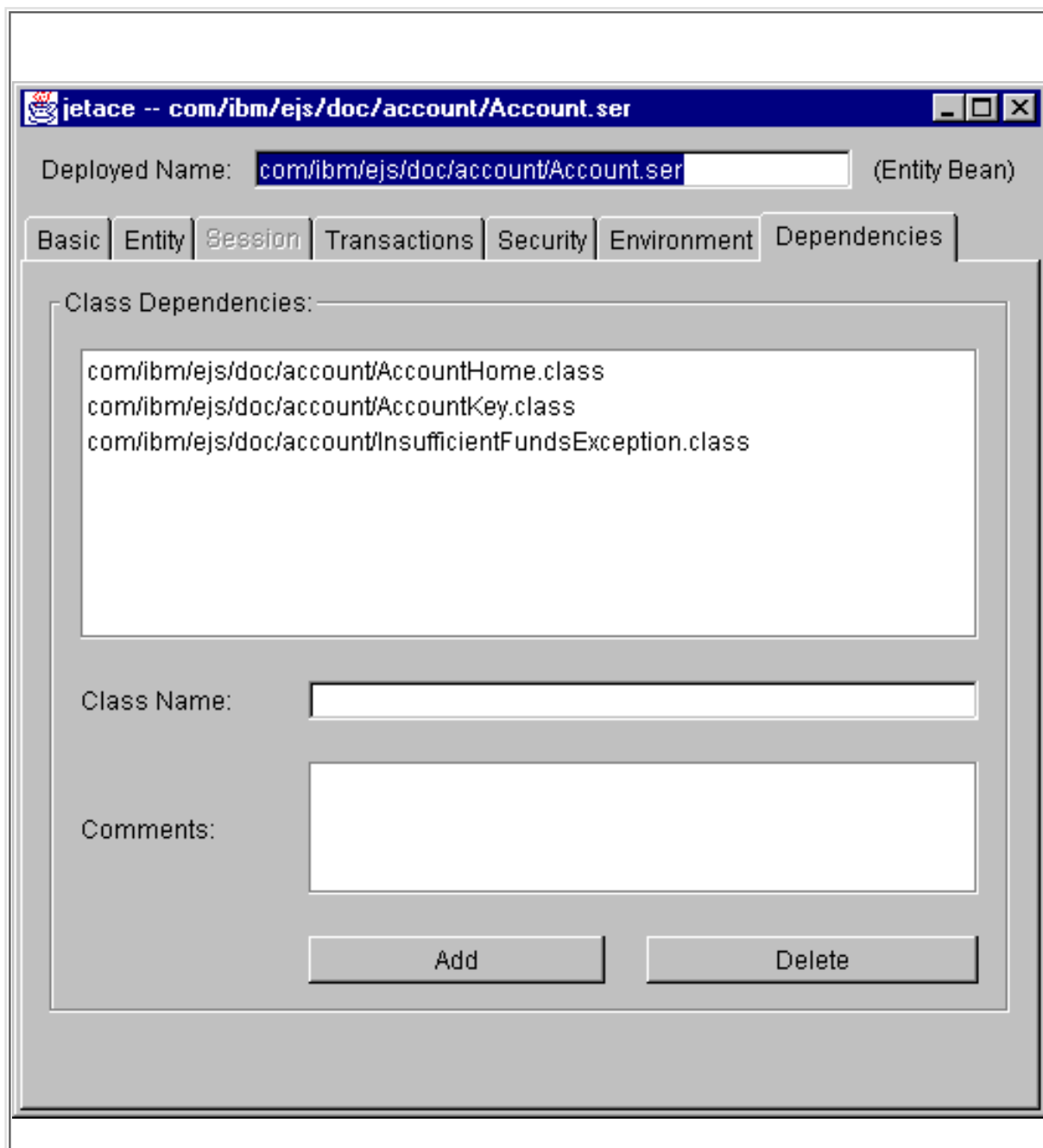
For more information on how these environment variables are used by the Transfer bean, see [Implementing](#)

[the ejbCreate methods.](#)

Setting class dependencies for an enterprise bean

The **Dependencies** page is used to specify classes on which the enterprise bean depends. To access the **Dependencies** page, click the **Dependencies** tab in the **jetace** tool. [Figure 15](#) shows an example of this page.

Figure 15. The **Dependencies** page of the **jetace** tool



Generally, the **jetace** tool discovers class dependencies automatically and sets them here. If there are other class dependencies required by an enterprise bean, you must set them here by entering the fully-qualified Java class name in the **Classname** field. If desired, use the **Comment** field to further identify the dependency. Press the **Add** button to set the value. To remove a dependency, highlight it in the **Class Dependencies** window and press the **Delete** button.

For the example Account bean, the **jetace** tool set the dependencies shown in [Figure 15](#).

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. If you are not using VisualAge for Java, it is strongly recommended that you use the WebSphere Administrative Console to automatically create database tables for CMP entity beans. The console names the database schema and table `ejb.beanNamebeantbl`, where *beanName* is the name of the enterprise bean (for example, `ejb.accountbeantbl`).

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.

Restrictions in the EJB server (AE) environment

The following restrictions apply when developing enterprise beans for the EJB server (AE) environment:

- 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.
 - 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`.
 - The value `TX_BEAN_MANAGED` is not a valid value for the *transaction* deployment descriptor attribute in entity beans. In other words, entity beans cannot manage transactions.
 - The *run-as identity* and *access control* deployment descriptor attributes are not used.
-

Appendix A. Example code provided with WebSphere Application Server

This appendix contains information on the example code provided with the WebSphere Application Server for both Advanced Edition and Enterprise Edition.

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 in the EJB server (AE) environment

[Table 2](#) provides a summary of the enterprise bean-specific examples provided with the EJB server (AE).

Table 2. Examples available with the EJB server (AE)

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.

Information about other examples in the EJB server (CB) environment

[Table 3](#) provides a summary of the enterprise bean-specific examples provided with the EJB server (CB). For more information about these examples, see the README file that accompanies each example.

Table 3. Examples available with the EJB server (CB)

Name	Bean types	EJB client types	Additional information
Hello	Stateless session	Java application.	Very simple example of a session bean.
Calculator	Stateful session	Java applet, ActiveX control	Demonstrates maintaining state information in a session bean.
Card Game	Stateful session, CMP entity	Java applet, ActiveX control	Demonstrates a session bean selecting entity beans using a variety of finder methods on the entity's home.
Travel	Stateful session, BMP entity, CMP entity	Java applet, ActiveX control	Demonstrates client-side transactions. Enterprise bean uses a PAA as a data source. One enterprise bean accesses another bean.

Appendix B. Using XML in enterprise beans

Note: In the EJB server (AE) environment, use of the XML feature described here is not recommended.

This appendix contains instructions for creating deployment descriptors for enterprise beans by using the extensible markup language (XML). This appendix does not contain general information on creating or using XML; for more information on XML, consult a commercially available book.

An XML file, which is a standard ASCII file, can be created manually or by using the graphical user interface (GUI) of the **jetace** tool. Once created, the XML file can be used to create an EJB JAR file from the command line by using the **jetace** tool. For more information, see [Creating a deployment descriptor and an EJB JAR file](#).

An XML-based deployment descriptor must contain the following major components:

- Standard header and EJB JAR tags. For more information, see [Creating the standard header and EJB JAR tags](#).
- The input file and output file tags. For more information, see [Creating the input file and output file tags](#).
- Session bean or entity bean tag, depending on the type of bean for which the deployment descriptor is being generated. An XML file can contain instructions for generating an EJB JAR file with multiple enterprise beans of all types. For more information, see [Creating the entity bean tags](#) and [Creating the session bean tags](#).
- The tags used by all enterprise beans. For more information, see [Creating tags used by all enterprise beans](#).

Creating the standard header and EJB JAR tags

Every XML-based deployment descriptor must have the standard header tag, which defines the XML version and the standalone status of the XML file. For enterprise beans, these properties must be set to the values shown in [Figure 109](#). Except for the header tag, which must be the first tag in the file, the remaining content of the XML file must be enclosed in opening and closing EJB JAR tags.

Figure 109. Code example: The standard header and EJB JAR tags

```
<?xml version='1.0' standalone='yes' ?>
<ejb-JAR>
<!-- Content of the XML file -->
...
</ejb-JAR>
```

Creating the input file and output file tags

The input file tag identifies the JAR or ZIP file or the directory containing the required components of one or more enterprise beans. The output file tag identifies the EJB JAR file to be created; by default a JAR file is created, but you can force the creation of a ZIP file by adding a .zip extension to the output file name. These components are described in [Creating an EJB JAR file](#). The input and output files for the example Account bean are shown in [Figure 110](#).

Figure 110. Code example: The input file and output file tags

```
<?xml version='1.0' standalone='yes' ?>
<ejb-JAR>
<input-file>AccountIn.jar</input-file>
<output-file>Account.jar</output-file>
...
</ejb-JAR>
```


Creating the entity bean tags

If you are creating a deployment descriptor for an entity bean, you must use an entity bean tag. The entity bean open tag must contain a `dname` attribute, which must be set to the fully qualified name of the deployment descriptor associated with the entity bean.

Between the open and close entity bean tags, you must create the following entity bean-specific attribute tags:

- `<primary-key>` -- Identifies the fully qualified name of the primary key class for this entity bean.
- `<re-entrant>` -- Specifies whether the entity bean is re-entrant. This tag must contain a value attribute, which must be set to either `true` (re-entrant) or `false` (not re-entrant).
- `<container-managed>` -- Identifies the persistent variables in a CMP entity bean that are container managed. You must use a separate tag for each persistent variable.

In addition to the entity bean-specific tags, you must create the tags required by all enterprise beans described in [Creating tags used by all enterprise beans](#).

[Figure 111](#) shows the entity bean-specific tags for the example Account bean.

Figure 111. Code example: The entity bean-specific tags

```
<?xml version='1.0' standalone='yes' ?>
<ejb-JAR>
<input-file>AccountIn.jar</input-file>
<output-file>Account.jar</output-file>
...
<entity-bean dname="com/ibm/ejs/doc/account/Account.ser">
<primary-key>com.ibm.ejs.doc.account.AccountKey</primary-key>
<re-entrant value=false/>
<container-managed>accountId</container-managed>
<container-managed>type</container-managed>
<container-managed>balance</container-managed>
<!--Other tags used by all enterprise beans--!>
...
</entity-bean>
...
</ejb-JAR>
```

Creating the session bean tags

If you are creating a deployment descriptor for a session bean, you must use a session bean tag. The session bean open tag must contain a `dname` attribute, which must be set to the fully qualified name of the deployment descriptor associated with the session bean. Between the open and close session bean tags, you must also create the following session bean attribute tags:

- `<session-timeout>` -- Defines the idle timeout in seconds associated with the session bean.
- `<state-management>` -- Identifies the type of session bean: `STATELESS_SESSION` or `STATEFUL_SESSION`.

In addition to the session bean-specific tags, you must create the tags required by all enterprise beans described in [Creating tags used by all enterprise beans](#).

[Figure 112](#) shows the session bean tags for the example Transfer bean.

Figure 112. Code example: The session bean-specific tags

```

<?xml version='1.0' standalone='yes' ?>
<ejb-JAR>
<input-file>TransferIn.jar</input-file>
<output-file>Transfer.jar</output-file>
...
<session-bean dname="com/ibm/ejs/doc/transfer/Transfer.ser">
<session-timeout>0<\session-timeout>
<state-management>STATELESS_SESSION<\state-management>
<!--Other tags used by all enterprise beans--!>
...
</session-bean>
...
</ejb-JAR>

```

Creating tags used by all enterprise beans

The following tags are used by all types of enterprise beans. These tags must be placed between the appropriate set of opening and closing session or entity bean tags in addition to the tags that are specific to those types of beans.

- `<remote-interface>` -- Identifies the fully qualified name of the enterprise bean's remote interface.
- `<enterprise-bean>` -- Identifies the fully qualified name of the enterprise bean's bean class.
- `<JNDI-name>` -- Identifies the JNDI home name of the enterprise bean.
- `<transaction-attr>` -- Defines the transaction attribute for the entire enterprise bean. This attribute can also be set for an individual bean method. The valid values are `TX_MANDATORY`, `TX_NOT_SUPPORTED`, `TX_REQUIRES_NEW`, `TX_REQUIRED`, `TX_SUPPORTS`, and `TX_BEAN_MANAGED`. For more information on the meaning of and restrictions on these values, see [Setting the transaction attribute](#).
- `<isolation-level>` -- Defines the transactional isolation level attribute for the entire enterprise bean. This attribute can also be set for an individual bean method. The valid values, which must be set by using a value attribute within the open tag, are `SERIALIZABLE`, `REPEATABLE_READ`, `READ_COMMITTED`, and `READ_UNCOMMITTED`. For more information on the meaning of and restrictions on these values, see [Setting the transaction isolation level attribute](#).
- `<run-as-mode>` -- Defines the run-as mode attribute for the entire enterprise bean. This attribute can also be set for an individual bean method. The valid values, which must be set by using a value attribute within the open tag, are `CLIENT_IDENTITY`, `SYSTEM_IDENTITY`, and `SPECIFIED_IDENTITY`. For more information on the meaning of these values, see [Setting the security attribute in the deployment descriptor](#).
- `<run-as-id>` -- Defines the run-as identity attribute for the entire enterprise bean. This attribute can also be set for an individual bean method. This attribute is not used with the EJB server environments contained in WebSphere Application Server.
- `<method-control>` -- Identifies individual bean methods with transaction or security attributes that are different from the attribute values for the entire bean.
- `<dependency>` -- Identifies the fully qualified names of classes on which this enterprise bean is dependent.
- `<env-setting>` -- Identifies environment variables (and their values) required by the enterprise bean. The environment variable name is specified with a name attribute, while the environment variable value is placed between the open and close tags.

[Figure 113](#) shows the enterprise bean tags for the example Transfer bean. A similar set is required by the Account bean.

Figure 113. Code example: The tags used for all enterprise beans

```

<?xml version='1.0' standalone='yes' ?>
<ejb-JAR>
<input-file>TransferIn.jar</input-file>
<output-file>Transfer.jar</output-file>
...
<session-bean dname="com/ibm/ejs/doc/transfer/Transfer.ser">
<!--Session bean-specific tags --!>
...
<remote-interface>com.ibm.ejs.doc.transfer.Transfer</remote-interface>
<enterprise-bean>com.ibm.ejs.doc.transfer.TransferBean</enterprise-bean>
<JNDI-name>Transfer </JNDI-name>
<transaction-attr value="TX_REQUIRED"/>
<isolation-level value="SERIALIZABLE"/>
<run-as-mode value="CLIENT_IDENTITY"/>
<dependency>com/ibm/ejs/doc/account/InsufficientFundsException.class</dependency>
...
<env-setting name="ACCOUNT_NAME">Account</env-setting>
...
</session-bean>
...
</ejb-JAR>

```

If you want to override the enterprise bean-wide transaction or security attribute for particular method in that bean, you must use the `<method-control>` tag. Between the open and close tags, you must identify the method with the `<method-name>` tag and the method's parameter types by using the `<parameter>` tag. In addition, the following tags can be used to identify those attribute values that are different in the method from the enterprise bean as a whole: `<transaction-attr>`, `<isolation-level>`, `<run-as-mode>`, and `<run-as-id>`.

For example, the XML shown in [Figure 114](#) is required to override the transaction attribute of the Transfer bean (TX_REQUIRED) in the `getBalance` method to TX_SUPPORTED. Because only the transaction attribute is overridden, the method automatically inherits the values of the `<isolation-level>` and `<run-as-mode>` tags from the Transfer bean.

Figure 114. Code example: Method-specific tags

```

<?xml version='1.0' standalone='yes' ?>
<ejb-JAR>
<input-file>TransferIn.jar</input-file>
<output-file>Transfer.jar</output-file>
...
<session-bean dname="com/ibm/ejs/doc/transfer/Transfer.ser">
<!--Session bean-specific tags --!>
...
<transaction-attr value="TX_REQUIRED"/>
<isolation-level value="SERIALIZABLE"/>
<run-as-mode value="CLIENT_IDENTITY"/>
...
<method-control>
<method-name>getBalance</method-name>
<parameter>long</parameter>
<transaction-attr value="TX_SUPPORTED"/>
</method-control>
</session-bean>
...
</ejb-JAR>

```


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

Note: This extension is supported only in the EJB server (AE) environment.

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.

Copyright [IBM Corporation 1999](#). All Rights Reserved

0.9: What are servlets?



Similar to the way applets run on a browser and extend the browser's capabilities, HTTP *servlets* run on a Java-enabled Web server and extend the Web server's capabilities.

For example, servlets can support dynamic Web page content, provide database access, serve multiple clients at one time, and filter data by MIME type.

Servlets are Java programs that use the Java Servlet Application Programming Interface (API). They exist as *ServletName.class* files, or could be included in a JAR (Java ARchive) file.

For the purposes of IBM WebSphere Application Server, discussions of servlets focus on HTTP servlets, which serve Web-based clients. Non-HTTP servlets are possible, but are not central to solutions providing Web-based e-business transactions.

Related information...

- [0.7: What are servlet engines?](#)

0.9.1: What are .servlet configuration files?



This servlet configuration method involves creating an XML servlet configuration file (which is an XML document named *servlet_instance_name*.servlet) that contains:

- The name of the servlet class file
- A description of the servlet
- The servlet initialization parameters
- A page list that contains the URIs (universal resource identifiers) of the JavaServer Pages (JSP) files that the servlet can call. The page list can include a default page, an error page, and one or more target pages that are loaded if their name appears in the HTTP request.

IBM WebSphere Application Server allows the administrator to store an XML servlet configuration file (*servlet_instance_name*.servlet) in the classpath of an application containing a servlet. When the application server hosting the application receives a request for a servlet instance, it obtains the servlet configuration information from the .servlet file.

Related information...

- [0.9: What are servlets?](#)

0.9.2: What are page lists?



Some servlets (or JSP files) process data related to a user request, then forward the user request to one of several Web pages, based on the outcome of processing the data. For each outcome, the "page list" part of the servlet configuration specifies the location to which to forward the request.

As a simple example, consider a logon servlet that:

1. Prompts for and reads a user ID and password
2. Submits these for authentication by a user registry or directory service
3. Waits to find out whether the directory service approves or denies the user
4. Based on the outcome, forwards the user either to the page that is the entry point to the requested resource, or to a "logon denied" page

A page list allows developers to avoid hard coding the locations (URIs or URLs) of servlets and JSP files. In the above example, the developer would not need to hard code URLs specifying the two possible pages to which the user could be directed.

By avoiding hard coding, a developer does not need to recompile a servlet in order to change the URLs to which the servlet refers users.

Instead, the administrator tending the servlet can simply change the page list in a servlet configuration file, which is separate from the servlet class file and is named *servlet_instance_name*.servlet. The list can contain a default page, error page, and other JavaServer Pages that are called, depending on the HTTP request. Whenever the .servlet file changes, the application server automatically reloads the corresponding servlet instance.

Servlets do not automatically have configuration files -- implementing a page list requires some effort. With WebSphere Application Server, you can either manually create a servlet configuration file (.servlet file), or use one generated by IBM WebSphere Studio. In addition, the servlet developer must ensure that the servlet supports page list functionality.

Related information...

- [0.9: What are servlets?](#)

4.2.1: Developing servlets



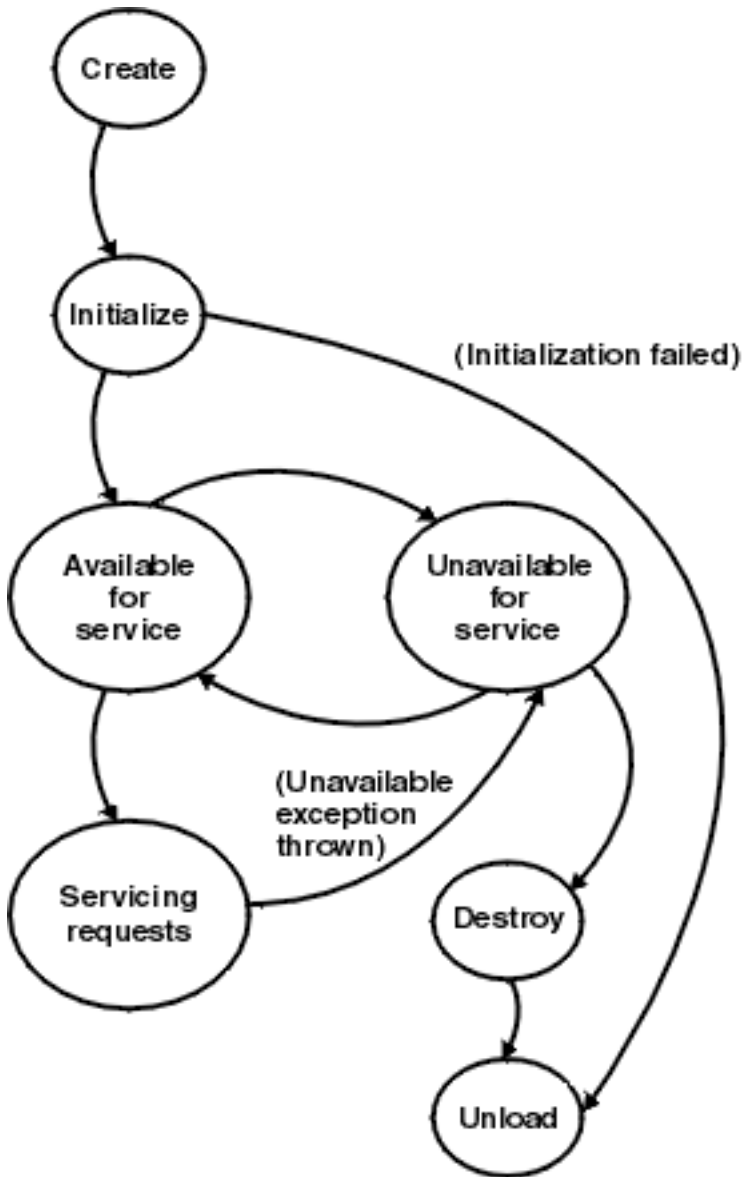
Review the servlet lifecycle to become familiar with servlets.

Section 4.2.1.2 outlines the features of the supported Java Servlet API and the IBM extensions to it. See section 4.2.1.3 when you are ready to delve into programming strategies and details.

Related information...

- [0.9: What are servlets?](#)
- [4.2.1.1: Servlet lifecycle](#)
- [4.2.1.2: Servlet support and environment in WebSphere](#)
- [4.2.1.3: Servlet content, examples, and samples](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2: Building Web applications](#)

4.2.1.1: Servlet lifecycle



Instantiation and initialization

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

If the initialization is successful, the servlet is available for service. If the initialization fails, the servlet engine 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 servlet engine creates a request object and a response object. The servlet engine invokes the servlet service method, passing the request and response objects.

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

Termination

The servlet engine 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 servlet engine 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 servlet engine loads the servlet. For the Application Server Version 3, you can configure the servlet to be loaded when the application starts or when a client first accesses 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 servlet engine stops the servlet. Typically, servlets are stopped as part of the process of stopping the application.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.1: Developing servlets](#)

4.2.1.2: Servlet support and environment in WebSphere



IBM WebSphere Application Server Version 3.5 supports the Java Servlet API 2.1 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.

Article 4.2.1.2.3 describes some complimentary servlets included with the product. Add them to Web applications for extended functionality. You can use the WebSphere servlets as they are, or use them as a basis for creating customized versions.

IBM WebSphere Application Server Version 3.5.2 supports the Java Servlet API 2.2 from Sun Microsystems. See [article 4.2.1.2.1a](#) for a description of the Servlet API 2.2 specification.

Related information...

- [4.2.1.2.1: Features of Java Servlet API 2.1](#)
- [4.2.1.2.1a: Features of Java Servlet API 2.2](#)
- [4.2.1.2.2: IBM extensions to the Servlet API](#)
- [4.2.1.2.3: Using the WebSphere servlets for a head start](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.1: Developing servlets](#)

4.2.1.2.1: Features of Java Servlet API 2.1



Some highlights of the Java Servlet API 2.1 are:

- A request dispatcher wrapper for each resource (servlet)

A request dispatcher is a wrapper for resources that can process HTTP requests (such as servlets and JSPs) and files related to those resources (such as static HTML and GIFs). The servlet engine generates a single request dispatcher for each servlet or JSP when it is instantiated. The request dispatcher receives client requests and dispatches the request to the resource.

- A servlet context per application

For the Java Servlet API 2.0, the servlet engine generated a single servlet context that was shared by all servlets. The Servlet API 2.1 provides a single servlet context per application, which facilitates partitioning applications. As explained in the description of the application programming model, applications on the same virtual host can access each other's servlet context.

- Deprecated HTTP session context

The Servlet API 2.0 `HttpSessionContext` interface grouped all of the sessions for a Web server into a single session context. Using the session context interface methods, a servlet could get a list of the session IDs for the session context and get the session associated with an ID. As a security safeguard, this interface has been deprecated in the Servlet API 2.1. The interface methods have been redefined to return null.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.1.2: Servlet support and environment in WebSphere](#)

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 2.1. 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 *AS_install_root\web\apidocs* directory.

The Application Server API packages and classes are:

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

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.db` package

Includes classes to simplify access to relational databases and provide enhanced access functions (such as result caching, update through the cache, and query parameter support).

- `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 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 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.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.1.2: Servlet support and environment in WebSphere](#)


4.2.1.2.3: Using the WebSphere servlets for a head start



IBM Application Server Version 3.5 provides internal (built-in) WebSphere servlets that you can add to your Web applications to enable optional functions.

The tables below describe each WebSphere servlet and how to use the Java console to add it to a Web application. To determine whether a WebSphere servlet currently belongs to a Web application, check the Web application configuration for the presence of the servlet by its administrative name.

Invoke servlets by class name

Objective	Invoke servlets by class or code names (such as MyServletClass)
Servlet administrative name	invoker
Servlet code	com.ibm.servlet.engine.webapp.Invoker
How to add to Web application	When using the Console -> Task -> Configure a Web application wizard , specify to serve servlets by classname. For an existing Web application, use the Console -> Tasks -> Add a servlet wizard.
More information	 Using the Invoker servlet is considered a security exposure that can be avoided by performing certain administrative tasks. See the Related information for details.

Serve files without specifically configuring them

Objective	Serve HTML, servlets, or other files in the Web application document root without extra configuration steps. For HTML files, you will not need to add a pass rule to the Web server. For servlets, you will not need to explicitly configure the servlets in the WebSphere administrative domain.
Servlet administrative name	file
Servlet code	com.ibm.servlet.engine.webapp.SimpleFileServlet
How to add to Web application	When using the Console -> Task -> Configure a Web application wizard , specify to enable the file servlet. For an existing Web application, use the Console -> Tasks -> Add a servlet wizard.

More information	This servlet handles files in the application document root whose URLs are not covered by the configured servlet URLs
------------------	---

Enable Web applications to serve JSP .91 or 1.0 files

Objective	Enable the JSP 0.91 or 1.0 page compiler to allow Web application to handle JSP files
Servlet administrative name	See section 4.2.1.2
Servlet code	See section 4.2.1.2
How to add to Web application	<p>When using the Console -> Task -> Configure a Web application wizard, specify a JSP level for the Web application.</p> <p>For an existing Web application, use the Console -> Tasks -> Add a JSP enabler wizard.</p>
More information	<p>Adding a JSP processor to an application is required if the Web application contains JSP files.</p> <ul style="list-style-type: none"> ● 4.2.1.2: JSP processors ● 6.6.10: Administering JSP files

Enable an error page without having to write one

Objective	Enable error reporting through an error page, without writing your own error page
Servlet administrative name	ErrorReporter
Servlet code	com.ibm.servlet.engine.webapp.DefaultErrorReporter
How to add to Web application	Configure the Web application , then add the ErrorReporter servlet by using the Console -> Tasks -> Add a servlet wizard.
More information	4.2.1.3.5: Enhancing servlet error reporting

Enable servlet chaining

Objective	Enable a servlet chain, in which servlets forward output and responses to other servlets for processing
Servlet administrative name	Chainer
Servlet code	com.ibm.websphere.servlet.filter.ChainerServlet
How to add to Web application	Configure the Web application , then add the Chainer servlet by using the Console -> Tasks -> Add a servlet wizard.

More information

- [4.2.1.3.4: Filtering and chaining servlets](#)

Related information...

- [4.2.1.2.3.1: Avoiding the security risks of invoking servlets by class name](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.1.2: Servlet support and environment in WebSphere](#)

4.2.1.2.3.1: Avoiding the security risks of invoking servlets by class name

Anyone enabling the Invoker servlet to serve servlets by their class names 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.

To protect each servlet, the administrator needs to:

1. Configure a Web resource based on the servlet class name, such as:
`/servlet/SnoopServlet`
for `SnoopServlet.class`
2. Add the Web resource to the Web Path list of the Invoker servlet in the Web application to which the servlet belongs.
3. Use the Configure Resource Security wizard in the Java administrative console to secure the Web resource.

Also, the administrator needs to secure the Invoker servlet itself.

Details

WebSphere security is based on defining, and then securing, URIs (known as Web resources) for servlets. This allows an administrator to apply different security levels to different paths for accessing the same servlet. Also, Web resources are logical designations that are not guaranteed to match servlet class names. For these reasons, actual class names are irrelevant to WebSphere security unless you explicitly specify that you want to protect the path for invoking a servlet by its class name.

When a Web application allows users to invoke servlets by class name, the administrator is able to drop servlets into a Web application without having to explicitly define them in WebSphere systems administration.

Suppose that the WebSphere administrator drops in a servlet class to be invoked by its class name. Even if a servlet corresponding to the same class name is defined and protected, users will be able to invoke the servlet by class name without any security checks. (The exception is if the administrator has created a Web resource corresponding to the servlet class name, as described in the above steps).

Undefined servlets remain unprotected unless steps are taken to assign secure Web resources to them based on their class names.

Related information...

- [5.1.3.1.2: Security considerations with Web applications](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.1.2.3: Using the WebSphere servlets for a head start](#)

4.2.1.3: Servlet content, examples, and samples



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

Related information...

- [4.2.1.3.1: Creating HTTP servlets](#)
- [4.2.1.3.2: Having servlets communicate](#)
- [4.2.1.3.3: Using page lists to avoid hard coding URLs](#)
- [4.2.1.3.4: Filtering and chaining servlets](#)
- [4.2.1.3.5: Enhancing servlet error reporting](#)
- [4.2.1.3.6: Invoking servlets by class name](#)
- [4.2.1.3.7: Serve files without specifically configuring them](#)
- [4.2.1.3.8: Obtaining the Web application classpath from within a servlet](#)
- [IBM WebSphere Samples Gallery](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.1: Developing servlets](#)

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

- `getParameterNames()`
 - `getParameter()`
 - `getParameterValues()`
4. Generate the HTTP response.

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

Related information...

- [4.2.1.3.1.1: Overriding HttpServlet methods](#)
- [The Java Tutorial from Sun Microsystems](#) (includes servlet threading)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.1.3: Servlet content, examples, and samples](#)

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), from the `<SERVLET>` tag in a JSP page, or from another servlet.

Methods to override

The `HttpServlet` class contains the `init`, `destroy()`, and `service` methods. The `init` and `destroy()` methods are inherited. Override the class methods you need.

- **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.

The Java Servlet API 2.1 provides a new `init` method: `init()`, the no argument `init` method that is inherited from the superclass `GenericServlet`. The `GenericServlet` also implements the `ServletConfig` object. The benefit is that when you use the no-argument `init` method in your servlet, you do not need to call `super.init(config)`. The reason is that servlet engines that implement the Servlet API 2.1 call the servlet's `init(ServletConfig config)` method behind the scenes. In turn, the `GenericServlet` calls the servlet's `init()` method.

If a servlet exception is thrown inside the `init` method, the servlet engine will unload the servlet. The `init` method is guaranteed to complete before the `service` method is called.

- **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 `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. A servlet should override the doXXX method for the HTTP methods that the servlet supports. Because the HttpServlet.service method checks the request method and calls the appropriate handler method, it is not necessary to override the service method itself. Only override the appropriate doXXX method.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.1.3.1: Creating HTTP servlets](#)

4.2.1.3.2: Having servlets communicate



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.putValue` and `getValue` 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 the Servlet API 2.1, an `HttpSession` object is only accessible to the Web applications and servlets that are a part of that session. In the Servlet API 2.1, a servlet cannot determine the ID of another session and request its `SessionContext`, because the `HttpSessionContext` and related methods are deprecated (returns null).

For application-scoped data, use the `RequestDispatcher`'s `forward` and `include` methods to share data among applications that are associated with the same servlet host (virtual host). 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 obtain the response object from its response parameter. The target servlet generates the response object and returns it to the client.

The `include` method enables a receiving servlet to include another servlet's response object in its response. The included servlet cannot set response headers. The receiving servlet only has access to the included servlet's `ServletOutputStream` or `PrintWriter`. 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 supported for JSPs developed.

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" );
```

Related information...

- [4.2.1.3.1.1: Forwarding and including data \(request and response\)](#)
- [4.2.1.3.1.2: Example: Servlet communication by forwarding](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.1.3: Servlet content, examples, and samples](#)

4.2.1.3.2.1: Forwarding and including data (request and response)



When the servlet engine calls the service method of an HTTP servlet, it passes two objects as parameters:

- `HttpServletRequest` (the Request object)
- `HttpServletResponse` (the Response object)

The servlet communicates with the server and ultimately with the client through these objects. The servlet reads the Request object from a `ServletInputStream`. The servlet can invoke the Request object's methods to get information about the client environment, the server environment, and any information provided by the client (for example, form information set by GET or POST). The servlet invokes the Response object's setter methods to return the client response. However, if the servlet is part of a servlet chain, it might pass its response object to another servlet for further processing.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.1.3.2: Having servlets communicate](#)

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 UpdateJSPTest 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=HelloWorldServlet></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>
```

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.1.3.2: Having servlets communicate](#)

4.2.1.3.3: Using page lists to avoid hard coding URLs



IBM WebSphere Application Server supports page lists, which allow application developers to prevent hard-coding URLs in servlets and JSP files. To learn how page lists work, and their advantages, see the page lists description cited in the Related information below.

Use IBM WebSphere Studio to develop (1) servlets that support page lists, and (2) their accompanying .servlet configuration files that specify the page lists. Alternatively, use materials supplied by IBM WebSphere Application Server Version 3.x to manually create the two items.

Regardless of how you obtain them, servlets and their .servlet configuration files can be deployed in an IBM WebSphere Application Server environment.

See the Related information for instructions for using .servlet configuration files obtained from either Studio or WebSphere Application Server.

Related information...

- [0.9.2: What are page lists?](#)
- [0.9.1: What are .servlet configuration files?](#)
- [4.2.1.3.3.1: Using .servlet files to configure servlets](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.1.3: Servlet content, examples, and samples](#)

4.2.1.3.3.1: Obtaining and using servlet XML configuration files (.servlet files)

The IBM WebSphere Studio provides wizards that generate servlets with accompanying XML servlet configuration files (.servlet files).

If you do not have access to the Studio, you can manually implement XML servlet configuration files. The servlet must also be modified or created to support the use of a .servlet file for its configuration.

Using .servlet files from IBM WebSphere Studio

1. Use IBM WebSphere Studio to create a servlet and .servlet files. See the Studio documentation for instructions.
2. Deploy the compiled servlet and its XML servlet configuration file on the application server.

Using manually configured .servlet files

1. Create or obtain a servlet that extends the PageListServlet class.
2. Use the XMLServletConfig class to create an XML servlet configuration file for the servlet instance.
3. Deploy the compiled servlet and its XML servlet configuration file.

Related information...

- [4.2.1.3.3.1.1: Extending PageListServlet](#)
- [4.2.1.3.3.1.2: Using XMLServletConfig to create .servlet configuration files](#)
- [4.2.1.3.3.1.3: XML servlet configuration file syntax \(.servlet syntax\)](#)
- [4.2.1.3.3.1.4: Example: XML servlet configuration file](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.1.3.3: Using page lists to avoid hard coding URLs](#)

4.2.1.3.3.1.1: Extending PageListServlet



IBM WebSphere Application Server Version 3.x supplies the PageListServlet, the superclass of servlets that load pages contained in the page list element (<page-list>) of an XML servlet configuration file.

Implement a servlet that supports the use of XML configuration files (.servlet files) and page lists by extending the PageListServlet class.

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

The PageListServlet.callPage() method receives as input:

- A page name from the page-list element of the XML configuration file
- The HttpServletRequest object
- The HttpServletResponse object

In structuring the servlet code, keep in mind that the PageListServlet.callPage() method is not an exit. Any servlet code that follows the callPage() method invocation will be run after the invocation.

See the Related information for an example of a servlet that extends the PageListServlet.

Related information...

- [4.2.1.3.3.1.1.1: Example: Extending PageListServlet](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.1.3.3.1: Obtaining and using servlet XML configuration files \(.servlet files\)](#)

4.2.1.3.3.1.1.1: Example: Extending PageListServlet



SimplePageListServlet is an example of a servlet that extends the PageListServlet class and uses its callPage() method to invoke a JSP:

```
public class SimplePageListServlet extends com.ibm.servlet.PageListServlet {
    public void service(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException
    {
        try{
            setRequestAttribute("testVar", "test value", req);
            setRequestAttribute("otherVar", "other value", req);

            String pageName = getPageNameFromRequest(req);
            callPage(pageName, req, resp);
        }
        catch(Exception e){
            handleError(req, resp, e);
        }
    }
}
```

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.1.3.3.1.1: Extending PageListServlet](#)

4.2.1.3.3.1.2: Using XMLServletConfig to create .servlet configuration files



IBM WebSphere Application Server Version 3.x supplies the XMLServletConfig class for creating XML servlet configuration files (*servlet_instance_name*.servlet files).

Write a Java program that uses the XMLServletConfig class to generate a servlet configuration file. The XMLServletConfig class provides methods for setting and getting the file elements and their contents.

See the comments in the XMLServletConfig class for an explanation of how to use it.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.1.3.3.1: Obtaining and using servlet XML configuration files \(.servlet files\)](#)

4.2.1.3.3.1.3: XML servlet configuration file syntax (.servlet syntax)

Each XML configuration file must be a well-formed XML document. The files are not validated against a Document Type Definition (DTD). This article describes the syntax, as illustrated by the example cited in Related information.

For the Application Server to use an XML servlet configuration file to load a servlet instance, the file must contain at least the code element. For a PageListServlet, the XML configuration file must contain at least the code element and the page-list element.

Although there is no DTD, it is recommended that all elements appear in the order shown in the example. The elements (also known as *tags*) are:

Tag	Description
servlet	The root element of an XML configuration file. The XMLServletConfig class automatically generates this element.
code	The class name of the servlet (without the .class extension), even if the servlet is in a JAR file
description	A user-defined description of the servlet
init-parameter	The attributes of this element specify a name-value pair to be used as an initialization parameter. A servlet can have multiple initialization parameters, each within its own init-parameter element.
page-list	Identifies the JavaServer pages to be called depending on the path information in the HTTP request. The page-list element can contain the following child elements: <ul style="list-style-type: none">● default-page: Contains a uri element that specifies the location of the page to be loaded, if the HTTP request does not contain path information● error-page: Contains a uri element that specifies the location of the page to be loaded, if the handleError() method sets the request attribute "error"● page: Contains a uri element that specifies the location of the page to be loaded if the HTTP request contains the page name. A page-list element can contain multiple page elements.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.1.3.3.1: Obtaining and using servlet XML configuration files \(.servlet files\)](#)

4.2.1.3.3.1.4: Example: XML servlet configuration file



```
<?xml version="1.0"?>
<servlet>
  <code>SimplePageListServlet</code>
  <description>Shows how to use PageListServlet class</description>
  <init-parameter name="name1" value="value2"/>
  <page-list>
    <default-page>
      <uri>/index.jsp</uri>
    </default-page>
    <error-page>
      <uri>/error.jsp</uri>
    </error-page>
    <page>
      <uri>/TemplateA.jsp</uri>
      <page-name>page1</page-name>
    </page>
    <page>
      <uri>templateB.jsp</uri>
      <page-name>page2</page-name>
    </page>
  </page-list>
</servlet>
```

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.1.3.3.1: Obtaining and using servlet XML configuration files \(.servlet files\)](#)

4.2.1.3.4: Filtering and chaining servlets



The Application Server Version 3 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.

Related information...

- [4.2.1.3.4.1: Servlet filtering with MIME types](#)
- [4.2.1.3.4.2: Servlet filtering with servlet chains](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.1.3: Servlet content, examples, and samples](#)

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 virtual hosts containing servlets.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.1.3.4: Filtering and chaining servlets](#)

4.2.1.3.4.2: Servlet filtering with servlet chains



To configure a servlet chain, use the Administrative Console to:

1. Add an instance of the internal servlet `com.ibm.websphere.servlet.filter.ChainerServlet` to your Web application. Assign the `ChainerServlet` a servlet URL, such as `abc`.
2. For the instance of the `ChainerServlet`, add the following initialization parameter and value:

Parameter	Value
-----------	-------

<code>chainer.pathlist</code>	<code>/first_servlet_URL /next_servlet_URL</code>
-------------------------------	---

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
-----------	-------

<code>chainer.pathlist</code>	<code>/servletA /servletB /servletC</code>
-------------------------------	--

3. To invoke a servlet chain, invoke the servlet URL of the `ChainerServlet` instance, for example, `http://your.server.name/webapp/example/abc`.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.1.3.4: Filtering and chaining servlets](#)

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 Version 3 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 `ServletErrorReport`). 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 Version 3, the `ServletResponse.sendError()` method has been overridden to provide the following function:

```
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);
}
```

Related information...

- [4.2.1.3.5.1: Public methods of the ServletErrorReport class](#)
- [4.2.1.3.5.2: Example: JSP file for handling application errors](#)
- [4.2.1.3.5.3: Using your error reporting servlet or JSP file](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.1.3: Servlet content, examples, and samples](#)

4.2.1.3.5.1: Public methods of the ServletException class



To create an error JSP or servlet, you need to know the public methods of the ServletException class (the error bean), which are:

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

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

}
```

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.1.3.5: Enhancing servlet error reporting](#)

4.2.1.3.5.2: Example: JSP file for handling application errors



An example of an error JSP file is:

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

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

Note: If you do not want to write your own error, consider adding the optional internal servlet, `com.ibm.servlet.engine.webapp.DefaultErrorReporter`, to your Web application.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.1.3.5: Enhancing servlet error reporting](#)

4.2.1.3.5.3: Using your error reporting servlet or JSP file



To enable this function for your Web application:

1. Use an administrative client to configure an error path for the Web application.
2. Create the error JSP or servlet.
3. Place the file in the Web application document root.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.1.3.5: Enhancing servlet error reporting](#)

4.2.1.3.6: Serving servlets by classname



IBM WebSphere Application Server provides a WebSphere servlet that you can add to your Web applications. Web applications that contain the servlet can serve servlets by the servlet classnames (such as MyServletClass). No additional steps are required.

See section 4.2.1.2.3 for details and instructions.

Related information...

- [4.2.1.2.3: Using the WebSphere servlets for a head start](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.1.3: Servlet content, examples, and samples](#)

4.2.1.3.7: Serving all files from application servers



IBM WebSphere Application Server provides a WebSphere servlet that you can add to your Web applications. Web applications that contain the servlet can serve HTML, eliminating the need to add a pass rule to the Web server for serving the same HTML files. No additional steps are required.

See section 4.2.1.2.3 for details and instructions.

Related information...

- [4.2.1.2.3: Using the WebSphere servlets for a head start](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.1.3: Servlet content, examples, and samples](#)

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`.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.1.3: Servlet content, examples, and samples](#)

0.10: What are JavaServer Pages (JSP) files?



The Application Server supports a powerful approach to dynamic Web page content: JavaServer Pages (JSP) files. JSP files are application building blocks coded to the Sun Microsystems JSP specification.

JSP files enable the separation of the HTML coding from the business logic in Web pages, allowing HTML programmers and Java programmers to more easily collaborate in creating and maintaining pages.

The IBM extensions to the JSP specification include JSP tags that resemble HTML, making it easy for HTML authors to add the power of Java to Web pages without being experts in Java.

JSP files support a division of roles:

Team member	Role
HTML authors	Develop JSPs that access databases and reusable Java components, such as servlets and JavaBeans
Java programmers	Create the reusable Java components and provide the HTML authors with the component names and attributes
Database administrators	Provide the HTML authors with the name of the database access and table information

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.

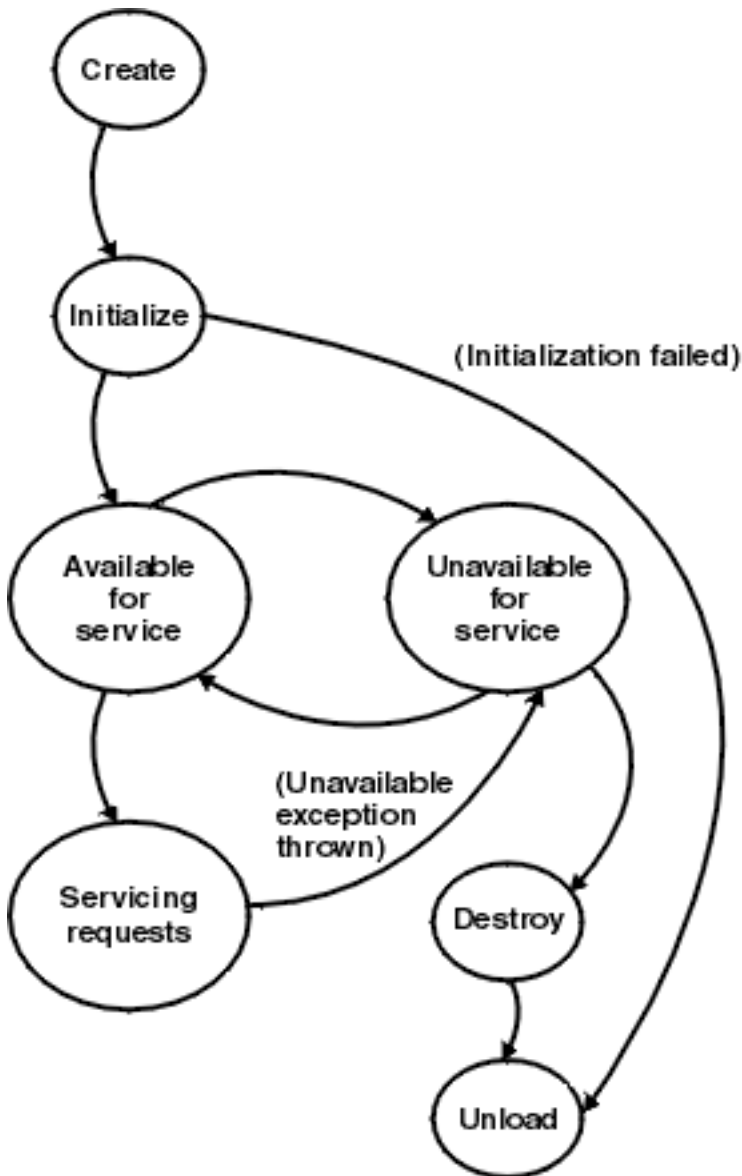
Related information...

- [4.2.2.1: JSP lifecycle](#)
- [4.2.2.1a: JSP access model](#)
- [4.2.2.2: JSP support and environment in WebSphere](#)
- [4.2.2.3: JSP file content, examples, and samples](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2: Building Web applications](#)

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 servlet engine 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 servlet engine 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 servlet engine 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 servlet engine 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 servlet engine no longer needs the servlet or a new instance of the servlet is being reloaded, the servlet engine invokes the servlet's `destroy()` method. The servlet engine 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()` method is called.

Related information...

- [4.2.2.1a: JSP access models](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.2: Developing JSP files](#)

4.2.2.1a: JSP access models

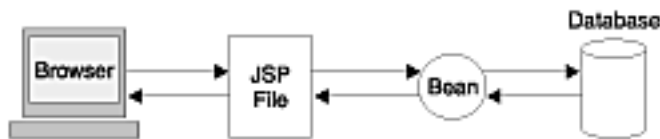


JSP files can be accessed in two ways:

- 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:

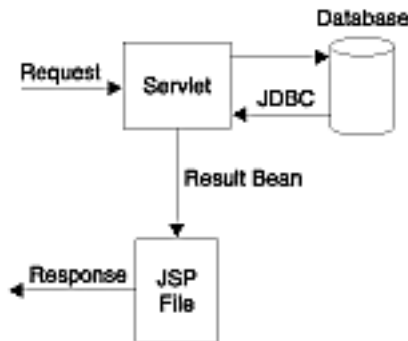
Request for a JSP file



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.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.1: JavaServer Pages \(JSP\) lifecycle](#)

4.2.2.2: JSP support and environment in WebSphere



IBM WebSphere Application Server Version 3.5 supports the JSP 1.0 Specification from Sun Microsystems. If you are going to develop new JSP files for use with IBM WebSphere Application Server, it is recommended you use JSP 1.0.

It also supports the JSP .91 Draft Specification, like IBM WebSphere Application Server Version 2.0x. Please consult the Related information for the necessary migration of JSP .91 APIs that are deprecated in Version 3.5.

Related information...

- [4.2.2.2.1: APIs for separating logic and content](#)
- [4.2.2.2.2: JSP processors](#)
- [4.2.2.2.3: Tools for JSP programming](#)
- [4.2.2.2.4: Batch compiling JSP files](#)
- [3.3.3: Migrating to supported JSP specifications](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.2: Developing JSP files](#)

4.2.2.2.1: JSP support for separating logic from presentation



Two interfaces support the JSP technology. These APIs provide a way to separate content generation (business logic) from the presentation of the content (HTML formatting).

This separation enables servlets to generate content and store the content (for example, in a bean) in the request object. The servlet that generated the context generates a response by passing the request object to a JSP file that contains the HTML formatting. The <BEAN> tag provides access to the business logic.

Goal	Interface
Set attributes in the request object.	<code>javax.servlet.http.HttpServletRequest.setAttribute()</code>
Forward a response object to another servlet or JSP file.	<code>javax.servlet.http.RequestDispatcher.forward()</code>

In IBM WebSphere Application Server Version 2.0x, these interfaces had different names. You might need to migrate code that is calling the old interfaces. See the Related information for details.

Related information...

- [3.3.3: Migrating to supported JSP specification](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.2: Developing JSP files](#)

4.2.2.2.2: JSP processors

IBM WebSphere Application Server provides a JSP processor for each supported level of the JSP specification, .91, 1.0 and 1.1. Each JSP processor is a **servlet** that you can add to a Web application to handle all JSP requests pertaining to the Web application.

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.

By specifying either a .91, 1.0 or 1.1 JSP Enabler for each Web application containing JSP files, you configure Web applications to pass JSP files in the Web application folder to the JSP processor corresponding to the JSP specification level of the JSP files.

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

- .java file, which contains the Java language code for the servlet (unless [otherwise specified](#), the .java files are automatically deleted from the server once the .class files have been compiled)
- .class file, which is the compiled servlet

The JSP processor puts the .java and the .class file in a path specific to the processor (see below). The .java and the .class file have the same filename. The JSP .91 processor uses a naming convention that includes adding underscore characters and a suffix to the JSP filename. For example, if the JSP filename is `simple.jsp`, the generated files are `_simple_xjsp.java` and `_simple_xjsp.class`.

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.

JSP 1.1 processor

Processor servlet name	JspServlet
Class name and location	org.apache.jasper.runtime.JspServlet in ibmwebas.jar
Where processor puts output	<i>AS_install_root</i> \temp\servlet_host_name\app_name\

For example, if the JSP file is in:

```
c:\WebSphere\AppServer\hosts\default_host\examples\web
the .java and .class files are put in:
```

```
c:\WebSphere\AppServer\temp\default_host\examples\
```

JSP 1.0 processor

Processor servlet name	JSP Servlet
Class name and location	com.sun.jsp.runtime.JspServlet in ibmwebas.jar
Where processor puts output	<i>AS_install_root</i> \temp\servlet_host_name\app_name\

For example, if the JSP file is in:

```
c:\WebSphere\AppServer\hosts\default_host\examples\web
the .java and .class files are put in:
```

```
c:\WebSphere\AppServer\temp\default_host\examples\
```

JSP .91 processor

Processor servlet name	PageCompileServlet
Class name and location	com.ibm.servlet.jsp.http.pagecompile.PageCompileServlet in ibmwebas.jar
Where processor puts output	<i>AS_install_root</i> \temp\ <i>servlet_host_name</i> \ <i>app_name</i> \pagecompile where <i>AS_install_root</i> is the path where the Application Server is installed and <i>app_name</i> is the name of the application root folder.

For example, if the JSP file is in:

c:\WebSphere\AppServer\hosts\default_host\examples\web

the .java and .class files are put in:

c:\WebSphere\AppServer\temp\default_host\examples\pagecompile

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.2: Developing JSP files](#)

4.2.2.2.3: Tools for developing JSP files



The following products make it easier to develop JSP files:

IBM WebSphere Studio Version 1.0 and Version 3.0

The Studio wizards create servlets, JavaBeans, and JSPs. The wizards include support for building SQL queries to a relational database and maintaining user information about Web visitors.

Use the wizard output files "as is," or tailor the output further to meet your requirements. For more information about the Studio, visit the Studio Web site:

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

NetObjects ScriptBuilder Version 2.01

The JSP support in ScriptBuilder Version 2.01 includes templates for JSP files, sample JSP files, and highlighting of JSP syntax. Visit the NetObjects Web site for more information about ScriptBuilder:

<http://www.netobjects.com>

IBM VisualAge for Java Enterprise 3

VisualAge for Java 3 provides the following tools for developing JSPs:

- Servlet Launcher - Allows you to start a Web server, open your Web browser, and launch a servlet. This tool is available for AIX and Windows NT systems.
- JSP Execution Monitor - Allows you to monitor the execution of JSP source, generated servlets, and HTML source as it is generated. This tool is available for Windows NT systems.

VisualAge for Java 3 also allows you to 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 VisualAge for Java 3, visit:

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

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.2: Developing JSP files](#)

4.2.2.2.4: Batch Compiling JSP files

3.5.2 

As an IBM enhancement to JSP support, IBM Application Server Version 3.5 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. Unless you have [configured the web application otherwise](#), 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.

The process of batch compiling JSP files is different for [JSP 0.91 files](#) and [JSP 1.0 files](#). Consult the page corresponding to the JSP level for your files.

Related information...

- [4.2.2.2.4.1: Compiling JSP .91 files as a batch](#)
- [4.2.2.2.4.2: Compiling JSP 1.0 files as a batch](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.2: Developing JSP files](#)

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><% @ and %></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
Class-wide variable and method declarations	<p>Use the <code><SCRIPT></code> and <code></SCRIPT></code> tags to declare class-wide variables and class-wide methods for the servlet class.</p>
Inline Java code (scriptlets) , enclosed within <code><% and %></code>	<p>You can embed any valid Java language code inline within a JSP file between the <code><% and %></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 (JSP .91 or JSP 1.0) or Java expressions enclosed within <code><%= and %></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><%= and %></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>
<BEAN> tag	<p>Use the <code><BEAN></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 .91](#) or [JSP 1.0](#))

The IBM extensions to JSP .91 and 1.0 make it easy for non-programmers to create Web pages that access databases.

HTML tags

A JSP file can contain any valid HTML tags. Refer to your favorite HTML reference for a description of those tags.

<SERVLET> tags

Using the <SERVLET> tag is one method for embedding a servlet within a JSP file.

NCSA tags

You might have legacy SHTML files that contain NCSA tags for server-side includes. If the IBM WebSphere Application Server Version 3.5 supports the NCSA tags in your SHTML files, you can convert the SHTML files to JSP files and retain the NCSA tags.

Related information...

- [4.2.2.3a: JSP 1.0 examples](#)
- [4.2.2.3b: JSP .91 examples](#)
- [4.2.2.3.1: JSP .91 syntax: JSP directives](#)
- [4.2.2.3.2: JSP .91 syntax: Class-wide variables and methods](#)
- [4.2.2.3.3: JSP .91 syntax: Inline Java code \(scriptlets\)](#)
- [4.2.2.3.4: JSP .91 syntax: Java expressions](#)
- [4.2.2.3.5: JSP .91 syntax: BEAN tags](#)
- [4.2.2.3.6: Supported NCSA tag reference](#)
- [4.2.2.3.7: IBM extensions to JSP 1.0 syntax](#)
- [4.2.2.3.8: IBM extensions to JSP .91 syntax](#)
- [WebSphere Samples Gallery](#)
- [4.1.1: Finding supported APIs and specifications](#) (including JSP)
- [www.w3c.org](#) for HTML information
- [Index to API documentation \(Javadoc\)](#)
- [4.2.2: Developing JSP files](#)

4.2.2.3.1: JSP .91 syntax: JSP directives



Use JSP directives (enclosed within `<%@` and `%>`) to specify:

- Scripting language being used
- Interfaces a servlet implements
- Classes a servlet extends
- Packages a servlet imports
- MIME type of the generated response

The general syntax of the JSP directive is:

```
<%@ directive_name = "value" %>
```

where the valid directive names are:

- **language**

The scripting language used in the file. At this time, the only valid value and the default value is `java` (the Java programming language). The scope of this directive is the JSP file. When used more than once, only the first occurrence of the directive is significant.

An example:

```
<%@ language = "java" %>
```

- **method**

The name of the method generated by the embedded Java code (scriptlet). The generated code becomes the body of the specified method name. The default method is `service`. When used more than once, only the first occurrence of the directive is significant. An example:

```
<%@ method = "doPost" %>
```

- **import**

A comma-separated list of Java language package names or class names that the servlet imports. This directive can be specified multiple times within a JSP file to import different packages. An example:

```
<%@ import = "java.io.*,java.util.Hashtable" %>
```

- **content_type**

The MIME type of the generated response. The default value is `text/html`. This information is used to generate the response header. When used more than once, only the first occurrence of this directive is significant. This directive can be used to specify the character set in which the page is to be encoded. An example:

```
<%@ content_type = "text/html; charset=iso-8859-1" %>
```

- **implements**

A comma-separated list of Java language interfaces that the generated servlet implements. You can use this directive more than once within a JSP file to implement

different interfaces. An example:

```
<%@ implements = "javax.servlet.http.HttpSessionContext" %>
```

- **extends**

The name of the Java language class that the servlet extends. The class must be a valid class and does not have to be a servlet class. The scope of this directive is the JSP file. When used more than once, only the first occurrence of the directive is significant. An example:

```
<%@ extends = "javax.servlet.http.HttpServlet" %>
```

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3: Overview of JSP file content](#)

4.2.2.3.2: JSP .91 syntax: Class-wide variables and methods

Use the `<SCRIPT>` and `</SCRIPT>` tags to declare class-wide variables and class-wide methods for the servlet class. The general syntax is:

```
<script runat=server>

// code for class-wide variables and methods

</script>
```

The attribute `runat=server` is required and indicates that the tag is for server-side processing. An example of specifying class-wide variables and methods:

```
<script runat=server>

// class-wide variables
    init i = 0;
    String foo = "Hello";

// class-wide methods
    private void foo() {
        // code for the method

    }

</script>
```

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3: Overview of JSP file content](#)

4.2.2.3.3: JSP .91 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.

The scriptlet can use a set of predefined variables that correspond to essential servlet, output, and input classes:

- **request**

The servlet request class defined by `javax.servlet.http.HttpServletRequest`

- **response**

The servlet response class defined by `javax.servlet.http.HttpServletResponse`

- **out**

The output writer class defined by `java.io.PrintWriter`. The content written to the writer is the client response.

- **in**

The input reader class defined by `java.io.BufferedReader`

An example:

```
<%  
foo = request.getParameter("Name");  
out.println(foo);  
%>
```

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>"); %>  
<% } %>
```

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3: Overview of JSP file content](#)

4.2.2.3.4: JSP .91 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 `<SCRIPT>` example in class-wide variables and methods:

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

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

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3: Overview of JSP file content](#)

4.2.2.3.5: JSP .91 syntax: BEAN tags



Use the <BEAN> tag to create an instance of a bean that will be accessed elsewhere within the JSP file. Then use JSP tags for variable data (such as the <INSERT> tag described later in this document) to access the bean.

The 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.

Related information...

- [4.2.2.3.5.1: BEAN tag syntax](#)
- [4.2.2.3.5.3: Setting bean properties](#)
- [4.2.2.3.5.2: Accessing bean properties](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3: Overview of JSP file content](#)

4.2.2.3.5.1: JSP .91 syntax: <BEAN> tag syntax



```
<bean name="bean_name" varname="local_bean_name"
      type="class_or_interface_name" introspect="yes|no"
      beanName="ser_filename" create="yes|no"
      scope="request|session|userprofile" >
</bean>
```

where the attributes are:

- **name**

The name used to look up the bean in the appropriate scope (specified by the scope attribute). For example, this might be the session key value with which the bean is stored. The value is case-sensitive.

- **varname**

The name used elsewhere within the JSP file to refer to the bean. This attribute is optional. The default value is the value of the name attribute. The value is case-sensitive.

- **type**

The name of the bean class file. This name is used to declare the bean instance in the code. The default value is the type Object. The value is case-sensitive.

- **introspect**

When the value is `yes`, the JSP processor examines all request properties and calls the set property methods (passed in the BeanInfo) that match the request properties. The default value of this attribute is `yes`.

- **beanName**

The name of the bean class file, the bean package name, or the serialized file (.ser file) that contains the bean. (This name is given to the bean instantiator.) This attribute is used only when the bean is not present in the specified scope and the create attribute is set to `yes`. The value is case-sensitive.

The path of the file must be specified in the Web application classpath.

- **create**

When the value is `yes`, the JSP processor creates an instance of the bean if the processor does not find the bean within the specified scope. The default value is `yes`.

- **scope**

The lifetime of the bean. This attribute is optional and the default value is `request`. The valid values are:

- `request` - The bean is added to the request object by a servlet that invokes the JSP file using the APIs described in JSP API. If the bean is not part of the request context, the bean is created and stored in the request context unless the create

attribute is set to no.

- session - If the bean is present in the current session, the bean is reused. If the bean is not present, it is created and stored as part of the session if the create attribute is set to yes.
- userprofile - This attribute value is an IBM extension to JSP 0.91 and causes the user profile to be retrieved from the servlet request object, cast to the specified type, and introspected. If a type is not specified, the default type is `com.ibm.websphere.UserProfile`. The create attribute is ignored.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3.5: JSP .91 syntax: BEAN tags](#)

4.2.2.3.5.2: JSP .91 syntax: Accessing bean properties



After specifying the <BEAN> tag, you can access the bean at any point within the JSP file. There are three methods for accessing bean properties:

- Using a JSP scriptlet
- Using a JSP expression
- Using the <INSERT> tag (as described in [the JSP .91 tags for variable data](#))

An example:

```
<!-- The bean declaration -->
```

```
<bean name="foobar" type="FooClass" scope="request" >  
  <param name="fooProperty" value="fooValue">  
</bean>
```

```
<!-- Later in the file, some HTML content that includes JSP syntax that calls  
a method of the bean -->
```

```
<p>The name of the row is <%= foobar.getRowName() %>.</p>
```

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3.5: JSP .91 syntax: BEAN tags](#)

4.2.2.3.5.3: JSP .91 syntax: Setting bean properties



You can set the bean properties by using the <PARAM> tag within the <BEAN> tag. The <PARAM> tag specifies a list of properties and the corresponding values. The properties are automatically set in the bean using introspection. The properties are set once when the bean is instantiated. The <PARAM> tag syntax is:

```
<PARAM name="property_name" value="property_value">
```

This syntax is an IBM extension to the JSP 0.91 <PARAM> tag. The IBM syntax is consistent with the syntax of the <PARAM> tag used within the <SERVLET> and <APPLET> tags.

In addition to using the <param> tag to set bean properties, there are three other methods:

- Specifying query parameters when requesting the URL of the JSP file that contains the bean. The introspect attribute must be set to yes. An example:

```
http://www.myserver.com/signon.jsp?name=jones&password=d13x
```

where the bean property name will be set to jones.

- Specifying the properties as parameters submitted through an HTML <FORM> tag. The JSP method directive must be set to post. The action attribute is set to the URL of the JSP file that invokes the bean. The introspect attribute must be set to yes. An example:

```
<form action="http://www.myserver.com/SearchSite.jsp" method="post">  
  <input type="text" name="Search for: ">  
  <input type="submit">  
</form>
```

- Using JSP syntax to set the bean property

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3.5: JSP .91 syntax: BEAN tags](#)

4.2.2.3.6: Supported NCSA tag reference



The product supports the following NCSA tags through their use in JSP files:

- config
- echo var=*variable* (see below)
- exec
- filesize
- include
- lastmodified
- Commands for formatting size and date outputs

For the echo command, the product supports the standard server-side include (SSI) environment variables and Common Gateway Interface (CGI) environment variables.

The SSI environment variables

Variable	Description
DATE_GMT	The current date and local time zone in Greenwich mean time (GMT)
DATE_LOCAL	The current date and local time zone
DOCUMENT_NAME	The current filename
DOCUMENT_URI	The path to the document (such as, /docs/tutorials/index.shtml)
QUERY_STRING_UNESCAPED	The unescaped version of any search query the client sent, with all shell special characters escaped with the \ character
LAST_MODIFIED	The last date the current document was changed

CGI environment variables

Variable	Description
AUTH_TYPE	The protocol-specific authentication method used to validate the user, if the server supports user authentication and the script is protected
CONTENT_LENGTH	The length of the content, as specified by the remote host
CONTENT_TYPE	The data content type for queries that have information attached (such as HTTP POST and PUT)
GATEWAY_INTERFACE	The revision level of the CGI specification to which the server complies
PATH_INFO	The extra path information given by the client in this request. The extra information follows the virtual pathname of the CGI script.

PATH_TRANSLATED	The server provides a translated version of PATH_INFO, which takes the path and performs any virtual-to-physical mapping.
QUERY_STRING	The information that follows the ? symbol in the URL request for a script
REMOTE_HOST	The hostname of the remote host sending the request. If the server does not have this information, the server should set REMOTE_ADDR and leave REMOTE_HOST unset.
REMOTE_ADDR	The IP address of the remote host sending the request
REMOTE_IDENT	If the HTTP server supports RFC 931 identification, the remote username retrieved from the server
REMOTE_USER	The username used for authentication, if the server supports user authentication and the script is protected
REQUEST_METHOD	The method with which this request was made. Methods include HTTP, GET, HEAD, POST, and so on
SCRIPT_NAME	The virtual path to the script being run. This variable is used for self-referencing URLs
SERVER_NAME	The IP address, hostname, or Domain Name Server (DNS) alias of the server
SERVER_PORT	The port number to which the request was sent
SERVER_PROTOCOL	The name and revision level of the protocol used to format this request
SERVER_SOFTWARE	The name and version of the server answering the request

Related information...

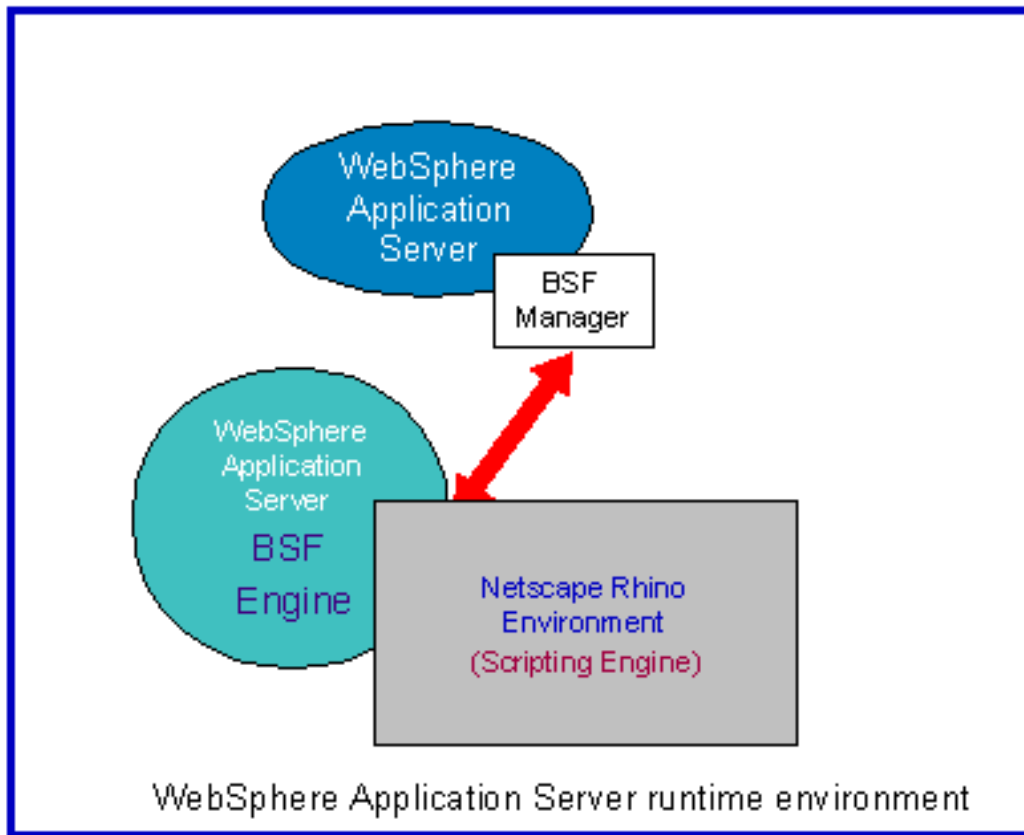
- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3: Overview of JSP file content](#)

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 version 3.5.2 provides the **Bean Scripting Framework (BSF)** which consists of a BSF Manager and a BSF Engine, and a scripting engine which is Netscape's **Rhino** version 1.5 environment.

Netscape's

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

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 [IBM's BSF project Web site](#) for more information on BSF functionality.

See section 4.2.5.1 when you are ready to delve into programming examples.

Related information...

- [4.2.5.1: BSF examples and samples](#)
- [4.2: Building Web applications](#)

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 article, [Placing files and setting classpaths](#), to administer your application.

The following steps and code samples describe how to implement BSF. To test these code samples, from a Browser window, copy the code samples and paste them into your own file. You can use any filename, 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.

To use 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 file, [JSP access models](#), for more JSP information.

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

```
<html>
<head>
  <title> Temperature Table >/title>
</head>
<body>
<h1>Temperature Table</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>
<tr>
<td> <h3>Java code example</h3></td></tr>

<%
  for (int i=0; i<=100; i+=10) {
    out.println ("<tr ALIGN=RIGHT BGCOLOR=\"#CCCCCC\">");
    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 >/title>
</head>
<body>
<h1>Temperature Table</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>
<tr>
<td> <h3>Java code example</h3></td></tr>

<%
  for ( i=0; i<=100; i+=10) {
    out.println ("<tr ALIGN=RIGHT BGCOLOR=\"#CCCCCC\">")
    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's 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.

Related information...

- [4.2.5: Using the Bean Scripting Framework](#)
- [4.2: Building Web applications](#)

4.2.2.3.7: IBM extensions to JSP 1.0 syntax



Refer to the Sun JSP 1.0 Specification for the base JavaServer Pages (JSP) APIs. IBM WebSphere Application Server Version 3.5 provides several extensions to the base APIs.

For JSP 1.0, 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.

Related information...

- [4.2.2.3.7.1: IBM extensions to JSP 1.0 syntax for variable data](#)
- [4.2.2.3.7.2: IBM extensions to JSP 1.0 syntax for database access](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3: Overview of JSP file content](#)

4.2.2.3.7.1: JSP 1.0 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>	This IBM extension of the Sun JSP 1.0 <jsp:getProperty> tag implements all of the <jsp:getProperty> function and adds the ability to introspect a database bean that was created using the IBM extension <tsx:dbquery> or <tsx:dbmodify>.
Repeat a block of HTML tagging that contains the <tsx:getProperty> syntax and the HTML tags for formatting content.	<tsx:repeat>	Use the <tsx:repeat> syntax to iterate over a database query results set. The <tsx:repeat> syntax iterates from the start value to the end value until one of the following conditions is met: <ul style="list-style-type: none">● The end value is reached.● An ArrayIndexOutOfBoundsException is thrown. The output of a <tsx:repeat> block is buffered until the block completes. If an exception is thrown before a block completes, no output is written for that block.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3.7: IBM extensions to JSP 1.0 syntax](#)

4.2.2.3.7.1.1: JSP 1.0 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 />
```

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3.7.1: IBM extensions to JSP 1.0 syntax for variable data](#)

4.2.2.3.7.1.2: JSP 1.0 syntax: <tsx:repeat> tag syntax



```
<tsx:repeat index=name start="starting_index" stop="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.

- **stop**

An optional ending index value for this repeat block. The maximum value is 2,147,483,647. If the value of the stop attribute is less than the value of the start attribute, the stop attribute is ignored.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3.7.1.2a: Repeat tag results set and the associated bean](#)
- [4.2.2.3.7.1: IBM extensions to JSP 1.0 syntax for variable data](#)

4.2.2.3.7.1.2a: JSP 1.0 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	<code><tsx:repeat></code> 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="col1(abc)" /> </tsx:repeat></pre> <p>Notes:</p> <ul style="list-style-type: none">● The bean (<code>myBean.class</code>) is a static bean.● The method to access the bean properties is <code>myBean.get(property(index))</code>.● You can omit the property index, in which case the index of the enclosing <code><tsx:repeat></code> is used. You can also omit the index on the <code><tsx:repeat></code>.● The <code><tsx:repeat></code> 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:dbconnect > <tsx:dbquery id="dynamic" connection="conn" > Select * from myTable; </tsx:dbquery> <tsx:repeat index=abc> <tsx:getProperty name="dynamic" property="col1(abc)" /> </tsx:repeat></pre> <p>Notes:</p> <ul style="list-style-type: none">● The bean (dynamic) is generated by the <code><tsx:dbquery></code> and does not exist until the syntax is executed.● The method to access the bean properties is <code>dynamic.getValue("property", index)</code>.● You can omit the property index, in which case the index of the enclosing <code><tsx:repeat></code> is used. You can also omit the index on the <code><tsx:repeat></code>.● The <code><tsx:repeat></code> 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" /t></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>

```

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3.7.1.2: <tsx:repeat> tag syntax](#)

4.2.2.3.7.2: JSP 1.0 syntax: Tags for database access



The Application Server Version 3 extends JSP 1.0 support by providing 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> <p>Examples: JSP10Employee.jsp</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 JSP10Employee.jsp example) that requests the database connection.</p> <p>The <tsx:userid> and <tsx:passwd> must be used within a <tsx:dbconnect> tag.</p> <p>Examples: None</p>

<p>Establish a connection to a database, submit database queries, and return the results set.</p>	<p><u><tsx:dbquery></u></p>	<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 connection 3. Retrieves and caches data in the results object 4. Closes the connection (releases the connection resource) <p>Examples: <u>Basic example</u> <u>JSP10Employee.jsp</u> <u>JSP10EmployeeRepeatResults.jsp</u></p>
<p>Establish a connection to a database and then add records to a database table.</p>	<p><u><tsx:dbmodify></u></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: <u>Basic example</u> <u>JSP10EmployeeRepeatResults.jsp</u></p>

Display query results.	<tsx:repeat> and <tsx:getProperty>	<p>The <code><tsx:repeat></code> loops through each of the rows in the query results. The <code><tsx:getProperty></code> uses the query results object (for the <code><tsx:dbquery></code> syntax whose identifier is specified by the <code><tsx:getProperty></code> bean attribute) and the appropriate column name (specified by the <code><tsx:getProperty></code> property attribute) to retrieve the value.</p> <p>Examples: Basic example</p>
------------------------	---	---

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3.7: IBM extensions to JSP 1.0 syntax](#)

4.2.2.3.7.2.1: JSP 1.0 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 table 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"
```

Note: 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.

All of the elements shown in the example XML file [JSP10Employee.jsp](#) need to be specified. However, an empty element (such as `<url></url>`) is valid.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3.7.2: IBM extensions to JSP 1.0 syntax for database access](#)

4.2.2.3.7.2.2: JSP 1.0 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.

See the [JSP10Login.jsp](#) and the [JSP10Employee.jsp](#) examples for an illustration of how to set the USERID and PASSWORD using parameters in the request object. The request parameters are set using an HTML form (JSP10Login.jsp). In the JSP10Employee.jsp, the values of the parameters are passed as hidden form values to the JSP10EmployeeRepeatResults.jsp.

- **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.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3.7.2: IBM extensions to JSP 1.0 syntax for database access](#)

4.2.2.3.7.2.3: JSP 1.0 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.

Related information...

- [4.2.2.3.7.2.3a: Basic example](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3.7.2: IBM extensions to JSP 1.0 syntax for database access](#)

4.2.2.3.7.2.4: JSP 1.0 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

Related information...

- [4.2.2.3.7.2.4a: Basic example](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3.7.2: IBM extensions to JSP 1.0 syntax for database access](#)

4.2.2.2.4.2: Compiling JSP 1.0 files as a batch



To use the JSP batch compiler for JSP 1.0 files, enter the following command on a single line at an operating system command prompt:

```
JspBatchCompiler -adminNodeName <node name> [ -serverName <server name>
[-application <application name> [-filename <filename>]]]
[-keepgenerated <true|false>]
```

where:

- **adminNodeName**

This is the name of the node as shown on the Administrative Console.

- **serverName**

[Optional: may only be used if *adminNodeName* is set] This is the name of the Application Server in the WebSphere environment on which you wish to perform this action. Unless you have set up other servers, this will be "Default Server" [Note that from the command-line, you will need to include quote marks around the name of the server if that name comprises two or more words separated by spaces. You do not have to do this if you use the *batchcompiler.config* file described below.]

- **application**

[Optional: may only be used if *serverName* is set] The name of a particular web application, should you wish to compile only those JSP files under that application.

- **filename**

[Optional: may only be used if *application* is set] The name of a single file in the web application you selected above, should you wish to compile only a single JSP file in an application.

- **keepgenerated**

[Optional] If set to "yes" this will keep the generated .java files used for compilation on your server. By default, this is set to "no" and the .java files are all erased after the class files have been compiled.

In lieu of specifying the parameters in a command line, you may specify them in the *batchcompile.config* file, located in the WebSphere Application Server bin directory. No quotation marks are necessary for any of the variables if you use this file. Any values you enter on the command-line will override the values specified in the *batchcompile.config* file.

Example

Suppose you want to precompile the JSP files associated with the *examples* application, shipped with WebSphere Application Server. Issue the following command in the *appserver* bin directory:

```
D:\WebSphere\AppServer\bin>JspBatchCompiler.bat -adminNodeName mynode
-serverName "Default Server" -application examples
```

You should receive the following response from the server

```
Server name: Default Server
Application Name: examples
JSP version: 1.0
docRoot: d:\WebSphere\AppServer\hosts\default_host\examples\web
Application Classpath:
d:\WebSphere\AppServer\hosts\default_host\examples\servlets;
Application output dir: d:\WebSphere\AppServer/temp/default_host/examples
URL: .jsp
URL: .jsv
URL: .jsw
Attempting to compile:
d:\WebSphere\AppServer\hosts\default_host\examples\web\debug_error.jsp
Compilation successful
Attempting to compile:
```

```
d:\WebSphere\AppServer\hosts\default_host\examples\web\HelloHTML.jsp  
Compilation successful
```

```
.  
.  
Attempting to compile:
```

```
d:\WebSphere\AppServer\hosts\default_host\examples\web\StockQuoteWMLRequest.jsp
```

```
Compilation successful
```

```
Attempting to compile:
```

```
d:\WebSphere\AppServer\hosts\default_host\examples\web\StockQuoteWMLResponse.jsp
```

```
Compilation successful
```

If you look in the appserver temp directory, you should see a directory named examples. All of the compiled class files for the examples application will be in this directory.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.2: Developing JSP files](#)
- [4.2.2.2.4: Batch Compiling JSP files](#)

4.2.2.3a: JSP 1.0 examples



The example JSP 1.0 application accesses the Sample database that you can install with IBM DB2. The example application includes:

JSP10Login.jsp	An interface for logging in to the application
JSP10Employee.jsp	A dialog for querying and updating database records
JSP10EmployeeRepeatResults.jsp	A dialog for displaying update confirmations and query results

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3: Overview of JSP file content](#)

4.2.2.3.7.2.3a: Example: JSP 1.0 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>
```

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.3.7.2.3: <tsx:dbquery> tag syntax](#)

4.2.2.3.7.2.4a: Example: JSP 1.0 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>
```

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3.7.2.4: <tsx:dbmodify> tag syntax](#)

4.2.2.3.7.2.5a: Example: JSP 1.0 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>
```

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3.7.2: JSP 1.0 syntax: Tags for database access](#)

4.2.2.3.8: IBM extensions to JSP .91 syntax



Refer to the Sun JSP .91 specification for the base JavaServer Pages (JSP) APIs. IBM WebSphere Application Server provides several extensions to the base APIs.

For JSP .91, 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.

Related information...

- [4.2.2.3.8.1: IBM extensions to JSP .91 syntax for variable data](#)
- [4.2.2.3.8.2: IBM extensions to JSP .91 syntax for data access](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3: Overview of JSP file content](#)

4.2.2.3.8.1: JSP .91 syntax: Tags for variable data



The variable data syntax enables you to put variable fields on your HTML page 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
Embed variables in a JSP file	<INSERT>	This is the base tag for specifying variable fields.
Repeating a block of HTML tagging that contains the <INSERT> tags and the HTML tags for formatting content	<REPEAT>	<p>Use the <REPEAT> tag to iterate over a database query results set. The <REPEAT> tag 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 <code>ArrayIndexOutOfBoundsException</code> is thrown. <p>The output of a <REPEAT> block is buffered until the block completes. If an exception is thrown before a block completes, no output is written for that block.</p>

The above tags are designed to pass intact through HTML authoring tools. Each tag has a corresponding end tag. Each tag is case-insensitive, but some of the tag attributes are case-sensitive.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3.8: IBM extensions to JSP .91 syntax](#)

4.2.2.3.8.1.1: JSP .91 syntax: <INSERT> tag syntax



```
<insert requestparm=pvalue requestattr=avalue bean=name  
  property=property_name(optional_index).subproperty_name(optional_index)  
  default=value_when_null>  
</insert>
```

where:

- **requestparm**

The parameter to access within the request object. This attribute is case-sensitive and cannot be used with the bean and property attributes.

- **requestattr**

The attribute to access within the request object. The attribute would have been set using the `setAttribute` method. This attribute is case-sensitive and cannot be used with the bean and property attributes.

- **bean**

The name of the JavaBean declared by a <BEAN> tag within the JSP file. The value of this attribute is case-sensitive.

When the bean attribute is specified but the property attribute is not specified, the entire bean is used in the substitution. For example, if the bean is type `String` and the property is not specified, the value of the string is substituted.

- **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. This attribute cannot be used with the `requestparm` and `requestattr` attributes.

- **default**

An optional string to display when the value of the bean property is null. If the string contains more than one word, the string must be enclosed within a pair of double quotes (such as "HelpDesk number"). The value of this attribute is case-sensitive. If a value is not specified, an empty string is substituted when the value of the property is null.

Use the alternate syntax instead if you need to embed the INSERT tag within another HTML tag.

Related information...

- [Alternate syntax for the JSP .91 INSERT tag](#)
- [Examples: JSP .91 INSERT tag syntax](#)
- [Index to API documentation \(Javadoc\)](#)
- [JSP .91 syntax for variable data](#)

4.2.2.3.8.1.1a: JSP .91 syntax: Alternate syntax for the <INSERT> tag



The HTML standard does not permit embedding HTML tags within HTML tags. Consequently, you cannot embed the <INSERT> tag within another HTML tag, for example, the anchor tag (<A>). Instead, use the alternate syntax.

To use the alternate syntax:

1. Use the <INSERT> and </INSERT> to enclose the HTML tag in which substitution is to take place.
2. Specify the bean and property attributes:
 - To specify the bean and property attributes, use the form:
`$(bean=b property=p default=d)`
where *b*, *p*, and *d* are values as described for the <INSERT> tag.
 - To specify the requestparm attribute, use the form
`$(requestparm=r default=d)`
where *r* and *d* are values as described for the <INSERT> tag.
 - To specify the requestattr attribute, use the form
`$(requestattr=r default=d)`
where *r* and *d* are values as described for the <INSERT> tag.

Related information...

- [Example: JSP .91 <INSERT> tag syntax](#)
- [Index to API documentation \(Javadoc\)](#)
- [<INSERT> tag syntax](#)

4.2.2.3.8.1.2: JSP .91 syntax: <REPEAT> tag syntax



```
<repeat index=name start=starting_index end=ending_index>
</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.

Related information...

- [???: JSP .91 REPEAT tag results set and the associated bean](#)
- [Index to API documentation \(Javadoc\)](#)
- [JSP .91 syntax for variable data](#)

4.2.2.3.8.1.2a: JSP .91 syntax: <REPEAT> tag results set and the associated bean



The <REPEAT> tag 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 [<DBQUERY> tag](#)). 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 [<DBQUERY> tag](#) 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 <REPEAT> tag iterates over the rows (beginning at the start row).

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

Static Bean Example	<DBQUERY> 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><repeat index=abc> <insert bean="myBean" property="col1(abc)"> </insert> </repeat></pre> <p>Notes:</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 <REPEAT> tag is used. You can also omit the index on the <REPEAT> tag.• The <REPEAT> tag iterates over the bean properties row by row, beginning with the start row.	<p>JSP file</p> <pre><dbconnect id="conn" userid="alice"passwd="test" url="jdbc:db2:sample" driver="COM.ibm.db2.jdbc.app.DB2Driver" </dbconnect> <dbquery id="dynamic" connection="conn" > Select * from myTable; </dbquery> <repeat index=abc> <insert bean="dynamic" property="col1(abc)"> </insert> </repeat></pre> <p>Notes:</p> <ul style="list-style-type: none">• The bean (dynamic) is generated by the <DBQUERY> tag and does not exist until the tag 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 <REPEAT> tag is used. You can also omit the index on the <REPEAT> tag.• The <REPEAT> tag 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 <REPEAT> tag. The examples produce the same output if all indexed properties have 300 or fewer elements. If there are more than 300 elements, Examples 1 and 2 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>
<repeat>
  <tr><td><insert bean=serviceLocationsQuery property=city></insert></tr></td>
  <tr><td><insert bean=serviceLocationsQuery property=address></insert></tr></td>
  <tr><td><insert bean=serviceLocationsQuery property=telephone></insert></tr></td>
</repeat>
</table>
```

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

```
<table>
<repeat index=myIndex start=0 end=2147483647>
  <tr><td><insert bean=serviceLocationsQuery
property=city(myIndex)></insert></tr></td>
  <tr><td><insert bean=serviceLocationsQuery
property=address(myIndex)></insert></tr></td>
  <tr><td><insert bean=serviceLocationsQuery
property=telephone(myIndex)></insert></tr></td>
</repeat>
</table>
```

The JSP compiler for the Application Server Version 3 is designed to prevent the `ArrayIndexOutOfBoundsException` with explicit indexing. Consequently, you do not need to place JSP variable data syntax before the `<INSERT>` tag to check the validity of the index.

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>
<repeat index=myIndex end=299>
  <tr><td><insert bean=serviceLocationsQuery property=city></insert></tr></td>
  <tr><td><insert bean=serviceLocationsQuery
property=address(myIndex)></insert></tr></td>
  <tr><td><insert bean=serviceLocationsQuery
property=telephone(myIndex)></insert></tr></td>
</repeat>
</table>
```

Nesting `<REPEAT>` tags

You can nest `<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 `<REPEAT>` blocks are nested to display the list of songs on each compact disc in the user's shopping cart.

```
<repeat index=cdindex>
  <h1><insert bean=shoppingCart property=cds.title></insert></h1>
  <table>
  <repeat>
    <tr><td><insert bean=shoppingCart property=cds(cdindex).playlist></insert>
    </td></tr>
  </table>
</repeat>
</repeat>
```

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [<REPEAT> tag syntax](#)

4.2.2.3.8.2: JSP .91 syntax: JSP tags for database access



The Application Server Version 3.5 extends JSP 0.91 support by providing a set of tags for database access. These HTML-like tags make 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 hardcoded 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	<DBCONNECT>	<p>The <DBCONNECT> tag does not establish the connection. Instead, the <DBQUERY> and <DBMODIFY> tags are used to reference a <DBCONNECT> tag 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 <DBCONNECT> tag to the servlet's service() method, which means a new database connection is created for each request for the JSP file.</p> <p>Examples: Employee.jsp example</p>
Avoid hard coding the user ID and password in the <DBCONNECT> tag	<USERID> and <PASSWORD>	<p>Use the <USERID> and <PASSWORD> tags to accept user input for the values and then add that data to the request object where it can be accessed by a JSP file (such as the Employee.jsp example) that requests the database connection.</p> <p>The <USERID> and <PASSWORD> tags must be used within a <DBCONNECT> tag.</p> <p>Examples: None</p>

<p>Establish a connection to a database, submit database queries, and return the results set.</p>	<p><u><DBQUERY></u></p>	<p>The <DBQUERY> tag:</p> <ol style="list-style-type: none"> 1. References a <DBCONNECT> tag in the same JSP file and uses the information provided by that tag to determine the database URL and driver. The user ID and password are also obtained from the <DBCONNECT> tag if those values are provided in the <DBCONNECT> tag. 2. Establishes a new connection 3. Retrieves and caches data in the results object 4. Closes the connection (releases the connection resource) <p>Examples: <u>Basic example</u> <u>Employee.jsp</u> <u>EmployeeRepeatResults.jsp</u></p>
<p>Establish a connection to a database and then add records to a database table.</p>	<p><u><DBMODIFY></u></p>	<p>The <DBMODIFY> tag:</p> <ol style="list-style-type: none"> 1. References a <DBCONNECT> tag in the same JSP file and uses the information provided by that tag to determine the database URL and driver. The user ID and password are also obtained from the <DBCONNECT> tag if those values are provided in the <DBCONNECT> tag. 2. Establishes a new connection 3. Updates a table in the database 4. Closes the connection (releases the connection resource) <p>Examples: <u>Basic example</u> <u>EmployeeRepeatResults.jsp</u></p>

Display query results	<REPEAT> and <INSERT> tags	<p>The <REPEAT> tag loops through each of the rows in the query results.</p> <p>The <INSERT> tag uses the query results object (for the <DBQUERY> tag whose identifier is specified by the <INSERT> bean attribute) and the appropriate column name (specified by the <INSERT> property attribute) to retrieve the value.</p> <p>Examples: Basic example</p>
-----------------------	--	---

- Related information...**
- [Index to API documentation \(Javadoc\)](#)
 - [4.2.2.3.8: IBM extensions to JSP .91 syntax](#)

4.2.2.3.8.2.1: JSP .91 syntax: <DBCONNECT> tag syntax



```
<dbconnect id="connection_id"
    userid="db_user" passwd="user_password"
    url="jdbc:subprotocol:database"
    driver="database_driver_name"
    jndiname="JNDI_context/logical_name"
    xmlref="configuration_file">
</dbconnect>
```

where:

- **id**

A required identifier for this tag. The scope is the JSP file. This identifier is referenced by the connection attribute of the <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 <USERID> and <PASSWORD> for an alternative to hardcoding this information in the JSP file.

- **passwd**

An optional attribute that specifies the user password for the userid. (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 <USERID> and <PASSWORD> for an alternative to hardcoding this attribute in the JSP file.

- **url and driver**

To establish a database connection, the URL and driver must be provided. If these attributes are not specified in the <DBCONNECT> tag, the xmlref attribute or the jndiname attribute must be specified.

The Application Server Version 3 supports connection to JDBC databases and ODBC databases. When connecting to an ODBC database, you can 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.

For a connection to a JDBC database, the URL consists of the following colon-separated elements: jdbc, the sub-protocol name, and the name of the database table 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"
```

If the database is an ODBC database, you can use an ODBC driver or the 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"
```

Note: 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.

Note: If your JSP accesses a different JDBC or ODBC database than the one the Application Server uses for its repository, ensure that you add the JDBC or ODBC driver for the other database to the Application Server's classpath.

- **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 `<DBCONNECT>` tag or in the file specified by the `xmlref` tag.

- **xmlref**

A file (in XML format) that contains the URL, driver, user ID, password information needed for a connection. This mechanism provides Web administrators an alternative method for specifying the user ID and password. It is an alternative to hardcoding the information in a `<DBCONNECT>` tag or reading the information from the request object parameters. This is useful when third-party vendors develop your JSP files and when you need to make quick changes or test an application with a different data source.

When the JSP compiler processes the `<DBCONNECT>` tag, it reads all of the specified tag attributes. If any of the required attributes are missing, the compiler checks for an `xmlref` attribute. If the attribute is specified, the compiler reads the configuration file.

The `xmlref` takes precedence over the `<DBCONNECT>` tag. For example, if the `<DBCONNECT>` tag and the `xmlref` file include values for the URL and the the driver, the values in the `xmlref` file are used.

The configuration file can have any filename and extension that is valid for the operating system. Place the file in the same directory as the JSP that contains the referring <DBCONNECT> tag. An example of a configuration file is:

```
<?xml version="1.0" ?>
<db-info>
  <url>jdbc:odbc:autos</url>
  <user-id>smith</user-id>
  <dbDriver>sun.jdbc.odbc.JdbcOdbcDriver</dbDriver>
  <password>v598m</password>
  <jndiName>jdbc/demo/sample</jndiName>
</db-info>
```

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

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3.8.2: JSP .91 tags for database access](#)

4.2.2.3.8.2.2: JSP .91 syntax: <USERID> and <PASSWD> tag syntax



```
<dbconnect id="connection_id"
  <font color="red"><userid></font><insert requestparm="userid"></insert><font
color="red"></userid></font>
  <font color="red"><passwd></font><insert requestparm="passwd"></insert><font
color="red"></passwd></font>
  url="protocol:database_name:database_table"
  driver="JDBC_driver_name">
</dbconnect>
```

where:

- **<INSERT>**

This tag is a JSP tag for including variable data. See JSP tags for variable data.

- **userid** tag

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.

See the [Login.jsp](#) and the [Employee.jsp](#) examples for an illustration of how to set the USERID and PASSWD using parameters in the request object. The request parameters are set using an HTML form (Login.jsp). In the Employee.jsp, the values of the parameters are passed as hidden form values to the EmployeeRepeatResults.jsp.

- **passwd** tag

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.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3.8.2: JSP .91 tags for database access](#)

4.2.2.3.8.2.3: JSP .91 syntax: <DBQUERY> tag



```
<!-- SELECT commands and (optional) JSP syntax can be placed within the DBQUERY tag.
-->
<!-- Any other syntax, including HTML comments, are not valid. -->
<dbquery id="query_id" connection="connection_id" limit="value" >
</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 <INSERT> tag 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 <DBCONNECT> tag in this JSP file. That <DBCONNECT> tag provides the database URL, driver name, and (optionally) the user ID and password for the connection.

- **limit**

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

- **SELECT command and JSP syntax**

Because the <DBQUERY> tag must return a results set, the only valid SQL command is SELECT. 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.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3.8.2: JSP .91 tags for database access](#)

4.2.2.3.8.2.4: JSP .91 syntax: <DBMODIFY> tag syntax



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

where:

- **connection**

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

- Database commands

Refer to your database documentation for valid database commands.

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 <INSERT> tags.

```
<dbmodify connection="conn" >
insert into EMPLOYEE
    (EMPNO , FIRSTNME , MIDINIT , LASTNAME , WORKDEPT , EDLEVEL )
values
    ('<INSERT requestparm="EMPNO"></INSERT>',
    '<INSERT requestparm="FIRSTNME"></INSERT>',
    '<INSERT requestparm="MIDINIT"></INSERT>',
    '<INSERT requestparm="LASTNAME"></INSERT>',
    '<INSERT requestparm="WORKDEPT"></INSERT>',
    '<INSERT requestparm="EDLEVEL"></INSERT>')
</dbmodify>
```

The [EmployeeRepeatResults.jsp](#) example illustrates this tag.

Displaying query results

To display the query results, use the <REPEAT> and <INSERT> tags. The <REPEAT> tag loops through each of the rows in the query results. The <INSERT> tag uses the query results object (for the <DBQUERY> tag whose identifier is specified by the <INSERT> bean attribute) and the appropriate column name (specified by the <INSERT> property attribute) to retrieve the value. An example is:

```
<repeat>
<tr>
    <td><INSERT bean="empqs" property="EMPNO"></INSERT>
    <INSERT bean="empqs" property="FIRSTNME"></INSERT>
    <INSERT bean="empqs" property="WORKDEPT"></INSERT>
    <INSERT bean="empqs" property="EDLEVEL"></INSERT>
    </td>
</tr>
</repeat>
```

JSP 0.91 APIs and migration

Two interfaces support the JSP 0.91 technology. These APIs provide a way to separate content generation (business logic) from the presentation of the content (HTML formatting). This separation enables servlets to generate content and store the content (for example, in a bean) in the request object. The servlet that generated the context generates a response by passing the request object to a JSP file that contains the HTML formatting. The <BEAN> tag provides access to the business logic.

The interfaces that supported JSP 0.91 for the Application Server Version 3 are:

- `javax.servlet.http.HttpServletRequest.setAttribute()`

Supports setting attributes in the request object. For the Application Server Version 2, this interface was `com.sun.server.http.HttpServiceRequest.setAttribute()`.

- `javax.servlet.http.RequestDispatcher.forward()`

Supports forwarding a response object to another servlet or JSP. For the Application Server Version 2, this interface was `com.sun.server.http.HttpServiceResponse.callPage()`.

Related information...

- [4.2.2.3.8.2.4a: Example: JSP .91 <DBMODIFY> tag syntax](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3.8.2: JSP .91 syntax: JSP tags for database access](#)

4.2.2.2.4.1: Compiling JSP .91 files as a batch



To use the JSP batch compiler for JSP .91 files:

1. Add the following JAR files (found in the Application Server lib directory) to your system classpath:
 - `ibmwebas.jar` (contains the batch compiler classes)
 - `servlet.jar` (contains the Java Servlet 2.1 APIs)
2. At an operating system command prompt, enter the following command on a single line:

```
java com.ibm.servlet.jsp.http.pagecompile.jsp.tsx.batch.JspBatch -s sourceRootDir
-t targetRootDir -c classPath -l libDirectory -v
```

where:

- **sourceRootDir**

The root directory of the paths where the batch JSP compiler will search JSP source files to process. The compiler processes all files with the extension `.jsp` that are in the source root and its subdirectories.

- **targetRootDir**

The root directory of the path where you want the compiler to place the resulting `.java` and `.class` files. The non-batch, JSP 0.91 processor (`PageCompileServlet`) places the `.java` and `.class` files in the path:

```
AS_install_root\temp\servlet_host_name\app_name\pagecompile
```

where *AS_install_root* is the path where the Application Server is installed and *app_name* is the name of the application root folder. It is recommended that you specify that path for the target root if you are batch compiling JSPs to run on your production Application Server. However, if you are batch compiling on a different system and plan to move them to the Application Server later, you can specify any valid target root directory.

If any of the `.class` files have package names, those names will become the names of subdirectories under the target root. For example, if the `.class` name is `security.login.login.class` and the target root is `d:\WebSphere\AppServer\temp\default_host\examples\pagecompile`, the batch compiler places the `.java` and `.class` files in

```
d:\WebSphere\AppServer\temp\default_host\examples\pagecompile\security\login
directory.
```

- **classPath**

An optional parameter that is the fully-qualified path for the classes and Java archives that the compiled classes need. If those resources are in multiple paths, use the semicolon character (`;`) to separate the path names. You do not need to specify the Application Server JAR files on this parameter.

- **libDirectory**

The fully-qualified path to the Application Server `ibmwebas.jar` (contains the JSP batch compiler and related JSP classes) and `servlet.jar` (contains the Java Servlet 2.1 APIs). The default path is *AS_install_root*\lib.

- **-v**

An optional parameter that causes more trace and progress messages to be displayed.

All of the command parameters, except `-v`, are required.

Example

Suppose you want to precompile the JSP files associated with the `examples` application, one of the two applications installed with the application server. If the JSP files are in the path:

```
d:\WebSphere\AppServer\hosts\default_host\examples\web
```

and you want the compiled files to be placed in:

```
d:\WebSphere\AppServer\temp\default_host\examples\pagecompile
```

the command would be (typed on a single line):

```
java com.ibm.servlet.jsp.http.pagecompile.jsp.tsx.batch.JspBatch
```

```
-s d:\WebSphere\AppServer\hosts\default_host\examples\web
-t d:\WebSphere\AppServer\temp\default_host\examples\pagecompile
-c d:\WebSphere\AppServer\hosts\default_host\examples\servlets;d:\devcntr\website
-l d:\WebSphere\AppServer\lib
```

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.2: Developing JSP files](#)
- [4.2.2.2.4: Batch Compiling JSP files](#)

4.2.2.3b: JSP .91 examples



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

(Login.jsp)	An interface for logging in to the application
(Employee.jsp)	A dialog for querying and updating database records
(EmployeeRepeatResults.jsp)	A dialog for displaying update confirmations and query results

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3: Overview of JSP file content](#)

4.2.2.3.8.1.1b: Example: JSP .91 syntax: INSERT tag syntax



Regular syntax

```
<insert bean=userProfile property=username></insert>
<insert requestparam=company default="IBM Corporation"></insert>
<insert requestattr=ceo default="Company CEO"></insert>
<insert bean=userProfile property=lastconnectiondate.month></insert>
```

In most cases, the value of the property attribute will be just the property name. However, you access a property of a property (sub-property) by specifying 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 [<REPEAT> tag](#). Some examples of using the full form of the property attribute:

```
<insert bean=staffQuery property=address(currentAddressIndex)></insert>
<insert bean=shoppingCart property=items(4).price></insert>
<insert bean=fooBean property=foo(2).bat(3).boo.far></insert>
```

Alternate syntax

```
<insert>
  <img src=${bean=productAds property=sale default=default.gif}>
</insert>
```

```
<insert>
  <a href="http://www.myserver.com/map/showmap.cgi?country=${requestparam=country
  default=usa}&city${requestparam=city default="Research Triangle Park"}
  &email=${bean=userInfo property=email}>Show map of city</a>
</insert>
```

Related information...

- [Alternate syntax for the JSP .91 <INSERT> tag](#)
- [Index to API documentation \(Javadoc\)](#)
- [<INSERT> tag syntax](#)

4.2.2.3.8.2.3a: Example: JSP .91 syntax: <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 <INSERT> tag to embed a variable data field. The value of that field is based on user input.

```
<dbquery id="empqs" connection="conn" >
select * from Employee where WORKDEPT='<INSERT requestparm="WORKDEPT"></INSERT>'
</dbquery>
```

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3.8.2.3: JSP .91 <DBQUERY> tag syntax](#)

4.2.2.3.8.2.4a: Example: JSP .91 syntax: <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 <INSERT> tags.

```
<dbmodify connection="conn" >
insert into EMPLOYEE
    (EMPNO, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT, EDLEVEL)
values
    ('<INSERT requestparm="EMPNO"></INSERT>',
    '<INSERT requestparm="FIRSTNME"></INSERT>',
    '<INSERT requestparm="MIDINIT"></INSERT>',
    '<INSERT requestparm="LASTNAME"></INSERT>',
    '<INSERT requestparm="WORKDEPT"></INSERT>',
    '<INSERT requestparm="EDLEVEL"></INSERT>')
</dbmodify>
```

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3.8.2.4: JSP .91 syntax: <DBMODIFY> tag syntax](#)

4.2.2.3.8.2.5a: Example: JSP .91 syntax: <INSERT> and <REPEAT> tags



```
<repeat>
<tr>
    <td><INSERT bean="empqs" property="EMPNO"></INSERT>
    <INSERT bean="empqs" property="FIRSTNAME"></INSERT>
    <INSERT bean="empqs" property="WORKDEPT"></INSERT>
    <INSERT bean="empqs" property="EDLEVEL"></INSERT>
</td>
</tr>
</repeat>
```

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.2.3.8.2: JSP .91 syntax: JSP tags for database access](#)

0.33: What is XML?



Extensible Markup Language (XML) is a framework for defining document markup languages and is predicted to become the primary approach to document exchange over the Internet. In simple terms, a document markup language is a set of elements (frequently called tags) that have one or more of the following functions:

- Describe the structure of the document
- Describe the content of the document
- Control how the document is presented to the user

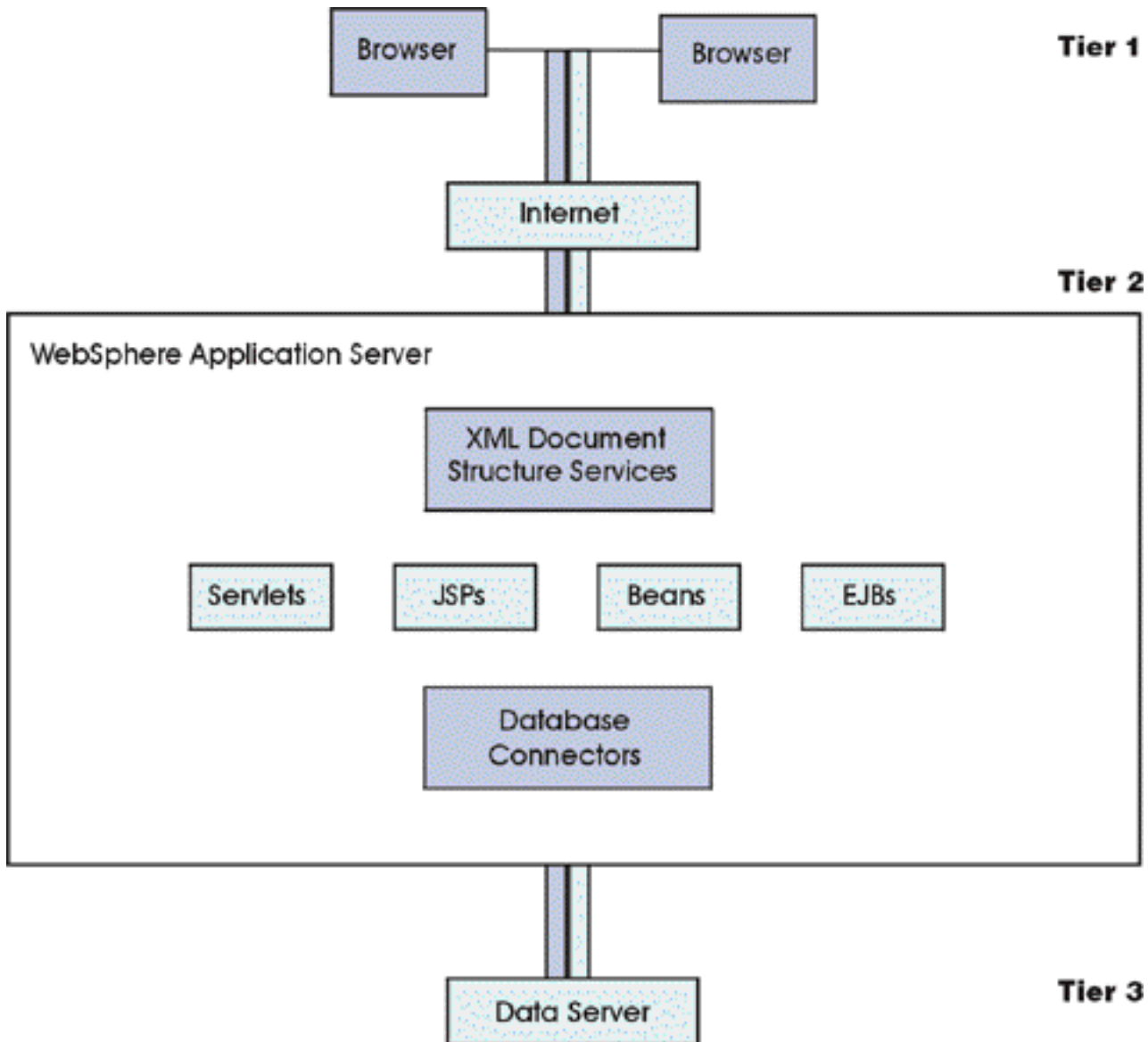
Related information...

- [0.33.1: XML application model](#)
- [0.33.2: XML compared to HTML](#)
- [0.33.3: Advantages of XML](#)
- [0.33.4: What are XML Document Type Definitions \(DTDs\)?](#)
- [0.33.5: What are well-formed XML documents?](#)
- [0.33.6: What are valid XML documents?](#)
- [0.33.7: What are style sheets?](#)
- [0.33.8: What are the Document Object Model \(DOM\) and DOM objects?](#)

0.33.1: XML application model



XML applications can be deployed in a logical three-tier environment as illustrated in the following figure.



- **Tier 1**

XML-enabled user agents parse an XML document, apply stylesheets, and present the document contents. Some user agents convert the XML document to HTML for presentation. Microsoft Internet Explorer 5 is an example of an XML-enabled user agent that can display XML documents without first converting them to HTML. Although only a small number of XML-enabled user agents are available today, their number is expected to grow as the popularity of XML increases.

- **Tier 2**

XML-based servlets and applications provide server-side XML processing. A Web server (an HTTP server) can be configured to serve static XML documents. However, to process and dynamically generate XML documents, the Web server base function must be

extended. The XML Document Structure Services in the Application Server provide such an extension of the Web server and enables Tier 2 servlets (database connectors and integration applications) to parse, generate, manipulate, and validate XML-based dynamic content. This content is sent to Tier 1 and interchanged with other servlets. Tier 2 can also be used to selectively apply stylesheets to XML documents when the tier 1 devices do not support applying XSL stylesheets to XML documents.

- **Tier 3**

The content for dynamically generated XML documents can be retrieved from data servers. Depending on the XML application, the extracted data can be returned as an XML document or returned in JDBC or some other format to a servlet that converts the data to an XML document. In the future, XML-capable JavaBeans for accessing databases should be commercially available.

Related information...

- [0.33: What is XML?](#)

0.33.2: XML compared to HTML



XML and Hypertext Markup Language (HTML) are derived from the more complex Standard Generalized Markup Language (SGML). SGML's complexity and high cost of implementation spurred the interest in developing alternatives.

HTML is the most widely used markup language for Web-based documents. As the popularity of HTML increases, the limitations of the language have become more apparent. Those limitations include restricting the user to a relatively small set of tags. HTML authors cannot create their own HTML tags, because commercially available Web browsers have no knowledge of tags that are not part of the HTML standards.

Another limitation of HTML is tags that control presentation are in the same file with tags that describe the document content. Although HTML 4 and Cascading Style Sheets enable HTML authors to separate content from presentation, HTML 4 remains weak in its ability to describe the content of a document.

XML overcomes limitations of HTML and other markup languages, while providing capabilities that are not a part of the earlier languages. Here's a simple XML document and an HTML document that contains of the same data:

XML document

```
<?xml version="1.0" standalone="yes" ?>
<state stateid="MN">
  <city cityid="12">
    <name>Johnson</name>
    <population>5000</population>
  </city>
  <city cityid="15">
    <name>Pineville</name>
    <population>60000</population>
  </city>
  <city cityid="20">
    <name>Lake Bell</name>
    <population>20</population>
  </city>
</state>
```

HTML document

```
<html>
  <h1 id="MN">State</h1>
  <h2 id="12">City</h2>
  <dl>
    <dt>Name</dt>
    <dd>Johnson</dd>
    <dt>Population</dt>
    <dd>5000</dd>
  </dl>
  <h2 id="15">City</h2>
  <dl>
    <dt>Name</dt>
    <dd>Pineville</dd>
    <dt>Population</dt>
    <dd>60000</dd>
  </dl>
  <h2 id="20">City</h2>
  <dl>
    <dt>Name</dt>
    <dd>Lake Bell</dd>
    <dt>Population</dt>
    <dd>20</dd>
  </dl>
```

</html>

In the XML document, the tag names convey the meaning of the data they contain. The structure of the document is easily discerned and follows a pattern. In contrast, the HTML tag names reveal little about the meaning of their content and the structure is not particularly useful for manipulating the document and exchanging it between applications.

Related information...

- [0.33: What is XML?](#)

0.33.3: XML advantages



XML:

- **Provides more accurate description of document content by enabling an extensible tag set**

XML implementers can define their own tag sets to describe document content. The precision of those descriptions is left to the implementer. For example, one implementation might use a <name> tag and another might find it more useful to use a <city_name> tag.

- **Enables validating document contents against a standardized grammar**

The content and structure of an XML document is defined by its grammar. The Document Type Definition (DTD) is an example of such a grammar. Other XML-based schemas are evolving. *Grammar* is used in this documentation as a generic reference to such schemas.

The grammar describes the valid tags, attributes (characteristics of tags, such as identifiers), and other content for the XML document. Whether an XML document is created as a static file or dynamically generated, the author is responsible for ensuring compliance with the grammar.

- **Makes it easier to exchange documents among users and applications**

Since the early days of computer networking, there's been a need to facilitate the exchange of information among users. The size of potential user populations expands with the use of large networks, such as the Internet, and with the increase in computer-based communication. XML is not the first common document format, but it has advantages over comparable document exchange formats.

XML is the best format for source documents, because it enables delivering content in the most appropriate output format (such as HTML, Portable Document Format, and PostScript) and formats for applications (EDI, electronic data interchange).

- **Supports advanced searching**

It's easier to find something within an XML document because the structure and meaning of the document content are known (as defined in its grammar). One requirement of XML documents is that they must be well-formed. One aspect of well-formedness is that every start tag must have an end tag. This requirement makes it easier to parse and manipulate the documents.

Grammars define structure and content of XML documents and can be used in performing more efficient searches. For example, user agents could support searching a collection of documents that were authored against a particular grammar. Searching by tag names, tag attributes, data content, and location within a document are other search strategies that XML documents make easier to implement.

- **Separates document structure from content from presentation**

Separating document content from document structure is especially important when the content for Web documents must be dynamically generated using programs. Such separation enables Web team members (Web page authors, business logic programmers, and graphics designers) to work in parallel with limited impact on one another's work.

Similar to Cascading Stylesheets in HTML, XSL stylesheets control the presentation of XML documents. XSL stylesheets can be inline or in a separate file. Putting the presentation controls in a separate file from content enables XML implementers to create multiple views for an XML document without changing the document itself.

Content can be presented easily to different users in different forms. For example, an auto parts catalog can be presented to a shopper as a view that includes the prices, descriptions, and order numbers for parts. The catalog view for the auto mechanic could include the information available to shoppers plus schematics that show the position of the installed part. The manufacturer's view could include information about subcomponents and materials.

- **Improves user response, network load, and server load**

XML implementations can have the Web server send an XML document and its associated XSL stylesheets to the client once. Each stylesheet can provide a different view of portions or all of the document data. The user selects the stylesheet to apply. Changing from one view (stylesheet) to the next would not involve sending another request to the server.

- **Supports Unicode**

XML applications support many document encodings, including Unicode double-byte character set (UTF-16) and a compressed version (UTF-8). Therefore, XML documents can include virtually any languages and scripts.

- **Supports advanced linking among documents**

In HTML, the <A> tag links a document to another document or to a target within the same document. Those links are unidirectional (from the source document to the target). The <A> tag includes the address (URL) of the target link and a text label for the link.

In contrast, XML supports two types of advanced linking: XLink and XPointer. The XLink and XPointer standards are evolving. With XLink, any tag can be a link. Optional XLink attributes provide additional information about the link itself and about the target document. Other attributes control how the link is activated and what happens when the link is activated. A single link (called an extended link) can even point to multiple targets.

In HTML, an <A> tag can point to a heading, paragraph, or list within a document. The target section must be a named anchor or tag that permits the ID attribute. In XML, XPointer links can refer to any part of an XML document, even those without identifiers. XPointers link to points in the Document Object Model (DOM) tree (that represents the XML document) and consist of object references, such as root().child(2, address).

XPointers can also point to ranges within a document.

Related information...

- [0.33: What is XML?](#)

0.33.4: What are XML Document Type Definitions (DTDs)?



As discussed in other sections, a DTD describes the content and structure of a specific type of XML document. In other words, the DTD describes the valid XML tags, the order of the tags, whether the tags are optional, the type of data contained within the tags, the tag attributes, the attribute values, processing instructions, entities, and so on.

DTDs are optional, but are almost essential in cases when XML documents must be exchanged among a large number of users. A DTD can be placed inline within the XML document or in an external file.

This section concerns DTDs. However, other grammars are being developed.

Related information...

- [0.33: What is XML?](#)

0.33.5: What are well-formed XML documents?



Well-formed versus valid XML documents

The structure of an XML document is governed by syntax rules for its tag set. There are general rules that are applied to all XML documents to determine whether the document is *well-formed*. To be parsed, an XML document must be well-formed. The XML 1.0 Recommendation describes the rules for well-formed documents, some of which are:

- The first line of the document must be the XML document declaration.
- The document must contain at least one *element* (or *tag*, the more popular term).
- Every starting tag must have a closing tag, such as `<tag></tag>`. The format `<tag/>` is also permitted for tags that do not have content (do not contain data).
- The document must contain a unique opening and closing tag within which all other tags in the document must be nested. For example, in the state XML document, the `<state>` and `</state>` tags are the unique opening and closing tags. All of the other tags are nested within the two tags.
- Tags cannot overlap. For example, for the state XML document, `<name><population></name></population>` is not valid.

XML 1.0 does not specify a set of tags that can appear in an XML document. Instead, XML implementers determine the actual tags that will be permitted in their documents. Those tags are defined formally in an XML-based grammar, such as a Document Type Definition (DTD). A DTD can be within the XML document or in a separate file that is referenced in the document's `<!DOCTYPE>` statement.

Related information...

- [0.33.6: What are valid XML documents?](#)
- [0.33: What is XML?](#)

0.33.6: What are valid XML documents?



An XML document that is well-formed and conforms to the rules specified in its grammar is a valid XML document. XML document validating processors (validators) must report validation errors, but they are not required to end the processing.

Related information...

- [0.33.5: What is well-formed XML](#)
- [0.33: What is XML?](#)

0.33.7: What are style sheets?



XML documents can be exchanged among intelligent applications without human intervention. In such cases, XML implementers do not need to be concerned about document presentation. In contrast, if users need to display a formatted XML document, implementers must determine the presentation format.

XSL stylesheets control how XML documents are presented. Stylesheets control whether all or parts of the document are displayed, as well as aspects of the appearance, such as fonts, color, and alignment.

XML also supports Cascading Style Sheet (CSS), which was developed for use with HTML. CSS is ideal for simple output, while XSL is designed for more complex document formatting. XSL supports advanced features, such as including JavaScript in style sheets, controlling formatting of tag content, and hiding content.

Related information...

- www.w3.org site XSL information
- [0.33: What is XML?](#)

0.33.8: What are the Document Object Model (DOM) and DOM objects?



The Document Object Model (DOM) is a language-independent API for XML and HTML documents. The API represents XML and HTML documents as objects that can be accessed by object-oriented programs (such as Web browsers, document search engines, conversion tools, business logic, and scripting languages). By using the DOM, these programs can create, navigate, manipulate, and modify the documents.

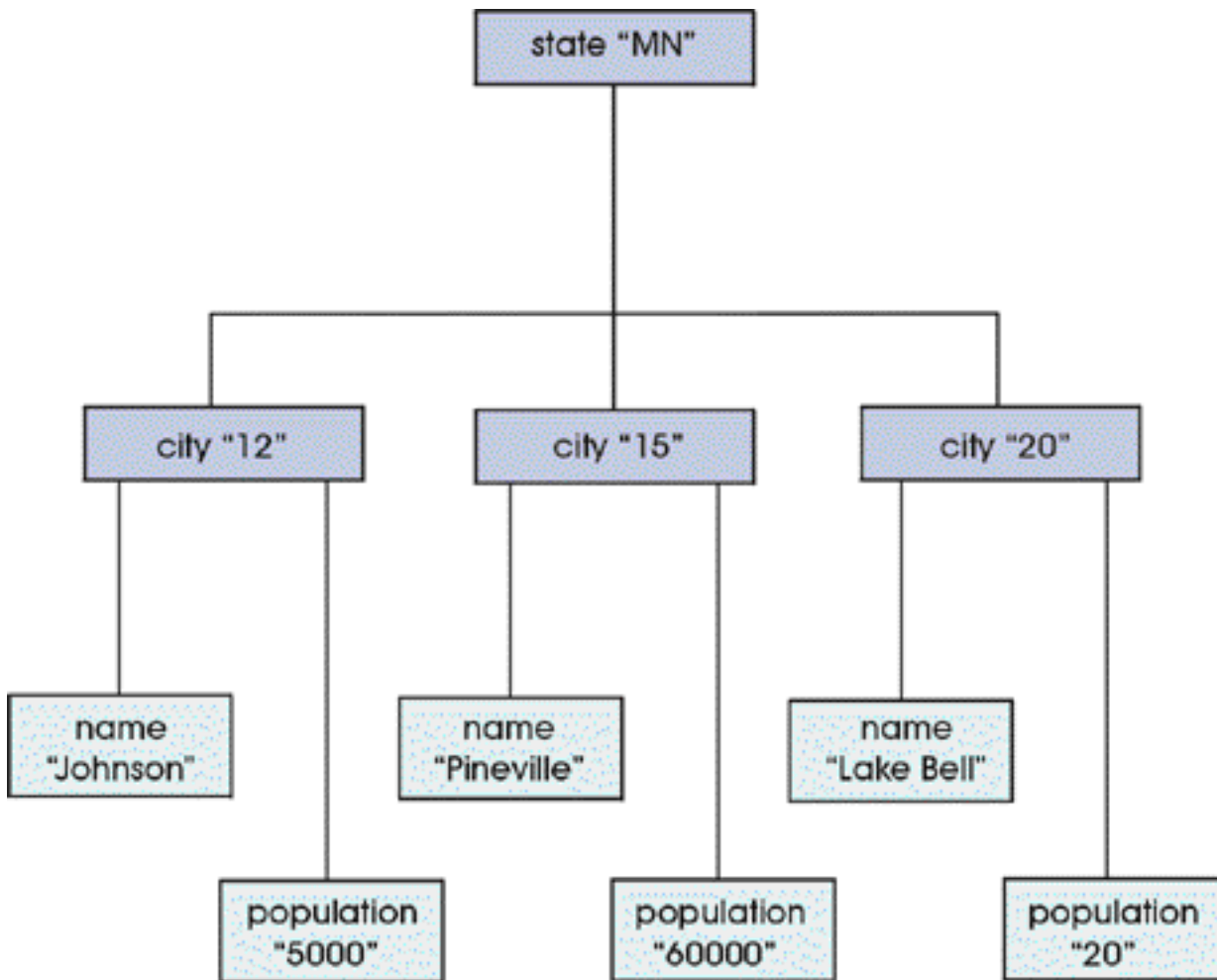
See article 4.1.1 to learn about the supported DOM specification.

DOM objects

The DOM can be used to represent an existing XML document or generate an XML document. The document is stored in computer memory. However, it is not persistent.

In the DOM, a document consists of a collection of Nodes that have parent/child relationships. The Node is the primary object and can be of different types (such as, Document, Element, Attribute, Text, Processing Instruction, CDATA Section, and Comment).

The logical structure of each document has a single Document node, which has no parent and zero or more children that are Element nodes. The following figure shows the DOM for the sample XML state document:



The Document node (state) is the root of the tree. Each of the Element nodes is a tag in the state document. The tag attributes are Attribute nodes. The text (data) within each tag is a Text node.

Related information...

- [4.1.1: Finding supported APIs and specifications](#)
- [0.33: What is XML?](#)

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 4.2.3.5 for all of the details about IBM WebSphere Application Server XML support. If you are just becoming familiar with XML, start with article 0.33, a primer on XML concepts, vocabulary, and uses.

XML is a tagging alternative to HTML that makes it easier to:

- Describe, determine, validate, and search document content
- Exchange information among disparate applications and users, including foreign language users

This documentation provides guidance as to XML document:

- structure -- defining and obeying the syntax for its tag set
- content -- determining the mechanism for filling tags with data
- presentation -- determining the mechanism for formatting and displaying 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 (xml4j.jar and lotusxsl.jar) are automatically added to the appropriate classpath, 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.

To learn which XML API is supported, and what to do if you are using APIs more recent than the supported version, see article 4.1.1.2.

Related information...

- [0.33: What is XML?](#) (and other XML concepts)
- [4.2.3.2: XML document structure](#)
- [4.2.3.3: XML document content](#)
- [4.2.3.4: XML document presentation](#)
- [4.2.3.5: Finding APIs, parsers, validators, and other XML resources](#)
- [4.2.3.6: Using DOM to incorporate XML documents into applications](#)
- [4.2.3.7: XML SiteOutliner example](#)
- [4.1.1.2: Using the newest XML APIs](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2: Building Web applications](#)

4.2.3.2: 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 the Document Type Definition (DTD). At the time of this publication, DTD is the most widely-implemented grammar. Therefore, this section discusses options for using DTDs.

The options for XML document structure are:

- **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.

- **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) the documents will contain.

Related information...

- [0.33.4: What are XML Document Type Definitions \(DTDs\)?](#)
- [0.33.5: What are well-formed XML documents?](#)
- [0.33.6: What are valid XML documents?](#)
- [4.2.3.2.2: Creating or obtaining DTDs](#)
- [4.2.3.2.3: Associating DTDs with XML documents](#)
- [4.2.3.2.4: Parsing XML documents](#)
- [4.2.3.2.5: XML syntax quick reference](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.3: Incorporating XML](#)

4.2.3.2.2: Creating or obtaining DTDs



This section familiarizes you with general and particular DTDs, with the goal of helping you select or create DTDs for your own XML documents. If you are not sure what DTDs or grammars are, navigate up to article 4.2.3.2.

Article 4.2.3.2.2.1 discusses some content basics that apply to all DTDs. See article 4.2.3.2.2.3 for pointers to some existing DTDs to use as examples in developing your own grammars, as well as for creating and validating XML documents of those types.

Related information...

- [4.2.3.2.2.1: Quick reference to DTD contents](#)
- [4.2.3.2.2.2: Example of XML document and its DTD](#)
- [4.2.3.2.2.3: Some existing DTDs](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.3.2: XML document structure](#)

4.2.3.2.2.1: Quick reference to DTD contents



Two of the key declarations in a DTD are the Element and Attribute declarations.

The Element declaration indicates the name and describes the valid contents. The declaration format is:

```
<ELEMENT element_name content_spec >
```

where:

- **element_name**

The tag name

- **content_spec**

The type of content the tag contains, if any. Valid values are:

- EMPTY - This keyword indicates that the tag does not contain content.
- A list of one or more child elements, enclosed within parentheses. As explained later, the order and number of children can be constrained.
- ANY - This keyword indicates that the tag can contain any of the specified child elements.
- (#PCDATA) - This keyword indicates that the tag contains parsed character data (character data that is parsed by the XML processor).
- A combination of PCDATA and children whose names appear in the content model. The order and number of children is not constrained.

The list of valid children is enclosed within parentheses. The list is called the *content model* for the element. There are several ways to specify an element content model:

- **Specify a sequence of children**

The children names appear in order, separated by commas. In this example, the city element must contain one name element and one population element, in that order:

```
<!ELEMENT city (name, population) >
```

- **Specify a choice of children**

The symbol | indicates that any one of the children in the |-separated list can appear within the element. In this example, the city element can contain one name or one population element, but not both:

```
<!ELEMENT city (name | population) >
```

- **Specify the occurrences of children**

Optional characters indicate the occurrence of children. The characters are + (one or more), * (zero or more), and ? (zero or one time).

In this example, the city element must contain at least one name element and, optionally, a population element:

```
<!ELEMENT city (name+, population?) >
```

Each child element must also have its own ELEMENT declaration. For example:

```
<!ELEMENT state (city)+ >
<!ELEMENT city (name, population?) >

<!ELEMENT name (#PCDATA)>
<!ELEMENT population (#PCDATA)>
```

An element can have zero or more attributes. The Attribute declaration format is:

```
<!ATTLIST element_name attribute_name type default >
```

where:

- **element_name**

The name of the tag of which this is an attribute

- **attribute_name**

The name of the attribute

- **type**

One of the following types: string (any literal string), a set of tokenized types (ID, IDREFS, ENTITY, NMTOKEN, NMTOKENS), and enumerated types (a set of NOTATIONS or NMTOKENS which have been declared).

- **default**

Can be one of three valid keywords:

- #REQUIRED - A value must be specified for the attribute
- #IMPLIED - An attribute value can be omitted
- #FIXED - The value of the attribute must have the specified value

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.3.2.2: Creating or obtaining DTDs](#)

4.2.3.2.2: Example of XML document and its DTD



The following example, shows the state.xml document and its DTD:

state.xml

```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE state SYSTEM "state.dtd">
<state stateid="MN">
<city cityid="mn12">
<name>Johnson</name>
<population>5000</population>
<country/>
</city>
<city cityid="mn15">
<name>Pineville</name>
<population>60000</population>
<country/>
</city>
<city cityid="mn20">
<name>Lake Bell</name>
<population>20</population>
<country/>
</city>
</state>
```

state.dtd

```
<!ELEMENT state (city)+ >
<!ATTLIST state stateid ID #REQUIRED>

<!ELEMENT city (name, population?, numpeople?, country)>
<!ATTLIST city cityid ID #IMPLIED>

<!ELEMENT name (#PCDATA)>
<!ELEMENT population (#PCDATA)>
<!ELEMENT numpeople EMPTY>
<!ATTLIST numpeople id ID #REQUIRED>
<!ELEMENT country (#PCDATA)>
<!ATTLIST country countryid ID #FIXED "US">
```

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.3.2.2: Creating or obtaining DTDs](#)

4.2.3.2.2.3: Some existing DTDs



DTDs and grammars describe the structure and content of a specific type of XML document. Although you can implement XML without using a grammar, grammars are useful in helping to ensure that your XML documents can be successfully exchanged among users outside an enterprise and among applications.

XML validators use the grammar associated with an XML document (typically through a DTD) to validate the document structure. The validator compares the grammar to the XML document. If the document deviates from the syntax rules specified in the grammar, the validator must report the error.

Several industry and interest groups have developed and proposed 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 Application Server. 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 in the path:

[*installation_root*](#)\web\xml\grammar\

Visit the XML Industry Portal at <http://www.xml.org/> for the DTD updates.

Some of the grammars in the library are:

- **Channel Definition Format (CDF)**

Developed by Microsoft Corporation, CDF enables subscription to Web-based channels (a Web site or a portion of a Web site). The subscription can be implemented to have the Web site automatically send the subscriber updates to the channel (using push technology) or transmit the updates at the request of the subscriber (using pull technology). In either case, the subscriber must be able to link to a CDF file on the channel site. The CDF file is an XML document that conforms to the CDF DTD. For more information about CDF, refer to the [CDF Specification](#).

- **Mathematics Markup Language (MathML)**

Developed by the W3C, MathML is a grammar for documents that contain math formulas, notations, and other data. Visit the [W3C Math page](#) for more information about MathML and related efforts.

- **Wireless Markup Language (WML)**

Accessing Web-based documents on mobile devices presents special challenges, such as presenting data on small displays. WML is a grammar for creating documents so that they can be efficiently transmitted and displayed on narrowband devices, such as cellular phones, personal digital assistants (PDAs), and pagers. Visit the [Specifications page](#) at the Wireless Application Protocol Forum Web site.

Related information...

- [4.2.3.7: SiteOutliner example](#) (shows CDF)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.3.2.2: Creating or obtaining DTDs](#)

4.2.3.2.3: Associating DTDs with XML documents



To point to an external DTD:

- In the XML document declaration, specify the attribute `standalone="no"`. An XML document declaration is enclosed within the XML tags for processing instructions, `<? and ?>`.
- In the document type declaration, specify the name of the root element of the XML document and whether the DTD is system or public DTD. For system DTDs, include the location (the absolute or relative URL) of the DTD. For public DTDs, include the name of the DTD and the URL of the repository where the DTD can be found. A document type declaration is enclosed within `<!DOCTYPE>`.

An example of declarations for a system DTD is:

```
<?xml version="1.0" standalone="no" ?>  
<!DOCTYPE state SYSTEM "doc/howto/state.dtd" >
```

An example of declarations for a public DTD is:

```
<?xml version="1.0" standalone="no" ?>  
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.0//EN"  
"http://www.wapforum.org/DTD/wml.xml" >
```

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.3.2: XML document structure](#)

4.2.3.2.4: Parsing XML documents



See article 4.2.3.2.4.1 for an overview of parser configurations and instructions for creating parsers. The subsequent articles discuss how to add sophistication to your parser in terms of error handling and performance.

Article 4.2.3.2.4.4 describes new parser methods, with respect to parsers predating the XML API 2.0.x. Please note, the XML API 2.0.x in itself is not very new. In fact, newer XML APIs are available and in use. See article 4.1.1.2 for details.

Related information...

- [4.2.3.2.4.1: Creating an XML parser](#)
- [4.2.3.2.4.2: Enabling XML parsers to handle errors without failing](#)
- [4.2.3.2.4.3: Improving XML parser processing time](#)
- [4.2.3.2.4.4: New XML parser methods in XML for Java Version 2.0.x](#)
- [4.1.1.2: Using the newest XML APIs](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.3.2: XML document structure](#)

4.2.3.2.4.1: Creating an XML parser



The API Version 2.0.x provides six parser configurations:

- Four parser configurations new to the API Version 2.0.x:
 - Non-validating SAX parser (`com.ibm.xml.parsers.SAXParser`)
 - Validating SAX parser (`com.ibm.xml.parsers.ValidatingSAXParser`)
 - Non-validating DOM parser (`com.ibm.xml.parsers.NonValidatingDOMParser`)
 - Validating DOM parser (`com.ibm.xml.parsers.DOMParser`)
- Two parser configurations first provided with the API Version 1.1.x:
 - Non-validating TX compatible parser (`com.ibm.xml.parser.Parser`)
 - Validating TX-compatible SAX parser (`com.ibm.xml.parser.Parser`)

There are two ways to create an instance of one of the API Version 2.0x parsers: passing a parser name or explicitly instantiating a parser class.

Related information...

- [4.2.3.2.4.1.1: Passing a parser name to create an XML parser](#)
- [4.2.3.2.4.1.2: Explicitly instantiating an XML parser](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.3.2.4: Parsing XML documents](#)

4.2.3.2.4.1.1: Passing a parser name to create an XML parser



This method is useful when your XML application must support switching between different parsers. The steps for creating a parser instance using this method are:

1. Import the packages for the parser, the SAX Parser, and the SAX ParserFactory.
2. Create a string that contains the fully-qualified name of the parser class.
3. Pass the string to the parser factory method to instantiate the parser.

An example of creating a validating DOM parser using this method is:

```
import org.xml.sax.Parser;
import org.xml.sax.helpers.ParserFactory;
import com.ibm.xml.parsers.DOMParser;
import org.w3c.dom.Document;
...
String parserClass = "com.ibm.xml.parsers.DOMParser";
String xmlFile = "input_file_URL";
Parser parser = ParserFactory.makeParser(parserClass);
try {
    parser.parse(xmlFile);
} catch (SAXException se) {
    se.printStackTrace();
} catch (IOException ioe) {
    ioe.printStackTrace();
}
// The next line is only for DOM Parsers
Document doc = ((DOMParser) parser).getDocument();
...
```

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.3.2.4.1: Creating an XML parser](#)

4.2.3.2.4.1.2: Explicitly instantiating an XML parser



If your XML application will use a specific parser, use this method to create the parser:

1. Import the parser package.
2. Instantiate the parser.

An example of creating a validating DOM parser using this method is:

```
import com.ibm.xml.parsers.DOMParser;
import org.w3c.com.Document;
...
String xmlFile = "input_file_URL";
DOMParser parser = new DOMParser();
try {
    parser.parse(xmlFile);
} catch (SAXException se) {
    se.printStackTrace();
} catch (IOException ioe) {
    ioe.printStackTrace();
}

// The next line is only for DOM Parsers
Document doc = parser.getDocument();
...
```

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.3.2.4.1: Creating an XML parser](#)

4.2.3.2.4.2: Enabling XML parsers to handle errors without failing



When you create a parser instance, the default error handler is not automatically registered. Consequently, the program will fail when it encounters an error. To register an error handler with the parser, supply a class that implements the `org.xml.sax.ErrorHandler` interface. This requirement applies to SAX-based and DOM-based parsers.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.3.4: Parsing XML documents](#)

4.2.3.2.4.3: Improving XML for Java Version 2.0.x parser processing time



The API Version 2.0.x validating DOM parser (`com.ibm.xml.parsers.DOMParser`) and the nonvalidating DOM parser (`com.ibm.xml.parsers.NonValidatingDOMParser`) support two approaches to expanding DOM tree nodes:

- Full expansion

Creates all nodes of the DOM tree by the end of the parsing. To invoke this type of node expansion, specify the parser's `setNodeExpansion(full)` method.

- Deferred expansion

Nodes in the DOM tree are created only when they are accessed. For example, `getDocument` returns a DOM tree that contains only the Document node. When your application accesses a child of the Document node, the children of Document are created. The immediate children of a Node are created when any of the Node's children are accessed. This shortens the time it takes to parse an XML file and create a DOM tree. Although this approach decreases the time required to parse an XML document and create a DOM tree, it increases the time required to access a node the first time. After a node has been created, it is cached so that subsequent accesses are faster.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.3.2.4: Parsing XML documents](#)

4.2.3.2.4.4: New methods for the XML for Java API Version 2.0.x parsers



The following table summarizes the new methods available to all six API Version 2.0.x parsers via the base class XMLParser:

New Method	Description
setAllowJavaEncodingName	If set to true, allows the Java language names for encoding schemes to be used. If false, you can only use names defined by the XML standard.
setWarningOnDuplicateAttDef	If set to true, warns of duplicate attribute definitions.
setCheckNamespace	If set to true, checks the syntax of namespaces.
setContinueAfterFatalError	If set to true, continues processing after a fatal error.
setDocumentTypeHandler	Sets the XMLDocumentHandler.
setEntityHandler	Sets the EntityHandler.
setValidationHandler	Sets the ValidationHandler.
setDocumentHandler	Sets the SAX DocumentHandler.
setLocale	Sets the locale (language) to use for messages.
setEntityResolver	Sets the SAX EntityResolver.
setDTDHandler	Sets the SAX DTDHandler.
setErrorHandler	Sets the SAX ErrorHandler.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.3.2.4: Parsing XML documents](#)

4.2.3.2.5: XML syntax quick reference



Use this table as a quick reference to XML syntax. The Java code is based on the TX compatibility classes.



The APIs below are significantly back-level compared to the XML API versions currently available. If you use them, it is likely you will need to migrate your code to the newer APIs in the near future. See the Related information below for a full discussion.

XML object	XML syntax	Example Java code based on TX compatibility classes
XML declaration	<code><?xml version="1.0" encoding="ISO-8859-1" ?></code>	<pre>TXDocument doc = new TXDocument(); doc.setVersion("1.0"); doc.setEncoding("ISO-8859-1");</pre>
Processing instruction	<code><?PINname data?></code>	<pre>TXPI pi = (TXPI) doc.createProcessingInstruction ("PIname", "data");</pre>
Processing instruction	<code><?PINname?></code>	<pre>TXPI pi = (TXPI) doc.createProcessingInstruction ("PIname", "");</pre>
XML comment	<code><-- comment text --></code>	<pre>TXComment comm = (TXComment) doc.createComment ("comment text");</pre>
Inline DTD declaration	<code><!DOCTYPE state [...]></code>	<pre>DTD dtd = doc.createDTD("state", null); dtd.addElement(...);</pre>
External DTD declaration	<code><!DOCTYPE state SYSTEM "state.dtd"></code>	<pre>DTD dtd = doc.createDTD("state", new) ExternalID("state.dtd");</pre>
Empty root element	<code><!ELEMENT state EMPTY></code>	<pre>ElementDecl ed = factory.createElementDecl("state", factory.createContentModel(ElementDecl.EMPTY));</pre>
Element content model	<code><!ELEMENT state (#PCDATA city)*;</code>	<pre>CMNode model = new CM1op('*', new CM2op(' ', new CM2op(' ', new CMLeaf("#PCDATA"), new CMLeaf("city"))); ContentModel cm = factory.createContentModel(model); ElementDecl ed = factory.createElementDecl("state", cm);</pre>
Element content model	<code><!ELEMENT state (#PCDATA city)*;</code>	<pre>ContentModel cm = factory.createContentModel(ElementDecl.MODEL_GROUP); cm.setPseudoContentModel("(#PCDATA FOO BAR)*"); ElementDecl ed = factory.createElementDecl("ROOT", cm);</pre> <p>A DTD that includes this instance cannot be used for validation. It can be used for only printing.</p>
Element content model	<code><!ELEMENT state (name?, (population numpeople)+, street*);</code>	<pre>CMNode model = new CM2op(',', new CM2op(',', new CM1op('?', new CMLeaf("capitol")), new CM1op('+', new CM2op(' ', new CMLeaf("population"), new CMLeaf("numpeople")))), new CM1op('*', new CMLeaf("city"))); ContentModel cm = factory.createContentModel(model); ElementDecl ed = factory.createElementDecl("state", cm);</pre>
Element content model	<code><!ELEMENT state (name?, (population numpeople)+, street*);</code>	<pre>ContentModel cm = factory.createContentModel(ElementDecl.MODEL_GROUP); cm.setPseudoContentModel("(capitol?, (population numpeople)+, city*"); ElementDecl ed = factory.createElementDecl("state", cm);</pre> <p>A DTD that includes this instance cannot be used for validation. It can be used for only printing.</p>
Attribute declaration	<code><!ATTLIST state att1 CDATA #IMPLIED att2 (A B O AB) "A"></code>	<pre>Attlist al = factory.createAttlist("state"); AttDef ad = factory.createAttDef("att1"); ad.setDeclaredType(AttDef.CDATA); ad.setDefaultType(AttDef.IMPLIED); al.addElement(ad); ad = factory.createAttDef("att2"); ad.setDeclaredType(AttDef.NAME_TOKEN_GROUP); ad.addElement("A"); ad.addElement("B"); ad.addElement("O"); ad.addElement("AB"); ad.setDefaultStringValue("A"); al.addElement(ad);</pre> <p>A DTD that includes this instance cannot be used for validation. It can be used for only printing.</p>

Entity declaration for inline DTD	<code><!ENTITY version "1.1.6">></code>	<code>Entity ent = factory.createEntity("version", "1.1.6", false);</code>
Entity declaration for external DTD	<code>Entity ent = factory.createEntity("version", new ExternalID("versionent"), null);</code>	<code>Entity ent = factory.createEntity("version", "1.1.6", false);</code>
General entity reference	<code>&version;</code>	<code>GeneralReference gr = factory.createGeneralReference("version");</code>
General entity reference for unparsed data	<code><!ENTITY logoicon SYSTEM "logo.gif" NDATA gif></code>	<code>Entity ent = factory.createEntity("logoicon", new ExternalID("logo.gif"), "gif");</code>
Notation declaration	<code><!NOTATION gif SYSTEM "browser.exe"></code>	<code>TXNotation no = doc.createNotation("gif", new ExternalID("browser.exe"));</code>
XML element and content	<code><state att1="stateid">any text</state></code>	<code>TXElement el = factory.createElement("state"); el.setAttribute("stateid", "MN"); el.addElement(factory.createText("any text"));</code>
CDATA section	<code><![CDATA[any text]]></code>	<code>TXCDATASection cd = factory.createCDATASection("any text");</code>

- Related information...**
- [4.1.1.2: Support for the newest XML APIs](#)
 - [Index to API documentation \(Javadoc\)](#)
 - [4.2.3.2: XML document structure](#)

4.2.3.3: 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, JavaBeans, and even inline Java code within a JavaServer Page (JSP) 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.

Related information...

- [4.2.3.3.1: Using document factory methods to generate XML document content](#)
- [4.2.3.3.2: Extracting XML document content from a database](#)
- [4.2.3.3.3: Using TXCatalog and XML Catalog for public identifier resolution](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.3: Incorporating XML](#)

4.2.3.3.1: Using document factory methods to generate XML document content

The basic steps for generating an XML document are:

1. Create a Document object.
2. Create the XML declarations, root element, child elements, element attributes and other document objects.
3. Append the objects to the root element and its descendants.

Related information...

- [4.2.3.3.1.1: Example: Extracting XML document content from a database](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.3.3: XML document content](#)

4.2.3.3.1.1: Example: Using document factory methods to generate XML document content



The following example uses the TX compatibility classes to create a state XML document that conforms to the state DTD.

```
import com.ibm.xml.parser.TXDocument;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Text;
import java.io.PrintWriter;

public class MakeStateDoc {
public static void main(String[] argv) {
    try {
        // Create Document object
        Document doc = new TXDocument();

        // Make tag as root, and add it
        Element root = doc.createElement("state");
        root.setAttribute("stateid", "MN");

        // Make element and add it
        Element elem = doc.createElement("city");
        elem.setAttribute("cityid", "mn12");
        root.appendChild(elem);

        // Make element and add it
        elem = doc.createElement("name");
        elem.appendChild(doc.createTextNode("Johnson"));
        root.appendChild(elem);

        // Make element and add it
        elem = doc.createElement("population");
        elem.appendChild(doc.createTextNode("5000"));
        root.appendChild(elem);

        // State has city, name, and population
        doc.appendChild(root);

        // Show the XML document
        ((TXDocument) doc).setVersion("1.0");
        ((TXDocument) doc).printWithFormat(new PrintWriter(System.out));

    } catch (Exception e) {
        e.printStackTrace();
    }
}
}
```

The MakeStateDoc class generates a state XML document and displays the document at the command prompt:

```
C:\>java MakeStateDoc
<?xml version="1.0"?>
<state stateid="MN">
```

```
<city cityid="mn12"/>
<name>Johnson</name>
<population>5000</population>
</state>
```

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.3.3.1: Extracting XML document content from a database](#)

4.2.3.3.2: Extracting XML document content from a database



In many cases, the content for your XML documents is in a database. As XML gains popularity, vendors will develop applications that make it easy to extract data and generate XML documents. For example, you could use the IBM database tag extensions to the JSP specification to access a database from a JSP file and use JSP syntax to embed Java code that creates the output XML document.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.3.3: XML document content](#)

4.2.3.3.3: Using TXCatalog and XML Catalog for public identifier resolution

In many cases, such as when specifying the DTD to use for validating your document, you may include XML public identifiers. Including such identifiers introduces parsing complexity because of the need to resolve external identifiers, possibly leading to unwanted network accesses.

Rather than having a parser attempt to resolve these references itself, you can provide it with an XML catalog for looking up the system identifier for each public identifier.

The supported XML API allows two catalog file formats:

- SGML Open catalog (TXCatalog), supported in the API Version 1.1.x
- [XML Catalog draft proposal](#)

Using TXCatalog

Using TXCatalog requires an SGML Open catalog file. IBM WebSphere Application Server includes such a catalog:

[installation_root](#)\web\xml\grammar\dtd\dtd.cat

To use the TXcatalog, set a TXCatalog instance as the EntityResolver for the parser, for example:

```
XMLParser parser = new DOMParser();
Catalog catalog = new TXCatalog(parser.getParserState());
parser.getEntityHandler().setEntityResolver(catalog);
```

After the catalog is installed, catalog files that conform to the TXCatalog format can be appended to the catalog by calling the loadCatalog method on the parser or the catalog instance.

For example, the following code loads two catalogs:

```
parser.loadCatalog(new InputSource("catalogs/cat1.xml"));
parser.loadCatalog(new InputSource("http://host/catalogs/cat2.xml"));
```

Using XML Catalog

Using XML Catalog requires a catalog in XML Catalog format. IBM WebSphere Application Server includes such a catalog:

[installation_root](#)\web\xml\grammar\dtd\xmlcatalog\xmlcatalog.dtd

The example catalog supports the XML Catalog draft proposal 0.1 and 0.2.

If you create your own XML Catalog to use with XML4J API Version 2.0.x, adhere to the following restrictions:

- Follow the XML Catalog grammar for draft 0.2
- Specify the `<!DOCTYPE>` declaration with the PUBLIC attribute set to `"-//DTD XCatalog//EN"` or ensure that the SYSTEM attribute points to the xmlcatalog.dtd file mentioned above.

- Specify the <XCatalog> document root.
- Specify the Version attribute as: `Version (0.1 | 0.2) "0.2"`

To use the XML Catalog, set an XCatalog instance in the EntityResolver of the parser, for example:

```
XMLParser parser = new DOMParser();
Catalog catalog = new TXCatalog(parser.getParserState());
parser.getEntityHandler().setEntityResolver(catalog);
```

After the catalog is installed, catalog files that conform to the XML Catalog format can be appended to the catalog by calling the loadCatalog method on the parser or the catalog instance.

For example, the following code loads two catalogs:

```
parser.loadCatalog(new InputSource("catalogs/cat1.xml"));
parser.loadCatalog(new InputSource("http://host/catalogs/cat2.xml"));
```



No error checking is done to avoid circular Delegate or Extend references. Do not specify catalog files that reference each other.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.3.3: XML document content](#)

4.2.3.4: XML document presentation



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 JavaScripts 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 Lotus XSL processor included with the Application Server Version 3 supports such server-side functions.

Related information...

- [4.2.3.4.1: Using XSL to convert XML documents to other formats](#)
- [4.2.3.4.2: Converting XML documents to HTML at the Web server](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.3: Incorporating XML](#)

4.2.3.4.1: Using XSL to convert XML documents to other formats



The Application Server Version 3 includes the Lotus XSL processor (LOTUSXSL.JAR), a set of APIs for formatting and converting XML documents. The processing can be server-side or at the browser. The transformation can be to HTML or to other formats.

To obtain updates and source code for LOTUSXSL, visit the IBM alphaWorks Web site at <http://alphaworks.ibm.com/>. The download package includes sample code, including the DefaultApplyXSL servlet that applies XSL stylesheets to XML data.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.3.4: XML document presentation](#)

4.2.3.4.2: Converting XML documents to HTML at the Web server



One of the options for presenting an XML document is for the server to convert the XML document to HTML and return the HTML document to the client. This approach is useful if the user agent on the client cannot format an XML document.

The conversion steps include:

1. Create an object to contain the HTML output.
2. Set document headers for MIME type and to prevent caching.
3. Get an output stream to send the HTTP response to the client.
4. Output static HTML content and dynamically generated content to the output object.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.3.4: XML document presentation](#)

4.2.3.5: Finding XML APIs, parsers, validators, and other resources



IBM WebSphere Application Server provides *XML Document Structure Services*, which consist of document parsers, document validators, and document generators for server-side XML processing.

The Document Structure Services support:

- [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](#)
- [Microstar SAX 1.0](#)

The XML Document Structure Services in IBM WebSphere Application Server include:

- XML for Java API Version 2.0.x
- XML for Java API Version 1.1.x (included to support XML applications that were developed under the Application Server Version 2)

Both versions of the XML for Java API are in the xml4j.jar file.

XML4J.JAR for core XML APIs

XML4J.JAR is contained in the [installation_root](#)\lib directory.

XML4J.JAR is an XML processor that provides support for parsing, validating, and generating XML data. The processor implements the base XML, namespace, and DOM W3C recommendations and SAX defacto standard. See the product Javadoc for more information.

XML4J.JAR follows the open source model and is also distributed standalone as the XML Parser for Java. To obtain updates and source code for XML4J and other XML-related resources, visit the IBM alphaWorks Web site at:

<http://alphaworks.ibm.com/>



Newer APIs are available, but not officially supported. However, using Version 2.0.x introduces an imminent need to migrate your code. See article 4.1.1.2 for a discussion.

XML API Version 2.0.x features

- A modular architecture that enhances extensibility by providing interfaces for:

- Pluggable validator
- Pluggable DOM implementation
- Pluggable catalog support
- A choice of four parser configurations, new to the API Version 2.0.x:

Non-validating SAX parser	com.ibm.xml.parsers.SAXParser
Validating SAX parser	com.ibm.xml.parsers.ValidatingSAXParser
Non-validating DOM parser	com.ibm.xml.parsers.NonValidatingDOMParser
Validating DOM parser	com.ibm.xml.parsers.DOMParser

- Support for two parser configurations first provided with the API Version 1.1.x:

Non-validating TX compatible parser	com.ibm.xml.parser.Parser
Validating TX-compatible SAX parser	com.ibm.xml.parser.Parser

- Ability to revalidate a DOM tree after the tree has been modified programmatically
- Improved performance
- Ability to parse larger documents for DOM-based and SAX-based applications
- Support for XML Catalog (an XML-format catalog module new to the API Version 2.0.x) and TXCatalog (first provided with the API Version 1.1.x)

LOTUSXSL.JAR for processing XML documents on the Web server

LOTUSXSL.JAR is contained in the [installation_root](#)\lib directory.

LOTUSXSL.JAR contains the APIs for the Lotus XSL processor, a mechanism for formatting and transforming XML documents on the Web server. To obtain updates and source code for LOTUSXSL, visit the IBM alphaWorks Web site at the URL provided above.

Reference documents for DTD development

For details about the DTD specifications and sample DTDs, refer to:

- IBM [Developers XML Zone](#) for education and other DTD resources
- [W3C XML 1.0 Recommendation](#)
- Local copies of the [Public DTDs](#) installed with IBM WebSphere Application Server
- Retail publications on XML

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.3: Incorporating XML](#)

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 0.33.8 introduces DOM concepts and vocabulary. Article 4.1.1 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. Subsequent articles provide instructions for specific skills, such as generating nodes and revalidating trees.

Related information...

- [0.33.8: What are the Document Object Model \(DOM\) and DOM objects?](#)
- [4.2.3.6.1: Quick reference to DOM object interfaces](#)
- [4.2.3.6.2: Manually generating an XML element node](#)
- [4.2.3.6.3: Revalidating a DOM tree after modifying it](#)
- [4.1.1: Finding supported APIs and specifications](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.3: Incorporating XML](#)

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).

For Nodes, the 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

The Document object represents the entire XML document. The 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 <code><!--</code> and <code>--></code> tags)

<code>createAttribute</code>	Creates an Attribute node of the specified name. Use the <code>setAttribute</code> 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.
<code>createProcessingInstruction</code>	Creates a Processing Instruction with the specified name and data (enclosed within <code><? and ?></code> tags). A processing instruction is an instruction to the application (such as an XML document formatter) that receives the XML document.

The Element node methods include:

Method	Description
<code>getAttribute</code>	Returns the value of the specified attribute or empty string
<code>setAttribute</code>	Adds a new attribute-value pair to the element
<code>removeAttribute</code>	Removes the specified attribute from the element
<code>getElementsByTagName</code>	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

`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.

Related information...

- [4.1.1: Finding supported APIs and specifications](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.3.6: Using DOM to incorporate XML documents into applications](#)

4.2.3.6.2: Manually generating an XML element node



You can create manually any XML element node using the PseudoNode construct:

```
new PseudoNode("literal");
```

If a DOM tree contains PseudoNode instances, you can use the tree for printing only. PseudoNode prevents validation against a DTD.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.3.6: Using DOM to incorporate XML documents into applications](#)

4.2.3.6.3: Revalidating a DOM tree after modifying it



You can revalidate a DOM tree after modifying the tree. For the native API Version 2.0.x, because the DOM implementation prevents the insertion of nodes that are not valid, revalidation is not a useful function for such applications. However, the `TXRevalidatingDOMParser` is a useful enhancement to Version 1.1.x applications.

To revalidate a DOM tree:

1. Import `TXRevalidatingDOMParser` or `RevalidatingDOMParser`.
2. Invoke the validating parser's `validate` method, passing in the name of a DOM node. The validity check is performed on the part of the DOM tree rooted at that node, using the DTD of the current document.

Related information...

- [4.2.3.6.3.1: Example: Revalidating a DOM tree after modifying it](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.3.6: Using DOM to incorporate XML documents into applications](#)

4.2.3.6.3.1: Example: Revalidating a DOM tree after modifying it



In the following example, the TXRevalidatingDOMParser is used to read an XML document, parse the document, modify the document by adding a node that is not valid, and revalidate the modified document:

```
import java.io.IOException;
import com.ibm.xml.parser.TXElement;
import com.ibm.xml.parsers.TXRevalidatingDOMParser;
import org.xml.sax.SAXException;
import org.w3c.dom.Document;
import org.w3c.dom.Node;

public class RevalidateSample {
    public static void main(String args[]) {
        String xmlFile = "input_file_URL";
        TXRevalidatingDOMParser parser = new TXRevalidatingDOMParser();
        try {
            parser.parse(xmlFile);
        } catch (SAXException se) {
            System.out.println("SAX error while parsing: caught "+se.getMessage());
            se.printStackTrace();
        } catch (IOException ioe) {
            System.out.println("I/O Error while parsing: caught "+ioe);
            ioe.printStackTrace();
        }

        Document doc = parser.getDocument();

        System.out.println("Doing initial validation");
        Node position = parser.validate(doc.getDocumentElement());
        if (position == null) {
            System.out.println("ok.");
        } else {
            System.out.println("Invalid at " + position);
            System.out.println(position.getNodeName());
        }

        // Now insert dirty data
        Node junk = new TXElement("invalid_node");
        Node corruptee = doc.getDocumentElement();
        System.out.println("Corrupting: "+corruptee.getNodeName());
        corruptee.insertBefore(junk, corruptee.getFirstChild().getNextSibling());

        System.out.println("Doing post-corruption validation");
        position = parser.validate(doc.getDocumentElement());
        if (position == null) {
            System.out.println("ok.");
        } else {
            System.out.println("Invalid at " + position);
            System.out.println(position.getNodeName());
        }
    }
}
```

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.3.6.3: Revalidating a DOM tree after modifying it](#)

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 some 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.

Related information...

- [WebSphere Samples Gallery](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.3: Incorporating XML](#)

0.8: What are Web applications?



Servlets and other files can belong to a *Web application* (servlet group). Indeed, every servlet in the administrative domain must belong to a Web application whose classpath specifies where to find the servlet class file.

A Web application is comprised of one or more related servlets, JavaServer Pages (JSP) files, and Web pages that can be managed as a unit. For example, you can start and stop the Web application in a single action.

The files in a Web application are related in the sense that they work together to perform a business logic function.

For example, one of the WebSphere Application Server samples is an "Expiring Page" Web application comprised of servlet, Web pages, and JSP files that work together to replace an "expired" Web page with a new one on an expiration date the user specifies.

The Web application is a concept supported by the Java Servlet specification. For IBM WebSphere Application Server purposes, a Web application needs to "wrapped" in a Web application in order to perform important administrative functions, such as securing it.

Related information...

- [0.8.1: What are WAR files?](#)
- [0.8.2: What is WAR file conversion?](#)
- [0.9: What are servlets?](#)
- [0.10: What are JSP files?](#)

4.2: Building Web applications



This section provides considerations, instructions, and tips for creating the building blocks that comprise Web applications. Article 4.2.4 focuses on how to assemble Web applications from these complementary pieces.

Related information...

- [4.2.1: Developing servlets](#)
- [4.2.2: Developing JSP files](#)
- [4.2.3: Incorporating XML](#)
- [4.2.4: Putting it all together](#)
- [4.2.5: Using the Bean Scripting Framework](#)
- [4.2.7: Securing Web applications from the inside and outside](#)
- [4.2.8: Programming high performance Web applications](#)
- [4.2.9: Setting language encoding in Web applications](#)
- [4.2.10: wartowebapp script](#)
- [0.8: What are Web applications?](#)
- [Index to API documentation \(Javadoc\)](#)
- [4: Developing applications](#)

4.2.4: Putting it all together (Web applications)

This section discusses some Web application features, such as data access and security, that can be implemented in a variety of ways.



Related information...

- [4.2.4.2: Obtaining and using database connections](#)
- [4.2.4.4: Providing ways for clients to invoke applications](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2: Building Web applications](#)

0.8.1: What are WAR files?

Web Archive (WAR) files are essentially JAR files containing the various files and configuration information of a Web Application. The WAR file is included as part of the Java Servlet specification.

WAR files are useful for importing complete web applications into a web server engine or to a development environment. While WebSphere Application Server does not support the direct importation of web applications from WAR files into the Application Server, a tool is provided that converts the WAR file to a format using XML that can be imported by the Application Server using either the [console](#) or a [command line](#). A separate [command line statement](#) can be used to convert the WAR file into a webapp format that can be imported into a stand-alone server environment or a development environment such as VisualAge for Java.

Related information...

- [0.8: What are Web applications?](#)
- [0.8.2: What is WAR file conversion?](#)

0.8.2: What is WAR file conversion?



IBM WebSphere Application Server Version 3.5 Fix Pack 2 (also known as Version 3.5.2) introduces support for WAR files. However, Fix Pack 2 has been designed to maintain compatibility with existing applications. As such, WAR files are only used as a deployment vehicle in Version 3.5.2. After a WAR file is installed into the server runtime, the WAR file itself is no longer used.

To install a WAR file into Version 3.5.2, a set of conversion tools are provided. The tools convert a WAR file using the following methodology:

1. Unjar the WAR file.
2. Create an IBM Web Application directory structure (classpath containing a "servlets" directory and a document root directory called "web")
3. Copy the contents of the WEB-INF/lib and WEB-INF/classes directories to the "servlets" directory of the IBM Web Application.
4. Copy the JSP Tag Library .tld files and the remainder of the files in the WAR into the "web" directory of the IBM Web Application.
5. Transform the information in the web.xml deployment descriptor into a format that is understood by Version 3.5.2.

Some tools are available for converting WAR files. See [article 6.6.8](#) for an overview of tools geared for system administrators. See [article 4.2.10](#) for a method geared to WebSphere programmers.

Related information...

- [0.8.1: What are WAR files?](#)
- [0.8: What are Web applications?](#)

6.6.8.1.3: Converting WAR files

To convert WAR files (see article 0.8.2 for a description) using the Java console:

1. Select the Convert WAR File task from the console Tasks menu.
2. Follow the instructions in the task wizard.

You will need to specify the following information:

- The servlet engine where the Web application will reside
- A name for the Web application
- A Web Path for the Web application
- The path to the WAR file

Related information...

- [6.6.8.1.4: Web application properties](#)
- [6.6.8.1.1: Configuring new Web applications](#)
- [0.8.1: What are WAR files?](#)
- [0.8.2: What is WAR file conversion?](#)
- [InfoCenter \(product documentation\)](#)
- [6.6.8.1: Administering Web applications](#)

6.6.0.2.1.4: wartoxmlconfig script

The wartoxmlconfig.(bat|sh) scripts performs from the command line the same function as the [Convert WAR File task](#) on the console. The command line syntax is shown below:

```
wartoxmlconfig [war filename] [webapp destination] [admin node] [node] [server]
[servlet engine] [virtual host] [webapp path] [webapp name]
```

- **war filename** - full path to the WAR file
- **webapp destination** - directory where web application will be rooted (a subdirectory will be created that matches the specified web app name)
- **admin node** -name of the node containing the admin server you are connecting to (as shown in the Admin GUI)
- **node** - name of the node you are installing the WAR on to (as shown in the Admin GUI)
- **server** - name of the server you are installing the WAR on to (as shown in the Admin GUI)
- **servlet engine** - name of the servlet engine you are installing the WAR on to (as shown in the Admin GUI)
- **virtual host** - name of the virtual host you wish this application to be accessible from (as shown in the Admin GUI)
- **webapp path** - the context path of the web application
- **webapp name** - the name of the web application to be shown in the Admin GUI

Example:

```
wartoxmlconfig c:\temp\servlet-tests.war c:\websphere\appserver\hosts\default_host
mynode mynode "Default Server" "Default Servlet Engine" default_host
/servlet-tests Servlet_Tests
```

Related information...

- [0.8.2: What is WAR file conversion?](#)
- [0.8.1: What are WAR files?](#)
- [6.6.8.1.3: Converting WAR files](#)

4.2.10: wartowebapp script

This script converts a [WAR file](#) into a format that can be used by a standalone Servlet engine runtime outside of the full WebSphere administration system. It will not affect the operation of the WebSphere Application Server by configuring new web applications on the server. This script is intended to provide a means for developers to execute a web application from XML on a Servlet Engine inside a development runtime environment (such as VisualAge for Java). The wartowebapp script converts the web.xml file into a .webapp format file used by the Servlet engine. The command line syntax is shown below:

```
wartowebapp [war filename] [webapp destination]
            [virtual host name] [webapp path] [webapp name]
```

where:

- **war filename** - full path to the WAR file
- **webapp destination** - directory where web application will be rooted (a subdirectory will be created that matches the specified web app name)
- **virtual host** - name of the virtual host you wish this application to be accessible from
- **webapp path** - the context path of the web application
- **webapp name** - the name of the web application you wish to use

Example:

```
wartowebapp c:\temp\servlet-tests.war c:\websphere\appserver\hosts\default_host
            default_host /servlet-tests servlet_tests
```

Related information...

- [4.2: Building Web applications](#)
- [0.8.1: What are WAR files?](#)
- [0.8: What are Web applications?](#)

0.14: What is data access?



The table summarizes data access concepts central to IBM WebSphere Application Server:

data store or database	A relational database such as DB2 or Oracle, or another product that manages and accesses data
application database	Holds data accessed by applications managed with IBM WebSphere Application Server
administrative database	Holds data for the IBM WebSphere Application Server administrative server. The same database can hold both application and administrative data
JDBC driver	Software that enables Java applications, such as those supported by the product, to connect to JDBC-compliant databases
JDBC driver configuration	An administrative configuration that specifies the location of the JDBC driver code for connecting Java applications to a database
data source configuration	<p>An administrative configuration that establishes a pool of connections to a database, for use by Java components. It also specifies connection pooling parameters, such as the maximum number of connections to maintain in the pool.</p> <p>A data source instance (and the associated connection pool) is created for every application server instance that uses the data source. The connection pool associated with a data source is in turn shared by all application components (servlets, JSP files, and enterprise beans) that are running in an application server.</p>
connection pooling	A scheme that addresses inefficiencies in obtaining and releasing database connections

Related information...

- [0.14.1: What are connection pools?](#)
- [0.14.3: What is administrative data?](#)
- [0.14.4: What is application data?](#)

0.14.1: What is connection pooling?



JDBC 1.0 defines the Java APIs for access to relational databases. With the introduction of JDBC 2.0, the APIs have been split into two parts:

The JDBC 2.0 Core API	It contains evolutionary improvements, but has been kept small and focused like the JDBC 1.0 API in order to promote ease of use. Code written for the 1.0 API continues to work with the 2.0 API. The 2.0 API classes remain in the java.sql package.
The JDBC 2.0 Standard Extension API	It defines specific kinds of additional functionality when vendors are ready to provide the functionality and programmers are ready to use the functionality. IBM has implemented these new classes and interfaces in new packages. You can use the new classes and interfaces in your applications by using the following import statements: <pre>import javax.sql.*; import com.ibm.ejs.dbm.jdbcext.*;</pre>

Connection pooling is defined as part of the JDBC 2.0 Standard Extension API. Another part of the Standard Extension API provides for the use of the Java Naming and Directory Interface (JNDI) and DataSource objects as an alternative to using DriverManager objects to access relational data servers.

IBM has implemented these extensions starting with IBM WebSphere Application Server Version 3.0x. This product documentation refers to the:

Connection manager/Reference model	Specifies how relational data servers are accessed by a servlet using the JDBC 1.0 APIs. This is the model used in IBM WebSphere Application Server Version 2.x. Connection manager provided a form of connection pooling before the JDBC 2.0 Standard Extension APIs became available.
Connection pooling	Specifies how relational data servers are accessed by a servlet using the new JDBC 2.0 Standard Extension APIs. This is the new model. Servlets running under Application Server Version 3.x can be coded to make efficient use of data server connection resources.

Related information...

- [0.14.1a: Connection pooling advantages](#)
- [0.14.2: How the product manages connection pools](#)
- [0.14: What is data access?](#)

0.14.1a: Connection pooling advantages



Each time a resource attempts to access a database, it must connect to that database. A database connection incurs overhead -- it requires resources to create the connection, maintain it, and then release it when it is no longer required.

The overhead is particularly high for Web-based applications because Web users connect and disconnect more frequently. In addition, user interactions are typically shorter, due to the surfing nature of the Internet.

Often, more effort is spent connecting and disconnecting than is spent during the interactions themselves. Further, because Internet requests can arrive from virtually anywhere, usage volumes can be large and difficult to predict.

To address the problem, WebSphere Application Server establishes a pool of database connections that is shared by applications on application servers.

Connection pooling lets the administrator control and reduce the resources used by Web-based applications. Connection pools spread the connection overhead across several user requests, conserving resources. Connection pooling can also improve the response times of Web-based applications.

When a user makes a request over the Web to a resource, the resource accesses a data source. Because the data source locates and uses an existing connection from the pool, the user request does not incur the overhead of creating a new connection.

Each connection is associated with a particular user request. When the request is satisfied and the response is returned to the user, the data source returns the connection to the connection pool for reuse. Again, the overhead of a disconnect is avoided.

Each user request incurs a fraction of the cost of connecting or disconnecting. After the initial resources are used to produce the connections in the pool, additional overhead is insignificant because the existing connections are reused.

Related information...

- [0.14.2: How the product manages connection pools](#)
- [0.14.1: What is connection pooling?](#)

0.14.2: How the product manages connection pools



IBM WebSphere Application Server establishes and maintains pools of connections as specified by the administrator.

Once the connections are set up, it manages them by parceling out connections in response to user requests and then performing housekeeping operations to maintain a balance between available connections and demand for connections. This ensures that an existing connection is available when a servlet or application server needs a connection.

For example, the connection pool periodically identifies idle or orphaned connections. It terminates idle connections and returns orphaned connections to the connection pool. This means fewer connections are available (and fewer resources are used) when demand for connections is low.

An idle connection is one that has not been used for the amount of time specified in the Idle timeout property of the data source. A connection is orphaned when its owning servlet or application server terminates or does not respond.

The above description applies to the connection pooling model. The alternative model is the connection manager (not recommended).

Related information...

- [0.14.2a: What is the connection manager?](#)
- [0.14: What is data access?](#)

0.14.2a: What is the connection manager?



Starting with IBM WebSphere Application Server Version 3.0, the JDBC 2.0 Standard Extension APIs provide connection pooling capability to make more efficient use of data server connections.

Prior to Application Server Version 3.0, the JDBC 2.0 Standard Extension APIs were not available. The connection pooling capability was provided by the IBM connection manager.

A servlet would communicate with a connection manager that maintained a pool of open data server connections to JDBC or ODBC data server products. When the servlet received a connection from the pool, it could communicate directly with the data server using its APIs.

The connection manager remains supported in Version 3.5 (but deprecated), and it is highly recommended you migrate applications to use the new connection pooling model.

Related information...

- [3.3.8: Migrating to supported database connection APIs \(and JDBC\)](#)
- [0.14.2: How the product manages connection pools](#)

0.14.3: What is administrative data?



IBM WebSphere Application Server has an administrative server that manages data about application configurations, application servers, and other resources known to the product.

The administrative server keeps its data in an administrative database, which can be the same database in which applications keep their data, or a different database.

The administrative database allows centralized administration -- several administrative servers on various machines can share the data.

Related information...

- [6.1.1: Administrative elements](#)
- [0.14: What is data access?](#)

0.14.4: What is application data?



Application data is simply data that applications access. Application building blocks such as servlets can establish database connections to:

- Query databases, possibly modifying their contents
- Store session and user profile data
- Store other data, such as application results

Related information...

- [0.14: What is data access?](#)

0.14.7: What are data access JavaBeans?



The data access JavaBeans are Java classes coded to the JavaBeans specifications. Data access JavaBeans provide a rich set of features and enhanced function with respect to the `java.sql` package, while hiding much of the complexity associated with accessing relational databases.

Because they are JavaBeans, the data access classes can be used in integrated development environments (IDEs), such as the IBM product VisualAge for Java. An IDE allows the programmer to manipulate Java classes in a visual way, rather than editing lines of Java code. Because JavaBeans are also Java classes, programmers can use them like ordinary classes when writing Java code. The IBM WebSphere Application Server Version 3.x documentation generally discusses the latter method.

The data access JavaBeans were available in the WebSphere Application Server prior to the Version 3.x environment. With the introduction of Application Server Version 3.x, the JDBC 2.0 Standard Extension APIs concerned with DataSource objects and connection pooling have also been available.

Using DataSource objects with connection pooling as a source of database connections enables increased efficiency in applications that use data access JavaBeans.

Related information...

- [0.14: What is data access?](#)

0.14.7.1: Features of data access JavaBeans



The data access JavaBeans (in the package `com.ibm.db`) offer:

Feature	Details
Caching of query results	<p>SQL query results can be retrieved all at once and placed into a cache. The application (or servlet) can move forward and backward through the cache or jump directly to any result row in the cache.</p> <p>In the <code>java.sql</code> package, in contrast, rows are retrieved from the database one at a time, in the forward direction only. A newly retrieved row overlays the last retrieved row unless additional code is written to expand functionality.</p> <p>For large result sets, the data access JavaBeans provide ways to retrieve and manage <i>packets</i>, subsets of the complete result set.</p>
Updates through result cache	<p>The servlet can use standard Java statements (rather than SQL statements) to change, add, or delete rows in the result cache. Changes to the cache can be propagated to the underlying relational table.</p>
Query parameter support	<p>The base SQL query is defined as a Java String, with parameters replacing some of the actual values. When the query is run, the data access JavaBeans provide a way to replace the parameters with values made available at runtime.</p> <p>For example, the parameter values might be submitted by the user on an HTML form.</p>
Metadata support	<p>A <code>StatementMetaData</code> object contains the base SQL query. Higher levels of data (<i>metadata</i>) can be added to the object to help pass parameters into the query and work with the returned results. Query parameters can be Java datatypes convenient for the Java program.</p> <p>When the query is run, the parameters are automatically converted using metadata specifications into forms suitable for the SQL datatype.</p> <p>For reading the query results, the metadata specifications can automatically convert the SQL datatype into the Java datatype most convenient for the Java application.</p>

Related information...

- [0.14.7: What are data access JavaBeans?](#)

4.2.4.2: Obtaining and using database connections



IBM WebSphere Application Server Version 3.5 provides two options for accessing database connections from application building blocks, such as servlets:

- Connection pooling (model based on JDBC 2.0)
- Connection manager (now deprecated, model based on JDBC 1.0)

Because connection pooling is the most efficient model for Web applications, it is recommended that you use connection pooling for all new applications requiring database access. You should consider migrating existing applications to connection pooling if your applications use connection manager or the standard JDBC 1.0 methods for getting database connections.

IBM WebSphere Application Server Version 3.5 also provides data access JavaBeans, which offer a rich set of features and functions for working with relational databases. Data access JavaBeans use database connections that you provide, such as through connection pooling.

See the Related information for a description of connection pooling, connection manager, and data access JavaBeans.

Related information...

- [0.14.1: What are connection pools?](#)
- [4.2.4.2.1: Connection pooling with the JDBC 2.0 Standard Extension APIs](#)
- [4.2.4.2.2: Database access with the JDBC 1.0 reference model](#)
- [4.2.4.2.3: Using data access JavaBeans to access relational databases](#)
- [4.2.4.2.4: Database access by servlets and JSP files](#)
- [Index to API documentation \(Javadoc\)](#)
- [3.3.8: Migrating to supported database connection APIs \(and JDBC\)](#)

4.2.4.2.1: Connection pooling with the JDBC 2.0 Standard Extension APIs



The model using the JDBC 2.0 APIs is based on the following steps. See the Related information for a tour through an example servlet that uses the model.

1. Using JNDI, obtain access to the relevant naming service. A naming service is a way to store things - Java objects for example - as well as a way to retrieve them.

Create a Hashtable object that will hold the parameters necessary to access the naming service. This step is typically performed once, in the `init()` method of the servlet.

```
Hashtable parms = new Hashtable();
```

2. Put the parameter information into the Hashtable object. The parameters should be available from the Web administrator. This step is typically performed once in the `init()` method of the servlet.

```
parms.put(Context.INITIAL_CONTEXT_FACTORY,  
"com.ibm.ejs.ns.jndi.CNInitialContextFactory");
```

3. Given the relevant parameters, get the naming context. The naming context will allow you to access things in the naming service. This step is typically performed once, in the `init()` method of the servlet.

```
Context ctx = new InitialContext(parms);
```

4. Use the JNDI naming context to obtain a DataSource object. The DataSource object is defined in the JDBC 2.0 Standard Extension APIs. This step is typically performed once, in the `init()` method of the servlet.

```
DataSource ds = (DataSource)ctx.lookup(source);
```

[Details about the above code example](#)

5. The DataSource object is a "factory" that is used to get an actual JDBC connection object. (A DataSource object is roughly comparable to the DriverManager introduced with JDBC 1.0).

The connection is typically obtained for each client request made to the servlet, typically in the `doGet()` or `doPost()` method.

```
conn = ds.getConnection("userid", "password");
```

6. Given the connection, perform the necessary data server interactions for each client request. This step is typically performed in the `doGet()` or `doPost()` method.

You can continue to use the same coding as with JDBC 1.0, or you can use some of the new JDBC 2.0 Standard Extension APIs.

7. At the end of each client request, free the connection resource. This step is typically performed at the end of the `doGet()` or `doPost()` method.

```
conn.close()
```

[Details about conn.close\(\)](#)

Related information...

- [4.2.4.2.1a: Example: Retrieving an object from a name service](#)
- [4.2.4.2.1b: How the JDBC 1.0 and 2.0 implementations differ in closing connections](#)
- [4.2.4.2.1.1: Implementing connection pooling with servlets](#)
- [4.2.4.2.1.2: Tips for utilizing connection pooling](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.4.2: Approaches to managing database connections](#)

4.2.4.2.1a: Example: Retrieving an object from a name service



The code snippet shows how a naming service is used: Specify a name and retrieve (in this case) a Java object from the service:

```
DataSource ds = (DataSource)ctx.lookup(source);
```

In this case, the source object is a string containing the context and logical name for the lookup. For example, the source object might be something like "jdbc/sample."

The Java object retrieved is a DataSource object stored in the "jdbc" context with the logical name of "sample." The DataSource object is created by a WebSphere administrator and stored in the naming service using an administrative client.

In the code fragment above, the string value of the source object can be read from an external property file, increasing servlet portability.

Alternatively, the WebSphere administrator could provide the context and logical name to use as an explicit string argument in the lookup() method. Vendor specific data server information and site specific information (such as the database name) are therefore hidden from the programmer.

In the code fragment above, the ds object is local to the init() method and is not available to other methods. In actual servlets, the ds object would first be defined as an instance variable *outside* of all methods, and then fully *initialized* within the init() method, making it available for use in all servlet methods.

In the code fragment above, the class name DataSource is shown before the ds variable (making it a local variable) to more clearly identify the type of the ds variable.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.4.2.1: Connection pooling with the JDBC 2.0 Standard Extension APIs](#)

4.2.4.2.1b: How the JDBC 1.0 and 2.0 implementations differ in closing connections

The `close()` method using JDBC 2.0 looks identical to the reference model using JDBC 1.0. Unlike the reference model, a real `Connection` object is not closed, so there is no close overhead.

Instead, the `Connection` object is returned to the connection pool for reuse. In a similar fashion, the earlier `getConnection()` method on the `DataSource` object probably did not go through the overhead of creating a new `Connection` object, but returned an already existing `Connection` object from the connection pool. A new `Connection` object is created only if the connection pool does not have an available `Connection` object.

Environments other than Application Server Version 3.x might lack the automatic connection pooling mechanism underneath a `DataSource` object. In such an environment, every `getConnection()` on a `DataSource` object and the related `close()` might in fact go through the overhead of creating a new connection and closing a connection.

Within Application Server Version 3.x, using a `DataSource` object to get a `Connection` object automatically provides the efficiencies of connection pooling.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.4.2.1: Connection pooling with the JDBC 2.0 Standard Extension APIs](#)

4.2.4.2.1.1: Implementing connection pooling with servlets



All servlets using connection pooling will follow the numbered steps illustrated in the fully working code sample [ConnPoolTest.java](#). Please note, the sample refers to a database table "owner," which you might instead know as the "schema" in terms of your database management product.

Consider displaying the code sample while reading the following steps:

Step	Details	When to perform
1. Import JDBC packages and IBM implemented extensions	Import the JDBC 2.0 Core classes, the IBM implementation of the JDBC 2.0 Standard Extension classes, and the JNDI naming classes. These are the classes needed for data server access.	Perform this step just one time
2. Create the initial naming context	The servlet will use the naming service to get access to a DataSource object placed there by a WebSphere administrator. The WebSphere Administrator will provide the parameters needed to create the naming context.	Perform this step just one time, in the init() method
3. Get a DataSource object	<p>The DataSource object is a "factory" that will be used by all requests to get a connection.</p> <p>Note that the code sample uses a String object named source to do a lookup to get a DataSource object. The text string would have a context part and a logical name part. A typical string might look something like "jdbc/sample" where the context is "jdbc" and the logical name is "sample."</p> <p>The WebSphere administrator can provide the information, which identifies the DataSource object placed in the naming service for use by the servlet programmer. Use the context and logical name appropriate for your servlet.</p>	Perform this step just one time, in the init() method
4. Get a Connection object using the DataSource factory	The connection is automatically a connection from the connection pool. Each individual client request gets its own connection.	Perform this step one time for each request, usually in the doGet() or doPost() method

<p>5. Interact with the data server</p>	<p>The servlet interacts with the data server -- retrieving data, updating data, and so on -- using methods of the connection object and objects derived from the connection. You can use the JDBC 2.0 Core APIs or JDBC 2.0 Standard Extension APIs.</p> <p>If you use the data connection for more than one data server interaction within the same user request, and the request extends over a period of time approaching the maximum age parameter, the connection will be considered an orphan connection and be taken away from the servlet. This behavior is a "preemption" feature that can be configured by the WebSphere administrator.</p> <p>In the event of preemption, the servlet might get a <code>ConnectionPreemptedException</code>. Consider adding code to handle the exception. One possible response is to use the <code>DataSource</code> object to get another connection.</p> <p>With proper connection pool tuning and servlet design, preemption should not be a problem to most servlets.</p>	<p>Perform this step one time for each request, usually in the <code>doGet()</code> or <code>doPost()</code> method</p>
<p>6. Close the connection</p>	<p>The servlet invokes the <code>close()</code> method on the connection, which does not actually close the connection but simply frees it and returns it to the connection pool for use by another servlet.</p> <p>Note that the close is done in a finally block, so you are assured that the close will take place regardless of whether the data server interactions are successful or cause an exception. Such coding will reduce the possibility of orphan connections.</p>	<p>Perform this step one time for each request, usually in the <code>doGet()</code> or <code>doPost()</code> method</p>
<p>7. Prepare and send the response</p>	<p>The servlet prepares and returns the response to the user request. In this step you will probably not be using any JDBC APIs.</p>	<p>Perform this step for each request, usually in the <code>doGet()</code> or <code>doPost()</code> method</p>

- Related information...**
- [4.2.4.2.1.2: Tips for coding connection pools](#)
 - [Index to API documentation \(Javadoc\)](#)
 - [4.2.4.2.1: Connection pooling with the JDBC 2.0 Standard Extension APIs](#)

4.2.4.2.1.2: Tips for utilizing connection pooling



Consider the following tips when developing applications that will use connection pools:

- Ensure that the appropriate files are in the system CLASSPATH to successfully compile the ConnPoolTest sample servlet, and any servlets you write. (See the Related information).
- Given a DataSource object ds, use one of two versions of the getConnection() method to obtain a connection:

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

```
Connection conn = ds.getConnection(useridString, passwordString);
```

In the first version, the default database user ID and password are used. In the second version, specify explicit character strings for the user ID and password.

- Consider coding for the possibility of a ConnectionWaitTimeoutException when getting a connection. This exception can happen if the pool is empty when the connection request is made, and no free connections are returned to the pool within the time specified by the connection timeout parameter.
- Note that the [ConnPoolTest sample servlet](#) reads in some information from an external property file, improving servlet portability. Consider writing your own servlets to promote portability.
- Note that a servlet does not access the connection pool directly. The servlet requests a connection through a DataSource object and the connection pooling is provided automatically.

Related information...

- [4.2.4.2.1.1: Implementing connection pooling with servlets](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.4.2.1: Connection pooling with the JDBC 2.0 Standard Extension APIs](#)

4.2.4.2.2: Database access with the JDBC 1.0 reference model



The reference model using the JDBC 1.0 APIs, which will still work under JDBC 2.0 and Application Server Version 3.x, is based on the code fragments shown in the following steps.

1. Load the driver for a specific relational database product. The specific driver class should be available from the WebSphere administrator.

This step is typically performed once, during the `init()` method of the servlet.

```
Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
```

2. Use the static `getConnection()` method of the `DriverManager` class to get a JDBC Connection to the relational database product, again using parameters for a specific database product. The WebSphere administrator can provide the subprotocol, database, userid, and password information.

This step is performed for each client request made to the servlet, typically in the `doGet()` or `doPost()` method. (The subprotocol and database information are combined into what is called the database URL, shown as "jdbc:subprotocol:database" below.)

```
Connection conn =
    DriverManager.getConnection("jdbc:subprotocol:database", // database URL
                               "userid",
                               "password");
```

3. Given the connection, do the necessary data server interactions for each client request. This step is typically performed in the `doGet()` or `doPost()` method.
4. At the end of each client request, free the connection resource. This step is typically performed at the end of the `doGet()` or `doPost()` method.

```
conn.close();
```

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.4.2: Approaches to managing database connections](#)

4.2.4.2.3: Using IBM data access JavaBeans to access relational databases



Java applications and servlets 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 JavaBeans. This gives you additional features and functions for data access beyond those available in the `java.sql` package.

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

Related information...

- [0.14.7: What are data access Javabeans?](#)
- [0.14.7.1: Features of data access JavaBeans](#)
- [4.2.4.2.3.1: Example: Servlet using data access JavaBeans](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.4.2: Approaches to managing database connections](#)

4.2.4.2.3.1: Example: Servlet using data access JavaBeans

The sample servlet uses the data access JavaBeans and is based on the sample servlet discussed in [Connection pooling](#). 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 JavaBeans, 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 JavaBeans) 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 JavaBeans.

The DA sample servlet differs slightly from the CP sample servlet. This discussion covers only the changes. See [How servlets use connection pooling](#) for the discussion of the CP sample servlet. The DA sample servlet shows the basics of connection pooling and the data access JavaBeans, 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 JavaBeans 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 JavaBeans.

- 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 `JavaBeans` 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 `JavaBean`.

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` `JavaBean`) is created to establish the link between the data access `JavaBeans` and the database connection - the `conn` object.
2. The `selectStatement` object (a `SelectStatement` `JavaBean`) 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 JavaBean) is a cache containing the results of the query, created using the getResult() method.
6. The data access JavaBeans 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 JavaBean.

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.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.4.2.3: Using IBM data access JavaBeans to access relational databases](#)

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);
```

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.4.2: Approaches to managing database connections](#)

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.

Related information...

- [4.4.1: Tracking sessions](#)
- [4.4.2: Keeping user profiles](#)
- [0.11: What are sessions and Session Managers?](#)
- [0.12: What are user profiles and User Profile Managers?](#)
- [6.6.11: Administering session support](#)
- [6.6.12: Administering user profile support](#)
- [Index to API documentation \(Javadoc\)](#)
- [4: Developing applications](#)

0.11: What are sessions and Session Managers?



A session is a series of requests to a servlet, originating from the same user at the same browser. Sessions allow servlets running on a servlet engine to keep track of individual users, a concept known as personalization.

For example, a servlet might use sessions to provide "shopping carts" to on-line shoppers. Suppose the servlet is designed to record the items each shopper indicates he or she will purchase from the Web site. It is important that the servlet be able to associate incoming requests with particular shoppers. Otherwise, the servlet might mistakenly add Shopper_1's choices to the cart of Shopper_2.

A servlet distinguishes users by their unique session IDs. The session ID arrives with each request. If the user's browser is cookie-enabled, the session ID can be stored as a cookie. Otherwise, it can be conveyed to the servlet by URL rewriting, in which the session ID is appended to the URL of the servlet or JavaServer Pages (JSP) file from which the user is making requests.

The product provides facilities, grouped under the heading *Session Manager*, that support the [javax.servlet.http.HttpSession](#) interface described in the Servlet API specification. Using the WebSphere Application Server 3.x implementation of the Java Servlet API (Version 2.1) session framework, your server can maintain session state information.

Related information...

- [0.11.1: What are session transactions?](#)
- [0.11.2: What are session clusters?](#)
- [0.9: What are servlets?](#)
- [0.14.4: What is application data?](#)

0.11.1: What are session transactions?



Programmers can implement session objects in a variety of ways, which provide different levels of performance, failover, and clustering. In all cases, IBM WebSphere Application Server defines the notion of a session *transaction*.

A session transaction begins when the servlet establishes a session by calling:

```
javax.servlet.http.HttpServletRequest.getSession\(boolean\)
```

If not manually unlocked, a session transaction ends with the completion of the servlet method:

```
javax.servlet.http.HttpServlet.service\(request, response\)
```

By default, the product locks sessions during the scope of these transactions to maintain session integrity. This means that only one instance of a servlet can access a session at a given time. In the case where several servlets are chained together to service an individual HTTP request, the session stays locked across each servlet in the chain until a response is sent back to the user.

A session transaction can be manually unlocked. See the Related information.

Related information...

- [4.4.1.1.2: Locking and unlocking session transactions](#)
- [0.11: What are sessions and Session Managers?](#)

0.11.2: What is session clustering?



The Session Manager session support allows multiple application server instances to share a common pool of sessions, known as a *session cluster*. A cluster is the binding of two or more virtual hosts that reside on separate nodes, such that each virtual host runs servlets across all of the nodes and processes sessions on any one of those nodes.

The implementation of clustering in WebSphere Application Server Version 3.x allows failover. This preserves session data integrity and the common pool of sessions in the event of a system failure in one or more of the clustered Java virtual machines (JVMs) running servlets within a virtual host group.

The session clustering implementation also allows load balancing, whereby the session workload is distributed among the virtual hosts comprising the cluster.

IBM WebSphere Application Server Version 3.x uses a database to maintain session clusters. With the use of a database and general architectural changes implemented in Version 3.x, the product no longer maintains the session cluster client and session cluster server concepts.

In a clustered environment, the session may be accessed on any virtual host in a cluster. Which virtual host is actually accessed will be transparent to the end user. During a session transaction, if the virtual host fails during the WebSphere HttpSession transaction, then the update to the database does not occur. Still, the common pool of sessions continues to function, including the session being processed during the failure, minus any updates made during the uncompleted transaction.

For non-catastrophic failures (such as when the virtual host remains functional), any session changes that cannot be completed are rolled back. The session reverts to its state prior to the start of the transaction. If instead the transaction is completed successfully and the changes are committed, the session is still accessible, regardless of the failure of an individual node.

Related information...

- [0.16: What are virtual hosts?](#)
- [0.11: What are sessions and Session Managers?](#)

0.11.3: What are cookies?



Netscape defines a cookie as "a general mechanism which server side connections (such as CGI scripts) can use to both store and retrieve information on the client side of the connection." See the [Netscape cookie specification](#) for additional information.

Related information...

- [0.11: What are sessions and Session Managers?](#)

0.12: What are user profiles and User Profile Managers?



Some applications collect data about the users with which they interact. The data is stored in a database. The next time the user interacts with the application, the application recalls the data.

Because the application already "knows" something about the user, it can provide the user with a more personalized experience.

Basically, user profiles allow a company to maintain and manage database tables containing fields for demographic data, and use those tables to interact with a database of individual customers or other users on the company system.

For example, when a repeat user logs onto a Web site that supports user profiles, the Web site can display headlines and advertising tailored to the shopping preferences of that user. The site can address the user by his or her logon name.

An application implementing user profiles requires database access for storing the user profile data it gathers.

WebSphere Application Server provides user profile support in the form of programming APIs (for the developer) and User Profile Managers that specify the Java classes and data sources providing user profile support (for the administrator).

Related information...

- [0.14.4: What is application data?](#)

4.4.1: Tracking sessions



IBM WebSphere Application Server Version 3.5 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.

Related information...

- [4.4.1.1: Session programming model and environment](#)
- [6.6.11: Administering session support](#)
- [4.4.1.1.7: Tuning sessions](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.4: Personalizing applications](#)

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 investigate the special 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

Use the

```
getSession()
```

method of the

```
javax.servlet.http.HttpServletRequest
```

object in the Java Servlet 2.1 API to obtain a session.

When you first obtain the

```
HttpSession
```

object, the Session Manager uses one of two ways to establish tracking the session: cookies or URL rewriting. 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. (Note that with the Java 2.1 servlet 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.

If Java objects will be added to a session, be sure to place the class files for those objects in the application server classpath or in the directory containing other servlets used in WebSphere Application Server. In the case of session clustering, this applies to every node in the cluster.

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. 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.

Related information...

- [4.4.1.1.1: Deciding between session tracking approaches](#)
- [4.4.1.1.2: Locking and unlocking sessions](#)
- [4.4.1.1.3: Securing sessions](#)
- [4.4.1.1.4: Deciding between single and multirow schema](#)
- [4.4.1.1.5: Using sessions in a clustered environment](#)
- [4.4.1.1.6: Session limitations](#)
- [4.4.1.1.7: Tuning sessions](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.4.1: Tracking sessions](#)

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 two 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. If the administrator enables URL rewriting, it will be used, even in situations in which cookies are feasible. When deciding whether to use URL rewriting, carefully review the coding requirements it imposes on applications requiring 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;$ssessionId$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 servlet engine recognizes

```
;$ssessionId$DA32242SSGE2
```

as the session ID and saves it for obtaining the proper `HttpSession` object for this user.

To use URL rewriting, applications must follow certain coding guidelines. Also, special preparation is required. See the Related information for details.

Related information...

- [0.11.3: What are cookies?](#)
- [4.4.1.1.1: Using cookies to track sessions](#)
- [4.4.1.1.1.2: Using URL rewriting to track sessions](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.4.1.1: Session programming model](#)

4.4.1.1.1: 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](#).

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.4.1.1.1: Deciding between session tracking approaches](#)

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 this 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.

With Version 3.x, if URL encoding is configured and `response.encodeURL` or `encodeRedirectURL` is called, the URL will be encoded, even if the browser making the HTTP request processed cookies. This differs from the behavior in previous releases, which checked for the condition and halted URL encoding in such a case.

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 above.

Sessions will be lost if one or more servlets in an application do not call the

```
encodeURL(String url)
```

or

```
encodeRedirectURL(String url)
```

methods.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.4.1.1.1: Deciding between session tracking approaches](#)

4.4.1.1.2: Locking and unlocking session transactions



You can manually override session transactions 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 will unlock the session. Unlocking the session sends any changes in the session to the database.

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()`.

To re-lock the session, you call the `getSession()` method of the request object.

Related information...

- [0.11.1: What are session transactions?](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.4.1.1: Session programming model](#)

4.4.1.1.3: Securing sessions



HTTP sessions and security are integrated in WebSphere. When WebSphere security is enabled, every resource from which sessions are created or accessed must be secured or it must be unsecured. You cannot mix secured and unsecured resources.

Summary

- 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. Unlike with product Versions 2.0x and earlier, the servlet cannot set a user name associated with a session; that functionality is incorporated into the Session Manager.

An identity or user name, readable by the `com.ibm.websphere.servlet.session.IBMSession` interface, is associated with a session. An unauthenticated identity is denoted by the user name "anonymous." WebSphere Application Server includes the interface:

`com.ibm.websphere.servlet.session.UnauthorizedSessionRequestException`
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, LTPA, and other methods.

After obtaining the identity of the current request, the Session Manager determines whether the session requested using a `getSession()` call should be returned.

The table lists possible scenarios, 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. The Session Manager changes the user name to "FRED"
A session ID for a valid session is passed in. The current session user name is "FRED"	The session is not returned. 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. UnauthorizedSessionRequest Exception is thrown*	The session is not returned. UnauthorizedSessionRequest Exception is thrown*

* UnauthorizedSessionRequest Exception is a runtime exception and does not need to be explicitly caught

Related information...

- [5: Securing applications](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.4.1.1: Session programming model](#)

4.4.1.1.4: Deciding between single to multirow schema for sessions



Using the "single" schema, each user session maps to a single database row. Using the "multirow" schema, each user session maps to multiple database rows.

In addition to allowing larger session records, using multirow schema can yield performance benefits, as discussed in [Tuning session support: Multirow schema](#). However, it requires a little work to switch to from single to multirow schema, as shown in the instructions below.

Switching from single to multirow schema

To switch from single 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 WebSphere uses to maintain HttpSession.

To access 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. Look for the table named "SESSIONS".
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.

In the case of multirow usage, design your application data objects not to have references to each other, and 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.

Related information...

- [4.4.1.1.3: Tuning session support: Multirow schema](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.4.1.1: Session programming model](#)

4.4.1.1.5: Using sessions in a clustered environment



Consider the following caveats regarding how session management works within a clustered environment. Clustered environments include any environments in which requests are being handled by multiple Web servers, multiple application servers, or both. See [section 7](#) for more about clustering and workload management.

For use in a clustered environment, objects placed in sessions must be serializable

To make your applications portable to a clustered environment, you must make any objects placed in a session serializable.

The definition of the `putValue()` method of the `HttpSession` interface in the current Java Servlet 2.1 API (as specified by Sun Microsystems) does not account for the possibility of a clustered environment.

If you add an object to a session that does not implement the `Serializable` interface, you lack a way to propagate the object along with a given session, disallowing proper persistence among servlets in the cluster.

Session binding occurs on a certain application server in the cluster

When `HttpSessionBindingListener` and `HttpSessionBindingEvent` are used in a clustered Web server environment, the event will be fired in the application server on which the session is currently being processed. The action will occur in situations where:

- The servlet explicitly invalidates the session
- The session times out
- A listener is removed from a session

Certain changes require restarting all Session Managers in a cluster

See the Related information for details.

Related information...

- [0.11.2: What is session clustering?](#)
- [6.4.2: Actions that require a restart](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.4.1.1: Session programming model](#)

4.4.1.1.6: Current limitations in session support



- The product does not provide non-JDBC, native access to a database version of session persistence
- For now, the administrator should use only the direct-to-database persistence type. The EJB persistence type is intended for securely and reliably accessing a HttpSession outside the scope of a servlet; however, it is not fully functional at this time.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.4.1.1: Session programming model](#)

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.

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?
Session caching	Minimize database read operations	Database
Manual update mode	Minimize database write operations	Database
Session affinity*	Minimize database read operations	Database
Multirow schema	Fully utilize database capacities	Database
Base in-memory session pool size	Fully utilize system capacity, without overburdening system	Either

* The product supports, but does not provide, session affinity

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.4.1.1: Session programming model](#)

4.4.1.1.7.1: Tuning session support: Session caching



IBM WebSphere Application Server session support maintains a list of the most recently used sessions in memory. It avoids using the database to read in or access the session when it is determined that the cache entry is still the most recently updated copy.

Use the [base in-memory session pool size](#) to tune the cache size.

It is also important to establish session affinity, so that the caching can be most effective. See the Related information for details.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.4.1.1.7: Tuning session support](#)

4.4.1.1.7.2: Tuning session support: Session affinity



To efficiently use its session cache, the product needs an "affinity" mechanism to help ensure cache hits. The necessary affinity features are provided by Network Dispatcher, an IBM product.

Two of the most desirable affinity features are:

- The "sticky port"

Network Dispatcher uses a load balancing functionality and the client's IP address to maintain the affinity.

- Content-based or cookie-based affinity

Network Dispatcher works with a proxy server to balance loads, using cookies to identify clients and maintain affinity.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.4.1.1.7: Tuning session support](#)

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.

Specifically, with an IBM DB2 database, the maximum space available is 2 Megabytes, using a BLOB data type.

With Oracle, the maximum amount is 2 Megabytes (where the only choices are "raw" for a maximum of 2 Kilobytes, or the "long raw" data type for 2 Megabytes).

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 remaining is the size of data object itself, where the above limits should be enough for any single Java object.

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.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.4.1.1.4: Deciding between single and multirow schema for sessions](#)
- [4.4.1.1.7: Tuning session support](#)

4.4.1.1.7.4: Tuning session support: Manual update mode



By default, session support automatically updates session information in a database when certain events occur. Depending on the application characteristics, performance can be improved by switching from automatic to manual updates. With manual updates, the servlet using sessions 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.

By default, IBM WebSphere Application Server always 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.

When manual update is turned on, the product session support no longer automatically updates the database at the end of a servlet's `service()` method. 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 the servlet processing, the servlet must manually call the `sync()` method of the

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

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.4.1.1.7: Tuning session support](#)

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. (For information about what happens after this number is surpassed, see the article about allowing session pool overflow (Related information).
- 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.

After the base in-memory session pool size is surpassed, sessions are simply not cached, and session updates are automatically sent back to the database, without checking for their presence in the cache.

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 stack sizes of the Java processes for the corresponding WebSphere application servers.

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 this value 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 url's and never utilizes the same session from one HTTP request to the next.

When overflow is disallowed, the Session Manager will still return a session with the `HttpServletRequest`'s `getSession(true)` method if the memory limit has currently been reached, but it will be an invalid session that is not saved in any fashion.

With the WebSphere extension to `HttpSession`, `com.ibm.websphere.servlet.session.IBMSession`, there is an `isOverflow()` method which will return "true" if the session is such an invalid session. An application could check this and react accordingly.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.4.1.1.7: Tuning session support](#)

4.4.2: Keeping user profiles



IBM WebSphere Application Server Version 3.5 provides a service for processing user profiles — the *User Profile Manager*. The service is provided in the form of an EJB entity bean that servlets can call whenever they are required to access a user profile.

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
 - Extending the User Profile enterprise bean itself to import existing databases

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.

Related information...

- [4.4.2.1: Data represented in the base user profile](#)
- [4.4.2.2: Customizing the base user profile support](#)
- [4.4.2.3: Accessing user profiles from a servlet](#)
- [6.6.12: Administering user profiles](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.4: Personalizing applications](#)

4.4.2.1: Data represented in the base user profile



WebSphere Application Server Version 3.5 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

Related information...

- [4.4.2.2.1: Extending data represented in user profiles](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.4.2: Keeping user profiles](#)

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
```

Extending the User Profile enterprise bean and importing legacy databases

Application developers can extend the User Profile enterprise bean itself and import legacy databases into the user profile. The main advantage in extending the User Profile enterprise bean is to gain the ability to import existing databases into the user profile. It can also be extended to simply add columns to the base user profile implementation.

Related information...

- [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](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.4.2: Keeping user profiles](#)

4.4.2.2.1: Extending data represented in user profiles



The following interface can be used to extend a user profile hash table, allowing you to place arbitrary name/value pairs in the User Profile:

```
com.ibm.websphere.userprofile.UserProfileProperties
```

Extending the hash table is similar to using the

```
java.util.Dictionary
```

class in the base JDK 1.x or any of the classes which extend it.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.4.2.2: Customizing the base user profile support](#)

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:

1. Extend the `com.ibm.servlet.personalization.userprofile.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 User Profile 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 <code>UPServletExample</code> 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 "USERPOFILE" 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.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.4.2.2: Customizing the base user profile support](#)

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



IBM WebSphere Application Server Version 3.x 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 EJB is more involved than extending the base user profile. The developer should be familiar with the concepts underlying enterprise beans (Java components coded to the EJB specification). For detailed information regarding enterprise beans and their implementation in IBM WebSphere Application Server Version 3.5, 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 Version 3.x, 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

3. Create read-only and read-write extensions of the UPBaseChild bean.

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.

Read-only bean extension classes:

- [Bean](#)
- [Remote interface](#)
- [Home interface](#)
- [Finder class](#)

Read-write bean extension classes:

- [Bean](#)
- [Remote interface](#)
- [Home interface](#)
- [Finder class](#)

4. Define a deployment descriptor for your beans

Use the Jetace tool included with WebSphere Application Server Version 3.x to define the container managed persistence (CMP) fields and properties of the bean. Alternatively, use IBM VisualAge for Java to develop the bean and define its deployment descriptor.

If using IBM VisualAge for Java, you can mark the methods as read-only using the "Const" method option. IBM VisualAge for Java also gives you the ability to map the fields to the existing table, so that you can map both of the beans to same table.

In the base read-only implementation provided, all the business methods are marked as read-only, so updating the fields using setter methods on the read-only bean will not update the persistent store.

Mark all the business methods in this example as constant (read-only):

- [Read only](#)

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:

- [Read write](#)

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.

5. Extend the User Profile data wrapper to include the new methods

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

[EJB extension](#)

6. Use a WebSphere Application Server administrative client to configure a User Profile Manager.

The administrative configuration includes:

- The data wrapper class name
- JNDI lookup names
- Home and remote interfaces for the read-only and read-write beans

User Profile Manager
[properties help](#)

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.4.2.2: Customizing the base user profile support](#)

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

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.4.2: Keeping user profiles](#)

4.2.4.4: Providing ways for clients to invoke applications



A Web application is of little use if users cannot access it. Users access a Web application by invoking a component that provides an entry point into the Web application, such as a JSP or servlet. The entry point is usually accessible from a Web page or the like. See the articles below for some ideas.

Related information...

- [4.2.4.4.1: Providing Web clients a way to invoke JSP files](#)
- [4.2.4.4.2: Providing Web clients a way to invoke servlets](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.4: Putting it all together \(Web applications\)](#)

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
Call JSP file from another JSP file	Open a JSP file that invokes the JSP file

Related information...

- [4.2.4.4.1.1: Invoking servlets and JSP files by URLs](#)
- [4.2.4.4.1.2: Invoking servlets and JSP files within HTML forms](#)
- [4.2.4.4.1.3: Invoking JSP files within other JSP files](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.4.4: Providing ways for clients to invoke applications](#)

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.

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.

Related information...

- [4.2.1.3.6: Enabling servlets to be invoked by their class names](#)
- [6.6.0.1.4: Editing resource properties](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.4.4.1: Providing Web clients access to JSP files](#)

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>
```

Related information...

- [4.2.4.4.1.2.1: Example: Invoking servlets within HTML forms](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.4.4.1: Providing Web clients access to JSP files](#)

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 HTML response (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.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.4.4.1.2.1: Invoking servlets within HTML forms](#)

4.2.4.4.1.3: Invoking JSP files within other JSP files



An application developer can enable users to invoke a JSP file from within another JSP file. Within the first JSP file, the developer should use one of the following methods for invoking the second JSP file:

- Specify the URL of the second JSP file on the FORM ACTION attribute.
- Specify the URL of the second JSP file on the anchor tag HREF attribute().
- Use the `javax.servlet.http.RequestDispatcher.forward()` method to invoke the second JSP file.
- For JSP 1.0, use the `jsp:forward` and `jsp:include` elements.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.4.4.1: Providing Web clients access to JSP files](#)

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

Related information...

- [4.2.4.4.2.1: Invoking servlets from within SERVLET tags](#)
- [4.2.4.4.2.2: Invoking servlets from within JSP files](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2.4.4: Providing Web clients access to JSP files](#)

4.2.4.4.2.1: Invoking servlets within SERVLET tags



The user can invoke a servlet from an HTML page containing a SERVLET tag.



This method is **not** recommended because it only works with JSP .91, and withdrawal of JSP .91 support is imminent.

The application developer includes a SERVLET tag in an HTML page to cause a *server-side include* in which everything between and including the two SERVLET tags is overlaid with the output from the servlet called within the tags. As the name suggests, all processing occurs on the server. The resulting HTML page is sent to the user.

The following HTML fragment shows how to use the SERVLET tag:

```
<SERVLET NAME="myservlet" CODE="myservlet.class" CODEBASE="url" initparm1="value">
<PARAM NAME="parm1" VALUE="value">
</SERVLET>
```

Servlet attributes

The loaded servlet will assume a servlet name matching the name specified in the NAME attribute. Subsequent requests for that servlet name will invoke the same servlet instance.

SERVLET attribute	Description
NAME	It specifies a servlet name, or the servlet class name without the .class extension
CODE	It specifies the servlet class name
CODEBASE	It specifies a URL on a remote system from which the servlet is to be loaded

Using the NAME and CODE attributes provides flexibility. Either or both can be used. CODEBASE is optional. With the product, it is recommended that you specify both NAME and CODE, or only NAME if the NAME specifies the servlet name. If only CODE is specified, an instance of the servlet with NAME=CODE is created.

If both NAME and CODE are present, and NAME specifies an existing servlet, the servlet specified in NAME is always used. Because the servlet creates part of an HTML file, the application developer will probably use a subclass of the HttpServlet class when creating the servlet and override the doGet() method, because GET is the default method for providing information to the servlet. Another option is to override the service() method.

Parameter attributes

In the previous tagging example, *parm1* is the name of an initialization parameter and *value* is the value of the parameter.

PARAM attribute	Description
NAME	It specifies the name of the parameter for use with this particular invocation of the servlet
VALUE	It specifies the value of the parameter for use with this particular invocation of the servlet

You can specify more than one set of name-value pairs. Use the getInitParameterNames() and getInitParameter() methods of the ServletConfig object (which the servlet engine passes to the servlet's init() method) to find a string array of parameter names and values.

In the example, *parm1* is the name of a parameter that is set to *value* after the servlet is initialized. Because the parameters set using the <PARAM> tag are available only through methods of a Request object, the server must have invoked the servlet service() method, passing a request from a user.

To get information about the user's request, the application developer should use the getParameterNames(), getParameter(), and getParameterValues() methods.

The parameters set within the <PARAM> attribute are for a specific invocation of the servlet. If a second JSP file invokes the same servlet with no <PARAM> parameters, the <PARAM> parameters set by the first invocation of the servlet are not available to the second invocation of the servlet.

In contrast, servlet initialization parameters are persistent. Suppose a client invokes the servlet by invoking an JSP file

containing some initialization parameters. Assume that a second client invokes the same servlet through a second JSP file, which does not specify any initialization parameters. The initialization parameters set by the first invocation of the servlet remain available and unchanged through all successive invocations of the servlet through any other JSP file. The initialization parameters cannot be reset until after the `destroy()` method has been called on the servlet.

For example, if another JSP file specifies a different value for an initialization parameter but the servlet is already loaded, the value is ignored.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.4.4.2: Providing Web clients a way to invoke servlets](#)

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

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4.2.4.4.2: Providing Web clients a way to invoke servlets](#)

4.2.7: Securing Web applications from the inside and outside



Application programmers can secure Web applications "from the inside" by using sound coding practices that prevent or minimize security vulnerabilities.

WebSphere administrators can secure Web applications "from the outside" by adding WebSphere security and designing environments that use firewalls and other security features to protect Web applications.

Related information...

- [5: Securing applications](#)
- [6.6.18: Administering product security](#)
- [Index to API documentation \(Javadoc\)](#)
- [4.2: Building Web applications](#)

4.2.8: Programming high performance Web applications



This article offers tips and guidelines for creating Web applications that perform well in the WebSphere Application Server environment. It also includes enterprise beans tips as appropriate.

Use calls to `ServletContext.log()` sparingly

Each calls to the `ServletContext.log()` method is recorded in the WebSphere administrative database. Overusing calls to this method will seriously degrade performance. Limit calls to only those events that should be considered `SeriousEvents`.

Related information...

- [4.2: Building Web applications](#)

4.2.9: Setting language encoding in Web applications



This article provides tips and guidelines for using various language encodings in WebSphere applications.

Setting encoding in servlets and JSP files

If Double Byte Character Set (DBCS) output streams are not being displayed correctly in Web browser clients accessing applications in the WebSphere Application Server environment, consider setting the encodings for the applications.

Setting encodings for servlets

To specify the character encoding of a servlet, add the following command line argument to the application server containing the servlets:

```
-Ddefault.client.encoding=encoding
```

where *encoding* is the encoding of your choice.

Setting encodings for JSP files

To specify the character encoding of a JSP page, add the following command line argument to the application server containing the JSP files:

```
-Dfile.encoding=encoding
```

where *encoding* is the encoding of your choice.

To specify the character encoding of the resulting stream, insert the encoding statement in the JSP file:

```
<%@ page contentType="encoding %>
```

where *encoding* is the encoding of your choice.

Writing to and from databases

When writing data to a database, a servlet (or other application component) must use the same encoding as that data stored in the database. Similarly, the database and a servlet obtaining data from the database must use the same encoding.

For example, a servlet writing data in a Korean encoding cannot write the data into a database configured with an English encoding, unless the servlet first converts the data to an English encoding. The same is true of any two encodings.

Related information...

- [4.2: Building Web applications](#)

4.5: Employing pervasive computing



The IBM WebSphere Application Server Web site contains a set of application programming interfaces (APIs) for pervasive computing.

The APIs present a set of reusable components for easy access to facilities such as printer, fax, mail, and calendar systems. Because these are commonly used in pervasive computing applications, this documentation refers to the APIs as the Common Application Functions.

Please note, the APIs are a technology preview, meaning they are not supported by IBM service personnel.

Pervasive computing components

Version 3.5 includes these components:

- SendMail
- ReadMail
- Calendar
- Fax
- Print

Each component has two implementations:

- Enterprise JavaBean implementation for environments offering EJB support
- JavaBean implementation for other environments

Clients of pervasive computing components

A set of client applications is included to demonstrate the main functionality of the pervasive computing components. Client applications for the enterprise bean and JavaBean implementations of the components use the same interface, providing the ability to switch between environments without changing client code.

Related information...

- [Index to API documentation \(Javadoc\)](#)
- [4: Developing applications](#)

4.2.6: Using applets as clients



Starting with IBM WebSphere Application Server Version 3.5, it is possible for Java *applets* to open connections and communicate with application servers using the IIOP protocol.

For an applet to connect to an application server using IIOP, it is necessary for the Java-enabled browser to use the Java Plug-in in order to access the runtime environment that enables the RMI over IIOP communication. JDK version 1.2.2 contains the Java Plug-in. It is automatically installed during the IBM WebSphere Application Server Custom installation, unless you indicate otherwise. See the Related information for example code to accompany the following instructions.

1. Install a Java-enabled browser that supports the use of plug-ins, such as Netscape Communicator or Internet Explorer.



Install the browser **before** installing the JDK. Otherwise, the browser plug-in will not be configured in the browser during JDK installation.

2. Install the IBM JDK, Version 1.2.2.

The Java Plug-in is automatically installed and configured into the installed browser. On Windows NT, a "Java Plug-in Control Panel" is installed in the Start -> Programs Menu for setting several configuration options.

3. Develop, write and compile the Java applet source code.

In addition to compiling the Java source code, it is necessary to also generate client "Stub" and server "Tie" Java classes for the object proxies necessary during remote method invocations. The command "rmic" invoked with the option "-iiop" is used for this purpose.



To invoke a remote method, an applet must first obtain an interoperable reference (IOR) that identifies the location of the remote object implementation. A naming service is a possible source of published IORs.

If the object implementation, identified by the IOR, is located on a different host than the Web server machine, the applet must be granted the necessary security permission for creating a socket connection to the given host.

4. For convenience, package all generated Java class files into a JAR archive. These will be loaded on demand by the browser in conjunction with the JDK applet class loader for execution.

Make a note to use the attribute "archive" in the applet source HTML to specify the location of the JAR archive in the Web server where the applet is deployed.

5. Find the location of the ZIP files containing the ORB Java classes. These are located in the lib directory of the product installation root:

- ujc.jar
- iioptools.jar

As with the applet JAR file, make a note to specify the above files in the archive attribute of the HTML source.

6. Write the HTML source that invokes the applet:

- Use the OBJECT tag instead of the APPLET tag (it supercedes the deprecated APPLET tag). Add the appropriate OBJECT attributes and parameters, depending on the browser that will be used. See the code example in Related information for guidance.

Sun Microsystems provides a tool to automatically convert HTML files from using APPLET to OBJECT tags, including the required attributes and parameters for Netscape Communicator and Internet Explorer.

See <http://java.sun.com/products/plugin/> for details.

- Do not forget to add the files to the archive attribute, as noted in the above steps.

Related information...

- [4.6.1: Example: Using applets as clients](#)
- [Index to API documentation \(Javadoc\)](#)
- [4: Developing applications](#)

4.2.6.1: Example: Using applets as clients



Sample HTML source code

```
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"  
WIDTH = 500 HEIGHT = 300 >  
<PARAM NAME = CODE VALUE = TestApplet.class >  
<PARAM NAME = ARCHIVE VALUE = "ujc.jar,iioptools.jar,applet.jar" >  
  
<PARAM NAME="type" VALUE="application/x-java-applet;version=1.2" >  
<COMMENT>  
<EMBED  
type="application/x-java-applet;version=1.2"  
CODE = TestApplet ARCHIVE = "ujc.jar,iioptools.jar,applet.jar"  
WIDTH = 500 HEIGHT = 300 >  
<NOEMBED>  
</COMMENT>  
</NOEMBED>  
</EMBED>  
</OBJECT>
```

Sample Java source code

```
import java.applet.Applet;  
import java.util.Properties;  
import javax.rmi.PortableRemoteObject;  
  
public class TestApplet extends Applet {  
  
    org.omg.CORBA.ORB orb;  
  
    public void start() {  
  
        try {  
            // obtain object reference and narrow to specific type as needed.  
            Object object = .... obtain reference to remote object, e.g. from naming service, etc.  
            TestInt test = (TestInt)PortableRemoteObject.narrow (object,TestInt.class); // narrow  
to specific object type  
  
            // invoke remote method now. It receives and returns an instance of itself.  
            Object result = test.testRemoteObject(object);  
        }  
        catch (Exception ex) {  
            ex.printStackTrace();  
        }  
    }  
}
```

```

}
public void init() {
    String orbKey = "org.omg.CORBA.ORBClass";
    String orbClass = "com.ibm.CORBA.iiop.ORB";
    Properties props = new Properties();
    props.put(orbKey, orbClass);

    // obtain a reference to the ORB, it will be needed to invoke ORB methods as needed,
    e.g. string_to_object(), etc
    orb = org.omg.CORBA.ORB.init(this, props);
}
}

```

Definition of remote object interface and a simple implementation

```

interface TestInt extends java.rmi.Remote {
    public Object testRemoteObject(Object o) throws java.rmi.RemoteException;
}

public class Test extends javax.rmi.PortableRemoteObject implements TestInt {
// The Tie and Stubs are generated from this class with the command: rmic -iiop Test.
// It generates the files _TestInt_Stub.class and _Test_Tie.class

    public Test() throws java.rmi.RemoteException {
    }

    public Object testRemoteObject(Object o) throws java.rmi.RemoteException {
        return o;
    }
}

```

Related information...

- [Example: Using applets as clients](#)
- [Index to API documentation \(Javadoc\)](#)
- [4: Developing applications](#)

5.4: Overview: Using programmatic and custom login



This section describes the use of programmatic login techniques and custom challenge capabilities (including the use of Single Sign-On) in WebSphere Application Server.

When applications require user to provide identifying information, the writer of the application must collect that information and authenticate the user. The work of the programmer can be broadly classified in terms of where the actual user authentication is performed:

1. In a client program
2. In a server program

Users can be prompted for authentication data in many ways. The challenge type configured for the application defines the mechanism used to collect this information. Programmers who want to customize login procedures, rather than relying on general-purpose devices like a 401 dialog window in a browser, can use a custom challenge to provide an application-specific HTML form for collecting login information.

No authentication work will occur unless WebSphere security is enabled. Additionally, if you want to use the custom challenge type, you must configure security as follows (click an item to link to detailed property help for the item):

- Set the [challenge type](#) to Custom.
- Set the [authentication mechanism](#) to LTPA.
- Set the [LoginURL and ReloginURL field](#) to the URL of your HTML login form.
- Enable [Single Sign-On](#), if used.

Related information...

- [5.4.1: Programmatic login on the client side](#)
- [5.4.2: Programmatic login on the server side](#)
- [5.4.3: Custom login challenges](#)
- [5: Securing applications](#)

5.4.1: Programmatic login on the client side



Use a client-side login when a pure Java client needs to log users into the security domain but does not need to use the authentication data itself.

Client-side login works in the following manner:

1. The user makes a request to the client application.
2. The client presents the user with a login form for collecting authentication data. The user inserts his or her user ID and password into the form and submits it.
3. The client programmatically places the user's authentication data into an ORB-related data structure called the *security context*.
4. The client program invokes a method on a server.
5. The server processes the request, extracting the authentication data from the context and performing authentication.
6. If the authentication was successful, the server grants the request and returns the security credentials for further use. If the authentication fails, the server denies service.

The client programmer is responsible for writing the code to extract the authentication data and insert it into the CORBA data structures. WebSphere provides a utility class, the LoginHelper class, that can be used to simplify the CORBA programming needed to do this kind of programmatic login. The TestClient application illustrates the use of the LoginHelper class.

In order to use the LoginHelper class, the client needs to know the security properties of the ORB, so you must load a properties file containing those values when you start the client program. The file `properties/sas.client.props` installed with WebSphere contains valid values. Specify the properties file on the command line as follows:

```
-D com.ibm.CORBA.ConfigURL=URL of properties file
```

For example, to load the `sas.client.props` file and run the TestClient program, issue the following command:

```
java -D com.ibm.CORBA.client.ConfigURL=file://<install_root>/properties/sas.client.props TestClient
```

The Component Broker online documentation provides information about the CORBA security classes. This information can be found on the Component Broker documentation Web site.

Related information...

- [5.4.1.1: The TestClient program](#)
- [5.4.1.2: The LoginHelper class](#)
- [Component Broker documentation](#)
- [5.4: Overview: Using programmatic and custom login](#)

5.4.1.1: The TestClient program



The TestClient program illustrates the use of the LoginHelper class, a utility class provided to help simplify programming client-side login. The excerpt below shows the performLogin method.

TestClient class

```
public class TestClient
{
    ...

    private void performLogin()
    {
        // Get the user's ID and password.
        String userid = customGetUserid();
        String password = customGetPassword();

        // Create a new security context to hold
        // authentication data.
        LoginHelper loginHelper = new LoginHelper();

        try {
            // Provide the user's ID and password for authentication.
            org.omg.SecurityLevel2.Credentials credentials =
                loginHelper.login(userid, password);

            // Use the new credentials for all future invocations.
            loginHelper.setInvocationCredentials(credentials);

            // Retrieve the user's name from the credentials
            // so we can tell the user that login succeeded.
            String username = loginHelper.getUserName(credentials);
            System.out.println("Security context set for user: "+username);
        }
        catch (org.omg.SecurityLevel2.LoginFailed e)
        {
            // Handle the LoginFailed exception.
        }
    }
    ...
}
```

Related information...

- [5.4.1: Programmatic login on the client side](#)
- [5.4.1.2: The LoginHelper class](#)

5.4.1.2: The LoginHelper class



The LoginHelper class is a WebSphere-provided utility class that provides wrappers around CORBA security methods. It can be used by pure Java clients that need the ability to programmatically authenticate users but don't need to use the authentication data on the client side.

The methods in this class give a client program a way to collect authentication information from a user and package it to be sent to a server. The server authenticates the user and returns security credentials to the client.

The following list summarizes the public methods in the LoginHelper class. The source file is installed at:

```
<installation_root>/hosts/default_host/examples/security/LoginHelper.java
```

and the class file is installed at:

```
<installation_root>/servlets/LoginHelper.class
```

LoginHelper()

The constructor obtains a new security-context object from the underlying ORB. This object is used to carry authentication information and resulting credentials for the client.

Syntax:

```
LoginHelper() throws IllegalStateException
```

login()

This method takes the user's authentication data (identifier and password), authenticates the user (validates the authentication data), and returns the resulting Credentials object.

Syntax:

```
org.omg.SecurityLevel2.Credentials login(String userID, String password)
throws IllegalStateException
```

setInvocationCredentials()

This method sets the specified credentials so that all future methods invocations will occur under those credentials.

Syntax:

```
void setInvocationCredentials(org.omg.SecurityLevel2.Credentials invokedCreds)
throws org.omg.Security.InvalidCredentialType,
      org.omg.SecurityLevel2.InvalidCredential
```

getInvocationCredentials()

This method returns the credentials under which methods are currently being invoked.

Syntax:

```
org.omg.SecurityLevel2.Credentials getInvocationCredentials()
throws org.omg.Security.InvalidCredentialType
```

getUserName()

This method returns the user name from the credentials in a human-readable format.

Syntax:

```
String getUserName(org.omg.SecurityLevel2.Credentials creds)
throws org.omg.Security.DuplicateAttributeType,
      org.omg.Security.InvalidAttributeType
```

Related information...

- [5.4.1.1: The TestClient program](#)
- [5.4.1: Programmatic login on the client side](#)

5.4.2: Programmatic login on the server side



Use a server-side when a program needs to log users into the security domain and to use the authentication data itself. A client-side login collects the authentication data and sends it to another program for actual authentication; a server-side login does both tasks.

Server-side login works in the following manner:

1. The user makes a request that triggers a servlet.
2. The servlet presents the user with a login form for collecting authentication data. The user inserts his or her user ID and password into the form and submits it.
3. The servlet presents the request to the server.
4. The server processes the request, extracting the authentication data from the context and performing authentication.
5. If the authentication was successful, the server grants the request. If the authentication fails, the server denies service.

The server programmer is responsible for writing the code to extract the authentication data, insert it into the CORBA data structures, and authenticate the user. WebSphere provides a utility class, the `ServerSideAuthenticator` class, that can be used to simplify the CORBA programming needed to do this kind of programmatic login. This class extends the `LoginHelper` class used for client-side login. The `TestServer` application illustrates the use of the `ServerSideAuthenticator` class.

The Component Broker online documentation provides information about the CORBA security classes. This information can be found on the Component Broker documentation website.

Related information...

- [5.4.2.1: The TestServer program](#)
- [5.4.2.2: The ServerSideAuthenticator class](#)
- [5.4.2.3: Accessing secured resources from a Java client](#)
- [Component Broker documentation](#)
- [5.4: Overview: Using programmatic and custom login](#)

5.4.2.1: The security TestServer program



The TestServer program illustrates the use of the ServerSideAuthenticator class, a utility class provided to help simplify programming server-side login. The excerpt below shows the performLoginAndAuthentication method.

TestServer class

```
public class TestServer
{
    ...

    private void performLoginAndAuthentication()
    {
        // Get the user's ID and password.
        String userid = customGetUserid();
        String password = customGetPassword();

        // Ensure immediate authentication.
        boolean forceAuthentication = true;

        // Create a new security context to hold
        // authentication data.
        ServerSideAuthenticator serverAuth = new ServerSideAuthenticator();

        try
        {
            // Perform authentication based on supplied data.
            org.omg.SecurityLevel2.Credentials credentials =
                serverAuth.login(userid, password, forceAuthentication);

            // Retrieve the user's name from the credentials
            // so we can tell the user that login succeeded.
            String username = serverAuth.getUserName(credentials);
            System.out.println("Authentication successful for user: "+username);
        }
        catch (Exception e)
        {
            // Handle exceptions.
        }
    }
    ...
}
```

Related information...

- [5.4.2: Programmatic login on the server side](#)

5.4.2.2: The ServerSideAuthenticator class



The ServerSideAuthenticator class is a WebSphere-provided utility class that provides wrappers around CORBA security methods. It extends the LoginHelper class for use by servers.

The following list summarizes the public methods in the ServerSideAuthenticator class. The source file is installed at:

`<installation_root>/hosts/default_host/examples/security/ServerSideAuthenticator.java`

and the class file is installed at:

`<installation_root>/servlets/ServerSideAuthenticator.class`

ServerSideAuthenticator()

The constructor obtains a new security-context object from the underlying ORB. This object is used to carry authentication information and resulting credentials.

Syntax:

```
ServerSideAuthenticator() throws IllegalStateException
```

login()

This method takes the user's authentication data (identifier and password), authenticates the the user (if the force_authn argument is set to TRUE), and returns the resulting Credentials object.

Syntax:

```
org.omg.SecurityLevel2.Credentials login(String userID, String password,
                                         boolean force_authn)
throws org.omg.SecurityLevel2.LoginFailed,
       com.ibm.IExtendedSecurity.RealmNotRegistered,
       com.ibm.IExtendedSecurity.UnknownMapping,
       com.ibm.IExtendedSecurity.MechanismTypeNotRegistered,
       com.ibm.IExtendedSecurity.InvalidAdditionalCriteria
```

authenticate()

This method does the actual authentication work.

Syntax:

```
org.omg.SecurityLevel2.Credentials authenticate(String userID, String password)
throws org.omg.SecurityLevel2.LoginFailed,
       org.omg.SecurityLevel2.InvalidCredential,
       org.omg.Security.InvalidCredentialType,
       com.ibm.IExtendedSecurity.RealmNotRegistered,
       com.ibm.IExtendedSecurity.UnknownMapping,
       com.ibm.IExtendedSecurity.MechanismTypeNotRegistered,
       com.ibm.IExtendedSecurity.InvalidAdditionalCriteria
```

Related information...

- [5.4.2: Programmatic login on the server side](#)

5.4.2.3: Accessing secured resources from Java clients



A Java client that needs to access a secured resource must know that resource is secured. This page describes how to provide clients with the information they need.

1. Create a text file. In it, specify the necessary property-value pairs:
 - `com.ibm.CORBA.SSLKeyRing=keyring to use`
 - `com.ibm.CORBA.securityEnabled=true`

You can use the file `properties/sas.client.props` installed with WebSphere Application Server as a model.

The `SSLKeyRing` property is required only if you are using digital certificates. For example, a demo keyring is provided as a WebSphere Application Server installation option. To use this keyring (if you installed it), specify:

```
com.ibm.CORBA.SSLKeyRing=com.ibm.websphere.DummyKeyring
```

The demo keyring is intended for demonstration purposes only, not for use in a production environment. In a production environment, you must create site-specific keyrings.

2. When you start the client, load the properties file you just created immediately following the "java" command and preceding the application class name.

Specify the properties file on the command line as follows: `-D`

```
com.ibm.CORBA.ConfigURL=URL of properties file created in previous step
```

For example, to load a properties file called `my.client.props` located in the product installation directory for a client called `MyClient App`:

```
java -D com.ibm.CORBA.client.ConfigURL=file:/install_root/properties/my.client.props MyClientApp
```

Related information...

- [5.4.2: Programmatic login on the server side](#)

5.4.3: Custom login challenges



Applications can present site-specific login forms by making use of WebSphere's custom challenge type. A custom challenge works in the following manner:

1. An unauthenticated user attempts to use a resource secured with a custom challenge.
2. The user is redirected to the LoginURL, which takes the user to an HTML form that collects authentication information.
3. The user inserts his or her user ID and password into the form and submits it.
4. The submission triggers a servlet that authenticates the user.

WebSphere provides two servlets that can be used as a basis for writing custom-challenge servlets, an `AbstractLoginServlet` and the `CustomLoginServlet`, which extends the `AbstractLoginServlet`.

In web-based applications, it is often desirable to maintain login information across multiple sites so each site doesn't have to require the user retype the information. You can use the WebSphere single sign-on framework to allow the authentication information to be passed along automatically. WebSphere provides a helper class called `SSOAuthenticator` that simplifies the handling of single sign-on.

Related information...

- [5.4.3.1: The AbstractLoginServlet class](#)
- [5.4.3.2: The CustomLoginServlet class](#)
- [5.4.3.3: The SSOAuthenticator class](#)
- [5.4: Overview: Using programmatic and custom login](#)

5.4.3.1: The AbstractLoginServlet class



The AbstractLoginServlet class that encapsulates all of the functionality of a server side login.

The following list summarizes the public methods in the AbstractLoginServlet class. The source and class files are installed in the directory

`<installation_root>/servlets/`

.

init()

This method initializes the servlet.

Syntax:

```
void init() throws ServletException
```

login()

This authenticates the user and, if requested, sets up the necessary information for single sign-on.

Syntax:

```
Object login (HttpServletRequest req, HttpServletResponse res,  
              boolean setSSO)  
throws ServletException
```

postLogin()

This is an abstract method.

Syntax:

```
void postLogin(HttpServletRequest req, HttpServletResponse res)  
throws ServletException
```

logout()

This destroys the user's credentials and, if necessary, unsets the single sign-on information.

Syntax:

```
void logout(HttpServletRequest req, HttpServletResponse res,  
            Object retCreds)  
throws ServletException
```

Related information...

- [5.4.3: Custom login challenges](#)

]



5.4.3.2: The CustomLoginServlet class

The CustomLoginServlet extracts the user ID, password, and redirect URL information from the HTML form by which the user logged in. The redirect URL specifies the Web site to which the user is requesting access. The CustomLoginServlet invokes the necessary methods on its parent servlet, AbstractLoginServlet.

The following list summarizes the public methods in the CustomLoginServlet class. The source and class files are installed in the directory

```
<installation_root>/servlets/
```

.

init()

This method initializes the servlet, reading the default redirect URL if it exists. The default redirect URL specifies the Web page to which the user will be forwarded if authentication is successful.

Syntax:

```
void init(ServletConfig conf) throws ServletException
```

doPost()

The primary entry point into the servlet, this method is designed to be called as the result of an HTML form post. The method reads and validates the posted parameters, then calls the login method in the base class LoginServlet.

Syntax:

```
void doPost(HttpServletRequest req, HttpServletResponse res)  
throws ServletException, IOException
```

postLogin()

This method is called after performing a successful login, so it is a good place to establish an HTTP session or perform other actions related to the logged-on user. This method runs under the identity of the user.

Syntax:

```
void postLogin(HttpServletRequest req, HttpServletResponse res)  
throws ServletException
```

Related information...

- [5.4.3: Custom login challenges](#)

5.4.3.3: The SSOAuthenticator class



The SSOAuthenticator class is a WebSphere-provided utility class that can be used by servlet developers to write custom-login servlets.

The following list summarizes the public methods in the SSOAuthenticator class. The class files is installed at `<installation_root>/servlets/SSOAuthenticator.class`

SSOAuthenticator()

The constructor creates an SSOAuthenticator object and initializes its based on the SSO configuration within WebSphere Application Server. These three conditions must be met for successful construction:

- WebSphere security is enabled
- LTPA is selected as the authentication mechanism
- Single Sign-On (SSO) is enabled

Syntax:

```
SSOAuthenticator() throws IllegalStateException
```

login()

These methods create an LPTA cookie and set the cookie on the HTTP response header. The first method takes a boolean argument, `force_auth`, whose value determines if the user (based on the identifier and password) is authenticated. If `force_auth` is `TRUE`, authentication occurs; if not, only the identifier and password are used in the cookie.

The second login method does not take the `force_auth` argument. It always attempts authentication and is equivalent to calling the first with the `force_auth` argument set to `TRUE`.

Both methods return CORBA security credentials.

Syntax:

```
org.omg.SecurityLevel2.Credentials login(String userID, String password,
                                         HttpServletRequest req,
                                         HttpServletResponse res
                                         boolean force_auth)
throws org.omg.SecurityLevel2.LoginFailed
```

```
org.omg.SecurityLevel2.Credentials login(String userID, String password,
                                         HttpServletRequest req,
                                         HttpServletResponse res)
throws org.omg.SecurityLevel2.LoginFailed
```

logout()

This method logs the user out. After this method runs, the user must be authenticated again before making any additional requests.

Syntax:

```
void logout(HttpServletRequest req, HttpServletResponse res)
```

Related information...

- [5.4.3: Custom login challenges](#)

7: Distribute and load balance



This section discusses topics related to multiple-machine environments, such as workload managed environments.

Related information...

- [7.2: Managing workloads](#)

7.2 Managing workloads



Workload management optimizes the distribution of client processing tasks in the WebSphere Application Server environment. Incoming work requests are distributed to the application servers and other objects that can most effectively process the requests. Workload management also provides failover when servers are not available.

Workload management is most effective when used in systems that contain servers on multiple machines. It also can be used in systems that contain multiple servers on a single, high-capacity machine. In either case, it enables the system to make the most effective use of the available computing resources.

Implementing workload management

The Advanced application server implements workload management for application objects as follows.

1. By using clones and models. They can be created for any object - from individual enterprise beans and servlets to entire application servers.
2. Enterprise beans and servlets have additional steps for implementing workload management.
 - For enterprise beans, by using workload management-enabled Java Archive (JAR) files. These JAR files can be created by using either the Administrative console or the **wlmjar** utility.
 - For servlets, by using servlet redirection to route client requests to remote servlet clones.

Administration servers can also participate in workload management.

Benefits of workload management

Workload management provides the following benefits to WebSphere applications:

- It balances client work loads, allowing processing tasks to be distributed according to the capacities of the different machines in the system.
- It provides failover capability by redirecting client requests if one or more servers is unable to process them. This improves the availability of applications and administrative services.
- It enables systems to be scaled up to serve a higher client load than provided by the basic configuration. With cloning and modeling, additional instances of servers, servlets, and other objects can easily be added to the configuration.
- It enables servers to be transparently maintained and upgraded while applications remain available for users.
- It centralizes administration of servers and other objects.

Related information...

- [0.22.1: What is workload management?](#)
- [7.2.1 Workload management for enterprise beans](#)
- [7.2.2 Workload management for servlets](#)
- [7.2.3 Workload management for administration servers](#)
- [7.2.4 Using models and clones](#)
- [7.2.5 Using workload management - a sample procedure](#)

7.2.1 Workload management for enterprise beans



Workload management for enterprise beans is enabled as follows:

1. [Create models and clones](#) of enterprise beans, containers, and application servers.
2. Create a [workload management-enabled Java Archive \(JAR\) file](#).

Creating models and clones

Models and clones can be created for objects at any level of the containment hierarchy. A model contains the objects that are deployed into the object from which it is created. For example, individual enterprise beans can be modeled and cloned without cloning the container in which they are deployed. Clones of containers include the enterprise beans that are deployed into the modeled container. Clones of application servers include the containers and enterprise beans that are deployed onto their model, and so forth.

Creating a model and clones of an application server and enterprise beans is the first step in enabling workload management of enterprise beans. Models must be created at the application server level for workload management to function. Client processing requests are managed and distributed among the clones of the server and enterprise beans.

Clones of enterprise beans, containers, and application servers can be created by using the WebSphere Administrative Console.

Creating workload management-enabled JAR files

A workload management-enabled JAR file enables EJB clients to access the enterprise beans through the workload management service. These JAR files can be created in two ways:

- By using the **wlmjar** command-line utility. Run this utility on a JAR file into which you have deployed one or more enterprise beans.
- By enabling workload management through the WebSphere Administrative Console. When you are deploying a JAR file (or specifying a deployed JAR file when creating an enterprise bean using the Administrative Console), you can enable workload management of the JAR file. This option is easier to use than the **wlmjar** utility.

Regardless of which method you use to enable workload management, the resulting JAR file contains stub code that allows EJB clients to access enterprise beans through the workload management service. Simply set the value of the client's CLASSPATH environment variable to the location of the workload management-enabled JAR file.

In general, deploy JAR files with workload management enabled. It does not affect their normal behavior unless models and clones are created, at which point workload management is activated.

Related information...

- [4.3 Writing enterprise beans](#)
- [6.6.5 Administering enterprise beans](#)

7.2.2 Workload management for servlets



Workload management for servlets can be enabled as follows:

1. [Create models and clones](#) of servlets.
2. Use [one of a few available methods](#) to distribute servlet processing across multiple machines.

Creating models and clones

Creating models and clones of servlets is the first step in enabling workload management for servlets. They can be created by using the WebSphere Administrative Console.

Related information...

- [6.6.13 Administering servlet redirectors](#)
- [7.2.4 Using models and clones](#)

7.2.3 Workload management for administration servers



Administration servers can participate in workload management. Workload management provides failover capability, improving the availability of administrative and naming services.



Workload management must be enabled (or disabled) for all administration servers in a domain.

When an administration server participates in workload management, an exception is thrown if the administration server fails during an administrative task. Subsequent requests are redirected to the other administration servers in the domain, minimizing the disruption to administrative operations.

For example, a command issued through the WebSphere Administrative Console can fail if an administration server goes off-line while the command is being executed. If workload management is enabled, any subsequent attempts to execute the command are redirected to another administration server. This allows the command to be successfully reissued, possibly with a delay for the initial redirection. Subsequent requests will not be noticeably slower. The original administration server will pick up its share of administrative requests when it comes back on-line.

Enabling workload management

To begin workload management, start all administration servers in the domain with workload management enabled. WebSphere Application Server provides two ways to enable workload management:

- By setting the following property in the `admin.config` file:

```
com.ibm.ejs.sm.AdminServer.wlm=true
```

This enables workload management for all administration servers that are started using this configuration file. (The actual value of the property is irrelevant; workload management is enabled as long as it is set in the file.)

- By specifying the `-wlm` argument when starting an administration server from the command line. For instance:

```
java com.ibm.ejs.sm.server.AdminServer -wlm ...
```

where ... represents any other arguments that are specified when starting the server.

Enabling workload management through the `admin.config` file is recommended because it is easier to administer than enabling it through the command line.

Disabling workload management

To discontinue workload management, stop all administration servers in the domain and restart

them with workload management disabled. Disable workload management in one of the following ways:

- By commenting out the `com.ibm.ejs.sm.AdminServer.wlm` property from the `admin.config` file.
- By restarting the server from the command line without specifying the `-wlm` argument.

Related information...

- [6 Administer applications](#)
- ?? Link to instructions on starting admin servers

7.2.4 Using models and clones



A *model* is a template for creating copies of a server or process instance, such as an application server or servlet engine. The copies are called *clones*. The act of creating the clones is called *cloning*.

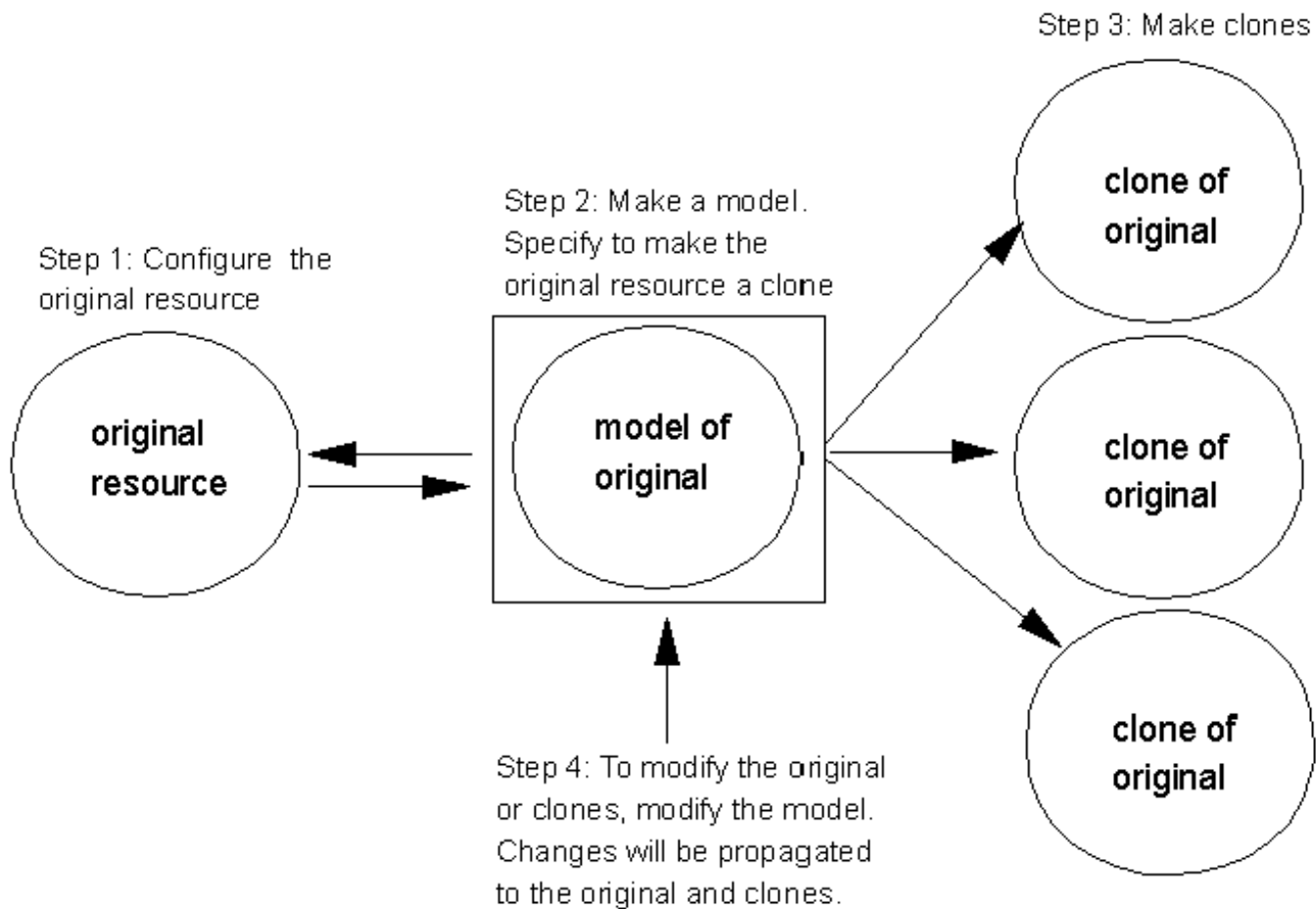
Cloning and modeling allows identical copies of objects (such as servlets, application servers, and so forth) to be created. A system administrator first creates a model that abstractly represents an object. From this model, one or more clones can be created. The clones represent real application server processes; when first created, they are identical to the model in every way.

Changes to a model are propagated to its clones when the clones are restarted. You can efficiently administer several copies of a server or other resource by administering its model.

Modeling and cloning objects at different levels of the application server containment hierarchy (from individual servlets and enterprise beans to entire application servers) gives administrators a great deal of flexibility for implementing and administering applications. For example, a system administrator can create models of an application server, servlets, containers and enterprise beans; adjust the properties of the models to optimize the performance of these objects; then deploy them by creating and starting clones from these models.

Working with models and clones

The following diagram illustrates the basic procedure for using models and clones:



First, create the original instance of the object that you want to clone (such as an application server, servlet, or enterprise bean). Configure it exactly as you would like it.

Next, create a model of the object by using the Administrative Console. When changes are necessary, apply them to the model, which in turn modifies the original instance (which is now a clone) and the other clones.

Making the original instance a clone is recommended but not required. The original instance can remain freestanding.

Determining which resources can be cloned

To tell whether a resource can be cloned, right-click the resource in the Topology tree to display a pop-up menu. If the menu contains a Create -> Model option, the resource can be cloned.

Resources that can be cloned include:

- Application servers
- EJB containers
- Enterprise beans
- Servlet engines
- Web applications
- Servlets

Related information...

- [6.6.22 Administering cloned applications](#)
- [7.2.4.1 Cloning for workload management, failover, and scaling](#)
- [7.2.4.2 Modifying models and clones](#)
- [7.2.4.3 Advice for cloning](#)
- [7.2.4.4 Containment relationships](#)
- [7.2.4.5 Considerations for cloning servers](#)

7.2.4.1 Cloning for workload management, failover, and scaling



Models and clones are used for workload management, failover, and scaling.

Workload management

Models and clones provide necessary support for workload management. When you change a model, the change is propagated to its clones when they are restarted. Besides making it easy to administer several servers as one logical server, this keeps the clones identical so that requests can be routed to any one of them, with the same results.

Being able to route a request to any server in a group of identical servers allows the servers to share work, improving throughput of client remote method invocations. Requests can be evenly distributed to servers to prevent workload imbalances in which one or more servers have idle or low activity while others are overburdened. This load balancing activity is a benefit of workload management.

Failover

With several clones available to handle requests, it is more likely that failures will not damage throughput and reliability. With clones distributed to various nodes, an entire machine can fail without producing devastating consequences (unless, of course, the failed machine is a single point of failure). Requests can be routed to other nodes if one node fails.

Scaling

Cloning is an effective way to perform horizontal and vertical scaling of application servers.

- In *horizontal scaling*, clones are defined on multiple machines in a system. This allows a single WebSphere application to run on several machines while presenting a single system image, making the most effective use of the resources of a distributed computing environment. Horizontal scaling is especially effective in environments that contain many smaller, less powerful machines. Client requests that would overwhelm a single machine can be distributed over several machines in the system. Failover is another benefit of horizontal scaling. If a machine becomes unavailable, its work can be routed to other machines containing server clones.
- In *vertical scaling*, clones are defined on a single machine to allow the machine's processing power to be more efficiently allocated. Vertical scaling is particularly useful if your environment contains large, under-utilized machines. A single application server is implemented by a single Java Virtual Machine (JVM) process and cannot fully utilize the power of a large machine. (This is especially true on large multiprocessor computers because of concurrency limitations within a single JVM process.) Vertical scaling allows multiple application server clones (and therefore JVM processes) to be created, which makes use of the machine's processing power more effectively.

Related information...

- [7.2.4: Using models and clones](#)

7.2.4.2 Modifying models and clones



To perform an administrative action on a clone (such as modifying the clone's properties), perform the action on the associated model. For example, to add or remove an enterprise bean from an application server, you must add or remove the bean in the server model. With one action, you can add or remove the enterprise bean from all clones of the application server.

Changes related to workload management (such as selection policy changes, starting clones, and stopping clones) are propagated to workload management clients. Other model changes (such as adding or removing enterprise beans from an application server) are picked up by the clones when they are restarted. System administrators can perform a *ripple restart* to propagate changes by stopping and restarting clones one after another without stopping the model.

If you modify a clone directly (instead of through its model), the clone no longer is identical to its model. However, it continues to be part of its model unless it is dissociated from the model.

Freestanding (disassociated) clones

A clone must be explicitly disassociated from its model. It then becomes a *freestanding* object and can be administered independently. Any changes you make to the former model of the clone will **not** be propagated to the clone.

Freestanding clones can be created by using the **wscp** command-line utility.

Related information...

- [6.6.22 Administering cloned applications](#)
- Need link to wscp help; couldn't find the file name

7.2.4.3 Advice for cloning



Create clones based on your knowledge of the application and on expected workload. Some considerations:

- Clients can have inconsistent views of configuration information in the model. This can occur when a server instance is stopped, started, added, or deleted.

The period of inconsistency is short-lived, however. Clients eventually refresh their caches of server model information. Server instances that are unchanged during the period of inconsistency remain available.

- If you make changes to a model, clients do not need to be restarted. The changes are eventually propagated to them.
- You can make changes to a model while servers are running. However, incremental changes (such as adding or removing one or two clones) have less impact on client performance than wholesale changes.
- It is always best to make changes when few clients and servers are running.
- You can add server instances as the load increases, or you can configure the initial number of instances based on expected load.
- If a machine becomes unavailable, you do not need to reconfigure clones of other servers to reflect unavailable servers on that machine. However, if the machine will be unavailable for an extended period, you can reconfigure the model to optimize performance.

7.2.4.4 Containment relationships



While cloning a resource, you can specify the relationship of the clone to its model and other models.

Freestanding or contained?

Specify a containment relationship for each model, which determines whether the model is freestanding or contained.

A model is *freestanding* if it is not contained by another resource's model. If a model is part of another resource's model, it is *contained*.

For example, a model of an enterprise bean can be *contained* by the model of an application server on which the bean is installed. Clones of the application server will contain clones of the enterprise bean.

Model recursively?

When you create a servlet engine model that contains a Web application, you can optionally specify to include the Web application, and any servlets it contains, in the model.

Note that containment relationships are preserved when you model and clone an instance that is contained by another instance. For example, if you create a model of a Web application that is contained by a particular servlet engine, the clones of the Web application model will also be associated with that servlet engine.

7.2.4.5 Considerations for cloning servers



When you are cloning an application server, you need to take the following things into account:

- [Application server models and clones](#)
- [Server selection policies](#)
- [Transaction affinity for application servers](#)

Application server models and clones

A *servlet engine* is a server process that works with your Web server to handle requests for servlets and Web resources such as HTML, JavaServer Pages (JSP) files, and such. A server object, such as an application server or servlet engine, can be used as a model for creating multiple clones of that server. The clones are defined by a model. Changes to the model are propagated to the clones when the server clones are restarted.

The clones remain basically identical to the model, allowing work to be distributed to any one of them. [Server selection policies](#) determine how clients choose server instances within the group.

Clones are created individually after creating the model. You can add or remove server clones later in response to the load on the application. Clones do not need to reside on the same machine.

Server selection policies

A server selection policy defines how clients choose among server clones (instances). Select among these policies:

Random

Server instances are selected randomly from within the group of clones associated with a model.

Round robin

A server instance is initially selected at random from an ordered list. Other server instances are selected from the ordered list in turn, until the initially selected server is selected again. The sequence repeats.

If a particular server instance is stopped or otherwise unavailable, that instance is skipped (no attempt is made to select it) until it can be verified as being back in service.

Random prefer local

Server instances on the same host as the client are selected according to the random server selection policy. If no local server instances are available, server instances on remote hosts are selected according to the random server selection policy.

Round robin prefer local

Server instances on the same host as the client are selected according to the round robin

server selection policy. If no local server instances are available, server instances on remote hosts are selected according to the round robin server selection policy.



The random prefer local and round robin prefer local policies try to select instances within the same Java Virtual Machine (JVM) process. For instance, if a client and an enterprise bean clone are located within the same JVM process, the local clone is always picked over a clone in a separate JVM process.

Transaction affinity for application servers

Regardless of the selection policy used, the client runtime attempts to choose an application server instance based on *transaction affinity*. Within a transaction, the first time a server is picked, the prevailing selection policy for the server group is applied. After a server is selected, it remains bound for the duration of the transaction.

For example, suppose the round-robin policy is specified for server group A with two servers, S1 and S2. A client has two concurrent threads, t1 and t2, with transaction contexts T1 and T2, respectively. Suppose t1 is first and needs to select a server from server group A. Suppose S2 is randomly chosen. When t2 tries to select a server from server group A, S1 is chosen based on the round-robin policy in effect for the server group. Subsequent requests to server group A are serviced by S2 for t1 and S1 for t2, based on transaction affinity.

7.2.5 Using workload management - a sample procedure



The following procedure shows how to implement basic workload management by cloning application servers and enterprise beans. In this scenario, client requests are distributed among the clones of the application server and enterprise beans on a single machine. (A client refers to any servlet, Java application, or other program or component that connects the end user and the enterprise beans being accessed.) In more complex workload management scenarios, you can clone servlets, distribute clones to remote machines, or configure a servlet redirector.

1. Decide which application server you are going to clone.
2. Deploy the enterprise beans that you plan to clone. Optionally, chose the deployment option to enable workload management for the JAR file.
3. After configuring the server and enterprise beans exactly as you want them to be, create a model of the server. This is the first step in cloning the server. Make sure that the model includes the enterprise beans that are deployed on the server. It is recommended that you make the original server instance a clone that is administered through the model.
4. Create one or more clones of the server model.
5. Start all of the application servers by starting the model.
6. If you did not enable workload management when you were deploying the enterprise beans, use the **wlmjar** command against the deployed JAR file of the enterprise bean to produce a WLM-enabled JAR file.
7. Add the WLM-enabled JAR file to the classpath of the client you want to enable to exercise workload management.
8. If the client is a servlet, also specify the WLM-enabled JAR file in the classpath of the application server on which the servlet resides.



You need to define a bootstrap host when a client is on a different machine than the application server and there is no administrative server for the client. Add the following line to the Java Virtual Machine (JVM) arguments for the client:

```
-Dcom.ibm.CORBA.BootstrapHost=machine_name
```

where *machine_name* is the name of the machine on which the administrative server is running.

Related information...

- [6.6.5: Administering enterprise beans](#)
- [6.6.22: Administering cloned applications](#)

3.3: Migrating APIs and specifications



IBM WebSphere Application Server supports a wide variety of technologies for building powerful enterprise applications. As technology advances, particularly in the area of Java components, the Application Server product releases advance to support and extend the most contemporary open specification levels.

If your existing applications currently support different specification levels than are supported by Application Server Version 3.5, it is likely you will need to update at least a few aspects of the applications to comply with the new specifications.

See [article 3.3.2a](#) for migration considerations for WebSphere Application Server Version 3.5.2.

In many cases, IBM extends the specification levels currently supported by IBM WebSphere Application Server to provide additional features and customization options. If your existing applications use extensions from past Application Server releases, mandatory or optional migration could be necessary to utilize the same kinds of extensions in Version 3.5.

From Version 3.0x to Version 3.5, the main migration areas concern the IBM extensions and the JDK. In contrast, migrating from Version 2.0x requires updating applications with respect to the open specifications, such as the Java Servlet API.

The table summarizes potential migration areas. See the Related information below for instructions pertaining to each area.

Functional area	Version 3.5.x support	Need to migrate from V3.0x?	Need to migrate from V2.0x?	Details
Enterprise beans	EJB 1.0 Specification	No*	Yes	* Although there are no EJB API changes for Version 3.5, changes in the underlying JDK 1.2 prerequisite require enterprise beans to be deployed again. Version 2.0x offered only limited EJB 1.0 Specification support

Servlets	Servlet 2.1 Specification and IBM extensions	No	Yes	Version 2.0x supported the Servlet 2.0 Specification and IBM extensions that were updated in Version 3.0x
Servlets	Servlet 2.2 Specification	No	Yes	Article 4.2.1.2.1a describes the new Servlet 2.2 APIs.
JSP files	Supported JSP specifications are: <ul style="list-style-type: none"> ● JSP 1.1 Specification recommended ● JSP 1.0 Specification supported ● JSP .91 Specification also supported 	No**	Yes	Version 2.0x supported only the JSP .91 Specification. ** If you did not already migrate JSP .91 files for use with Version 3.0x, small migration is required for use with Version 3.5 It is recommended you migrate to JSP 1.0, despite .91 support.
XML	XML 2.0.x supported XML 1.1.x supported with restrictions	No***	No***	*** Migration from 1.1.x to 2.0.x is not required, but you might decide to migrate based on criteria and 1.1.x restrictions described in section 3.3.4 (see Related information)

JDBC and IBM database connection support APIs	JDBC 2.0; connection pooling model	No	Yes	V2.0x supported JDBC 1.0 and a connection manager model. If still using connection manager in Version 3.0x, it is recommended you switch to connection pooling
User profiles	IBM user profile APIs	No	Yes	Need to migrate from V2.0x deprecated classes for use with V3.0x or V3.5
Sessions	IBM session support APIs	No	Yes	Need to migrate from V2.0x deprecated classes, changes to clustering, URL encoding for use with V3.0x or V3.5
Security	IBM security support	No	No	No action required
Transactions	Java 1.2 transactions support	Yes	Yes	Version 3.0x provided proprietary IBM packages to simulate Java 1.2 functionality. Version 2.0x did not provide any support. Migrate to Version 3.5 if your applications require this kind of support.

XML configuration	XMLConfig tool	Yes	Yes	The XML Configuration Management Tool (XMLConfig) was introduced in Version 3.02. Some of the interfaces have changed in Version 3.5
-------------------	----------------	-----	-----	--

Related information...

- [3.3.1: Migrating to supported EJB specification](#)
- [3.3.2: Migrating to supported Servlet specification and extensions](#)
- [3.3.3: Migrating to supported JSP specification](#)
- [3.3.4: Migrating to supported XML specification](#)
- [3.3.5: Migrating to supported user profile APIs](#)
- [3.3.6: Migrating to supported session APIs](#)
- [3.3.7: Migrating to supported security APIs](#)
- [3.3.8: Migrating to supported database connection APIs \(and JDBC\)](#)
- [3.3.9: Migrating to supported transaction support](#)
- [3.3.10: Migrating to supported XML configuration](#)
- [3: Migration overview](#)

3.3.1: Migrating to supported EJB specification



Migrating from Version 3.0x

The EJB specification level for Version 3.5 has not changed from that of Version 3.0x, however changes due to the prerequisite of JDK 1.2 are required:

- In Version 3.0, the JavaSoft standard packages:

```
javax.sql.*  
javax.transaction.*
```

were present under non-standard names. In Version 3.5, they are present under their standard names.

Any code using WebSphere Application Server data sources, including BMP entity beans and session beans that access databases, will need to be modified.

See articles 3.3.8 and 3.3.9 for instructions.

- Some of the stub classes for deployed enterprise beans have changed in JDK 1.2. [Repdeploy all EJB server JAR files](#) to generate the correct stub file references.

Be aware that, in general, JAR files generated prior to JDK 1.2 are source and binary code compatible on a JDK 1.2 base. However, there are some incompatible cases. For further details, see:

<http://java.sun.com/products/jdk/1.2/compatibility.html>

Migrating from Version 2.0x

No action is required.

Related information...

- [3.3.8: Migrating to supported database connection APIs](#)
- [3.3.9: Migrating to supported transaction support](#)
- [3.3: Migrating APIs and specifications](#)

3.3.2: Migrating to supported Servlet specification and extensions



Servlets will require migration if they are not of the supported specification level (2.1) or they rely on deprecated or removed IBM servlet extensions.

See [article 3.3.2a](#) for migration considerations for WebSphere Application Server Version 3.5.2.

Migrating to the supported Servlet specification

Refer to the [Java Servlet API 2.1 specification](#) for complete information concerning new and deprecated APIs. This table highlights a few of the new and deprecated classes and methods.

Recall, IBM WebSphere Application Server supported the Servlet 2.1 Specification; if you already migrated servlets to 2.1 for use with that release, no further action is required.

[Article 4.2.1.2.1a](#) describes the new Servlet 2.2 APIs.

Method or Class	Status and recommendation
RequestDispatcher	New. Use the forward method to forward a servlet response from one servlet to a second servlet for further processing. Use the include method to include part of the one servlet's response in the body of another servlet's response. Refer to the code example .
HttpSessionContext	Deprecated.
HttpSession.getSessionContext	Deprecated. For security reasons, no equivalent.
HttpSession.getMaxInactiveInterval	New. Sets the maximum time a session will be maintained by the servlet engine without a client request.
ServletRequest.getRealPath	Deprecated. Use <code>ServletContext.getRealPath</code> .
ServletContext.getServlet	Deprecated. Use <code>ServletContext.getRequestDispatcher</code> .
ServletContext.getResource	New. Use this method to obtain a servlet resource by requesting its URL.

ServletContext.getResourceAsStream	New. Use this method to obtain a servlet resource (as an InputStream) from its servlet context.
encodeUrl and encodeRedirectUrl methods of HttpServletResponse	Deprecated. But the fix is easy. Change <code>Ur1</code> to <code>URL</code> in the method names.
HttpSession.isRequestedSessionIdFromUrl	Deprecated. Another easy fix. Change <code>Ur1</code> to <code>URL</code> in the method name.
HttpServiceRequest.setAttribute()	Deprecated. See Migrating to supported JSP specification for details.
HttpServiceResponse.callPage()	Deprecated. Refer to the code example .

Migrating IBM extensions to the Servlet API

The following packages were part of the Application Server Version 2.0x, but were removed or deprecated as of Version 3.

Method or Class	Status and recommendation
com.ibm.servlet.personalization.sam()	Removed -- no recommended replacement
com.ibm.servlet.servlets.personalization.util	Removed -- no recommended replacement
com.ibm.servlet.connmgr	<p>Deprecated. Starting with Version 3.0x, IBM WebSphere Application Server has provided a built-in connection pooling function that eliminates the need for servlet programmers to use the connection manager APIs directly. Instead, servlets can be written to use the JDBC APIs to access the connection pool.</p> <p>You are encouraged to migrate servlets that use the deprecated connection manager APIs. See the Related information for details.</p>

Related information...

- [3.3.2.1: Example: Migrating HttpServiceResponse.callPage\(\)](#)
- [3.3.8: Migrating to supported database connection APIs \(and JDBC\)](#)
- [3.3: Migrating APIs and specifications](#)

Example: Migrating `HttpServletResponse.callPage()`



Calls to `HttpServletResponse.callPage()` need to be replaced by calls to `RequestDispatcher`, as shown.

Before -- Using `HttpServletResponse.callPage()`

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

public class UpdateJSPTest extends HttpServlet
{
    public void doGet (HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException
    {
        String message = "This is a test";
        ((com.sun.server.http.HttpServletRequest)req).setAttribute("message",
message);
        ((com.sun.server.http.HttpServletResponse)res).callPage("/Update.jsp", req);
    }
}
```

After -- Using `RequestDispatcher`

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

public class UpdateJSPTest 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);
        //((com.sun.server.http.HttpServletRequest)req).setAttribute("message",
message);
        //((com.sun.server.http.HttpServletResponse)res).callPage("/Update.jsp",
req);
    }
}
```

● [Related information...](#)

3.3.3: Migrating to supported JSP specification



If using JSP 1.0 already, no action is required, with one exception, Version 3.5 does not have Bean Scripting Framework (BSF) support for using LotusXSL and other scripting languages in JSP 1.0 files. Version 3.0x applications depending on this support will require modification.

Version 3.5.2 supports the Bean Scripting Framework (BSF) for Netscape's Rhino JavaScript language. See [article 4.2.5](#) for more information.

Version 3.5.2 also supports JSP 1.1 and continues to support JSP 1.0 and JSP .91 files. See [article 3.3.2a](#) for migration details.

IBM WebSphere Application Server Version 3.5 continues to support the JSP .91 Specification, but it is recommended that you migrate applications containing JSP .91 files to JSP 1.0, if you have not already done so.

See the Related information for the migration steps required to use JSP .91 files with Version 3.5, and tips for migrating them to JSP 1.0 instead.

Related information...

- [3.3.3.1: Updating JSP .91 files for use with Version 3.5](#)
- [3.3.3.2: Tips for migrating JSP .91 files to JSP 1.0](#)
- [3.3: Migrating APIs and specifications](#)
- [3.3.2a: New Servlet Engine option for migrating Applications to Servlet 2.2](#)

3.3.3.1: Updating JSP .91 files for use with Version 3.5



If using JSP .91 files or servlets that you have not already been using with Version 3.0x, and the JSP .91 files or servlets cast to either of these methods:

- `com.sun.server.http.HttpServletRequest`
- `com.sun.server.http.HttpServletResponse`

either replace the deprecated calls **or** recompile the files.

Additionally, if migrating JSP .91 files last used with IBM WebSphere Application Server Version 1.x, you need to eliminate `<REPEATGROUP>` tags. See below for details.

Option 1: Modify deprecated calls

The tables summarize calls to the deprecated `HttpServletRequest` and `HttpServletResponse` classes, and provide replacement code.

Before:	<code>com.sun.server.http.HttpServletRequest.setAttribute()</code>
After:	<code>javax.servlet.http.HttpServletRequest.setAttribute()</code>

A [code example](#) is provided to show how to migrate to `RequestDispatcher`:

Before:	<code>com.sun.server.http.HttpServletResponse.callPage()</code>
After:	<code>javax.servlet.RequestDispatcher</code>

Option 2: Recompile files

As an alternative to replacing the deprecated calls to `HttpServletRequest` and `HttpServletResponse`, recompile your JSP .91 files or servlets developed for Application Server Version 2.0x before using them with Application Server Version 3.5.

Recompiling is necessary because starting with Version 3.0x, `HttpServletRequest` and `HttpServletResponse` are provided as interfaces (instead of classes) that are implemented by the WebSphere servlet engine.

If you do not recompile the servlets or JSP files, the Java Virtual Machine (JVM) will crash on Windows NT systems due to a suspected bug in the JDK.

It is possible that you already recompiled the files for use with Version 3.0x. In such a case, it is not necessary to compile them again for Version 3.5.

Migrating JSP .91 files from IBM WebSphere Application Server Version 1.x

The Application Server Version 1.x supported an additional tag, `<REPEATGROUP>` for repeating a block of HTML tagging for data that is already logically grouped in the database. Because this release does not support the `<REPEATGROUP>` tag, remove that tag from any JSP files that you want to use on the Application Server Version 3.5.

Related information...

- [3.3.3: Migrating to supported JSP specification](#)

3.3.3.2: Tips for migrating JSP .91 files to JSP 1.0



Referring to WebSphere example code for the purposes of illustration, the tips below cover some main steps in migrating JSP .91 to JSP 1.0.

Replacing <SERVLET> with <jsp:include>

Use the JSP 1.0 equivalent of <SERVLET> to include data in a JSP page from another file.

Example	CounterServletOutputPage.jsp
JSP .91	<pre><SERVLET CODE="WebSphereSamples.Counter.CounterServlet"></SERVLET></pre>
JSP 1.0	<pre><jsp:include page="/servlet/WebSphereSamples.Counter.CounterServlet" /></pre>
Discussion	<p>The CounterServletOutputPage.jsp file and the servlet it invokes are part of the Version 3.5 Web application named WSamples_app, with the Web Application Web Path setting "/WebSphereSamples."</p> <p>Using a WebSphere administrative client to view the WSamples_app Web application, you will find that it contains the Auto-Invoker servlet, which enables you to call servlets by classname.</p> <p>Specified within the Auto-Invoker servlet is the Servlet Web Path List, which has the single entry "default_host/WebSphereSamples/servlet." Now, the JSP and CounterServlet servlet are both under the Web Application Web Path of /WebSphereSamples. Relative to /WebSphereSamples, the servlet needs the additional qualifier of /servlet to properly locate it (as specified in the Auto-Invoker). This results in the "/servlet/WebSphereSamples.Counter.CounterServlet" in the <jsp:include> tag.</p>

Replacing <BEAN> with <jsp:useBean>

Use the JSP 1.0 equivalent of <BEAN> to make an existing or newly created bean available from within the JSP file. Four variations are possible.

Variation 1: JSP is to create the bean

Example	PollServletInputPage.jsp
JSP .91	<pre><BEAN NAME="getQuestionDBBean" TYPE="WebSphereSamples.Poll.GetQuestionDBBean" CREATE="YES" INTROSPECT="YES" SCOPE="request"> </BEAN></pre>
JSP 1.0	<pre><jsp:useBean id="getQuestionDBBean" type="WebSphereSamples.Poll.GetQuestionDBBean" class="WebSphereSamples.Poll.GetQuestionDBBean" scope="request" /></pre>
Discussion	<p>You no longer have the explicit attribute of CREATE="YES". Instead, if the bean with the name specified by the id attribute is not found within the specified scope, then an instance of bean will be created according to the class attribute.</p> <p>JSP NAME attribute corresponds to the JSP 1.0 id attribute. It is no longer an INTROSPECT attribute. (The JSP .91 scope of requests and sessions carry over to JSP 1.0, plus some new ones for JSP 1.0.) Compare with variation 1 with variation 2.</p>

Variation 2: JSP is to use existing bean

Example	PollServletResultPage.jsp
JSP .91	<pre><BEAN NAME="pollQueryDBBean" TYPE="WebSphereSamples.Poll.PollQueryDBBean" CREATE="NO" INTROSPECT="NO" SCOPE="request"> </BEAN></pre>
JSP 1.0	<pre><jsp:useBean id="pollQueryDBBean" type="WebSphereSamples.Poll.PollQueryDBBean" scope="request" /></pre>
Discussion	Compare variation 2 with variation 1, which creates a bean if one does not exist. JSP 1.0 version no longer has the class attribute from JSP .91. If a bean instance corresponding to the id attribute not found in the specified scope, there will be an error. As a result, the bean will not be created.

Variation 3: Properties are to be set for bean

Example	CenterGeneric.jsp
JSP .91	<pre><BEAN NAME="getQuestionDBBean" TYPE="WebSphereSamples.YourCo.Poll.GetQuestionDBBean" CREATE="YES" INTROSPECT="NO" SCOPE="request"> <PARAM NAME="userID" VALUE="wsdemo"> </BEAN></pre>
JSP 1.0	<pre><jsp:useBean id="getQuestionDBBean" type="WebSphereSamples.YourCo.Poll.GetQuestionDBBean" class="WebSphereSamples.YourCo.Poll.GetQuestionDBBean" scope="request" /> <jsp:setProperty name="getQuestionDBBean" property="userID" value="wsdemo" /></pre>
Discussion	<p>The example above has been shortened somewhat to set only one parameter.</p> <p>Note that with JSP .91, the <PARAM> tag used within the <BEAN> tag. In JSP 1.0, you must instead use the <jsp:setProperty> tag outside of <jsp:useBean> tag.</p> <p>You must properly link setting the property of an existing bean by the name attribute within <jsp:setProperty> pointing to the bean identified by the id attribute within <jsp:useBean>. Similar considerations for <jsp:getProperty>.</p>

Variation 4: Invoke methods on a bean

Example	FeedbackServletResultPage.jsp
JSP .91	<pre><% try { java.lang.String _p0_1 = feedbackQuery.getWSDemo_Feedback_Name(0); %></pre>

JSP 1.0	No change from JSP .91 to JSP 1.0
Discussion	The NAME attribute in <BEAN> tag and id attribute in <jsp:useBean> tag are equivalent. Both identify a bean named feedbackQuery. For either JSP specification, invoking a method on a bean is identical.

Incorporating IBM extensions to JSP 1.0

See the Related information for references of IBM tags that extend JSP 1.0. The tags might give you additional ideas for replacing JSP .91 tagging functionality. In some cases, IBM extensions provide functionality that was removed in the switch from JSP .91 to JSP 1.0. The remainder of this article focuses on one such tag, <tsx:repeat>.

Replacing <REPEAT> with <tsx:repeat>

<tsx:repeat> provides for repeating information, which is useful in creating HTML tables. <REPEAT> from JSP .91 is usually used with <INSERT> to actually insert data from specified bean. <REPEAT> from JSP .91 does not have an equivalent in the JSP 1.0 specification. However, the IBM extension <tsx:repeat> provides much the same function.

Example	timeout.jsp (available in Advanced Edition only)
JSP .91	<pre><REPEAT INDEX="i"> <%timeoutBean.getBalance(i);%> <TD><INSERT BEAN="timeoutBean" PROPERTY="balance"></INSERT></TD> </REPEAT></pre>
JSP 1.0	<pre><tsx:repeat index="i"> <TD> <%= timeoutBean.getBalance(i) %> </TD> </tsx:repeat></pre>
Discussion	Note that there is an actual call, using Java syntax, of getBalance method of timeoutBean, within loop of <tsx:repeat>, rather than use of IBM JSP 1.0 extension <tsx:getProperty>. This is because getBalance method requires an explicit argument to specify which row of data from underlying array in timeoutBean is to be returned. Thus, <tsx:getProperty> not suitable.

Related information...

- [JSP 1.0 syntax: Tags for variable data](#)
- [JSP 1.0 syntax: Tags for database access](#)
- [3.3.3: Migrating to supported JSP specification](#)

3.3.4: Migrating to supported XML specification



If you developed XML applications using the XML API Version 1.1.x, you can use those applications with the Application Server Version 3.0x with certain restrictions, or you can migrate those applications to the API Version 2.0.x.

Migrating to XML for Java API Version 2.0.x from Version 1.1.x

Most XML applications are partly based on the DOM or SAX APIs. The primary decision factor for migration to the API Version 2.0.x is whether the application must strictly adhere to the DOM or SAX APIs. If strict adherence is a requirement, use the API Version 2.0.x. If the applications employ some of the convenience features of the compatibility APIs, use the API Version 1.1.x.

Also, there are some inherent performance improvements in the API Version 2.0.x APIs, but you can gain additional performance improvements by explicitly using non-validating parsers (instead of the API Version 1.1.x parsers, which force validation) in application environments where the data can be trusted.



You cannot mix Version 2.0.x APIs with Version 1.1.x APIs to create an XML application.

API Version 1.1.x classes deprecated or not supported in API 2.0.x

The Application Server Version 3.5 XML4J.JAR file includes the following API Version 1.1.x packages for TX compatibility:

- com.ibm.xml.parser package
- com.ibm.xml.xpointer package

If you need parser function that is provided in the API Version 1.1.x but not in the API Version 2.0.x, you can use the Version 1.1.x classes for TX compatibility. Use the API Version 1.1.x methods to create a parser, cause the parser to read input, and set options. The DOM tree returned by the TX compatibility classes is an instance of the TX classes from the API Version 1.1.x.

The table summarizes the methods of the API Version 1.1.x class com.ibm.xml.parser.Parser that are not supported or implemented in the API Version 2.0.x.

Method	Status
Parser.addNoRequiredAttributeHandler	Not supported. Throws java.lang.IllegalArgumentException.
Parser.getReaderBufferSize	Not supported. Throws java.lang.IllegalArgumentException.
Parser.setErrorNoByteMark	Not supported. Throws java.lang.IllegalArgumentException.

Parser.setReaderBufferSize	Not supported. Throws <code>java.lang.IllegalArgumentException</code> .
Parser.setProcessExternalDTD	Not implemented. Does not function the same as in the API Version 1.1.x.
Parser.setWarningNoDoctypeDecl	Not implemented. Does not function the same as in the API Version 1.1.x.
Parser.setWarningNoXMLDecl	Not implemented. Does not function the same as in the API Version 1.1.x.
Parser.stop	Not implemented. Does not function the same as in the API Version 1.1.x.

In the API Version 2.0.x, some Version 1.1.x methods are deprecated, as summarized by the table.

Deprecated Method	Recommendation
<code>com.ibm.xml.parser.Parent.addElement(Child)</code>	Use <code>appendChild()</code> .
<code>com.ibm.xml.parser.EntityDecl.getName()</code>	Use <code>getNodeName()</code> .
<code>com.ibm.xml.parser.TXNotation.getName()</code>	Use <code>getNodeName()</code> .
<code>com.ibm.xml.parser.TXElement.getName()</code>	Use <code>getNodeName()</code> or <code>getTagName()</code> .
<code>com.ibm.xml.parser.EntityDecl.getNDATAType()</code>	Use <code>getNotationName()</code> .
<code>com.ibm.xml.parser.Namespace.getUniversalName()</code>	See <code>createExpandedName()</code> .
<code>com.ibm.xml.parser.TXElement.getUniversalName()</code>	Use <code>createExpandedName()</code> .
<code>com.ibm.xml.parser.TXAttribute.getUniversalName()</code>	Use <code>createExpandedName()</code> .
<code>com.ibm.xml.parser.TXElement.isEmpty()</code>	See <code>hasChildNodes()</code> .
<code>com.ibm.xml.parser.EntityDecl.isNDATAType()</code>	This method will be removed in a future release.
<code>com.ibm.xml.parser.TXAttribute.setAttribute(TXAttribute)</code>	Use <code>setAttributeNode()</code> .
<code>com.ibm.xml.parser.TXNotation.setName(String)</code>	This method will be removed in a future release.
<code>com.ibm.xml.parser.DTD.setName(String)</code>	This method will be removed in a future release.
<code>com.ibm.xml.parser.TXText.splice(Element, int, int)</code>	This method will be removed in a future release.

3.3.5: Migrating to supported user profile APIs



Migrating from Version 3.0x

No action is required.

Migrating from Version 2.0x

Starting with IBM WebSphere Application Server Version 3.0x, the User Profile implementation differs significantly from that in Version 2.0. The table summarizes the changes.

Area	Change from Version 2.0x
Profile management functions	<p>The User Profile management functions are separated from the data elements (the elements mapped to the columns in the database schema).</p> <p>The management functions in the <code>com.ibm.websphere.userprofile.UserProfile</code> class are deprecated and disabled. The class is to be used solely for getting and setting data for individual instances of users.</p>
Extending the base implementation	<p>You can now extend the base User Profile implementation to include custom database columns and import legacy databases.</p> <p>See the Related information for instructions.</p>

Related information...

- [4.4.2.2.2: Extending the base User Profile implementation](#)
- [3.3: Migrating APIs and specifications](#)

3.3.6: Migrating to supported session APIs



If your existing applications successfully use session support provided by IBM WebSphere Application Server Version 3.0x, these same applications will be able to use Version 3.5 session support.

IBM WebSphere Application Server Version 3.0x introduced some changes to Version 2.0x session support (see Related information).

Between Version 3.0x and Version 3.5, session support did not change in ways that require application updates.

Related information...

- [3.3.6.1: Migrating from Version 2.0x session support](#)
- [3.3: Migrating APIs and specifications](#)

Migrating from Version 2.0 session support



Note these changes to the implementation of sessions in IBM WebSphere Application Server Version 2.x.

- The public classes in the `com.ibm.servlet.personalization.sessiontracking` package have been deprecated.

Application developers can still compile servlets using the old classes. (Specifically, the `IBMSessionData` class typecast will still work). However, the functions will return null or constant values, and no processing or setting of values will occur.

- Clustering is now handled using a database, and the Version 2.0 concept of session cluster client and server is no longer valid because all nodes within a cluster are now considered equal.
- Extensions for sessions to the Java Servlet API are now encapsulated in the `com.ibm.websphere.servlet.session.IBMSession` interface.
- If URL encoding is configured and `response.encodeURL` or `encodeRedirectURL` is called, the URL will be encoded, even if the browser making the HTTP request processed cookies. This differs from the behavior in previous releases, which checked for the condition and halted URL encoding in such a case.

Related information...

- [3.3.6: Migrating to supported session APIs](#)

3.3.7: Migrating to supported security APIs



No action is required.

Related information...

- [3.3: Migrating APIs and specifications](#)

3.3.8: Migrating to supported database connection APIs (and JDBC)



Migrating servlets from Version 3.0x connection pooling to Version 3.5 connection pooling

Connection pooling (provided through DataSource objects) was introduced in IBM WebSphere Application Server Version 3.0x. Version 3.0x servlets using connection pooling need to be changed slightly and recompiled to run under Version 3.5.

You will need to change one import statement from this:

```
import com.ibm.db2.jdbc.app.stdext.javax.sql.*;
```

to this for Version 3.5:

```
import javax.sql.*;
```

Other JDBC considerations:

- For Version 3.5, an application component that obtains two or more connections to the same database manager, using either the same data source or different data sources, must use data sources with JTA-enabled drivers.

With the Version 3.02 Connection Manager, obtaining two connections to the same resource resulted in only one actual connection, allowing JDBC drivers to be used in this scenario.

Migrating servlets from connection manager to Version 3.5 connection pooling

If existing applications contain servlets (or JSP files) that require database connections, and the servlets are using the "connection manager" from Version 2.0x, it is recommended that you update the servlets to use the newer "connection pooling" model (see the Related information). The shift in models corresponds to a change in supported JDBC specification levels.

Servlets written to use the connection manager should continue to work in the Application Server Version 3.5 environment, provided that the servlets use a subset of the connection manager APIs that are deprecated but still supported. See the Related information for the API subset, which is anticipated to cover most existing servlets.

For most servlets, the migration consists of simple coding changes. Because you should not write new servlets using the connection manager, the details of connection manager coding are not discussed, except as needed in the migration.

Migration involves these activities:

- [Update import statements](#)
- [Modify servlet init\(\) methods](#)
- [Modify how servlets obtain connections](#)

- [Utilize JDBC APIs for data access](#)
- [Modify servlets to perform conn.close\(\)](#)
- [Modify how servlets handle preemption](#)

Considerations for new servlets

The connection manager APIs are deprecated in the Application Server Version 3.5 environment and might not work with releases beyond this one.

Therefore, you should not write new servlets using the connection manager. You should instead write new servlets using the connection pooling model from Version 3.5.

Related information...

- [3.3.8.1: Modifying import statements](#)
- [3.3.8.2: Modifying init\(\)](#)
- [3.3.8.3: Obtaining connections](#)
- [3.3.8.4: Utilizing JDBC data access](#)
- [3.3.8.5: Closing connections](#)
- [3.3.8.6: Modifying preemption handling](#)
- [3.3.8.7: Deprecated connection manager APIs](#)
- [3.3: Migrating APIs and specifications](#)

3.3.8.1: Modifying import statements -- Migrating servlets from connection manager



Make sure you have the necessary import statements. Consider dropping any unnecessary imports.

Connection manager (previous way)

```
import java.sql.*;           // for data server access      (keep)
import com.ibm.servlet.connmgr.*; // connection manager classes (drop)
```

Connection pooling (new way)

```
import java.sql.*;           // for data server access      (retained)
import javax.sql.*;
import com.ibm.ejs.cm.*;      // IBM implementations...    (new)
import com.ibm.ejs.cm.pool.*;
import javax.naming.*;       // to get at naming service (new)
```

Related information...

- [3.3.8: Migrating to support database connection APIs](#)

3.3.8.2: Modifying init() -- Migrating servlets from connection manager



In the servlet `init()` method, one-time initializations establish variables for use by all client requests.

Connection manager (previous way)

```
// create specification for desired connection
IBMConnSpec spec = new IBMJdbcConnSpec("poolname",
                                     true,
                                     "COM.ibm.db2.jdbc.app.DB2Driver",
                                     "jdbc:subprotocol:database",
                                     "userid",
                                     "password");

// establish connection manager access to use its facilities
IBMConnMgr connMgr = IBMConnMgrUtil.getIBMConnMgr();
```

Connection pooling (new way)

The WebSphere administrator provides information on the arguments for the `put()` and `lookup()` methods.

```
// create parameter list to access naming system
Hashtable parms = new Hashtable();
parms.put(Context.INITIAL_CONTEXT_FACTORY,
          "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
// access naming system
Context ctx = new InitialContext(parms);
// get DataSource factory object from naming system
DataSource ds = (DataSource)ctx.lookup("jdbc/sample");
```

Note that the `spec`, `connMgr`, and `ds` variables are actually instance variables. They *are not* local to the `init()` method. The class names are shown to help you more easily identify the variables.

In actual servlets, first declare these variables *outside* of all methods. When setting their values in `init()`, do not precede them with the class names `IBMConnSpec`, `IBMConnMgr`, and `DataSource` as shown.

Related information...

- [3.3.8: Migrating to support database connection APIs](#)

3.3.8.3: Obtaining connections -- Migrating servlets from connection manager



For every client request, the `doGet()` or `doPost()` method will begin by obtaining a connection.

Connection manager (previous way)

The connection manager is explicitly told the connection pool to use, through the `spec` variable.

```
// use spec to get connection manager connection
IBMJdbcConn cmConn = (IBMJdbcConn)connMgr.getIBMConnection(spec);
// use connection manager connection to get data server connection
Connection conn = cmConn.getJdbcConnection();
```

Connection pooling (new way)

Connection pooling uses a *ds* `DataSource` variable. The servlet programmer does not need to be aware of connection pooling.

```
// use DataSource factory to get data server connection
Connection conn = ds.getConnection("userid", "password");
```

Related information...

- [3.3.8: Migrating to support database connection APIs](#)

3.3.8.4: Utilizing JDBC data access -- Migrating servlets from the connection manager



A servlet employing sessions uses a database connection to access the data server for each client request.

The access usually occurs in the `doGet()` or `doPost()` method. Standard JDBC APIs are used for data access. Because no connection manager APIs are used, no code migration is necessary.

However, the JDBC 2.0 Standard Extension APIs offer some new possibilities to consider for data server interaction.

Related information...

- [3.3.8: Migrating to support database connection APIs](#)

3.3.8.5: Closing connections -- Migrating servlets from connection manager



Towards the end of every client request, usually in the `doGet()` or `doPost()` method, connection related resources should be released. There are subtle differences between the connection manager and connection pooling models, the latter being the model to which you should migrate servlets.

Connection manager (previous way)

Release the connection manager connection but do not close the actual data server connection, because the connection manager must manage the actual data server connection.

```
// release connection manager connection
cmConn.releaseIBMConnection();
// do not issue conn.close();
```

Connection pooling (new way)

Invoke the `close()` method on the data server connection. Under Application Server Version 3.x, the connection is not really closed, but is instead returned to the connection pool for reuse.

```
// "close" connection, placing it back in the connection pool
conn.close();
```

Related information...

- [3.3.8: Migrating to support database connection APIs](#)

3.3.8.6: Modifying preemption handling -- Migrating servlets from the connection manager

It is possible in rare situations for a servlet using connection pooling to have its connection taken away. This "preemption" feature is disabled by default. The WebSphere administrator can use an administrative client to enable it.

If preemption is enabled, a problem can arise if a single request makes multiple uses of the connection, separated by extended periods. The connection might be considered an orphan connection owned by a servlet that has failed or otherwise become unresponsive.

The connection pooling facility can take away an orphaned connection, returning it to the connection pool for reuse. Servlets using the connection manager have the same problem. Servlets need to be migrated to use a new way of detecting the problem.

Connection manager (previous way)

Use the `verifyIBMConnection()` method to check whether a connection has been preempted from a servlet. This way, you can code a means of recovery.

Connection pooling (new way)

Instead, check for a `ConnectionPreemptedException` and code a means of recovery. Migrate any servlet coding using the `verifyIBMConnection()` method to instead respond to a `ConnectionPreemptedException`.

Related information...

- [3.3.8: Migrating to support database connection APIs](#)

3.3.8.7: Deprecated connection manager APIs



Some connection manager APIs are intended only for monitoring purposes or internal connection manager use, and do not have any practical use in production servlets. Therefore, such APIs were not migrated to the Application Server Version 3.x environment and are not likely to be found in existing production servlets.

The table lists the connection manager classes and associated methods that continue to be supported. The classes are now deprecated. Because you should not use the connection manager to write new servlets, the details of connection manager coding will not be discussed.

Deprecated connection manager class	Methods
<p>com.ibm.servlet.connmgr.IBMConnMgrUtil</p> <p>The last three of the four methods are intended for IBM WebSphere Studio use only.</p>	<ul style="list-style-type: none"> ● public static IBMConnMgr getIBMConnMgr() ● public static IBMConnPoolSpec getPoolProperties(String poolName) ● public static void addPoolProperties(IBMConnPoolSpec spec) ● public static String urlToPoolName(String url)
<p>com.ibm.servlet.connmgr.IBMConnMgr</p>	<ul style="list-style-type: none"> ● public IBMConnection getIBMConnection(IBMConnSpec connSpec) ● public IBMConnection getIBMConnection(IBMConnSpec connSpec, String ownerClass)
<p>com.ibm.servlet.connmgr.IBMConnection</p>	<ul style="list-style-type: none"> ● public boolean verifyIBMConnection() ● public void removeIBMConnection() ● public void releaseIBMConnection()
<p>com.ibm.servlet.connmgr.IBMJdbcConn</p> <p>This class is derived from the IBMConnection class above and it implements one additional method, as shown.</p>	<ul style="list-style-type: none"> ● public Connection getJdbcConnection()
<p>com.ibm.servlet.connmgr.IBMConnPoolSpec</p> <p>This class and the associated methods are intended for WebSphere Studio use only. Both methods are constructors.</p>	<ul style="list-style-type: none"> ● public IBMConnPoolSpec(String poolName, String poolType, int maxConnections, int minConnections, int connectionTimeOut, int maxAge, int maxIdleTime, int reapTime) ● public IBMConnPoolSpec(String poolName, String poolType)

com.ibm.servlet.connmgr.IBMJdbcConnSpec

The first three methods are constructors.

- public IBMJdbcConnSpec(String poolName, boolean waitRetry, String dbDriver, String url, String loginUser, String loginPasswd)
- public IBMJdbcConnSpec(String poolName)
- public IBMJdbcConnSpec()
- public void verify()

Related information...

- [3.3.8: Migrating to support database connection APIs](#)

3.3.9: Migrating to supported transaction support



Version 3.0x of the product ran with a 1.1 level of JDK. Version 3.0x included packages written by IBM to provide transaction support features usually provided by JDK 1.2. Now that Version 3.5 runs with JDK 1.2, applications should no longer import the proprietary IBM packages, but instead import the open Java 1.2 packages that provide the required functionality.

1. In Java source files, find the import statement:

```
import com.ibm.db2.jdbc.app.jta.javax.transaction.*
```

2. Change the import statement to:

```
import javax.transaction.*
```

3. Recompile the Java files using JDK 1.2.

Other transaction considerations for Version 3.5:

- One database connection cannot be used across multiple user transactions. If an application component obtains a connection to a database, then begins a transaction, the connection is closed automatically when the transaction ends. The connection must be obtained again before beginning another transaction.
- Transactions that began by using UserTransaction now use the isolation level specified when the enterprise bean is deployed.

In Version 3.02, the transaction isolation level defaulted to:

- REPEATABLE_READ for DB2
- SERIALIZABLE for Oracle
- The transaction inactivity timeout units are in milliseconds.

Related information...

- [3.3: Migrating APIs and specifications](#)

3.3.10: Migrating to supported XML configuration



The XMLConfig tool was introduced in Version 3.02 for importing and exporting XML representations of the WebSphere administrative domain. Some of the interfaces have changed in Version 3.5.

The changes in XML descriptions of the elements are as follows.

Element	Changed syntax
container	<container name= <i>container name</i> ></container>
jdbc-driver	Now supports <install-info/> and <uninstall-info/> elements (optional)
model	Additional clone support information (optional)
olt-controller	No OLT Controller resource in Version 3.5
servlet	Additional clone support information (optional)
servlet-engine	Additional clone support information (optional)
session-manager	<session-manager name= <i>session manager name</i> ></session-manager>
userprofile-manager	<userprofile-manager name= <i>user profile manager name</i> ></userprofile-manager>
web-application	Additional clone support information (optional)

As for programmatic access:

- The XMLConfig constructor now throws NamingException and InvalidArgumentException.
- XMLConfig now supports variable substitution and variable Hashtable setter.

The XMLConfig command line now supports the substitute option for variable replacement.

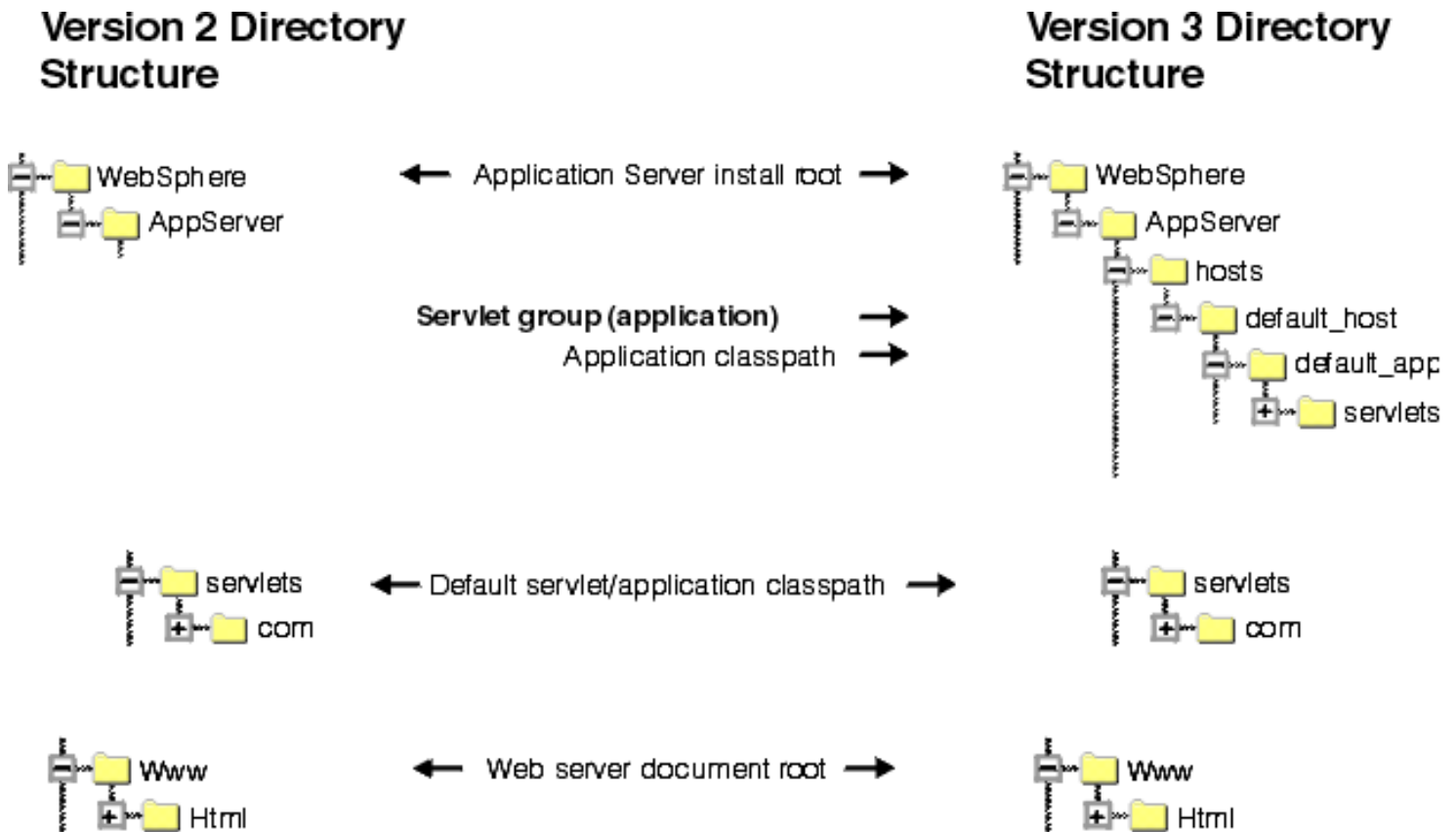
Related information...

- [3.3: Migrating APIs and specifications](#)

3.4.2: Migrating Web application files from Version 2.0x directories

In addition to migrating Web applications to use supported APIs and specifications (see Related information below), you need to move the actual application files from the directories in the previous version to the Version 3.5 directory structure.

The figure illustrates the Application Server Version 2 and Version 3 (including Version 3.5) directory structures for the default Web application (`default_app`).



For the Application Server Version 2.0x:

- Servlet and other application components were placed in the following paths:
 - `<AS_install_root>\servlets`
 - `<AS_install_root>\classes`
 - A user-defined reloadable classpath
- The static HTML and JSP files were placed in the Web server HTML document root

Optimal migration or quick migration

The above placement is not the optimal placement for Version 3.5. However, a `default_app` is provided with Version 3.5 in case you want to quickly migrate using a placement similar to that of Version 2.0x.

For instructions, select one of two Version 2.0x to Version 3.5 file migration paths in the Related information below. Again, the optimal path is the recommended one because withdrawal of support for the suboptimal path is imminent.

Related information...

- [3.4.2.1: Optimally migrating Version 2.0x Web application files](#)
- [3.4.2.2: Quickly migrating Version 2.0x Web application files](#)
- [6.1.5.3: The default application](#)
- [3.4: Migrating from Version 2.0x](#)

3.5.1: Optimally migrating Version 2.0x Web application files



To reorganize your Version 2 applications to take full advantage of the Version 3 Web application programming model, you should ultimately use the following migration method.

Task	Instructions
Use a WebSphere Application Server administrative client to configure a Web application to include the servlets and Java components from the Version 2.0x Web application.	6.6.8: Administering Web applications
Move the servlets and Java components to the classpath of the new Web application that you configured.	6.3: Placing files and setting classpaths
Keep any servlets and Java components that were in the JVM classpath (application server classpath) in that location. These are servlets that you did not want to have reloaded on your Version 2 installation. The same restriction applies to Version 3.	6.3: Placing files and setting classpaths
Move the static HTML and related resources to the new application document root directory.	6.3.2: Placing files
To enable serving of the HTML files, use one of the following methods: <ul style="list-style-type: none">● Add the SimpleFileServlet to the application.● To enable serving of the JSP files, add the JSP compiler to the application.● If needed, add other, optional internal servlets to the application.● Add a pass rule for the document root to the Web server's configuration.	4.2.1.2.3: Using the WebSphere servlets for a head start 6.6.8.1.1: Configuring new Web applications For the last method, consult the Web server documentation.

Related information...

- [3.4.2: Migration tips for Version 2.0x users](#)

3.5.2: Quickly migrating Version 2.0x Web applications



For the fastest deployment, maintain the Version 2 file organization on your Version 3.5 installation:

1. Move the servlets and other Java components that were in the Version 2 Application Server \servlets directory to the Version 3 *AS_install_root*\servlets directory. If your servlets use package names, create subfolders under the \servlets directory to mirror the package name.
2. Move the servlets and Java components that were in another Version 2 reloadable classpath to the Version 3 *AS_install_root*\servlets directory, or add the Version 2 reloadable classpath to the Version 3 *default_app* application classpath.
3. Keep any servlets and Java components that were in the system classpath in that location. These are servlets that you do not want to have reloaded on your Version 2 installation. The same restriction applies to Version 3.
4. If your servlets or applications use .servlet configuration files, move those files to the same directory as the servlet.
5. Move the static HTML and related resources to the Web server document root.
6. Invoke the servlets and applications in the same manner you used with the Application Server Version 2.

Related information...

- [6.1.5.3: The default application](#)
- [3.4.2: Migrating application files from Version 2.0x directories](#)