IBM

Programming J2EE APIs with WebSphere Advanced

Feel the power of the J2EE-compliant environment

Learn J2EE APIs by practical application development

Deploy J2EE applications to WebSphere

Osamu Takagiwa
Adrian Spender
Anthony Stevens
Julien Bouyssou

Redbooks

ibm.com/redbooks

IBM

International Technical Support Organization

Programming J2EE APIs with WebSphere Advanced

August 2001

First Edition (August 2001)

This edition applies to VisualAge for Java Enterprise Version 3.5.3, and WebSphere Application Server Advanced Edition Version 3.5.3 for use with Windows NT/2000.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. QXXE Building 80-E2
650 Harry Road
San Jose, California 95120-6099

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Figures

Tables

Preface

This IBM Redbook provides you with sufficient information to effectively use the IBM WebSphere and VisualAge for Java environments to create, manage and deploy Web-based applications using the Java 2 Enterprise Edition API specification.

Part 1 describes the overview of J2EE and the products used in our environment. Following this, we give an overview of the PiggyBank application scenario which is an integrated application used to illustrate various principles and techniques for enterprise software development. Many of the code samples provided throughout this redbook are taken from the PiggyBank application.

Part 2 describes the EJB container of the J2EE specification, which includes transactional EJBs, transactions, messaging with JMS, WebSphere and MQSeries. Chapters 5 and 6 examine the programming model for EJBs and explore the use of transactions. Chapter 7 covers JMS.

Part 3 describes the Web container of J2EE specification. It starts with the basic concepts of the Java Servlet technology and then shows the difference between Java Servlet specifications 2.1 and 2.2 and how to develop servlets and integrate them into an enterprise level application. Following this, we cover the JavaServer Pages, especially the custom tag libraries which were introduced in JSP1.1.

This book is intended to be read by anyone who requires both introductory and detailed information on software development in a WebSphere environment using J2EE APIs. We assume that you have a good understanding of Enterprise Java Development.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, San Jose Center.

Osamu Takagiwa is an Advisory I/T Specialist at the International Technical Support Organization, San Jose Center. He writes extensively and teaches IBM classes worldwide in all areas of application development. Before joining the ITSO at the beginning of 2001, Osamu worked for IBM in Japan as an I/T Specialist.

Adrian Spender is an I/T Specialist at the IBM Hursley development laboratory in the United Kingdom. He has five years of experience in Java and Web application development and currently works as a WebSphere consultant. He holds a degree in Information Systems from the University of Leeds in the United Kingdom. His areas of expertise include JSP programming and WebSphere performance. He has taught many WebSphere application development related classes throughout Europe.

Anthony Stevens is an I/T Specialist for Global Services Federal Systems at IBM in Austin, Texas. He has 15 years of experience in object-oriented analysis and design and holds a master's degree in Computer Science. He has worked at IBM for 12 years and has three years of experience in Java and Web application development. His areas of expertise include XML and EJB development.

Julien Bouyssou is an I/T Professional working for IBM Global Services in Bordeaux, France. He has one year of experience in e-business application development. He holds a degree in Information Technology from the University of Bordeaux. His areas of expertise include UML and Java development.

Special notice

This publication is intended to help WebSphere and VisualAge for Java developers build Web server applications using J2EE APIs. The information in this publication is not intended as the specification of any programming interfaces that are provided by WebSphere Application Server, WebSphere Studio, and VisualAge for Java Enterprise. See the PUBLICATIONS section of the IBM Programming Announcement for WebSphere Application Server, WebSphere Studio, and VisualAge for Java Enterprise for more information about what publications are considered to be product documentation.

IBM trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

e (logo)®	Redbooks
IBM ®	Redbooks Logo
AIX	MQSeries
AS/400	OS/390
400	S/390
AT	SP
CT	SP1
Current	SupportPac
DB2	VisualAge
Domino	WebSphere
Lotus	XT

Comments welcome

Your comments are important to us!

We want our IBM Redbooks to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

► Use the online **Contact us** review redbook form found at:

    ibm.com/redbooks

► Send your comments in an Internet note to:

    redbook@us.ibm.com

► Mail your comments to the address on page ii.

Part 1

Introduction

Chapter 1 shows our development conditions concerning tools and applications.

Chapter 2 begins with an overview of J2EE.

Chapter 3 discusses the suite of software products and tools that we used in writing this book.

Chapter 4 introduces the PiggyBank integrated application used to illustrate various principles and techniques for enterprise software development. Many of the code samples provided throughout this book are taken from the PiggyBank application.

Our development conditions

This chapter describes the various tools and applications used.

1.1 Development and deployment tools

Here are the various tools we used in our environment.

- ► Development tools
  - – VisualAge for Java 4.0
- ► Deployment tools
  - – Application Assembly Tool
    - • Creates and maintains EAR, WAR, JAR files
    - • Performs deployment tasks
  - – EJB Deploy
    - • Generates deployment code for EJBs
  - – SEApp Install
    - • Installs applications packaged in an EAR file
  - – launchClient
    - • Used to start an EJB client application

► Troubleshooting and debug tools

   – Traces, logs, logAnalyzer

   – Object level trace, Object level debugger

> **Note:** This redbook does not delve into the details for development and deployment tools due to the upcoming changes with WebSphere 4.0 tooling.

1.1.1 VisualAge for Java 4.0

Here are our conditions for VisualAge for Java 4.0. Upcoming changes in VisualAge for Java 4.0 include:

► Enhanced EJB export smartguide supports WebSphere 4.0

► Code to EJB 1.0, deploy to EJB 1.0 or EJB 1.1

► Enhanced EAB smartguide generates JCX 1.0 compliant code

► WebSphere Test Environment supports:

   – Servlet 2.2
   – JSP 1.1
   – EJB 1.0

► Application development environment using JDK 1.2.2

   – Deploy to JDK 1.2.2 or JDK 1.3

> **Note:** VisualAge for Java 3.x was missing EJB 1.1 support, although the VAJ class editor could be used to add EJB 1.1 method calls.

1.2 WebSphere and J2EE

Figure 1-1 depicts the changes in servlet, JSP and EJB support between WebSphere 3.x and WebSphere 4.0.
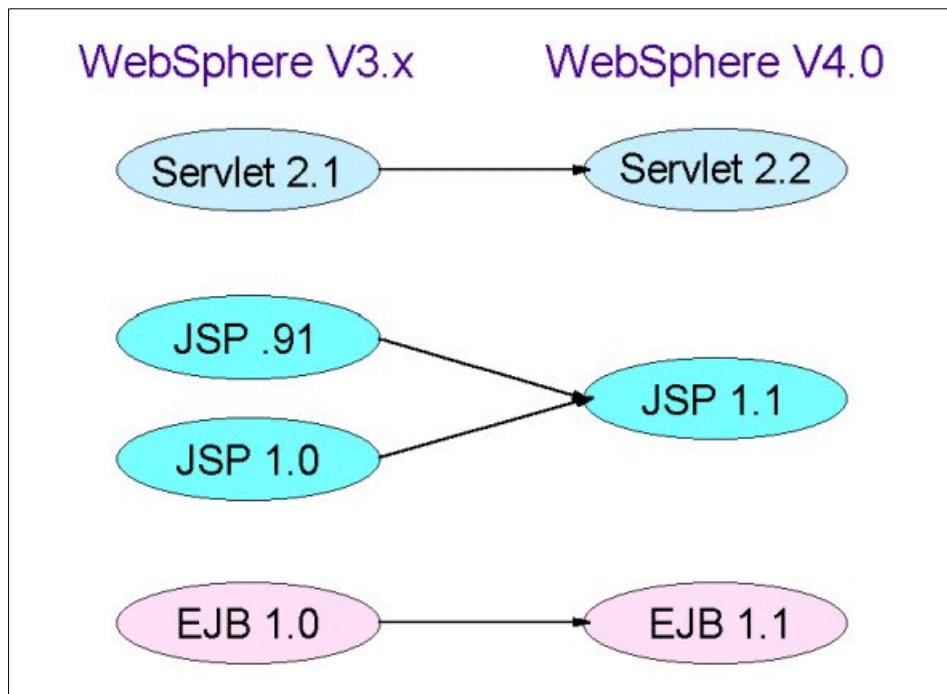
Figure 1-1   WebSphere 3.x to 4.0

WebSphere is J2EE 1.2 compliant:

► The components are:

    – EJB 1.1
    – JSP 1.1
    – Servlet 2.2

► The services are:

    – JNDI 1.2
    – JDBC 2.0

► The communications are:

    – JMS 1.0
    – JavaMail 1.1

2

J2EE overview

This chapter introduces the concept of J2EE, the standard for developing multi-tier enterprise applications. We describe the different parts of the J2EE runtime environment: the containers, the standard services, the application components and the resource manager drivers. Also, we deal with the concept of deployment of a J2EE application with the introduction of deployment descriptors.

2.1 What is J2EE?

J2EE stands for Java 2 Platform, Enterprise Edition. The Java 2 Platform, Enterprise Edition defines a simple standard that applies to all aspects of architecting and developing multi-tier server based applications.

J2EE defines a standard architecture composed of an application model, a platform for hosting applications, a Compatibility Test Suite (CTS) and a reference implementation.

The primary concern of J2EE is the platform specification: it describes the runtime environment for a J2EE application. This environment includes application components, containers, resource manager drivers, and databases. The elements of this environment communicate with a set of standard services that are also specified. Figure 2-1shows the J2EE server model and in which tiers the J2EE components reside.
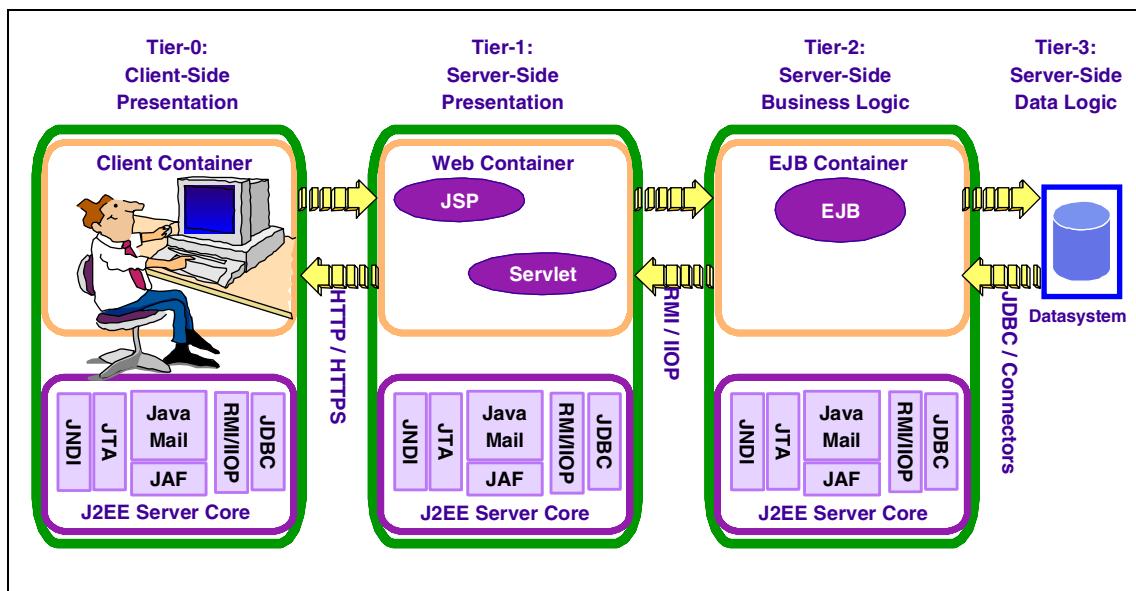
Figure 2-1   J2EE server model

2.2 Application components

A J2EE product must support four types of application components. Each type of component must execute in a container (see 2.4, "J2EE containers" on page 14).

Application components can be packaged in JAR files and include a deployment descriptor, giving platform independent information to the container that manages them (see Part 2.6, "Deployment" on page 17).

Figure 2-2 shows an overview of the application components in their respective containers along with the logic they usually handle in an enterprise application.
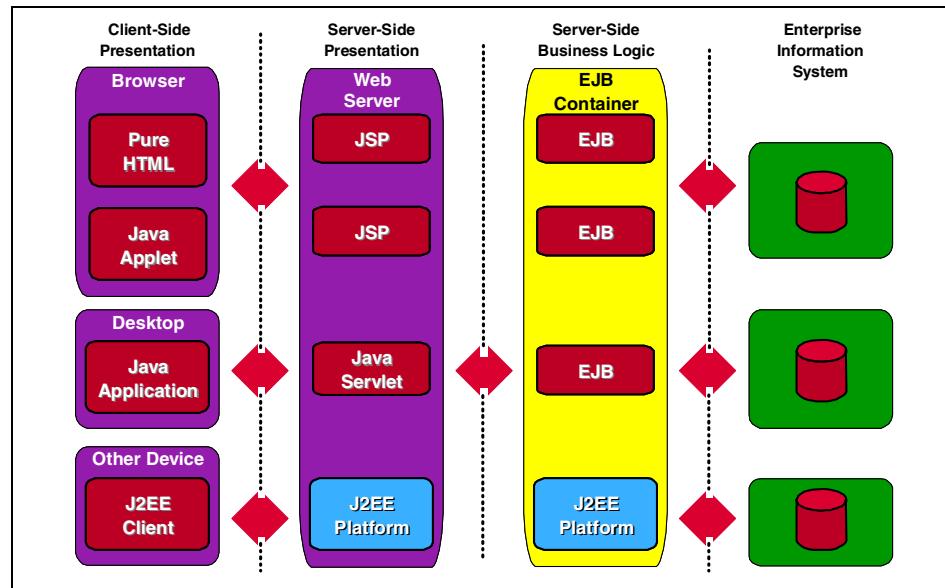
Figure 2-2   J2EE application component overview

2.2.1 Application clients

Application clients are Java programs, such as a Graphical User Interface (GUI) that executes on a desktop computer. They have access to all the facilities of J2EE as they run in a container. Furthermore, they run in their own Java Virtual Machine.

An application client is packaged in a JAR file and may be used to run the presentation logic of an application. Stand-alone applications or programs that do not fit in any of the other categories can be placed in this one.

Chapter 11, "Application clients and J2EE communications" on page 299 deals with the application clients.

2.2.2 Applets

An applet is a component that typically executes in a Web browser. It is a Java class that can also run in a variety of other applications or devices.

An applet must be loaded, initialized and run by a Web browser; it can be used to process presentation logic, and it provides a powerful user interface for J2EE applications. However, simple HTML pages may also be used.

Applets embedded in an HTML page are deployed and managed on a J2EE server although they run into a client machine. They are considered as belonging to the HTML page and an HTML page is managed by a J2EE server.

2.2.3 Servlets and JavaServer Pages

Servlets and JavaServer Pages are components used to handle HTTP requests from Web clients. They generally process the presentation logic of an application. Servlets can also act as a controller of an application and provide several services such as session management.

A Servlet is a Java program extending the capacities of a Web server. A Servlet interacts with a user through a request response paradigm and, basically, generates dynamic content.

JavaServer Pages is a technology used to return dynamic content to a client (in its Web browser, for example). It uses HTML or XML elements, server-side Java objects, custom and scripting elements to process requests.

These components may be used to generate formatted data (XML, for example) for use by other application components. Servlets and JavaServer Pages are deployed, managed and executed on a J2EE server and are generally called "Web components".

2.2.4 Enterprise JavaBeans

Enterprise JavaBeans components are deployed, managed and executed on a J2EE server. Their environment must support transactions. Enterprise beans usually contain the business logic for a J2EE application; they interact with the databases.

The Enterprise Java Bean architecture described in the EJB specification defines the contract enabling Java code written by multiple vendors to interoperate in a distributed programming environment. The reusable code components are referred to as Enterprise Java Beans. EJBs add the scalability and reliability aspects inherent to industrial strength server applications.

EJBs incorporate several Java technologies to support scalable, secure client-server applications:

► JDBC — database access
► JNDI — Java Naming and Directory Interface
► JTS — Java Transaction Service
► JSP — JavaServer pages
► Servlet
► RMI — Remote Method Invocation
► JMS — Java Message Service

EJBs are either entity beans (javax.ejb.EntityBean) or session beans. Additional functionality is achieved through the use of CMP (container managed persistence) and BMP (bean managed persistence) for entity beans. Similarly, session beans can be either stateless or stateful. Since container-managed persistence obviously transfers a great deal of the burden for transaction management and other enterprise concerns to the application server, CMP is considered to be a heavyweight implementation, and careful consideration should be given to when to make a Java Bean container-managed versus bean-managed. In this regard, experience with using EJBs has provided "best practice" design considerations which are useful in avoiding common errors affecting both performance and code maintenance. In the same vein, "best of best practices" are starting to surface for architecture, design and implementation of EJBs.

2.3 Standard services

J2EE components may use a set of standard services to interact with each other. Here is a list of these services with a short explanation on the functionality they provide to the components.

2.3.1 HTTP and HTTPS

The HTTP protocol is the protocol used across the Internet to get objects (pages, images, and so on) from remote hosts. HTTP messages are client requests and responses from the server.

A J2EE platform must implement this protocol: the client-side API is defined by the java.net package which provides the classes for implementing networking applications. The server-side API is defined by the servlet and JSP interfaces.

For HTTP layered over the Secure Socket Layer protocol (HTTPS), the same client and server APIs as HTTP are required to support this protocol.

2.3.2 Java Naming and Directory Interface (JNDI)

This API allows the J2EE components to look up other remote objects that they may need to access. The JNDI API was designed to standardize access to a variety of naming and directory interfaces and has two parts:

- ► An application-level interface: these APIs are used by the application components to access naming and directory services.

- ► A service provider interface: this part of the API is used to attach a provider of a naming and directory service to the J2EE platform.

In EJB1.0, JNDI lookup was similar to:

```
initialContext.lookup("com/ibm/piggyapp");
```

In EJB1.1, JNDI the same lookup becomes:

```
initialContext.lookup("java:comp/env/ejb/com/ibm/piggyapp");
```

2.3.3 Java DataBase Connectivity (JDBC)

JDBC is the standard API that enables connectivity with database systems. An application's components can use JDBC to handle data from relational databases and other repositories.

The JDBC API can also be divided into two parts, an application-level interface used by the components for accessing databases and a service-provider interface to attach a provider to the platform.

JDBC 2.0 has been supported since WAS 3.5.

2.3.4 Java Message Service (JMS)

This API defines a standard mechanism for the components to send and receive messages. The API is required to be available on a J2EE platform but not implemented. It enables the use of enterprise messaging systems (IBM MQSeries, for example).

Figure 2-3 depicts the types of messaging possible with JMS.
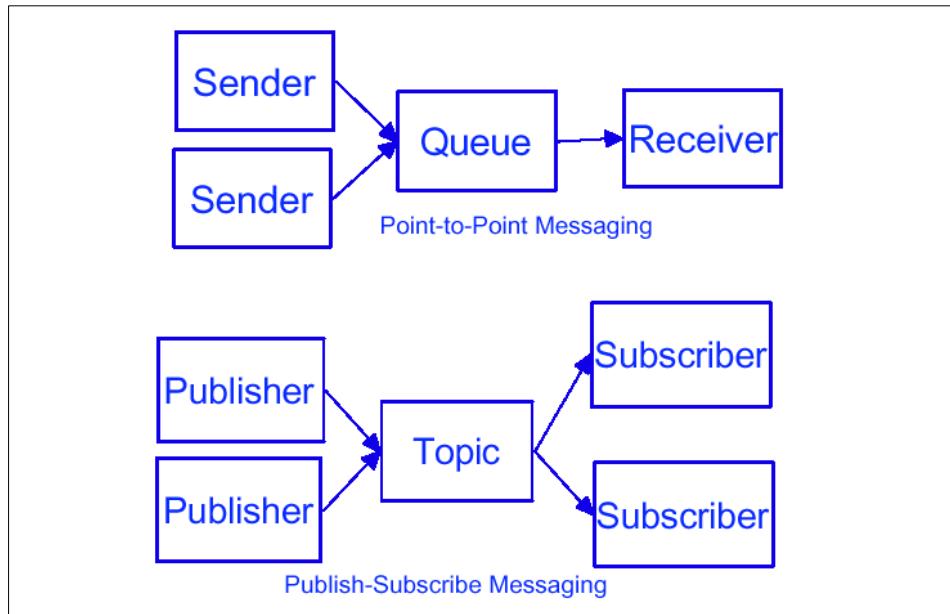
Figure 2-3   JMS allows for asynchronous messaging

2.3.5 JavaMail and JavaBeans Activation Framework (JAF)

The JavaMail API allows an application to send e-mail notifications. It is divided into two parts: an application-level interface used by the application to send mail and a service provider interface to attach a provider to the platform.

JavaMail includes the JavaBeans Activation Framework API (JAF).

2.3.6 Java Transaction API (JTA and JTS)

The JTA API provides a way for J2EE components and clients to manage their own transactions. It also allows multiple components to participate in a single transaction. JTA is the API, and JTS is the implementation.

The application-level interface of the API is used by the container and the application components to demarcate transaction boundaries. The second part of the API defines an interface between the transaction manager and the resource manager used.

JTA does not support nested transactions. JTA does support two phase commit.

2.3.7 Remote Method Invocation — Internet Inter-ORB protocol

This API supports the use of the RMI APIs with the IIOP protocol.

> **Important:** RMI and IIOP separately.
>
> The Remote method Invocation (RMI) is a technology that allows an object running in one Java Virtual Machine to access another object running in a different Java Virtual Machine.
>
> The Internet Inter-ORB (Object Request Broker) Protocol (IIOP) is a protocol used for communication between CORBA object request brokers. An object request broker is a library that enables CORBA objects to locate and to communicate with one another.

RMI-IIOP allows J2EE applications to access CORBA services defined by components living outside the J2EE product. This API will also be used to access Enterprise JavaBeans Components in an application.

2.3.8 Java IDL

IDL stands for Interface Definition Language. This API allows J2EE application components to invoke external CORBA objects using the IIOP protocol. These CORBA objects are completely independent of the J2EE application (they may not be Java objects) and may be used by other applications.

2.3.9 XML deployment descriptors

J2EE defines a set of descriptors in XML consisting of DTDs (Document Type Definition) allowing an easier implementation of customizable components.

2.4 J2EE containers

The J2EE Specification defines a container as being responsible for providing the runtime support of the application components (see 2.2, "Application components" on page 8).

There must be one container for each application client type in a J2EE application. The reason for having a container between the J2EE application components and the set of services is to provide a federated view of the APIs for the application components.

A container provides the APIs to the application components in order to access the services and may handle security, resource pooling or state management. The container also has to deal with naming and transaction issues.

Figure 2-4 shows the containers, the services they use and their components.
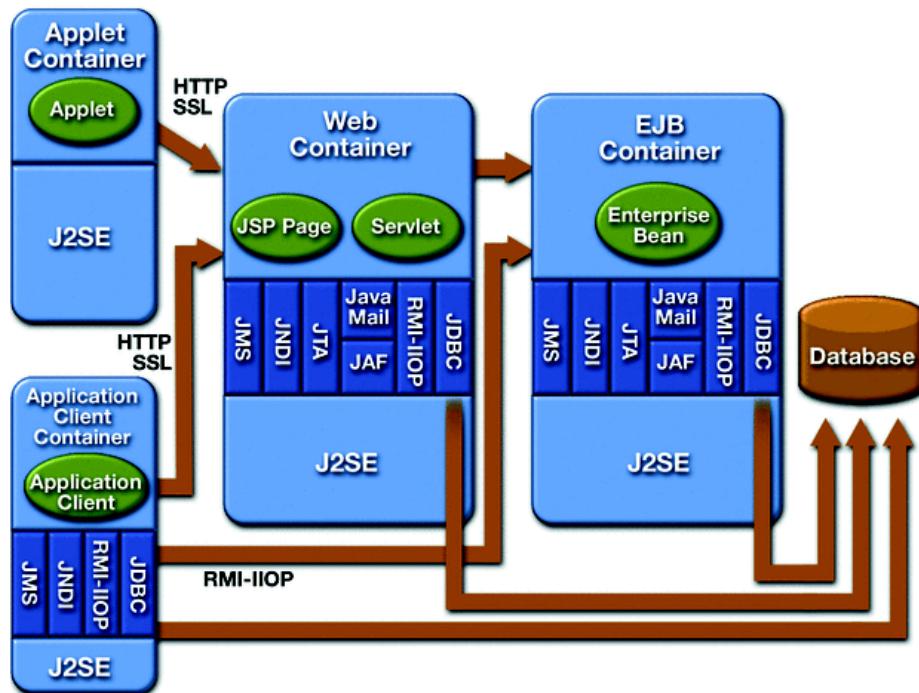
Figure 2-4   J2EE containers, standard services and components

These are the various containers and the services they perform:

► **Application client container**: This container supports application client components. It must provide access to the set of services required by J2EE but is not required to manage transaction. Application clients have access to the Java API and are packaged in a JAR file.

This container is the central topic of Chapter 11, "Application clients and J2EE communications" on page 299.

► **Applet container**: The applet container includes support for the applet programming model. Typically a J2SE (Java 2 Platform, Standard Edition) 1.2 compatible applet execution environment acts as a container. The Java plug-in may be added to the browser to obtain such a container. Applets

communicate over HTTP if they run in a browser, but they can also communicate using serialized objects.

► **Web container**: The Web container is used to run servlets and JSPs (referred as "web components"). It is responsible for providing a runtime environment with a set of services.

The Web container is the topic of Part 3, "The Web container" on page 163 of this book.

Important: A Web container is not a Web server.

A Web container provides runtime environment for Web components (a Web component is a Servlet or a JSP) and services for these components: security, concurrency, life cycle management, transaction, deployment, etc.

A Web server is software that provides services to access the Internet; it hosts Web sites. It also performs certain functions for the Web container (such as message handling) and hosting.

► **EJB container**: The EJB container on the application server is where all deployed EJBs reside. The EJB specification defines a container contract for remote access to the bean, as well as specific container interaction, which may be implemented differently by various system vendors. In addition to the container contract, the EJB specification also defines EJB composition and deployment requirements. Each of these functional characteristics is supported by one or more APIs in J2EE.

Part 2, "The EJB container" on page 37 deals with this subject.

2.5 Resource managers and database

The last elements of a J2EE runtime environment are the resource manager drivers and the database.

► A resource manager provides access to a set of shared resources. From the J2EE point of view a driver will implement network connectivity to an external resource manager.

This driver can extend the functionality of the J2EE platform by implementing one of the J2EE standard service APIs or by defining and implementing a resource manager driver for a connector to an external application system (MQ Series for example).

► The provision for connecting to a database is included in the J2EE architecture. This database is accessible through the JDBC API. Web

components, enterprise beans and application clients may access the database (applets do not need to).

2.6 Deployment

The J2EE Specification gives instructions for packaging and deploying components to their runtime environment. It introduces the concept of deployment descriptors and specifies the content that can be grouped together.

2.6.1 Deployment and deployment descriptors

Application components can be packaged so that they can be deployed in many environments. Application components can be packaged independently; whereas, elements of an application client are packaged in a single JAR file. Web components may be packaged in a WAR (Web Archive) file with other Web resources. Enterprise Beans can be packaged together as well. These packages will be used for deployment in the container dedicated to each application component.

Application components can also be packaged together to construct a J2EE application, that is to say, an application containing Enterprise Beans, Web components and eventually, an application client. The intent of the application is to provide for the possibility of deployment to several operating environments.

The deployment task generally consists of three steps: installation of the components on the server (or container), followed by configuration and execution of the application. Deployments descriptors assist in this deployment task.

A deployment descriptor is an XML file describing how to assemble and deploy a unit into a specific environment. It is provided with the application and specifies the required external resources, as well as the security requirements and environment parameters. The purpose of a deployment descriptor is to enable developers to create reusable components. These components can be customized to run in several environments.

2.6.2 The J2EE packages and their deployment descriptor

Figure 2-5 shows the different packages that can be built for deployment; each of these contains its deployment descriptor.

JAR files are the standard way of packaging modules and applications but other formats may be used.
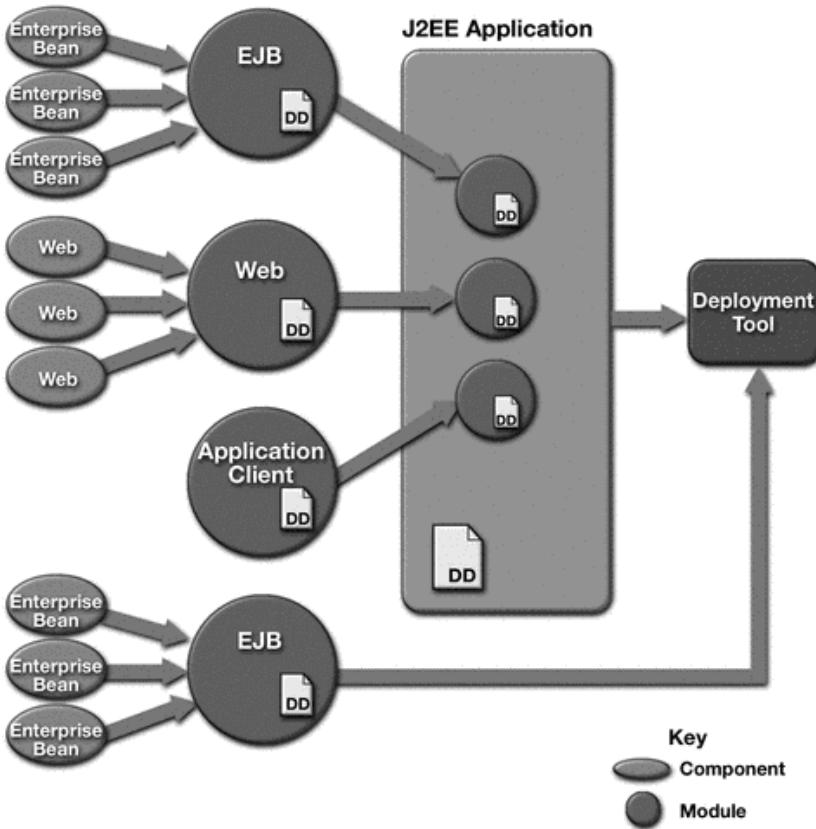
Figure 2-5   J2EE deployment descriptors overview

There are four deployment modules:

► **EJB**: EJBs are packaged in a J2EE module (a JAR file, for example). The EJB specification defines the DTD (Document Type Definition) of the deployment descriptor associated with this module. Part 5.7, "XML deployment descriptor" on page 102 deals with the deployment descriptor for EJBs.

► **Web application**: A Web application also represents a J2EE module, containing servlets, JSPs and other resources. The Java servlet specification defines the DTD of the deployment descriptor for a Web application (see "The deployment descriptor" on page 173).

► **Application client**: An application client represents a J2EE module. It is packaged in a JAR file and contains a deployment descriptor. Chapter 11, "Application clients and J2EE communications" on page 299, deals with application client issues such as deployment.

► **J2EE application**: A J2EE application is a set of J2EE modules with a J2EE deployment descriptor. It is packaged using the JAR file format into an Enterprise ARchive (EAR) file. This file has the .ear extension, it contains a deployment descriptor and one or more J2EE modules, each of these contains its own deployment descriptor. Chapter 12, "Deploying J2EE applications to WebSphere" on page 313, will address the deployment of a J2EE application.

3

Products used within this book

This chapter provides an overview of the various IBM software products we use in this redbook. All of the products belong to the IBM WebSphere software platform.

In addition, we discuss how these products combine to support the APIs and standards mandated by J2EE.

21

3.1 WebSphere software platform overview

The WebSphere software platform comprises a series of products that deliver a full set of J2EE and Web Services APIs and related technologies. In addition, the platform supplies the tools to develop, test, deploy and scale solutions built upon J2EE. The software platform is split into segments as shown in Figure 3-1.

Customer and Partner Applications

WS Commerce Suite   Lotus Collaboration

MQSeries Workflow   **Application accelerators**   B2B Integrator

Development
WS Homepage Builder
WS Studio
VisualAge for Java
Versata Studio
WS Business Components
VisualAge Generator

Presentation
WS Transcoding Publisher
WS Everyplace Suite
WS Voice Server
WS Personalization
WS Portal Server
Web Content Management

Deployment
WS Edge Server
WS Site Analyzer
Tivoli Policy Director

Integration
MQSeries Integrator
MQ Adapter Offering
Host Integration

Foundation Extensions

WS Application Server, MQSeries
Foundation

Figure 3-1   The WebSphere software platform

Each segment is introduced below:

- Foundation — Contains the products that provide the deployment environment for Web-enabling your business as quickly as possible. WebSphere Application Server provides a leading J2EE certified and Web Services enabled environment for running your Web applications. MQSeries provides industry leading assured messaging over more than 35 platforms.

- Foundation extensions — The extensions are the set of tools and products which provide the capability to expand the foundation products in four areas:

  - Development: Providing the tools to create, update and maintain your Web application from end-to-end. Web page editors can utilize markup creation tools such as WebSphere Homepage Builder and the WebSphere Studio PageDesigner. WebSphere Studio provides a team environment for the organization and management of your Web application, while VisualAge

for Java provides developers with an award-winning Java development environment.

– Presentation: Providing the tools to extend the reach of your Web applications to multiple channels such as voice, mobile and Internet appliances. WebSphere Personalization allows you to deliver pages tailored to the interests and needs of the individual user, while WebSphere Portal Server allows you to build customized points of entry to your enterprise applications.

– Deployment: WebSphere Edge Server provides the capability to make your Web site more scalable and resilient by moving content closer to the user and intelligently load-balancing your servers. WebSphere Site Analyzer helps your Web master to monitor and perform trends analysis on your site. Tivoli PolicyDirector allows you to provide a robust common security policy across your sites.

– Integration: MQSeries Integrator (MQSI) provides a message brokering layer to your applications allowing seamless message transformation and integration of disparate applications into a unified business process.

► Application accelerators — Offers scalable solution-oriented products based upon the WebSphere foundation, from WebSphere Commerce Suite for the user to online buying scenario, to WebSphere Business-to-Business Integrator for spanning the gap between enterprise systems.

► Customer and partner applications — Utilizes the open software strategy of the WebSphere software platform to provide tailored solutions for specific industries and scenarios.

We now describe the particular products used in this redbook in more detail.

3.2 Foundation products

The two foundation products: WebSphere Application Server and MQSeries provide the underlying implementation of the APIs and technologies used to build J2EE applications. WebSphere Application Server provides servlet, JSP and EJB implementations, as well as standard J2EE services, while MQSeries is the resource manager providing the underlying messaging system for JMS.

3.2.1 WebSphere Application Server

WebSphere Application Server (WAS) is the IBM base deployment platform for e-business applications. Based on the J2EE platform, WAS provides a scalable, robust environment for your Java based Web applications. As an application server, WAS sits behind a Web server and handles requests for dynamic data

generation from a browser by invoking the appropriate Java servlet or JSP. WAS also allows Java-based clients such as Java applications and applets to directly invoke Java objects. Figure 3-2 shows how WAS supports the J2EE container types:
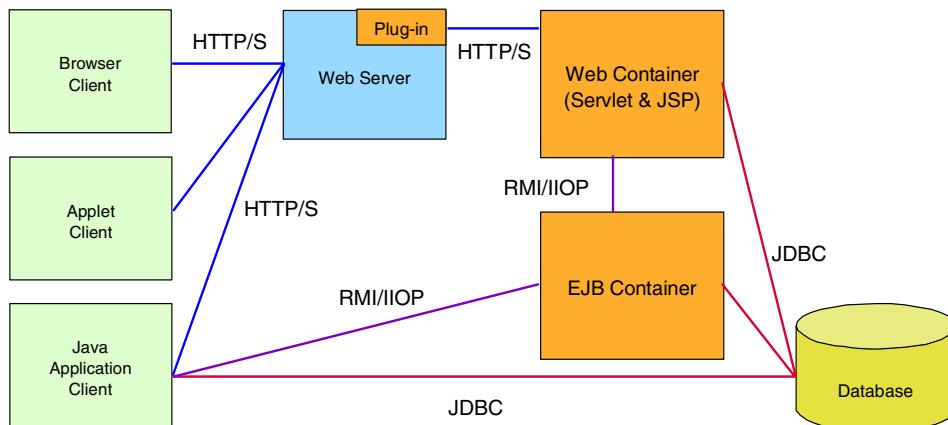
Figure 3-2   WebSphere architecture

Within this diagram, the Web server plug-in, Web container and EJB container are provided by WAS. The plug-in works with the Web server to redirect certain URLs requests from application clients to servlets or JSPs within the Web container. The Web container components can then call Enterprise JavaBeans within the EJB container and may make calls directly to a database or databases. It is not shown in the diagram, but messages may also be placed on or read from queues by components from either the Web or EJB container. In addition to requests via the Web server, Java application clients running in a client container can make direct calls to EJB components, the database, and indeed a message queue.

WebSphere Application Server versions

WebSphere Application Server ships in Standard, Advanced and Enterprise editions, each with differing levels of functionality:

Note: The functionality described below is true as of WebSphere Application Server 3.5 Fixpack 3. Fixpacks for WebSphere can be downloaded from `http://www.ibm.com/software/webservers/appserv/support.html`

► WebSphere Application Server Standard Edition provides a single node implementation of the Servlet 2.2 and JSP 1.1 APIs in an easy to install and configure environment. Standard Edition provides JDBC 2.0 extensions for

database connection pooling, and support for XML and XSL. It ships with the IBM HTTP Server 1.3.12 based on the Apache Web Server, and InstantDB, a Java based database.

► Advanced Edition adds an EJB 1.0 compliant EJB container for both session and entity beans (container-managed and bean-managed.) Database support is extended for distributed transactional support for multiple databases, namely DB/2, Oracle, Sybase and Microsoft SQL Server, and transactional JMS/JTA support with MQSeries. Advanced Edition also supports multi-node distribution of Web applications via cloning and Web container/EJB workload management. Finally, Advanced Edition supports a rich security environment including LDAP directory authentication and trust association.

► Enterprise Edition is a fully distributed-object, CORBA based enterprise environment combining TxSeries, IBMs world-class transactional application environment, with the object distribution capabilities of Component Broker.

Standard Edition is available on Windows NT, Windows 2000, AIX, Sun Solaris, HP/UX, Linux, iSeries (AS/400) and Novell Netware. Advanced Edition is also available for Linux/390. For the mainframe, WebSphere Application Server V4.0 for z/OS and OS/390 is the first IBM J2EE compliant application server platform.

> **Note:** As this book was being written, WebSphere Advanced 4.0 was in beta and may well be released as you read this. Where appropriate within our discussions we make reference to functionality contained in WebSphere Advanced 4.0. However, all the functionality and APIs we describe (with the exception of XML deployment descriptors for EJBs) is implemented in WebSphere Advanced 3.5.3.

3.3 Product/J2EE API matching

Table 3-1 shows the J2EE 1.2 APIs and the support within WebSphere Application Server Advanced Edition 3.5.3 and 4.0.

Table 3-1   3.5.3 versus 4.0

J2EE technology or API	WAS 3.5.3	WAS 4.0
J2EE 1.2 Certification	Not fully certified	**Fully J2EE certified**
EJB 1.1	1.0 plus extensions	**1.1 full support including XML deployment descriptors**
HTTP/HTTPS	Yes	Yes
JavaIDL/CORBA	Yes, via Enterprise Edition	Yes, via Enterprise Edition

J2EE technology or API	WAS 3.5.3	WAS 4.0
JavaMail/JAF	Yes, via 3rd party vendor implementations and Lotus Domino	Yes, via 3rd party vendor implementations and Lotus Domino
JDBC 2.0	Yes	Yes
JDK 1.2	Yes, JDK 1.2.2	Yes, **JDK 1.3**
JMS 1.0	Yes, via MQSeries	Yes, via MQSeries
JNDI 1.1	Yes	Yes
JSP 1.1	Yes	Yes
JTS/JTA	Yes, with distributed transactions	Yes, with distributed transactions
RMI/IIOP	Yes	Yes
Servlet 2.2	Yes	Yes
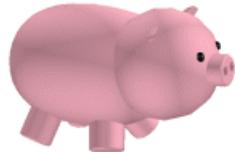
4

Introducing PiggyBank

This chapter introduces the PiggyBank application used to demonstrate the concepts discussed in this redbook.

This chapter discusses:

► Patterns for e-business
► PiggyBank scenario

4.1 Patterns for e-business overview

Patterns for e-business define reusable assets that can help speed the process of developing applications. The developerworks Web site at `http://www.ibm.com/developerworks/patterns/` describes business patterns as an interaction between the participants in an e-business solution.

The important business patterns are:

► User-to-Business
► User-to-Online Buying
► Business-to-Business
► User-to-Data
► User-to-User

4.1.1 User-to-Business

User-to-Business is the general case of users (internal to the enterprise or external) interacting with enterprise transactions and data. It is relevant to those enterprises that deal with goods and services not normally listed in and sold from a catalog. It covers all user-to-business interactions not covered by the User-to-Online Buying pattern. This Business pattern also covers the more complex case where there is a need to access back-end applications and data.

Examples of the User-to-Business pattern:

► Convenience Banking

– View account balances
– View recent transactions
– Pay Bills-Transfer funds
– Stop Payments
– Manage bank card

► Discount Brokerage

– Portfolio summary
– Detailed holdings
– Transaction history
– Quotes and news
– Buy and sell stocks

4.1.2 User-to-Online Buying

User-to-Online Buying is the special case (a subset of the User-to-Business pattern) where products are sold through a catalog using a shopping cart, a wallet, and so forth. This business pattern can also include links to back end systems to allow for inventory updates and credit checking.

Here are examples of the User-to-Online Buying pattern:

▶ Consumers purchasing goods online
▶ Buyers purchasing goods online from a supplier

4.1.3 Business-to-Business

Business-to-Business is between parties who do not belong to the same company. There are two styles of this pattern:

▶ Business-to-Business Integration: This style includes programmatic links between arms-length businesses (where a trading partner agreement might be appropriate).

▶ e-Marketplace or B2BeMP: In this second style of the Business-to-Business pattern, an e-Marketplace supports multiple buyers and suppliers. The buying function can be performed online or programmatically.

4.1.4 User-to-Data

The User-to-Data pattern encompasses the provision of Business Intelligence capabilities to an organization.

The user is someone connected to the data through one of four paths:

▶ Internet: The user is external to the company, or an agent of the company.

▶ Intranet: A staff (internal) user.

▶ Extranet or privileged Internet: An associate of the client's organization who acts as a business agent on the company's behalf.

▶ Fat client-connected as in a client-server system: Applies to internal users only.

> **Note:** There is a possibility of a "chaining" effect: the external user can trigger the user-to-data scenario while the user, actually connected to the data, is an internal staff member or agent. For example, a customer phones a query into an organization and a staff member connects to a data store to respond to the query. In this scenario, the user is defined as the staff member, not the customer.

The data can be held in:

- A Web-content store: Holding Web pages and cached information. The data may include copies of operational detailed records, such as consolidated account information for a customer reference. The data is read-only; if an update of the data is required, the pattern is User-Business, not User-Data.

- A Data mart: The data may be read-write with local scope-of-effect only.

- A Data warehouse: The data is read-only for applications.

- A Tool-specific store (proprietary): Some tools, such as Essbase, require specialized data stores for efficiency.

This pattern has several distinguishing characteristics:

- The user is not connecting to a traditional transactional system performing an operational business process.

- The user perceives him or herself to be interacting directly with the data rather than with a system.

- Normally, the user has significant freedom and flexibility in his or her access of available data. The data sets are often specially prepared in advance to suit the user or the tool he or she is using.

- The data sets being accessed:
  – Are not the company's prime operational data
  – Include a copy of relevant operational data and other data as necessary
  – Include a historical set of data

4.1.5 User-to-User

User-to User describes users collaborating with one another by e-mail, shared documents, and so forth, for example, collaborating across teams on document development.

4.2 PiggyBank application scenario

Our fictitious PiggyBank uses the common banking application scenario and employs the User-to-Business pattern. Interaction of the user with his or her available accounts at the local PiggyBank branch include transfer of funds between accounts, obtaining balances and making a mortgage payment. The samples are within the context of multiple users interacting with a Web site that is implemented with servlets and JSPs which accesses the underlying application business logic (implemented with EJBs). The middle tier accesses a backend

datastore (DB2) holding the account information for all customers of PiggyBank. Our scenario of online banking can be implemented with Application Topology6 (Figure 4-1), which allows for online bill payment from an existing customer account.
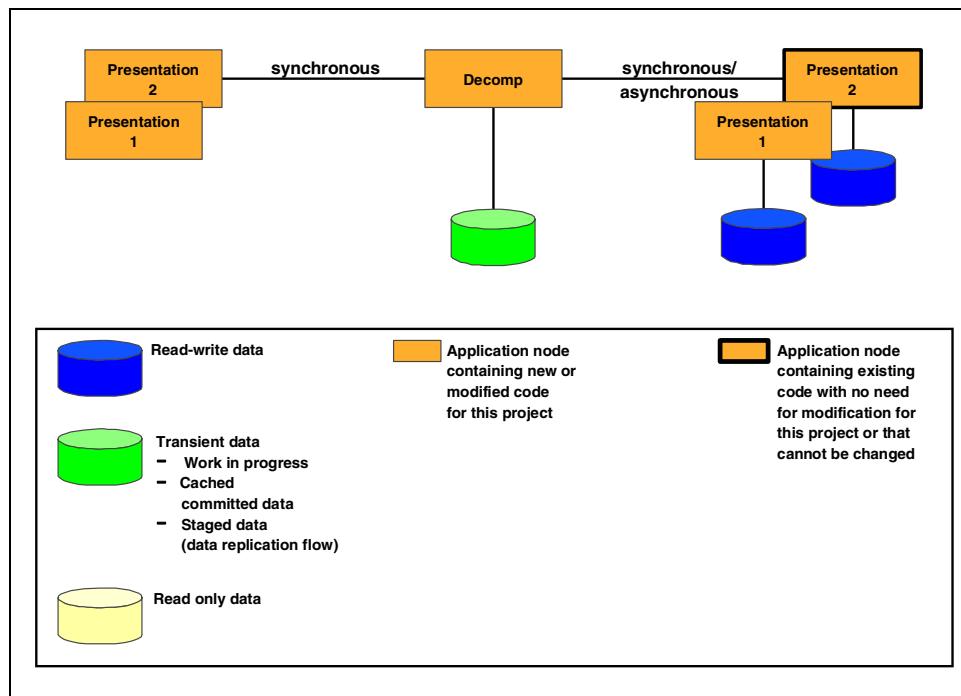
Figure 4-1   Application Topology6: User-to-Business pattern

4.2.1 Review of application topology

The presentation node in the first tier is responsible for the user interface into the Web application. It is responsible for all the Web-based presentation logic of the application. The application nodes in the third tier are responsible for a portion of the business logic and for data access. New applications can be developed specifically for the Web application or there can be existing legacy applications that might require modification.

A presentation node is linked to the decomposition node, which controls the application flow (Figure 4-2). The decomposition node breaks down the user request into individual application requests, forwards the requests to the applications, and then gathers the results to send back to the presentation node. The business logic of the application is divided between the applications and the decomposition node.
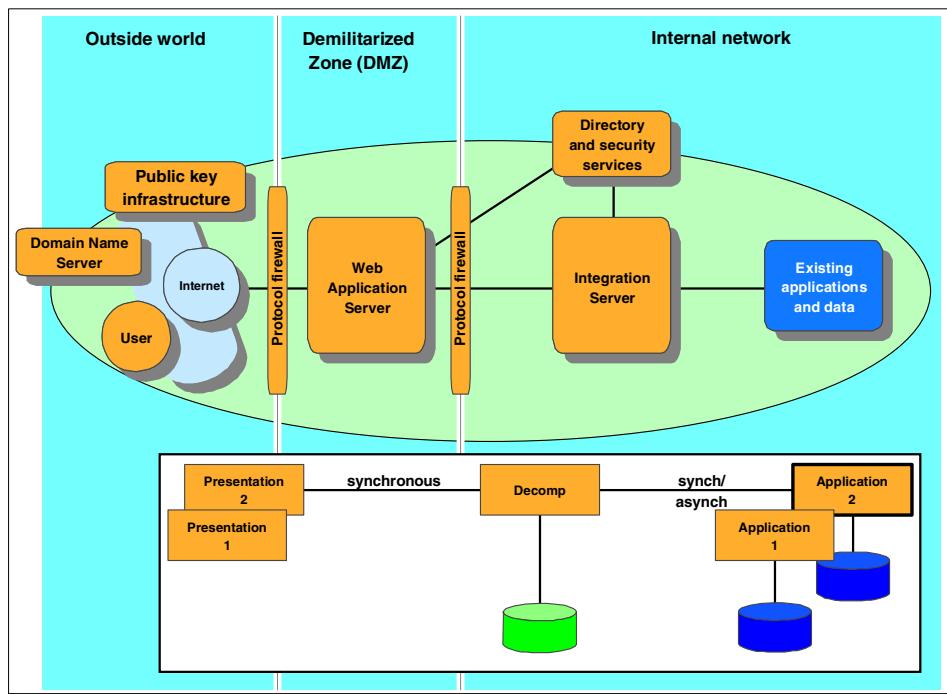
Figure 4-2   Runtime Topology6: User-to-Business pattern

Because the presentation logic and business logic are separated, it is easy to adapt the presentation node to new kinds of clients. The easiest approach is to use thin browser-based clients. You can also extend the presentation logic to new client platforms, for example, client Java applications or Web appliances like Web-enabled cellular phones or personal digital assistants (PDAs), without changing the business logic in the application node.

A banking company that purchased a mortgage company now wants to offer its customers the ability to make mortgage payments directly from their checking or savings account. A topology 6 solution can let users take money from their accounts and make mortgage payments. The application was designed to ensure that there is enough money in the account to cover the mortgage payment. It also was designed to ensure that if something goes wrong during the transaction, any changes made to either database are backed out.

The presentation control flow resides in the Web application server. The application logic implementing the routing and decomposition functions resides on the integration server, available for reuse by the presentation tiers.

The third-tier data and applications to be accessed are behind the domain firewall in the internal network. The integration server is also placed in the internal network for protection. Access to the Web application server resources is protected by the Web application server's security features, while access to the integration server's resources is protected by the integration server's security features. User information, needed for authentication and authorization by both servers, is stored in the directory and security services node behind the domain firewall in the internal network.

4.3 PiggyBank components

Within the PiggyBank application (see the logo Figure 4-3), the user can view his or her accounts, select an account, view balances, transfer funds between accounts and pay utility bills. The following components form the application.

Figure 4-3   PiggyBank

4.3.1 Servlets

Our controller servlet in the MVC paradigm is invoked by various user actions and interacts directly with a Helper interface which utilizes the required access beans to satisfy a particular user request. The controller servlet has no knowledge of the EJB implementation nor of the various access beans required to accomplish a given task; it is concerned only with performing a logical task and returning the results to a JSP.

There will be one servlet for the whole application, the *ControllerServlet*. Each client request will be handled by this servlet and a parameter will identify the requested functionality of the application (login, displaying a balance for a specific account, making a transfer, and so on). The servlet will act as a controller and manages the session.

For each call, the servlet will handle the request: it will retrieve the session information, check the validity of the request (for example, a user can't display a balance account after logging out) and calls a specific method to handle the execution of the required task.

This method will check the parameters sent by the user, transform it into objects understandable for the business components which will perform their business logic. The results will then be formatted for the presentation layer and sent to it.

The methods implemented in the servlet to control the execution of a logical task are:

- ► *performLogin*: Logins into the application.
- ► *performAccountList*: Displays a list of his or her accounts.
- ► *performAccountDisplay*: Displays the balance for a selected account.
- ► *performOperationInput*: Displays the form to fill in order to transfer money from an account to another or to pay a bill with an account.
- ► *performStatementDisplay:* Displays a list of the transactions associated with a particular account.
- ► *performTransfer*: Transfers money from an account to another one.
- ► *performMortgagePayment*: Performs a mortgage payment from funds within the users account.
- ► *performLogout*: Logs out from the application.

> **Note:** For simplicity, our controller servlet does not perform user authentication. In a real-world application, authentication would be performed against a directory, typically LDAP based. WebSphere's in built security can handle the challenge, authentication and authorization of users and is beyond the scope of this book. For more information, refer to the WebSphere 3.5 Security White paper at `http://www.ibm.com/websphere/`

To perform various actions, the controller servlet calls methods on various helper classes. The helpers use the singleton pattern so that multiple servlets will all interact with a single instance of each helper object.

- ► Naming Helper
  - – Provides access to properties files containing information required to perform actions against the WebSphere namespace, such as EJB home or MQSeries QueueConnectionFactory lookups.
- ► Transfer Helper
  - – Uses the Bank Transfer access bean to transfer funds between two accounts. This results in a session bean initiating a transaction with two Account entity beans.

- ▶ Mortgage Payment Helper
  - – Uses the Mortgage Payment access bean to pay some money towards the users mortgage. This results in a session bean initiating a transaction involving an entity bean and JMS.
- ▶ Account and Customer Helpers
  - – These two helpers use the access beans for our customer and account entity beans.
  - – Access beans are provided as an interface to the business logic.

4.3.2 JSPs

The presentation logic of the application is handled by JavaServer Pages. The JSPs are called by the controller servlet and passed a data bean containing the information to be displayed. Within the JSPs, custom tags are used to provide select functionality.

4.3.3 EJB specifics

Our ITSOEJBGroup contains the following EJBs:

- ▶ Account CMP
- ▶ Customer CMP
- ▶ BankTransfer stateless session bean
- ▶ Transaction Record CMP

The following access beans are also defined:

- ▶ Customer
- ▶ Account
- ▶ BankTransfer

Direct JDBC calls are used to populate the list of account for a customer as wells as to retrieve the transactions associated with an account (for example, to print a monthly statement).

In a real-world application where scalability was an issue, the BankTransfer EJB would act as a session facade to the underlying entity beans. A simple access bean wrapper provides the necessary functionality for purposes of this scenario.

4.3.4 MQSeries and JMS

Through a recent merger with a leading mortgage provider, PiggyBank has extended their online banking application to allow account holders to make payments to their mortgage account. The mortgage system implemented by the mortgage provider is not an EJB based system and in order to interact with it the PiggyBank application must use MQSeries to send asynchronous messages. These messages may then be handled by a message broker such as MQSeries Integrator (MQSI) and converted into a form that the mortgage system understands. To send messages, the application uses the JMS API. The messages are part of a transaction, and involve these objects:

► MortgagePayment stateless session EJB

► BankTransfer stateless session EJB (to transfer an amount from the account holders account to a special holding account)

► MessageSender stateless session EJB (a wrapper to the JMS code used to send the message)

4.3.5 Database

The underlying persistent store for all of our bank's data is a DB2 database. The format of the database is explained in Chapter 5, "Working with Enterprise JavaBeans" on page 39.

4.3.6 Setting up and using the PiggyBank scenario

The PiggyBank scenario is used throughout the chapters to demonstrate programming for the various APIs we discuss. All of the code is available to accompany this redbook, and setup instructions are provided in Appendix , "Using the Web material" on page 323.

The EJB container

Working with Enterprise JavaBeans

Enterprise Java Beans are becoming more accepted as the defacto method for enabling enterprise application development and for access to legacy systems in the Internet environment. Application Server functionality is being enhanced to fully support a multitude of client types, such as mobile phones, PDAs, kiosks and Web browsers. The servlet programming model replaces CGI scripts as the programming model of choice in the MVC paradigm and a multitude of reusable design patterns for various user-to-business and B2B (business-to-business) EJB architectures is prevalent. The common 3-tier programming model is enhanced by multi-layered application architectures providing greater separation between various client types and persistence mechanisms. It is now possible for application support for a particular database solution to be extended to application support for multiple databases. J2EE and EJBs help move Java into this new object space.

This chapter covers EJB functionality as defined in the EJB 1.1 specification. The goal is to make the technology understandable with a succinct presentation without delving into complex issues affecting only a fraction of the existing user scenarios. However, where there are known "best practices", the recommended approach is mentioned. From the application development standpoint, the changes which occurred moving from EJB 1.0 to EJB 1.1 primarily deal with the new XML deployment descriptor, enabling greater portability of EJBs across application servers (one format usable as input to alternate vendors).

In this chapter, we discuss these topics:

► EJB container contract
► EJB composition
► Session beans
► Entity beans
► Database connections
► XML deployment descriptor

Code fragments are used to illustrate the concepts as they are introduced, and a succinct, yet comprehensive, example is provided by the complete PiggyBank application.

5.1 EJB specification

An EJB is a server-side component representing an object in the business model of the enterprise application. Each EJB either represents business logic or provides access to a datasource. Business logic is normally represented by a session EJB, while data access is through one or more entity EJBs. The EJB specification defines home and remote interfaces on the bean that expose the capabilities of the bean to the client. Access to the bean is through these interfaces, effectively hiding the underlying implementation in the bean class.

EJBs handle the complexities of transactions, persistence and security transparently, allowing the developer to concentrate on the business logic of the application. EJBs make distributed programming concepts, such as distributed objects and transaction processing, readily available to the masses.

The EJB specification is from the point of view of a client, that is, client-centric. It is hoped that a later version of the specification will address in more detail the relationship of the application server to the EJB container, allowing for greater portability of components and tools for building EJBs. Current toolkits produce EJBs with a specific application server in mind as the target deployment platform.

The EJB 2.0 specification introduces new features:

► Message-driven beans

    – Process asynchronous messages via JMS
    – Transaction aware
    – Stateless

► Improved architecture for container-managed persistence

► Container-managed relationships for entity beans with CMP

    – One to one relationships
    – One to many relationships

- ► Remote and Local interfaces

  - – Does not use RMI
  - – Can have both Remote and Local interface
  - – Accessing a bean through its Local Interface avoids overhead of a remote call

- ► Enterprise JavaBeans Query Language

  - – Specified in the deployment descriptor

In the current 2.0 specification, instance variables can be designated in the deployment descriptor either as:

- ► Container-managed persistent fields (CMP Fields)

  - – Defined through the deployment descriptor

- ► Container-managed relationship fields (CMR Fields)

  - – Defined through the deployment descriptor

Local interfaces for session and entity beans are intended to provide light weight access from enterprise beans that are local clients.

5.1.1 EJB container

EJB containers manage enterprise beans at runtime. The developer specifies deployment characteristics of the bean and the application server toolkit generates the runtime code necessary for the application server to add the `EJB Group` to the container. `EJB Group` is a VisualAge specific concept that allows EJBs to be treated and deployed in groups; it is not a requirement outlined by the EJB specification. Note that EJBs live in a container and are only accessible to clients after deployment (Figure 5-1).
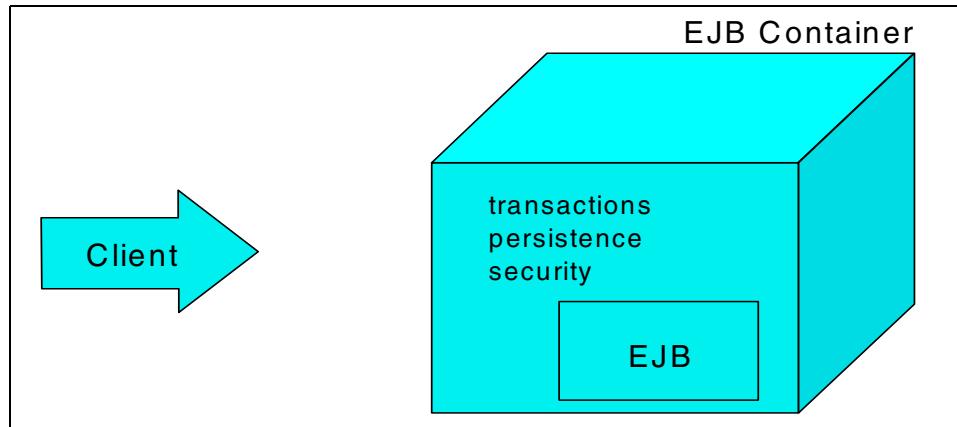
Figure 5-1   EJB container manages transactions, persistence and security

EJB container contract

The EJB specification establishes a contract for remote access to the bean by mandating that every EJB define a home interface and a remote interface. The container manages all run time characteristics of the EJB: remote access, transactions, persistence, security, concurrency, and pooled access.

- ► Remote Access
    - – J2EE Remote and Home Interfaces
    - – J2EE Bean Implementation Class
- ► Transactions
    - – Container-managed
    - – Bean-managed
- ► Persistence
    - – CMP — Container-managed
    - – BMP — Bean-managed
- ► Security
- ► Concurrency
- ► Pooled Resource Access

Each of these characteristics deserves further explanation. Code samples and additional details are provided under the appropriate section(s).

EJB container interaction

The container intercepts the client request and determines how to handle that request by looking at the runtime characteristics of the bean specified during deployment (for example, transactional, security, persistence). The contract between the bean and the container outlined in the EJB specification allows a developer to implement an enterprise bean that is portable across various application servers (vendor neutrality).

All access to the enterprise bean by the client is strictly through the EJB container. The container resolves object references to actual bean instances. Remember that EJBs are distributed Java components. In the standard distributed object paradigm, client-side code stubs are used to access the actual objects that may reside remotely. Saying this another way, the client program invokes methods on the remote interface.

In the J2EE implementation, the bean-container contract includes the notion of callback methods, an EJB context and a remote reference via JNDI. Callback methods notify the enterprise bean of various events in the life cycle of the bean. The EJB container invokes the callback methods to notify the bean of various events, allowing the bean to prepare for being saved to the database, for instance. There are also callback methods that allow the bean to do post processing tasks, if needed (for example, at the end of a transaction). Every bean creates and uses an `EJBContext` object that allows the bean to query the status of a transaction and obtain a handle to itself. Among other uses of the Java Naming and Directory Interface by the enterprise bean, JNDI allows the bean to obtain reference to other enterprise beans and gain access to the JDBC connection. Therefore, various technologies play different roles in providing the enterprise bean with a secure, transactional operating environment in the distributed object space.

5.1.2 EJB composition

What then, are the core requirements for implementing an EJB object or an EJB client? For the application developer, a *home interface,* a *remote interface*, and a *bean implementation class* must all be provided for the EJB object. A deployment descriptor is also defined specifying various attributes of the beans. The EJB is then packaged through the use of one or more tools and deployed by the application server to an EJB container where it is made available for client interaction. Integrated development environments (IDE) may further allow for testing the EJB independently or in combination with a client. VisualAge allows for the ability to construct and execute a test client wholly within the IDE before deployment to Websphere Advanced Server.

EJBs implement the **Proxy** design pattern. The home and remote interfaces refer to proxy objects. Stubs for the home and remote interface reside on the client-side. The application server manages instances of EJBs and access to the beans is through the client-side home and remote interfaces via RMI over IIOP.

Home and remote references are the primary way for a client to access the functionality of the underlying bean (Figure 5-2). The remote interface is used to access the underlying business methods on an EJB through container control. The home interface follows the **Factory** design pattern and is used in locating the bean and obtaining an instance of the bean. A remote reference, or handle, to the object must be obtained before an EJB method may be invoked by the client (Figure 5-2). The EJB container wrappers the client invocation request with whatever transactional characteristics are defined for that method or bean, placing the request in the appropriate transactional context. Finally, the bean implementation class provides methods that implement the business logic of the application.
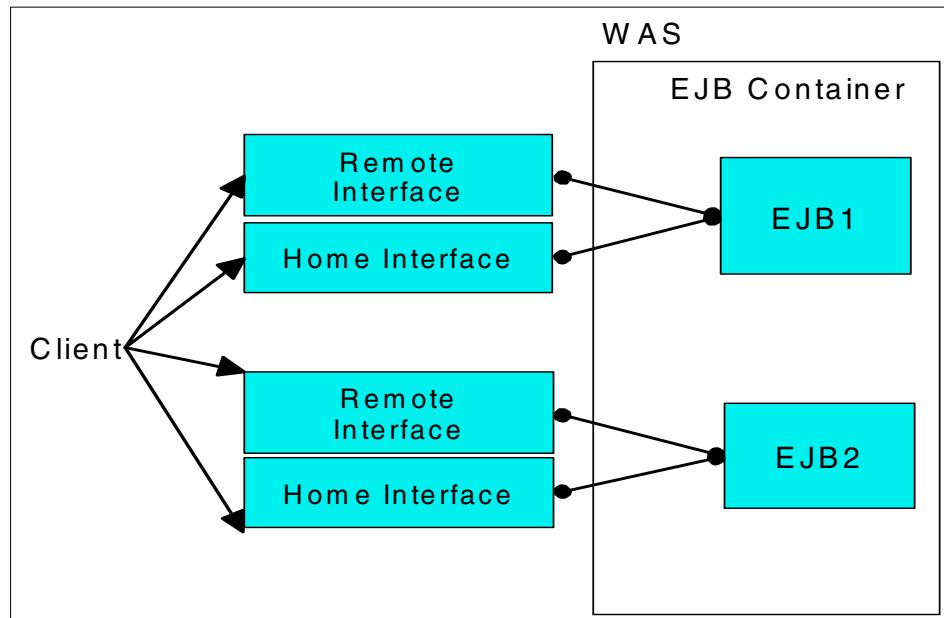
Figure 5-2   EJB client access via home and remote interfaces

EJB client

Servlets are the preferred client for accessing EJBs, since Websphere is able to optimize calls to the EJBs if the EJB and the EJB client are physically in the same JVM, thereby minimizing network overhead. The key steps a client must perform to access the business methods on an EJB are:

- ▶ Obtain the JNDI context
- ▶ Use the context to obtain the Home interface
- ▶ Use the Home interface to create the bean
- ▶ Invoke methods on the bean

Public interfaces

The EJB Home and EJB Remote interfaces comprise the set of public interfaces on the enterprise bean. All EJBs must define a Home interface, a Remote interface and a bean implementation class.

Home interface

Also referred to as the EJBHome, our Home interface must implement one or more create methods. For each create method, a corresponding `ejbCreate` and `ejbPostCreate` must be defined with the same number and type of arguments in the bean implementation. Stateless session beans would define a single `create()` method with no arguments, while stateful session beans would define one or more create methods that take arguments used in initializing the bean instance.

The purpose of the Home interface is to define the create methods that a client may invoke. With this in mind, the Home interface extends `EJBHome` and must define one or more create methods with a return type matching the remote interface type.

To summarize, the Home interface create method(s) must adhere to the:

- ▶ Number of arguments & type of arguments match corresponding `ejbCreate` method in bean implementation class

- ▶ Arguments and return types are valid RMI types

- ▶ Return type is the remote reference type

- ▶ Throws clause includes *java.rmi.RemoteException* and *javax.ejb.CreateException*

```
import javax.ejb.EJBHome;
import java.rmi.CreateException;
import java.rmi.RemoteException;

//Life Cycle Methods for Home Interface
public interface AccountHome extends EJBHome {

  //for Stateless Session Beans
  remoteType create ( )

  //for Entity Beans and Stateful Session Beans
  remoteType create ( args )
  throws RemoteException, CreateException;
```

```
//for Entity Beans only
public remoteType findByPrimaryKey( args )
throws FinderException, RemoteException;

//optional, for Entity Beans only
public Enumeration findByMethod( args )
throws FinderException, RemoteException;

}
```

Remote interface

Remote interfaces must extend *javax.ejb.EJBObect* or one of its descendants. Since Java RMI is utilized in J2EE to implement the underlying functionality, methods in the Remote interface can throw *java.rmi.RemoteException*.

The purpose of the Remote interface is to define the *business methods* that a client may invoke. With this in mind, the Remote interface extends `EJBObject` and defines business methods with return types that are valid RMI types.

To summarize, the methods defined in the Remote interface must adhere to the:

► Methods defined in the interface must be implemented in the bean class

► Arguments and return types are valid RMI types

► Throws clause must include *java.rmi.RemoteException*

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

// Methods for Remote Interface
public interface Account extends EJBObject {

  // Business Methods
  public void deposit ( java.math.BigDecimal amount )
  throws RemoteException;

  public void withdraw ( java.math.BigDecimal amount )
  throws RemoteException;

}
```

Bean Implementation Class

To understand the methods defined in the bean implementation class, it is important to remember that object life cycle methods exist for bean creation, passivation, activation and removal. The required methods defined in the implementation class correspond to these life cycle methods and are in addition to any business methods that may be defined on the class.

Therefore, the bean implementation class supplies the following standard methods common to both session beans and entity beans:

- ▶ `ejbCreate()`
- ▶ `ejbActivate()`
- ▶ `ejbPassivate()`
- ▶ `ejbRemove()`

Specifically, the `SessionBean` interface extends `EnterpriseBean` which extends `Serializable`. Therefore, session beans *implement* these methods:

- ▶ public void **ejbCreate**(*args*)
    - – One of these for a *stateless* session bean but with no args
    - – One or more of these for *stateful* session beans
    - – Return type is void
    - – Every `create` method in the home interface must be paired with an `ejbCreate` method in the bean class that has the same number and type of arguments
- ▶ **ejbActivate**()
    - – *stateless* session beans will always have an empty implementation
- ▶ **ejbPassivate**()
    - – *stateless* session beans will always have an empty implementation
- ▶ **ejbRemove**()
- ▶ **setSessionContext**(SessionContext theContext)

Similarly, the `EntityBean` interface extends `EnterpriseBean` which extends `Serializable`. Therefore, entity beans *implement* these methods:

- ▶ public *primaryKeyType* **ejbCreate**(*args*)
    - – One or more of these
    - – All ejbCreate methods on entity beans must return the primary key type with EJB 1.1
- ▶ public *primaryKeyType* **ejbFindByPrimaryKey**(*primaryKeyType*)
    - – One of these
- ▶ public `Enumeration` **ejbFindBy*Method***(*args*)
    - – Zero or more of these
    - – Can also return `Collection` in EJB 1.1
- ▶ **ejbActivate**()
- ▶ **ejbPassivate**()
- ▶ **ejbRemove**()
    - – BMP entity beans will delete the instance from the database

- ► **ejbPostCreate**(*args*)
  - – Optional
  - – If provided, must have an associated ejbCreate(*args*) in addition to a create(*args*) method in the home interface
- ► **setEntityContext**(EntityContext theContext)
- ► **unsetEntityContext**()
- ► **ejbLoad**()
  - – Empty implementation for CMP
  - – Developer provides implementation for BMP
- ► **ejbStore**()
  - – Empty implementation for CMP
  - – Developer provides implementation for BMP

> **Note:** VisualAge for Java 3.5.3 generates EJB 1.0 compatible code that has a `void` return type in `ejbCreate()` for CMP entity beans. For compatibility with EJB 1.1, `ejbCreate()` needs to return the primary key type (for all entity beans).

```
// AccountBean ejbCreate
public void ejbCreate(java.lang.String argAccountID, java.math.BigDecimal
argBalance) throws javax.ejb.CreateException, java.rmi.RemoteException
```

The remaining steps involved in writing an EJB include defining runtime and deployment characteristics for the bean and deploying the bean to an application server which reads in the deployment descriptor and adds the bean to an EJB container.

5.1.3 PiggyBank scenario

The PiggyBank example is used to illustrate the concepts as they are introduced. It uses the EJBBank database (Figure 5-3).
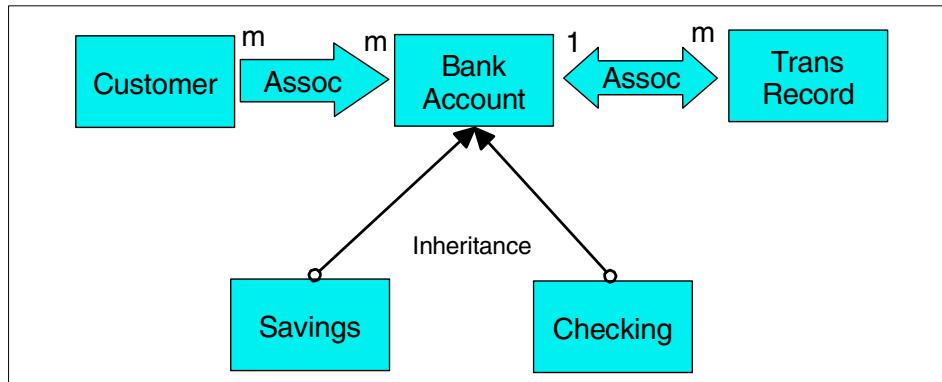
Figure 5-3   EJBBank scenario

In our PiggyBank scenario, we discuss:

- Bank Account interfaces and "`AccountBean`" implementation class

  – A simple entity bean that uses CMP

- Bank Transfer interface and "`BankTransferBean`"

  – Stateless session bean used as a session facade

- Customer interface and "`CustomerBean`"

  – Entity bean that uses CMP
  – Converted to illustrate BMP

- TransactionRecord interface and "`TransactionRecordBean`"

  – CMP enterprise bean used to record account transactions

The CMP fields for the `TransactionRecordBean` are:

- public String accID
- public java.math.BigDecimal transamt
- public java.sql.Timestamp transID
- public String transtype

The CMP fields for the `AccountBean` are:

- public String accountID
- public java.math.BigDecimal balance

The CMP fields for the `CustomerBean` are:

- public int customerID
- public String firstName
- public String lastName
- public String password

- ▶ public String title
- ▶ public String userID
- ▶ public CustomerAddress address

EJB inheritance is supported by VisualAge but is not part of the EJB specification. Inheritance might be used to define different kinds of bank accounts, such as "checking" and "savings" with different behavior. In our example, we do not implement the checking and savings subclasses.

5.2 Session beans

A session bean is the simplest form of EJB you can create. Session beans are not persisted to a datastore, but rather, are transient objects that may or may not hold state during a series of client invocations in the context of a single user session. Application state does not have to be persisted; it may be cached in memory. Session beans are useful if persistence is not required. It is important to point out, however, that a session bean may, in fact, choose to save or retrieve data directly from a database or some other persistence mechanism (although the state of the bean itself is not saved). Stateful beans are dedicated to a single client and maintain *conversational state* across all the methods of the bean. Another way of saying this is that a stateful session bean maintains *client state*.

Stateless session beans, on the other hand, can be shared across multiple clients, so any information kept in instance variables should not be visible to a client. Information kept in instance variables might include a connection that is shared by all client invocations and is not specific to any particular client.

Client state, sometimes referred to as application state, may be cached in the HTTPSession object (in memory on the application server). We may wish to *harden* the session information so that load balancing among a pool of available application servers can take place. Websphere persistent session storage supports this functionality. Multiple requests from the same servlet application must otherwise be serviced by the same application server (*server affinity*).

We may also want the ability to store and retrieve the session information to support a *fail over* capability where the existing session is not lost when a server fails, but rather, subsequent requests from the same client application could be serviced by another server. Also, in the case of load balancing to service large numbers of users, each server in the pool would have to have access to the same shared session information. WebSphere Application Server 3.5.3 supports hardening of the session with its shared HTTPSession implementation. An application server that did not support session persistence would require the developer to implement their own persistence mechanism. This could be done by

using entity EJBs to store session state. Invoking `findByPrimaryKey` would locate the appropriate EJB to restore the state. A stateful session EJB might also be used; however, the application server may not support fail over capability for stateful session EJBs.

The point to be made is that the data stored in the HTTPSession object may be persisted for a number of reasons, either direct to datastore, through the use of an EJB, or hand-coded using the JDBC API and serialized objects. The EJB solution is not recommended if the HTTPSession information cannot be modeled with a standard database schema. This is because complex associations will introduce a greater likelihood of errors and maintenance problems. Navigating EJB associations can be an expensive proposition.

When to use session beans
Session beans can access a database directly and typically represent *workflow* on behalf of the client application. Entity beans are simply an object representation of the shared data, which may be frequently updated. Entity beans do not contain workflow.

A stateless session bean should be used for **read-only objects**. Also, for any activity that can be accomplished in a single method call, such as generating a report.

For **write-only objects**, such as log files, a session EJB in combination with a set of dependent objects cached in memory on a per-server basis and saved to datastore under the control of the session EJB (using the Singleton design pattern) makes sense.[1]

For objects which are **frequently read,** but very infrequently updated, a session bean can be used to manage a set of dependent objects (which are not EJBs).[2]

Restricting the use of EJBs to where they are really needed will increase the performance of the application server rather noticeably. Not every Java Bean should be implemented as an EJB; similarly, not every EJB should be implemented as an entity bean. In addition, note that session beans have very low overhead, since the EJB container does not have to address any persistence requirements.

Session beans can be used in a *session facade* to restrict direct access to entity beans by the client. Access to entity beans is through the interface defined on the session facade. The facade delegates any required persistence operations to the underlying entity beans (Figure 5-4).

---

[1] Kyle Brown, et al., *Enterprise Java Programming with IBM WebSphere*, Addison Wesley Professional, 2001
[2] Ibid.
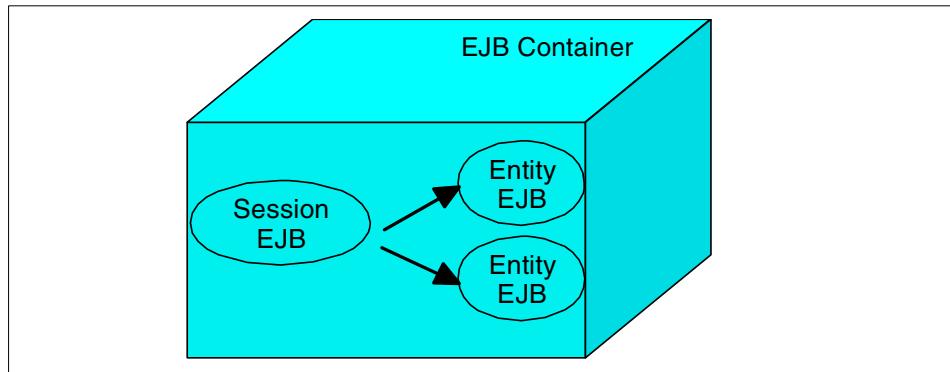
Figure 5-4   Session facade

Session bean: life cycle

Stateful Session EJBs are kept in memory by the application server for the duration of all invocations in a single session coming from any particular client. The life cycle of a session bean is managed by the EJB Container (see Figure 5-5 on page 53). The state transitions for the life cycle events are:

1. Client invokes `create`

2. EJB container invokes `setSessionContext`

3. EJB container invokes `ejbCreate` on the bean implementation class

4. EJB container may decide to invoke `ejbPassivate`

5. EJB container invokes `ejbActivate` if a passivated bean is requested by a client business method

6. Client invokes `remove`

7. EJB container invokes `ejbRemove`

Stateless Session EJBs, on the other hand, can be returned to the pool of EJB instances managed by the container once a client request has been completed. When the bean is requested through a client invocation, the EJB container invokes `setSessionContext()`, followed by `ejbCreate()`. The session context can be saved in an instance variable to be referred to as needed. When the business method completes, the container calls `ejbRemove()` on the bean implementation class.
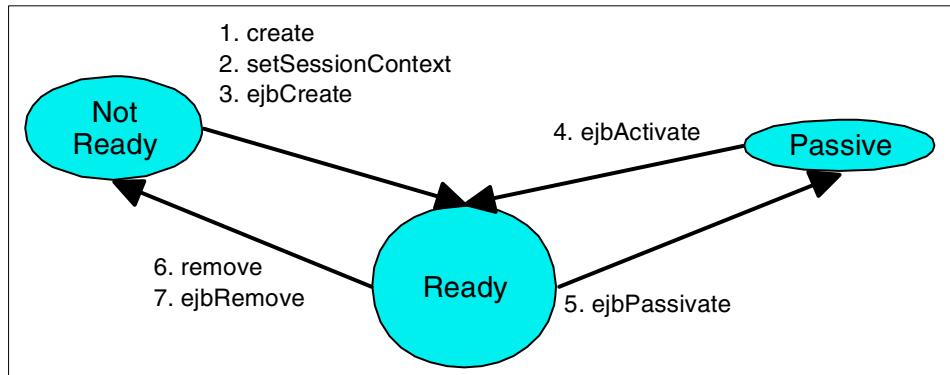
Figure 5-5   Stateful session bean life cycle

The `BankTransferBean` (a stateless session bean) is used in the example as a session facade to coordinate the activity of withdrawing funds from one bank account and depositing the funds into another bank account. The two accounts are implemented as entity beans.

Exceptions
If a business method is unable to connect to the database, then `javax.ejb.EJBException` should be thrown, which will be wrapped and returned to the client by the EJBContainer as a `RemoteException`.

Note: The EJB 1.1 specification specifies the container's behavior for the cases when a client attempts to access the primary key of a session object. In summary, the container must throw an exception on a client's attempt to access the primary key of a session object.

Session bean: Home interface
The Home interface of a *stateless* session bean declares a `create` method that takes no arguments and returns the remote type. The Home interface of a *stateful* session bean declares a `create` method with arguments necessary to initialize the instance variables of the stateful session bean.

```
public interface BankTransferHome extends javax.ejb.EJBHome {
  com.ibm.itso.j2eebook.ejb.BankTransfer create()
  throws javax.ejb.CreateException, java.rmi.RemoteException;
}
```

Session bean: Remote interface

The Remote interface requirements are the same for both stateful and stateless session beans. The Remote interface contains the business methods. Additional Helper classes may exist which are not necessarily exposed to the client through the Remote interface. The methods which the client may invoke are said to be *promoted* to the Remote interface.

```
public interface BankTransfer extends javax.ejb.EJBObject {

void transferFunds(java.lang.String fromAccountID,
                   java.lang.String toAccountID,
                   java.math.BigDecimal anAmount)
throws java.rmi.RemoteException,
com.ibm.itso.j2eebook.exceptions.InsufficientFundsException;
}
```

Session bean: bean class

The bean class has the same requirements for stateful and stateless beans with one exception: the ejbCreate method takes no arguments for *stateless* session beans.

Stateless session beans cannot participate in transactions and therefore, cannot implement the javax.ejb.SessionSynchronization interface. Furthermore, stateless beans are not passivated, so their implementation of the ejbActivate() and ebjPassivate() methods would be empty, although their ejbRemove() method may be used to close a resource connection.

Stateful beans may, however, wish to manage database data and would implement the afterBegins(), beforeCompletion() and afterCompletion() methods on the SessionSynchronization interface in this case. An additional complexity arises if the stateful session bean wrappers any remote object references, such as when accessing Java RMI objects. In this case, ejbPassivate() would have to be coded to handle passivation of the remote objects themselves and ejbActivate() would have to restore those object references, since the container will not do that automatically.

BankTransferBean class

```
import java.rmi.RemoteException;
import java.security.Identity;
import java.util.Properties;
import javax.ejb.*;
import javax.naming.*;
import com.ibm.itso.j2eebook.ejb.*;
import com.ibm.itso.j2eebook.helpers.*;
import com.ibm.itso.j2eebook.exceptions.*;
/**
```

```
 * This is a Session Bean Class
 */
public class BankTransferBean implements SessionBean {
    private javax.ejb.SessionContext mySessionCtx = null;
    private final static long serialVersionUID = 3206093459760846163L;

public void ejbCreate() throws javax.ejb.CreateException,
java.rmi.RemoteException {}
public void ejbActivate() throws java.rmi.RemoteException {}
public void ejbPassivate() throws java.rmi.RemoteException {}
public void ejbRemove() throws java.rmi.RemoteException {}
public void setSessionContext(javax.ejb.SessionContext ctx) throws
java.rmi.RemoteException {
    mySessionCtx = ctx;
}

public void transferFunds(String fromAccountID, String toAccountID,
java.math.BigDecimal amt)
throws InsufficientFundsException{
    Account fromAcct = null;
    Account toAcct = null;
    try {
     fromAcct = getAccountHome().findByPrimaryKey(new
AccountKey(fromAccountID));
     toAcct = getAccountHome().findByPrimaryKey(new AccountKey(toAccountID));
     fromAcct.withdraw(amt);
     toAcct.deposit(amt);
    } catch (Exception e) {
       if(e instanceof InsufficientFundsException)
          throw (InsufficientFundsException)e;
    }
}
```

The Home interface for a deployed bean is made available for client invocation by the container through the use of JNDI. In order to invoke methods on the remote interface, a client first obtains a reference to the EJB Home. This is done by obtaining a handle to the JNDI context and using the context to lookup the home reference. Methods are then invoked on the EJB Home, such as findByPrimaryKey.

Retrieving the home is cookie-cutter code that can be placed in its own method and invoked from ejbCreate. Since the getAccountHome() method retrieves the home on first use and it will be cached thereafter, it is not necessary to invoke getAccountHome() from ejbCreate in our sample code.

> **Note:** We define a resource bundle in an associated properties file that is used to look up JNDI names by supplying a key to getJNDIName(), which returns the corresponding value, thereby avoiding hardcoded JNDI names for various properties.

getAccountHome method

```
protected AccountHome getAccountHome() throws NamingException {

    AccountHome accountHome =
    (AccountHome) javax.rmi.PortableRemoteObject.narrow(
            HomeHelper.singleton().getHome("piggy.Account"),
            AccountHome.class);
    return accountHome;
}

public Object getHome(String homeName) throws NamingException{
    Object obj = (Object)homes.get(homeName);
    if (obj == null) {
        // get InitialContext and perform lookup
        obj = NamingHelper.singleton().getInitialContext().lookup(
            NamingHelper.singleton().getJNDIName(homeName));
        homes.put(homeName, obj);
    }
    return obj;
}
```

5.3 Entity beans

Entity beans provide a richer set of functionality to the application developer, but also carry with them a performance penalty due to the additional overhead in managing their persistence and other attributes. Entity beans may employ either CMP (container managed persistence) or BMP (bean managed persistence). BMP provides additional flexibility by allowing the developer to fully manage the persistence of the bean. The downside is the additional complexity involved in manually writing the necessary SQL code. The persistence code is generated automatically in the case of CMP (but does not allow for fine-grained performance tuning by the developer). EJB 2.0 enhances the flexibility of CMP through the use of local and remote interfaces. When to use CMP versus BMP is an important design consideration, as the number of CMP beans will affect the startup time of the application server and the runtime performance of the application itself.

The life cycle of an entity bean involves the EJB container managing a pool of available instances. Instances in the pool are not associated with any particular EJB object (Figure 5-6).
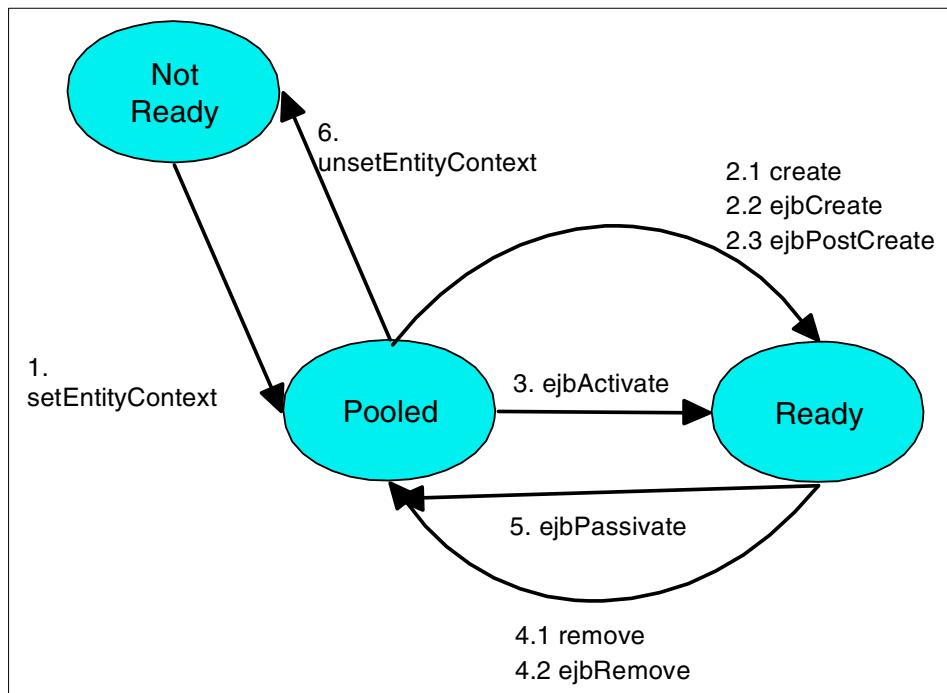
Figure 5-6 Entity bean life cycle

The state transitions for the life cycle events are:

1. EJB container invokes `setEntityContext`

2. Client invokes `create`, causing:

   a. EJB container to invoke `ejbCreate`
   b. EJB container to invoke `ejbPostCreate`

3. Alternatively, the EJB Container may simply call `ejbActivate`

4. Client invokes `remove`, causing:

   a. EJB container to invoke `ejbRemove`

5. Alternatively, the EJB Container may simply call `ejbPassivate`

6. EJB Container invokes `unsetEntityContext`

5.3.1 Examining CMP

Container-managed persistence implies that the EJB container on the application server will manage the persistence of the entity bean. The code to persist the bean is generated by a vendor-specific tool. Note that the Home and Remote interfaces are coded in the same fashion regardless of whether BMP or CMP are utilized.

Entity bean (CMP)

An entity bean with CMP is used to transfer funds between two bank accounts for a particular logged-on user. The entity bean may have additional dependent classes which are not enterprise beans. An example would be an `Account` object that had dependent `Name` and `Address` objects.

Home interface (CMP)

A CMP entity bean contains one or more `create` methods and one or more finder methods. The `findByPrimaryKey(`*primaryKey*`)` method is required.

```
remoteType create (args)
throws RemoteException, CreateException;

public remoteType findByPrimaryKey(args) // required
throws FinderException, RemoteException;

public Enumeration findByMethod(args) // optional, one or more
throws FinderException, RemoteException;
```

```
Home interface for CMP Entity Bean

public interface AccountHome extends javax.ejb.EJBHome {

com.ibm.itso.j2eebook.ejb.Account create(java.lang.String argAccountID,
java.math.BigDecimal argBalance)
throws javax.ejb.CreateException, java.rmi.RemoteException;

com.ibm.itso.j2eebook.ejb.Account create(java.lang.String argAccountID)
throws javax.ejb.CreateException, java.rmi.RemoteException;

com.ibm.itso.j2eebook.ejb.Account
findByPrimaryKey(com.ibm.itso.j2eebook.ejb.AccountKey key) throws
java.rmi.RemoteException, javax.ejb.FinderException;
}
```

Remote interface (CMP)

The Remote interface declares the business methods available to a client.

Remote interface for CMP Entity Bean

```
public interface Account extends javax.ejb.EJBObject {
//business methods
void withdraw(java.math.BigDecimal amount)
throws java.rmi.RemoteException,
com.ibm.itso.j2eebook.exceptions.InsufficientFundsException;

void deposit(java.math.BigDecimal amount) throws java.rmi.RemoteException;
}
```

Bean class (CMP)

Entity beans implement *javax.ejb.EntityBean*. Container-managed fields should
be initialized in the ejbCreate(*args*) method. The method signatures for a CMP
entity bean are followed by a brief description of any salient points along with
sample code to illustrate key concepts where necessary.

CMP Entity Bean class

```
import java.rmi.RemoteException;
import java.security.Identity;
import java.util.Properties;
import javax.ejb.*;
import javax.naming.*;
import com.ibm.itso.j2eebook.ejb.*;
import com.ibm.itso.j2eebook.helpers.*;
import com.ibm.itso.j2eebook.exceptions.*;

/**
 * This is an Entity Bean class with CMP fields
 */
public class AccountBean implements EntityBean {
    public String accountID;
    public java.math.BigDecimal balance;
    private javax.ejb.EntityContext entityContext = null;
    private final static long serialVersionUID = 3206093459760846163L;

public void ejbActivate() throws java.rmi.RemoteException {}
public void ejbPassivate() throws java.rmi.RemoteException {}
public void ejbRemove() throws java.rmi.RemoteException,
javax.ejb.RemoveException {}
public void ejbLoad() throws java.rmi.RemoteException {}
public void ejbStore() throws java.rmi.RemoteException {}
public void setEntityContext(javax.ejb.EntityContext ctx) throws
java.rmi.RemoteException {
    entityContext = ctx;
```

```
    }
    public void unsetEntityContext() throws java.rmi.RemoteException {
        entityContext = null;
    }

    public void ejbCreate(java.lang.String argAccountID) throws
    javax.ejb.CreateException, java.rmi.RemoteException {
        _initLinks();
        // All CMP fields should be initialized here.
        accountID = argAccountID;
        balance = new java.math.BigDecimal(0);
    }
    public void ejbCreate(java.lang.String argAccountID, java.math.BigDecimal
    argBalance)
    throws javax.ejb.CreateException, java.rmi.RemoteException {
        _initLinks();
        // All CMP fields should be initialized here.
        accountID = argAccountID;
        balance = argBalance;
    }
    public void ejbPostCreate(java.lang.String argAccountID) throws
    java.rmi.RemoteException {}

    public void deposit(java.math.BigDecimal amount) {
        balance = balance.add(amount);
        try {
            getTransactionRecordHome().create(accountID, amount, "D");
        } catch (Exception e) {
            System.out.println("==> TxnRecord could not be created!");
        }
    }

    public void withdraw(java.math.BigDecimal amount) throws
    InsufficientFundsException {
        if (balance.compareTo(amount) == -1) {
            throw new InsufficientFundsException("Need more allowance");
        } else {
            balance = balance.subtract(amount);
        }
        try {
            getTransactionRecordHome().create(accountID, amount, "W");
        } catch (Exception e) {
            System.out.println("==> TxnRecord could not be created!");
        }
    }

    protected TransactionRecordHome getTransactionRecordHome() throws
    NamingException {
        TransactionRecordHome txnRecordHomeCMP =
```

```
            (TransactionRecordHome) javax.rmi.PortableRemoteObject.narrow(
                HomeHelper.singleton().getHome("piggy.TransactionRecord"),
                TransactionRecordHome.class);
        return txnRecordHomeCMP;
    }
```

ejbLoad & ejbStore

```
  public void ejbLoad ( ) throws RemoteException {
    System.out.println("Bean is loaded from the database");
  }
  public void ejbStore ( ) throws RemoteException {
    System.out.println("Bean is stored to the database");
  }
```

The client cannot call either `ejbLoad` or `ejbStore`. These methods are invoked by the container to load or save the state of a bean. For CMP, this is handled transparently by the container, and these methods will have an empty implementation. For BMP, the persistence mechanism is managed by the application developer.

setEntityContext & unsetEntityContext

```
  public void setEntityContext (EntityContext theContext)
  throws RemoteException {
    System.out.println("entity context is set");
  }
  public void unsetEntityContext () throws RemoteException {
    System.out.println("container is removing our context ");
  }
```

The default implementation of `setEntityContext` and `unsetEntityContext` is usually sufficient. The `EntityContext` is *transient* since it represents runtime information about the bean.

ejbActivate & ejbPassivate

```
  public void ejbActivate () throws RemoteException {
    System.out.println("Bean about to be activated");
  }
  public void ejbPassivate () throws RemoteException {
    System.out.println("Bean about to be passivated");
  }
```

For entity beans, the `ejbPassivate` method notifies the bean that it will no longer be associated with a particular entity (for reuse) or will be de-referenced and can be garbage collected. Remember that stateful beans are usually evicted from the container when passivated, while entity beans and stateless beans are pooled for later reuse.

The `ejbActivate` method is called when the passivated bean is being activated from the pool.

ejbCreate, ejbPostCreate & ejbRemove

```
public void ejbCreate(java.lang.String argAccountID)
throws javax.ejb.CreateException, java.rmi.RemoteException {
  System.out.println("set all instance variables here");
}
public void ejbPostCreate(java.lang.String argAccountID) throws
java.rmi.RemoteException {
  System.out.println("post create actions, if any");
}
public void ejbRemove () throws RemoteException {
  System.out.println("Bean is removed from the database");
}
```

The bean's `ejbCreate` method is called by the container after the client calls `create` on the remote interface. All instance variables should be set either from the supplied arguments or by other means (for example, set to default values) in this method.

The `ejbPostCreate` method can invoke `getPrimaryKey` and `getEJBObject` on `EntityContext`, since the bean has already been created when `ejbPostCreate` is called. Any additional initialization that may have to be performed before business methods are invoked on the bean can be done in this method.

The `ejbRemove` method is called by the container when the bean is about to be removed from the server, so the bean should be removed from the datastore here as well. Note that with CMP, the bean is removed by the container.

findByPrimaryKey

```
Account findByPrimaryKey(AccountKey key)
throws java.rmi.RemoteException, javax.ejb.FinderException;
```

The `findByPrimaryKey` method is a required method. The container manages persistence for CMP, so the method implementation is not provided.

5.3.2 Examining BMP

Bean managed persistence requires the application developer to implement the methods necessary to persist the state of the bean.

Note on exceptions: If an ejbCreate method cannot create an entity bean due to a duplicate key, it should throw `javax.ejb.DuplicateKeyException`, a subclass of `CreateException`.

Entity bean (BMP)

An entity bean implementation of the `Customer` object is used to illustrate the concepts. The `Customer` object is a good candidate for a BMP entity bean, since it may contain one or more *dependent objects* which could require multiple SQL select statements in a CMP implementation. The BMP implementation could handle arbitrarily complex relationships and table joins.

For illustrative purposes, we begin with code fragments that use a simple DDL combining the customer address information into the `Customer` table. Later, in the more realistic PiggyBank scenario, we define the address information as a dependent object. If the two tables share the same primary key, then CMP would still be a good choice to handle the table join. The idea here is that BMP can be used when necessary to load a group of related objects in a single SQL statement, as in the case of multiple tables joined together by a single relationship table.

> **Definition:** Dependent objects are normal JavaBeans that are created by an owning object, such as an EJB. The dependent objects are normally created when the owning object is marshalled. This marshalling may be delayed until the dependent objects are actually referenced if lazy initialization is employed.

Home interface (BMP)

There are no coding differences in the home interface due to whether BMP or CMP is employed.

Note that the `create` method shown has a single argument, the customer key. The matching `ejbCreate` is still responsible for initializing all the instance variables; variables not supplied as arguments would be initialized to default values, such as `null` string for `String` arguments.

Home Interface for a BMP Entity Bean

```
public interface CustomerHome extends javax.ejb.EJBHome {

com.ibm.itso.ejb.Customer create(com.ibm.itso.ejb.CustomerKey primaryKey)
  throws javax.ejb.CreateException, java.rmi.RemoteException;

com.ibm.itso.ejb.Customer findByPrimaryKey(com.ibm.itso.ejb.CustomerKey key)
  throws java.rmi.RemoteException, javax.ejb.FinderException;
}
```

Remote interface (BMP)

There are no coding differences in the remote interface due to whether BMP or CMP is employed.

Remote Interface for a BMP Entity Bean

```
public interface Customer extends javax.ejb.EJBObject {
  //business methods
  com.ibm.itso.ejb.Address getMyAddr() throws java.rmi.RemoteException;
  com.ibm.itso.ejb.Name getMyName() throws java.rmi.RemoteException;
}
```

Bean class (BMP)

Entity beans implement *javax.ejb.EntityBean*. The relevant method signatures
are followed by a brief description of any salient points along with sample code to
illustrate the concepts.

BMP Entity bean class

```
public class CustomerBean implements EntityBean {
    private javax.ejb.EntityContext entityContext = null;
    private final static long serialVersionUID = 3206093459760846163L;
    Name myName;
    Address myAddr;
    String custID;
    private static DataSource ds;

    public static final String EJBLOAD =
"SELECT cust_ln, cust_fn, cust_addr, cust_city, cust_st, cust_ctry FROM
DB2ADMIN.CUSTOMER WHERE cust_id=?";

    public static final String EJBCREATE =
"INSERT INTO
DB2ADMIN.CUSTOMER(cust_id,cust_ln,cust_fn,cust_addr,cust_city,cust_st,cust_ctry
) VALUES (?,?,?,?,?,?,?)";

    public static final String EJBSTORE =
"UPDATE DB2ADMIN.CUSTOMER SET cust_ln=?, cust_fn=?, cust_addr=?, cust_city=?,
cust_st=?, cust_ctry=? WHERE cust_id=?";

    public static final String EJBFINDBY =
"SELECT cust_id FROM DB2ADMIN.CUSTOMER WHERE cust_id=?";

    public static final String EJBREMOVE =
"DELETE from DB2ADMIN.CUSTOMER WHERE cust_id = ?";
}
```

The `ejbCreate` method is invoked by the container when the client invokes the
`create` method. It creates a new record in the database.

ejbCreate and ejbRemove (BMP)

```
public com.ibm.itso.ejb.CustomerKey ejbCreate(com.ibm.itso.ejb.CustomerKey key)
throws javax.ejb.CreateException, java.rmi.RemoteException {
    Connection conn = null;
    PreparedStatement sqlStmt = null;
    try{
        custID = key.primaryKey;//set all the instance variables for our bean
        myName = new Name();
        myName.lastName = "";
        myName.firstName = "";
        myAddr = new Address();
        myAddr.street = "";
        myAddr.city = "";
        myAddr.state = "";
        myAddr.country = "";

        conn = getConnection();
        sqlStmt = conn.prepareStatement(EJBCREATE);
        sqlStmt.setString(1,key.primaryKey);
        sqlStmt.setString(2,"");//myName.lastName
        sqlStmt.setString(3,"");//myName.firstName
        sqlStmt.setString(4,"");//myAddr.street
        sqlStmt.setString(5,"");//myAddr.city
        sqlStmt.setString(6,"");//myAddr.state
        sqlStmt.setString(7,"");//myAddr.country
        sqlStmt.executeUpdate();
    } catch (java.sql.SQLException sqle) {
        System.out.println("ejbCreate failed for CustomerBean");
    } finally {
        cleanup(sqlStmt,conn);
    }
    return key;
}

public void ejbRemove() throws java.rmi.RemoteException,
javax.ejb.RemoveException {
    Connection conn = null;
    PreparedStatement sqlStmt = null;
    CustomerKey key = (CustomerKey)entityContext.getPrimaryKey();
    try {
        conn = this.getConnection();
        sqlStmt = conn.prepareStatement(EJBREMOVE);
        sqlStmt.setString(1,key.primaryKey);
        if (sqlStmt.executeUpdate() == 0) {
            throw new RemoveException("oops ejbRemove failed for CustomerBean");
        }
    } catch (java.sql.SQLException sqle) {
        System.out.println("oops ejbLoad failed for CustomerBean");
```

```
        } finally {
         cleanup(sqlStmt,conn);
        }
}
```

The `ejbFindByPrimaryKey` method must be implemented for BMP and provides a
way to retrieve a particular EJB from the database given its primary key. Note
that only the key is returned from this method. A subsequent call must be made
to actually create the object.

ejbFindByPrimaryKey (BMP)

```
public com.ibm.itso.ejb.CustomerKey
ejbFindByPrimaryKey(com.ibm.itso.ejb.CustomerKey primaryKey)
throws java.rmi.RemoteException, javax.ejb.FinderException {
    Connection conn = null;
    PreparedStatement sqlStmt = null;
    boolean wasFound = false;
    try {
     conn = this.getConnection();
     sqlStmt = conn.prepareStatement(EJBFINDBY);
     sqlStmt.setString(1,primaryKey.primaryKey);
     ResultSet results = sqlStmt.executeQuery();
     wasFound = results.next();
    } catch (java.sql.SQLException sqle) {
     System.out.println("ejbFindBy failed for CustomerBean");
    } finally {
     cleanup(sqlStmt,conn);
    }
    return primaryKey;
}
```

The `ejbLoad` and `ejbStore` methods *must* be implemented for BMP and are
called to store and retrieve, or synchronize, the state of the bean with the
database.

ejbLoad and ejbStore (BMP)

```
public void ejbLoad() throws java.rmi.RemoteException {
    Connection conn = null;
    PreparedStatement sqlStmt = null;
    boolean wasFound = false;
    try {
     CustomerKey key = (CustomerKey)entityContext.getPrimaryKey();
     conn = this.getConnection();
     sqlStmt = conn.prepareStatement(EJBLOAD);
     sqlStmt.setString(1,key.primaryKey);
     ResultSet results = sqlStmt.executeQuery();
     wasFound = results.next();
```

```
        if (wasFound) {
           myName = new Name();
           myName.lastName = results.getString("cust_ln");
           myName.firstName = results.getString("cust_fn");
           myAddr = new Address();
           myAddr.street = results.getString("cust_addr");
           myAddr.city = results.getString("cust_city");
           myAddr.state = results.getString("cust_st");
           myAddr.zip = results.getString("cust_ctry");
        }
     } catch (java.sql.SQLException sqle) {
        System.out.println("ejbLoad failed for CustomerBean");
     } finally {
       cleanup(sqlStmt,conn);
     }
}

public void ejbStore() throws java.rmi.RemoteException {
   Connection conn = null;
   PreparedStatement sqlStmt = null;
   try{
      CustomerKey key = (CustomerKey)entityContext.getPrimaryKey();
      conn = getConnection();
      sqlStmt = conn.prepareStatement(EJBSTORE);
      sqlStmt.setString(1,myName.lastName);
      sqlStmt.setString(2,myName.firstName);
      sqlStmt.setString(3,myAddr.street);
      sqlStmt.setString(4,myAddr.city);
      sqlStmt.setString(5,myAddr.state);
      sqlStmt.setString(6,myAddr.zip);
      sqlStmt.setString(7,key.primaryKey);
      sqlStmt.executeUpdate();
   } catch (java.sql.SQLException sqle) {
      System.out.println("ejbStore failed for CustomerBean");
    } finally {
     cleanup(sqlStmt,conn);
    }
}
```

Utility methods are often useful in hiding the implementation details of a particular approach. Changes to the underlying implementation can also be accomplished in a few routines instead of throughout the code. One way of implementing getConnection() is shown in the example here.

Utility Methods

```
private DataSource getDatasource() throws java.sql.SQLException {
   javax.naming.InitialContext ctx = null;
   Properties prop = new Properties();
```

```
        if (ds == null) {
          try {
          prop.put( javax.naming.Context.PROVIDER_URL, "iiop://localhost:900/");
          prop.put( javax.naming.Context.INITIAL_CONTEXT_FACTORY,
                  "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
           ctx = new javax.naming.InitialContext(prop);
          ds = (DataSource) ctx.lookup("jdbc/EJBBANK");
          } catch (javax.naming.NamingException e) {
          System.out.println("Error retrieving datasource");
          }
        }
        return ds;
}

private Connection getConnection() throws SQLException {
    Connection conn = null;
    try{
      conn = getDatasource().getConnection("db2admin","db2admin");
    } catch (Exception e) {
      System.out.println("couldn't connect to EJBBANK");
    }
    return conn;
}
```

ejbLoad & ejbStore

For BMP, the persistence mechanism is managed by the application developer, so the implementations of these methods must be provided. The ejbLoad method will need to obtain a database connection and the primary key. The primary key can be obtained from the EntityContext.

```
CustomerKey key = (CustomerKey)entityContext.getPrimaryKey();
```

The entity bean obtains the result set and maps columns from a row in the result set to values of corresponding instance variables on the bean. In ejbStore the instance variables are mapped to SQL substitution parameters in a preparedStatement. Executing the prepared statement updates the applicable row in the database. In the case of a more complex bean with dependent objects, the dependent objects would have to be loaded and stored as part of the ejbLoad or ejbStore for the enterprise bean.

Note: A *prepared* statement separates the compilation process and execution process while a *callable* statement avoids compilation by executing a stored procedure. WebSphere uses a prepared statement cache for its JDBC connection pooling and skips the compilation for anything that is already in cache, for instance, previously executed prepared statements.[3]

---

[3] Kyle Brown, et al., *Enterprise Java Programming with IBM WebSphere*, Addison Wesley Professional, 2001

Data type conversion, if necessary, is done manually in these methods.

setEntityContext & unsetEntityContext

The default implementations for these methods provided by VisualAge normally suffice.

```
public void setEntityContext(javax.ejb.EntityContext ctx)
throws java.rmi.RemoteException {
  entityContext = ctx;
}

public void unsetEntityContext() throws java.rmi.RemoteException {
  entityContext = null;
}
```

ejbActivate & ejbPassivate

For BMP, the ejbActivate and ejbPassivate methods are implemented if desired to perform any additional actions which may be necessary before the bean is created (in this case, use ejbActivate) and before the bean's state is written to datastore (in this case, use ejbPassivate).

```
public void ejbPassivate() throws java.rmi.RemoteException {
  System.out.println("activating the BMP");
}

public void ejbActivate() throws java.rmi.RemoteException {
  System.out.println("passivating the BMP");
}
```

ejbCreate, ejbPostCreate & ejbRemove

For BMP, `ejbCreate` must save the object to datastore, which requires obtaining a database connection. A `finally` clause should be included to close the connection if an exception occurs. If an object has associations, the associated objects can be created as well.

The `ejbPostCreate` method gives the developer access to the bean instance immediately after creation. This is useful in creating dependent objects. Dependent objects could be created after the main object had already been instantiated.

For BMP, a bean must also delete itself from the datastore in `ejbRemove`.

findByPrimaryKey

This method is required for BMP. Additional `findBy` methods may also be provided, as desired.

5.3.3 Custom key class

For primary keys that are common java types, a primary key class can be defined during deployment. For other key types consisting of multiple fields, a custom primary key class can be defined by the developer.

For example, if `AcctID` was defined as the primary key for `Acct`, then a simple key class might look like:

```
public AcctKey(int argAcctID) {
acctID = argAcctID;
}

public boolean equals(Object o) {
if (o instanceof AcctKey) {
  AcctKey otherKey = (AcctKey) o;
  return (((this.acctID == otherKey.acctID)));
}else
  return false;
}

public int hashCode() {
  return ((new java.lang.Integer(acctID).hashCode()));
}
```

When defining primary keys, the key requirements are:

► For CMP, the field names in the primary key class must match the corresponding container-managed fields.

► The class must implement `hashCode()` and `equals(Object other)`.

► The class implements a default public constructor.

► The class is `Serializable`.

```
public AcctKey(int argAcctID, String argBankID) {
  this.acctID = argAcctID;
  this.bankID = argBankID;
}
```

5.3.4 Bean finder helpers

The BeanFinderHelper interface is generated when CMP entity beans are created to assist the developer in creating *custom finder* helpers. These are in addition to the `findByPrimaryKey` method and any other finder methods which may have been generated for supporting associations.

Additional `findBy` methods are often used in order to find an entity by something other than its primary key. `FindBy` methods can return either an `Enumeration` or a `Collection`. This is because the possibility exists that more than one entity may match the search criteria.

```
public Enumeration findAccountsWithBalanceGreaterThan(BigDecimal
anExorbitantAmount)
```

Custom finders are often implemented for the convenience of the client in retrieving a set of desired entities. There are two recommended ways of implementing custom finders:

► Where clause custom finders

 – Requires a string constant for the WHERE clause be defined in the `BeanFinderHelper` interface

 – Requires a matching method in the Home interface

► Method custom finders

 – Requires the signature of the finder method to be defined in the `BeanFinderHelper` interface

 – Requires a matching method in the Home interface

 – Requires an implementation class be written that extends `com.ibm.vap.finders.VapEJBJDBCFinderObject` and implements the `BeanFinderHelper` interface

For each finder method an SQL query string or a method declaration must be defined in the `BeanFinderHelper` interface. All method declarations further require that an implementation class be written.

In the case of a *where customer finder*, if the Home interface contains the following method declaration:

```
public Enumeration findAccountsWithBalanceGreater(BigDecimal threshold)
throws java.rmi.RemoteException, javax.ejb.FinderException;
```

Then, a where clause must be specified for the associated SQL query in the `BeanFinderHelper` interface:

```
public interface AccountBeanFinderHelper {
public static final String findAccountsWithBalanceGreaterWhereClause =
"T1.threshold > ?";
}
```

In the case of a *method custom finder*, if the Home interface contains the following method declaration:

```
public Enumeration findGoldAccounts(BigDecimal threshold) throws
Exception;
```

Then, we can define the associated method in the `BeanFinderHelper` interface:

```
public interface AccountBeanFinderHelper {
public java.sql.PreparedStatement findGoldAccounts(BigDecimal
threshold) throws Exception;
}
```

In this case, the method implementation must be provided. The name of the class must be <beanClassName>BeanFinderObject (that is, AccountBeanFinderObject) or the name of the class can be specified in the environment properties of the enterprise bean with certain restrictions.

Defining Custom Finders in the BeanFinderHelper Interface

```
* WHERE Custom Finder:
*     public static final String findGreaterThanWhereClause = "T1.VALUE > ?";
*
*     In case there is no where clause in the SQL statement such as
*     "SELECT * FROM MYTABLE",
*     use a query string that always evaluates to true. For example,
*     public static final String findAllWhereClause = "1 = 1";
*
* Method Custom Finder:
*     public java.sql.PreparedStatement findGreaterThan(int threshold) throws
*     Exception;
*
*     An implementation of this method must be provided in a class that follows
*     these rules:
*     1.  The name of the class is either <beanClassName>FinderObject or the
*         name of the class is specified in the environment properties of the
*         enterprise bean. The name of the property must be
*         CustomFinderClassName. The value of the property must be the fully
*         qualified class name (including the package name).
*     2.  The class must be in the same package as the deployed code for the
*         bean.
*     3.  The class must extend com.ibm.vap.finders.VapEJSJDBCFinderObject and
*         must implement this finder helper interface.
* This implementation will be referenced when deployed code is generated.
```

To summarize, where clause finders can be used where the parameters to the finder method map to simple data types amenable to a static SQL query string. Custom method finders are able to handle complex queries and data types but require additional coding on the part of the developer. There are `VapEJSJDBCFinderObject` utility methods that make the task easier.

Remember that a finder instantiates the resulting entity beans, so they should not be used unless the actual beans will be needed. For instance, when attempting to populate a large drop down list from which only the actual selection will need to be populated, a session bean with direct JDBC calls would be much more efficient.

Note on exceptions: If a finder method returns a primary key and the associated EJB does not exist or cannot be retrieved, *javax.ejb.ObjectNotFoundException* should be thrown (a subclass of `FinderException`).

5.3.5 Entity context and session context

These extensions of the `EJBContext` interface allow an entity bean to obtain its primary key and query for a remote reference to itself. A session bean can also query for a remote reference to itself.

The `getCallerPrincipal` and `isCallerInRole` methods are useful if the bean manages its own security.

The `getRollbackOnly` method and the `setRollbackOnly` methods are used to check whether a roll back has occurred and to force a roll back, respectively.

The `getEJBObject` and `getPrimaryKey` methods are used to obtain a reference to the EJB object or the primary key of the instance, respectively. You may want to do this if you want to invoke a method on another bean and pass the current bean instance as a parameter. This is due to the fact that it is illegal to pass in "`this.myBean`".

The `getEJBHome` method is useful in obtaining the bean's home interface. This method might be used, for instance, in the event that a bean wanted to create more instances of itself.

EJBContext Method Summary

```
java.security.Principal getCallerPrincipal()
Obtain the java.security.Principal that identifies the caller.

EJBHome getEJBHome()
Obtain the enterprise bean's home interface.

boolean getRollbackOnly()
Test if the transaction has been marked for rollback only.

javax.transaction.UserTransaction getUserTransaction()
Obtain the transaction demarcation interface.
```

```
boolean isCallerInRole(java.lang.String roleName)
Test if the caller has a given security role.

void setRollbackOnly()
Mark the current transaction for rollback.
    For Entity Beans
EJBObject getEJBObject()
Obtain a reference to the EJB object that is currently associated with the
instance.

java.lang.Object getPrimaryKey()
Obtain the primary key of the EJB object that is currently associated with this
instance.
    For Session Beans
EJBObject getEJBObject()
Obtain a reference to the EJB object that is currently associated with the
instance.
```

5.3.6 Working with EJB metadata

Working with EJB metadata allows the developer to obtain the bean's home interface type, determine the class for its home and remote interface, determine the class of its primary key (entity beans), and query whether the bean is a session bean or an entity bean. This class is typically generated by container tools.

The `isSession` method and the `isStatelessSession` method can be used to determine the type of bean.

The `getHomeInterfaceClass` method, `getPrimaryKeyClass` method and `getRemoteInterfaceClass` method are used to obtain the class type of the home class, key class and remote class, respectively.

EJBMetaData Method Summary

```
EJBHome getEJBHome()
Obtain the home interface of the enterprise Bean.

java.lang.Class getHomeInterfaceClass()
Obtain the Class object for the enterprise Bean's home interface.

java.lang.Class getPrimaryKeyClass()
Obtain the Class object for the enterprise Bean's primary key class.

java.lang.Class getRemoteInterfaceClass()
Obtain the Class object for the enterprise Bean's remote interface.
```

```
boolean isSession()
Test if the enterprise Bean's type is "session".

boolean isStatelessSession()
Test if the enterprise Bean's type is "stateless session".
```

5.3.7 Handling exceptions

Application exceptions are not mapped to another exception by the container. System exceptions may cause the container to destroy the instance, so they cannot be handled by the client. All application exceptions should be handled by the client.

The exceptions in the `javax.ejb` package are:

- ► `CreateException`
    - – `DuplicateKeyException`
- ► `FinderException`
    - – `ObjectNotFoundException`
- ► `RemoveException`
- ► `EJBException`

`CreateException` is thrown by `ejbCreate` if an input parameter is invalid.

`ObjectNotFoundException` is thrown by `ejbFindByPrimaryKey` and other finder methods.

`RemoveException` is thrown by `ejbRemove` if the entity bean's row cannot be deleted from the datastore.

`EJBException` is thrown for system problems.

In addition, `NoSuchEntityException` can be thrown by `ejbLoad` and `ejbStore`.

5.3.8 Message driven beans

EJB 2.0 provides for a new kind of EJB: the message driven bean or MDB. It is essentially a message listener that can reliably consume messages from a queue or subscription. The MDB contains an `onMessage()` method that is part of the `javax.jms.messageListener` interface.

```
void onMessage(javax.jms.Message aMessage);
```

The EJB container automatically performs these tasks:

- ► Creates a message consumer to receive the messages
  - – Associates an MDB with a connection factory and destination at deployment
- ► Specifies a message acknowledgement mode
- ► Registers the message listener

The MDB has no home or remote interface, only an implementation class; in addition, it must implement the `MessageDrivenBean` and `MessageListener` interfaces. Furthermore, it can use container-managed or bean-managed transactions, but container managed is the recommended approach.

5.3.9 Programming for portability

The EJB1.1 specification details a list of programming restrictions that include:

- ► Rules against using sockets
- ► Restrictions on creating class loaders
- ► Writable static fields on EJB bean classes

In addition, to enable RMI over IIOP for exporting enterprise beans, the `javax.rmi.PortableRemoteObject.narrow()` should be used to cast all home and remote object references.

5.4 Database connections with EJB

The EJB container manages a pool of database connections. A single connection may be used by multiple beans. In this way, the connection can be created once by the application server and later reused without dropping the connection, thereby improving overall performance. The particular container implementation strategy employed by a specific vendor may dictate whether CMP or BMP will be used by the application developer, since complex table joins and arbitrary relationships, among other advanced features of database connectivity, are normally not handled well, if at all, by CMP. In other words, use CMP if it meets your needs.

The JNDI ENC provides support for handling standard resources like JDBC and JMS. For stateless session beans, non-standard resource connections can be opened in the `setSessionContext()` method and closed in the `ejbRemove()` method. For entity beans, resource connections can be opened in the `setEntityContext()` method and closed in the `unsetEntityContext()` method.

5.4.1 Mapping EJBs to the database

For CMP, a tool is normally provided that will generate a default mapping from an EJB to tables and rows in the datastore (as well as the reverse direction). A specific tool may allow some alteration of the default mapping before deployment of the bean to the container, depending on the application server used. If more flexibility is needed, as in the case of complex table relationships or table joins, BMP allows arbitrary relationships to be constructed by the developer.

The persistence type of the bean determines whether the connection code must be hand-coded or not. For entity beans with CMP, the connection methods are generated. For entity beans with BMP and for sessions beans that access a database (stateful), the connection method must be provided by the developer.

There are three common approaches to generating the required mapping for a CMP entity bean:

► Top-down

– Generate the table from the EJB

► Bottom-up

– Generate the EJB from the table

► Combination "meet-in-the-middle"

– Take an existing EJB and an existing table and map them to one another

EJB inheritance
Selecting one of these mapping approaches has implications when implementing inheritance with EJBs. There are two types of generalization relationships:

► Implementation inheritance (extends in Java)

► Interface inheritance (implements in Java) or "realization"

In the context of EJBs, the inheritance is supported for:

► Interfaces

– A component may derive home and remote interfaces from a parent home or remote interface.

– A home interface cannot currently realize a parent interface directly, since the EJB spec requires a `findByPrimaryKey` method be implemented that returns an instance of the class. To get around this, the root EJB class in the home interface would implement finders that formulated queries over its tables and tables of its subclasses.

► Bean implementation classes

– Standard Java class inheritance can be used.

– Cannot mix stateless and stateful session beans.

Now, database inheritance can be one of:

► Single-table inheritance (can produce large tables), shown in Figure 5-7.

► Root-leaf inheritance (joins can be time-intensive), shown in Figure 5-8.

► Multiple-table inheritance (implies loss of normal form).

key	discriminator	name	office	yearlySalary
101	EmployeeBean	Fred	B105	null
202	SalariedEmplyee	Bob	C238	40000

EmployeeBean

key
name
office

SalariedEmployeeBean

yearlySalary

The 'discriminator' column identifies the type of the bean
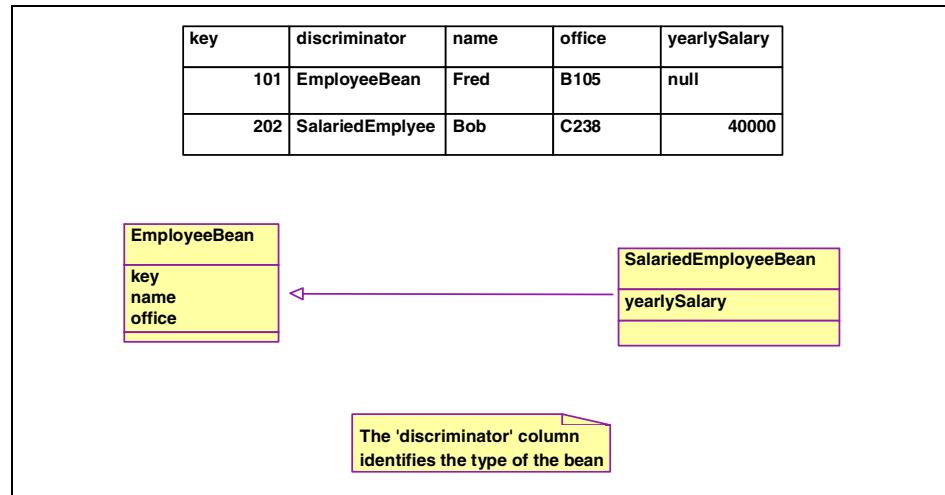
Figure 5-7   Single table inheritance

key	discriminator	name	office
101	EmployeeBean	Fred	B105
202	SalariedEmplyee	Bob	C238

key	yearlySalary
202	40000

EmployeeBean

key
name
office

SalariedEmployeeBean

yearlySalary

The 'discriminator' column identifies the type of the bean
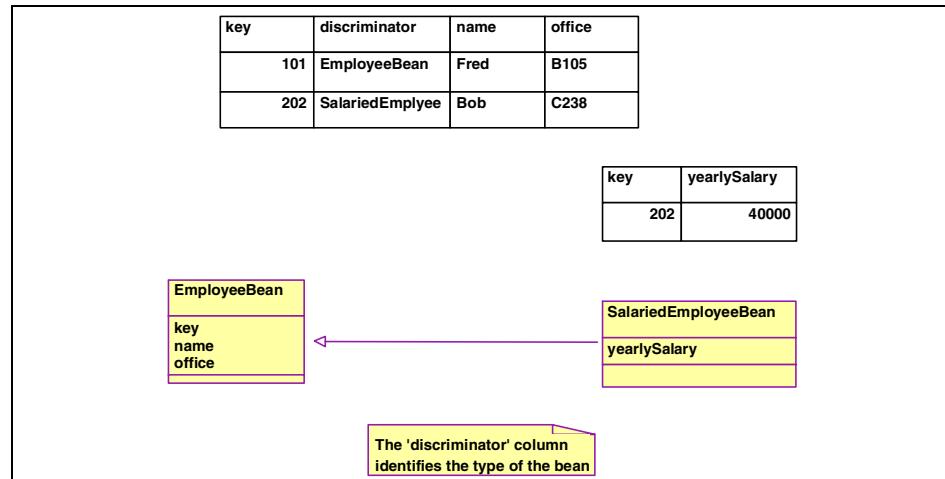
Figure 5-8   Root leaf inheritance

To summarize the possible combinations, when using the meet-in-the-middle mapping approach, all three types of database inheritance are supported. WebSphere supports a single table model for top-down mapping and does not support inheritance mapping for a bottom-up approach.

The meet-in-the-middle, or combination approach, is the most flexible and is frequently used as it is quite often the case that an existing database is used in combination with a newly developed object model.

To create the database schema using the VisualAge Schema Browser we select **Import/Export Schema** and **Import Schema from Database** from the Schemas menu item (Figure 5-9).

In our example, the name given to the schema is **ITSO**. In the Database Connection Info window, we specify the connection type and data source. We use the "com.ibm.db2.jdbc.app.DB2Driver" driver and specify the data source as "jdbc:db2:EJBBANK". Also specify the database user ID and password in this window.

In the Select Tables window we select the **ITSO** qualifier. After pressing the **Build Table List** button we select the **ACCOUNT** table from the table list.

Finally, we select **Save Schema** and provide a package name of "com.ibm.itso.j2eebook.schema".
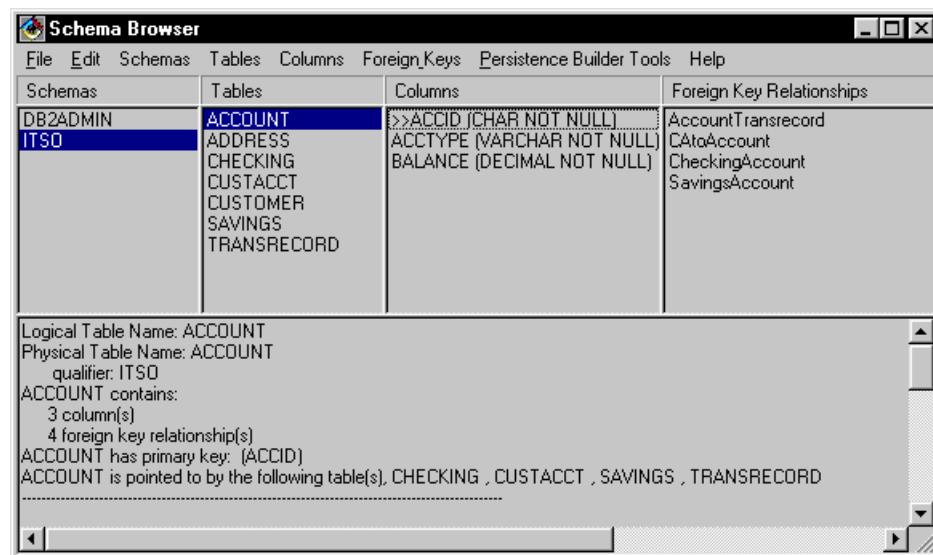
Figure 5-9   VisualAge Schema Browser

To create the schema mapping we use the VisualAge Map Browser (Figure 5-10).

We select **New EJB Group Map** supplying the requested information in the New Datastore Map window.

Next, select the map and the ACCOUNT persistent class. Then select **New Table Map** and **Add Map with No Inheritance** from the Table Maps menu item.

Now, in the Table Maps pane, selecting **Edit Property Maps** from the popup menu, lets us assign **Table Columns** in the database to **Class Attributes** of our EJB.

Complex map types may require a *secondary table map* as in the case of table columns which do not map to simple types. For example, if `address` is stored as a `BLOB` in the database but maps to an `Address` class with individual `String` fields in our object model, we would first need to define a *composer* in order to map the complex class attribute `address` to multiple table columns (Figure 5-11). For more information on composers, refer to the VisualAge documentation.

To summarize the required steps, in the primary table map, the complex map type is left unspecified (that is, `<Not mapped>`). In the secondary table map, the *Address* table is defined to have a foreign key relationship for *CustomerAddress.* Furthermore, the `address` attribute is specified as a *Complex* Map Type (Figure 5-11). Selecting the square button in the Table Column field opens the Complex Attribute Editor where the newly defined *AddressComposer* class can be specified to define the complex mapping between `Composer Attributes` and `Table Columns` in the database.
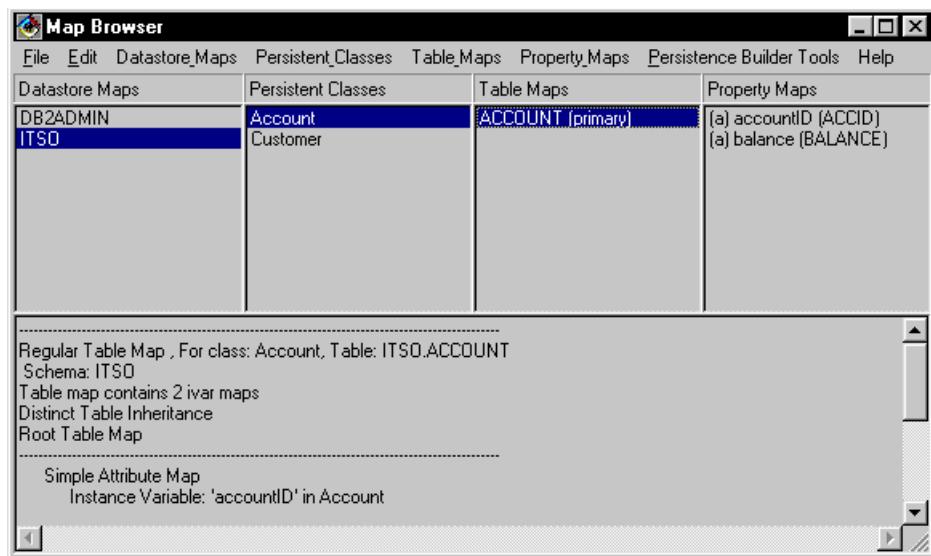
Figure 5-10 VisualAge Map Browser

AddressComposer class

```
public class AddressComposer extends com.ibm.vap.composers.VapAttributeComposer
{
    private static AddressComposer singleton;

    public static String[] getAttributeNames() {
        String[] attributes = {"street","city","state","zipcode"};
        return attributes;
    }
    public static String[] getSourceDatatype() {
        String[] types = {"String","String","String","String"};
        return types;
    }
    public static String getTargetClassName() {
        return CustomerAddress.class.getName();
    }
    public static void reset() { singleton = null; }
    public static AddressComposer singleton() {
      if (singleton == null)
        singleton = new AddressComposer();
      return singleton;
    }
    public Object[] dataFrom (Object anObject) {
        Object[] anArray = new Object[] {null, null, null, null};
        if (anObject != null) {
```

```
            CustomerAddress address = (CustomerAddress) anObject;
            anArray[0] = address.getFieldStreet();
            anArray[1] = address.getFieldCity();
            anArray[2] = address.getFieldState();
            anArray[3] = address.getFieldZip();
        }
        return anArray;
    }
    public Object objectFrom (Object[] anArray) {
        String name, street, city, state, zipcode;
        street  = (String) anArray[0];
        city    = (String) anArray[1];
        state   = (String) anArray[2];
        zipcode = (String) anArray[3];
        return new CustomerAddress( street, city, state, zipcode);
    }
}
```


Figure 5-11   Secondary table map for address BLOB (complex map type)

Typical relationships which can be mapped are (Figure 5-12):

- ► One-to-one
- ► One-to-many
- ► Many-to-many
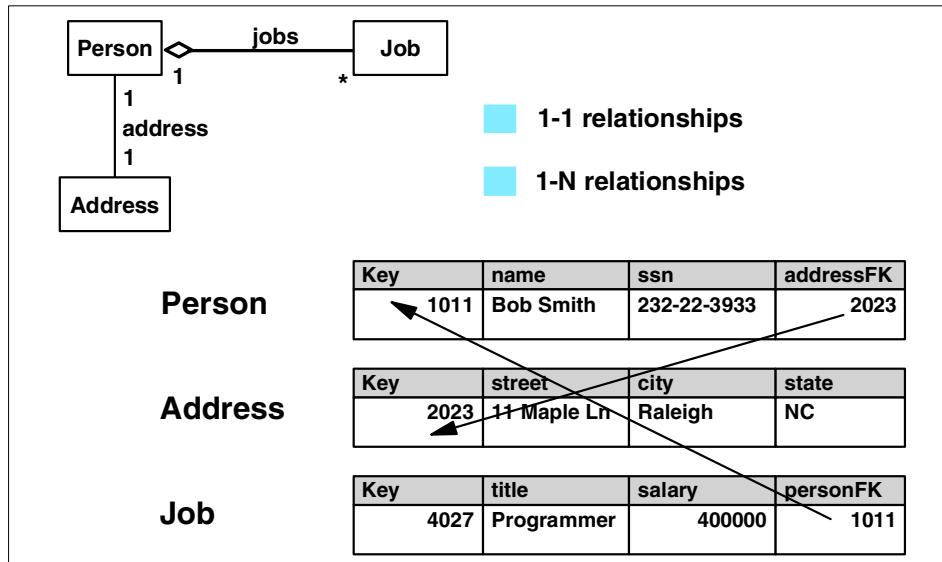
Figure 5-12   Relationships

Support for *associations* in VisualAge is limited to container-managed entities. This suffices in many cases. For instance, if an `Account` CMP entity has a one-to-many relationship with a `TransactionRecord` CMP entity, then we would get the `TransactionRecordHome` and create the TransactionRecord(s) associated with this transaction in the appropriate business method (that is, `deposit` or `withdraw`). VisualAge generates many of the methods to maintain the relationship between the two objects that would otherwise have to be written by hand, such as `getTransactions` and `addTransactions(TransactionRecord)` on the `Account` object as well as `getOwningAccountKey` and `setOwningAccount(Account)` on the `TransactionRecord` object.

The DDL file used for the code fragments in this chapter is purposefully simplistic. A more realistic DDL is provided with the `PiggyBank` scenario integrating servlets and JSP with the `Account`, `Customer` and `BankTransfer` EJBs.

The DDL file used for the code fragments

```
CONNECT TO PIGGY USER db2admin USING db2admin;

CREATE TABLE DB2ADMIN.CUSTOMER
(
 CUST_ID INTEGER NOT NULL,
 CUST_LN VARCHAR(30) NOT NULL,
 CUST_FN VARCHAR(30) NOT NULL,
 CUST_ADDR VARCHAR(50) NOT NULL,
```

```
  CUST_CITY VARCHAR(30) NOT NULL,
  CUST_ST   VARCHAR(5) NOT NULL,
  CUST_CTRY VARCHAR(30) NOT NULL,
  PRIMARY KEY (CUST_ID)
);

CREATE TABLE DB2ADMIN.ACCOUNT
 (
  ACCT_NUMBER INTEGER NOT NULL,
  CUST_ID     INTEGER NOT NULL,
  BALANCE     DECIMAL(10,2),
  CONSTRAINT  FK_CUST_ID FOREIGN KEY (CUST_ID)
              REFERENCES DB2ADMIN.CUSTOMER(CUST_ID) ON DELETE CASCADE
 );

 alter table DB2ADMIN.account add primary key (acct_number);

insert into DB2ADMIN.CUSTOMER
values(1,'LastName','FirstName','myStreet','myCity','TX','US');
insert into DB2ADMIN.CUSTOMER
values(2,'Mouse','Mickey','DisneyLand','Orlando','FL','US');
insert into DB2ADMIN.CUSTOMER
values(3,'Mouse','Minnie','DisneyLand','Orlando','FL','US');
insert into DB2ADMIN.CUSTOMER values(4,'Duck','Daffy','DisneyWorld','Los
Angeles','CA','US');
insert into DB2ADMIN.CUSTOMER values(5,'Dog','Pluto','DisneyWorld','Los
Angeles','CA','US');

insert into DB2ADMIN.ACCOUNT values(1,1,100.00);
insert into DB2ADMIN.ACCOUNT values(2,1,200.00);
insert into DB2ADMIN.ACCOUNT values(3,2,300);
insert into DB2ADMIN.ACCOUNT values(4,3,400);
insert into DB2ADMIN.ACCOUNT values(5,4,500);
insert into DB2ADMIN.ACCOUNT values(6,5,600);
insert into DB2ADMIN.ACCOUNT values(7,5,700);

CONNECT RESET;
```

Here is the more realistic DDL which we use for the integrated PiggyBank scenario. Just execute "db2 -fejbbank.ddl"

The DDL file used for the integrated PiggyBank scenario (ejbbank.ddl)

```
echo --- create the EJBBANK database ---
CREATE DATABASE EJBBANK

echo --- connect to EJBBANK database ---
CONNECT TO EJBBANK
```

```
echo --- drop tables ---
DROP TABLE ITSO.TRANSRECORD
DROP TABLE ITSO.CUSTACCT
DROP TABLE ITSO.ADDRESS
DROP TABLE ITSO.CHECKING
DROP TABLE ITSO.SAVINGS
DROP TABLE ITSO.ACCOUNT
DROP TABLE ITSO.CUSTOMER

echo --- create tables ---
CREATE TABLE ITSO.CUSTOMER (                                   \
   customerID  INTEGER    NOT NULL,                            \
   title       CHAR(3)    NOT NULL,                            \
   firstName   VARCHAR(30) NOT NULL,                           \
   lastName    VARCHAR(30) NOT NULL,                           \
   userID      CHAR(8),                                        \
   password    CHAR(8),                                        \
   address     BLOB(2000),                                     \
                       PRIMARY KEY (CUSTOMERID)     \
)
CREATE TABLE ITSO.ADDRESS (                                    \
   customerID  INTEGER    NOT NULL,                            \
   street      CHAR(20),                                       \
   city        CHAR(12),                                       \
   state       CHAR(12),                                       \
   zipcode     CHAR(10),                                       \
                       PRIMARY KEY (CUSTOMERID)     \
)
CREATE TABLE ITSO.CUSTACCT (                                   \
   customerID  INTEGER    NOT NULL,                            \
   accID       CHAR(8)    NOT NULL,                            \
                       PRIMARY KEY (CUSTOMERID,ACCID) \
)
CREATE TABLE ITSO.ACCOUNT (                                    \
   accid       CHAR(8)    NOT NULL,                            \
   balance     DEC(8,2)   NOT NULL,                            \
   acctype     VARCHAR(8) NOT NULL DEFAULT 'CHECKING',   \
                       PRIMARY KEY (ACCID)          \
)
CREATE TABLE ITSO.CHECKING (                                   \
   accid       CHAR(8)    NOT NULL,                            \
   overdraft   DEC(8,2)   NOT NULL DEFAULT 200.00,       \
                       PRIMARY KEY (ACCID)          \
)
CREATE TABLE ITSO.SAVINGS (                                    \
   accid       CHAR(8)    NOT NULL,                            \
   minamount   DEC(8,2)   NOT NULL DEFAULT 100.00,       \
                       PRIMARY KEY (ACCID)          \
)
```

```
CREATE TABLE ITSO.TRANSRECORD (                            \
  transid    TIMESTAMP  NOT NULL,                          \
  accid      CHAR(8)    NOT NULL,                          \
  transtype  CHAR(1)    NOT NULL,                          \
  transamt   DEC(8,2)   NOT NULL,                          \
                        PRIMARY KEY (TRANSID)              \
)

echo --- referential integrity ---
ALTER TABLE ITSO.TRANSRECORD                               \
   ADD CONSTRAINT "AccountTransrecord" FOREIGN KEY (ACCID) \
   REFERENCES ITSO.ACCOUNT ON DELETE RESTRICT

ALTER TABLE ITSO.CUSTACCT                                  \
   ADD CONSTRAINT "CAtoCustomer"  FOREIGN KEY (CUSTOMERID) \
   REFERENCES ITSO.CUSTOMER ON DELETE RESTRICT

ALTER TABLE ITSO.CUSTACCT                                  \
   ADD CONSTRAINT "CAtoAccount"  FOREIGN KEY (ACCID)       \
   REFERENCES ITSO.ACCOUNT ON DELETE RESTRICT

ALTER TABLE ITSO.ADDRESS                                   \
   ADD CONSTRAINT "CustAddress"  FOREIGN KEY (CUSTOMERID)  \
   REFERENCES ITSO.CUSTOMER ON DELETE RESTRICT

ALTER TABLE ITSO.CHECKING                                  \
   ADD CONSTRAINT "CheckingAccount"  FOREIGN KEY (ACCID)   \
   REFERENCES ITSO.ACCOUNT ON DELETE RESTRICT

ALTER TABLE ITSO.SAVINGS                                   \
   ADD CONSTRAINT "SavingsAccount"  FOREIGN KEY (ACCID)    \
   REFERENCES ITSO.ACCOUNT ON DELETE RESTRICT

echo --- execute GRANT statements ---
GRANT CONNECT ON DATABASE         TO PUBLIC
GRANT ALL     ON ITSO.CUSTOMER    TO PUBLIC
GRANT ALL     ON ITSO.ACCOUNT     TO PUBLIC
GRANT ALL     ON ITSO.CHECKING    TO PUBLIC
GRANT ALL     ON ITSO.SAVINGS     TO PUBLIC
GRANT ALL     ON ITSO.TRANSRECORD TO PUBLIC
GRANT ALL     ON ITSO.CUSTACCT    TO PUBLIC
GRANT ALL     ON ITSO.ADDRESS     TO PUBLIC

echo --- connect reset ---
CONNECT RESET
```

Here is the SQL file used to populate the tables for PiggyBank. Just execute
"db2 -fejbbank.sql"

Populating the PiggyBank database (ejbbank.sql)

```
echo --- load the EJBBANK database ---

echo --- connect to EJBBANK database ---
CONNECT TO EJBBANK

DELETE FROM ITSO.TRANSRECORD
DELETE FROM ITSO.SAVINGS
DELETE FROM ITSO.CHECKING
DELETE FROM ITSO.CUSTACCT
DELETE FROM ITSO.ACCOUNT
DELETE FROM ITSO.CUSTOMER

echo --- insert into CUSTOMER tables ---
INSERT INTO ITSO.CUSTOMER                                           \
   (customerid, title, firstname, lastname, userid, password) VALUES  \
   (101, 'Mr', 'Dom',     'Faidherbe',   'cust101', 'DF'),          \
   (102, 'Mr', 'Akis',    'Laftsidis',   'cust102', 'AL'),          \
   (103, 'Mr', 'Kart',    'Ponnalagu',   'cust103', 'KP'),          \
   (105, 'Ms', 'Unknown', 'Lady',         null,      null),          \
   (106, 'Mr', 'Ueli',    'Wahli',       'cust106', 'UW')

INSERT INTO ITSO.ADDRESS                           \
   (customerid, street, city, state, zipcode) VALUES \
   (106, 'Steinway Ave', 'Campbell', 'California', '95008')
INSERT INTO ITSO.ADDRESS (customerid) VALUES (101),(102),(103),(105)

echo --- insert into ACCOUNT tables ---
INSERT INTO ITSO.ACCOUNT                  \
   (accid, acctype, balance) VALUES        \
   ('101-1001', 'CHECKING',    80.00),     \
   ('101-1002', 'SAVINGS',    375.26),     \
   ('102-2001', 'SAVINGS',   9375.26),     \
   ('102-2002', 'CHECKING',    75.50),     \
   ('103-3001', 'SAVINGS',    100.00),     \
   ('105-5001', 'CHECKING',     0.00),     \
   ('106-6001', 'CHECKING',  1000.00),     \
   ('106-6002', 'SAVINGS',   2000.00),     \
   ('106-6003', 'SAVINGS',   3000.00)

INSERT INTO ITSO.CHECKING                   \
   (accid, overdraft) VALUES                \
   ('101-1001',200.00), ('102-2002',200.00), \
   ('105-5001',200.00), ('106-6001',300.00)

INSERT INTO ITSO.SAVINGS                                          \
   (accid, minamount) VALUES                                       \
   ('101-1002',100.00), ('102-2001',100.00), ('103-3001',150.00), \
```

```
                ('106-6002',100.00), ('106-6003',250.00)

        echo --- insert into CUSTACCT table ---
        INSERT INTO ITSO.CUSTACCT                   \
           (customerid, accid) VALUES               \
           (101,'101-1001'), (101,'101-1002'),   \
           (102,'102-2001'), (102,'102-2002'),   \
           (103,'103-3001'), (105,'105-5001'),   \
           (106,'106-6001'), (106,'106-6002'), (106,'106-6003'), (106,'105-5001')

        echo --- insert into TRANSRECORD table ---
        INSERT INTO ITSO.TRANSRECORD                                       \
           (transid,          accid,      transtype, transamt) VALUES  \
           (CURRENT TIMESTAMP, '101-1001', 'C',          80.00 )
        INSERT INTO ITSO.TRANSRECORD                                       \
           (transid,          accid,      transtype, transamt) VALUES  \
           (CURRENT TIMESTAMP, '101-1002', 'D',         200.00 )
        INSERT INTO ITSO.TRANSRECORD                                       \
           (transid,          accid,      transtype, transamt) VALUES  \
           (CURRENT TIMESTAMP, '106-6001', 'D',          66.66 )
        INSERT INTO ITSO.TRANSRECORD                                       \
           (transid,          accid,      transtype, transamt) VALUES  \
           (CURRENT TIMESTAMP, '106-6002', 'C',          66.66 )

        echo --- connect reset ---
        CONNECT RESET
```

5.4.2 When to use EJBs

Clearly, not everything should be an EJB. Remember that entity EJBs are
distributed, transactional and persistent. If an application contains a set of
read-only objects (very rarely updated, if at all), then an all EJB solution does not
make sense. The state of an entity EJB is typically read once per transaction (the
default with WebSphere) regardless of whether the object has been updated.
Therefore, a simple stateless session bean with a set of dependent objects that
is read only once at program startup is a more efficient approach. The dependent
objects can be simple JavaBeans and are kept in memory during the life cycle of
the stateless session bean. Remember, though, that stateless session beans are
shared across multiple clients. So, any information kept in instance variables in
the bean should not be accessible to the client. The stateless session bean could
maintain a reference to the dependent objects (singletons), or else the
dependent objects may contain a static class variable that contains the single
instance of its associated class.[4]

---

[4] Kyle Brown, et al., *Enterprise Java Programming with IBM WebSphere*, Addison Wesley Professional, 2001

A strong case for EJBs can be made if more than one of the following considerations enter into the application use cases:

► Need to support multiple client types
► Need for concurrent read and update access to shared data
► Need to access of multiple disparate data sources
► Method-level object security
► Standards-based architecture
► Portable component-based architecture
► Multiple servers to handle throughput requirements
► Availability and scalability requirements
► Security requirements

Using entity EJBs simply to map the data model to the object model used by the underlying business logic is not the best use of EJBs. Minimally, a session EJB could be used with the associated entities to minimize network traffic. In this way, we avoid multiple fine-grained client requests and the associated client-initiated transactions. Also, if the servlets are not physically located on the same machine as the Web server there will be additional network overhead. It has been demonstrated that use of the IIOP redirector necessarily entails a performance penalty. Architectural requirements may dictate whether the presentation logic (servlets and JSP) is a separate layer from the business logic (EJBs). The EJBs used to implement the business logic may be kept behind a separate firewall in this case and use of the redirector becomes necessary.

Our PiggyBank scenario writes transaction records to the database for every deposit into or withdrawal from an account. This is basically a logging requirement; we write records to the database frequently but read the records very infrequently. In the case of a true event log where the records are essentially write-only, a stateless session bean would make sense to use in combination with a broker to write the individual records out using direct JDBC calls.

In our case, a simple CMP entity bean suffices for writing the transaction records to datastore (we may be concerned with write contention). Furthermore, since an account maintains a 1::m relationship with transaction records belonging to that account, the `TransactionRecord` entity is dependent on the `Account` entity and is accessed directly only by the Account object. An access bean for the TransactionRecord entity is, therefore, not provided.

In the `Account` object, we create a transaction record every time a customer deposits or withdraws funds from one of their accounts. Because the `Account` object is an EJB and uses container managed transactions, if the transaction fails, it will be automatically rolled back, and the newly created transaction record will not actually be written to datastore.

Creating Transaction Records from the AccountBean

```
public void withdraw(java.math.BigDecimal amount)
throws InsufficientFundsException {
   if (balance.compareTo(amount) == -1) {
      throw new InsufficientFundsException("Need more allowance");
   } else {
      balance = balance.subtract(amount);
   }
   try {
      getTransactionRecordHome().create(accountID, amount, "W");
   } catch (Exception e) {
      System.out.println("==> TxnRecord could not be created!");
   }
}

protected TransactionRecordHome getTransactionRecordHome()
   throws NamingException {
   TransactionRecordHome txnRecordHomeCMP =
       (TransactionRecordHome) javax.rmi.PortableRemoteObject.narrow(
           HomeHelper.singleton().getHome("piggy.TransactionRecord"),
           TransactionRecordHome.class);
   return txnRecordHomeCMP;
}
```

An application that needs to process the records associated with a particular
account might choose to use direct JDBC calls to retrieve those records rather
than EJBs (may be a different application from our PiggyBank application,
entirely). Client-initiated transactions could be used to synchronize access with
any updates in progress from within the same application or other applications.

We implement the required functionality with direct JDBC calls in our servlet
using the helper classes in the package com.ibm.itso.j2eebook.dataaccess
and com.ibm.itso.j2eebook.helpers.

Data access helpers (com.ibm.itso.j2eebook.dataaccess)

```
public class TransactionListHelper {

// SQL statement for retrieving transactions
public static final String ACCTTXNS =
"SELECT * from ITSO.TRANSRECORD WHERE ITSO.TRANSRECORD.accid=? ORDER BY transid
DESC";
}

public class AccountListHelper {

// SQL to retrieve account ids for all accounts for the given customerID
// uses a Table Join
public static final String ACCTLIST = "SELECT ITSO.ACCOUNT.accid from
ITSO.ACCOUNT, ITSO.CUSTACCT " +
```

```
"WHERE ITSO.CUSTACCT.customerid=? and ITSO.ACCOUNT.accid=ITSO.CUSTACCT.accID;";
}
```

Our client (in this case a servlet) creates an instance of a
`TransactionListHelper` and a static instance of a `DatabaseHelper` and invokes
helper routines to retrieve and process a multi-row resultset from the database. It
is good practice to process the result set and return a standard Java type instead
of using a `ResultSet` as the return type. This supports the RMI over IIOP protocol
and maintains the separation of our persistence layer from our middle-tier
business logic.

```
// in our servlet, we retrieve the list of transactions
TransactionListHelper tlh = new TransactionListHelper();
Vector txns = tlh.getTransactionList(accountId);

// in getTransactionList we obtain a database connection
Connection conn = null;
conn = DatabaseHelper
.singleton()
.getDataSource(NamingHelper.singleton().getJNDIName("piggy.ds"))
.getConnection();
```

Retrieving transaction records (com.ibm.itso.j2eebook.dataaccess)

```
public Vector getTransactionList(String accountID) throws
TransactionListException{
    // Define method instance variables
    Connection conn = null;
    PreparedStatement sqlStmt = null;
    boolean wasFound = false;
    Vector txns = new Vector();
    try {
        // use the DatabaseHelper to retrieve a cached DataSource
        conn = DatabaseHelper
                .singleton()
                .getDataSource(NamingHelper.singleton().getJNDIName("piggy.ds"))
                .getConnection();

        sqlStmt = conn.prepareStatement(ACCTTXNS);
        sqlStmt.setString(1, accountID);
        ResultSet results = sqlStmt.executeQuery();

        // Process the result set to retrieve the last ten transactions, or
        // all transactions if there are less than 10
        int count = 0;
        while (count < 10 && results.next()) {
```

```
                // create a TransactionDataBean by making calls on the ResultSet
                 TransactionDataBean transactionDB =
                     new TransactionDataBean(
                         results.getString("accid"),
                         results.getBigDecimal("transamt"),
                         results.getTimestamp("transid"),
                         results.getString("transtype"));
                //add result to Vector
                txns.add(transactionDB);
                count++;
           }
        // check the size of the vector. If no transactions were returned
        // then we should tell the user.
        if(txns.size() <= 0)
            throw new TransactionListException("No transactions were found.");
     } catch (NamingException ne){
        // thrown during DataSource lookup
        throw new TransactionListException("DataSource lookup exception");
     } catch (SQLException sqle){
        // thrown during the query
        throw new TransactionListException("SQL query exception");
     } finally {
         cleanup(sqlStmt, conn);
     }
     //return Vector or HashTable
     txns.trimToSize();
     return txns;
}
```

Our `DatabaseHelper` class is shown as follows:

DatabaseHelper (com.ibm.itso.j2eebook.helpers)

```
import java.sql.*;
import javax.sql.*;
import javax.naming.*;
import java.util.*;
import com.ibm.itso.j2eebook.helpers.NamingHelper;
/**
 * Singleton helper for accessing DataSource objects.
 *
 * @author: Anthony Stevens
 */
public class DatabaseHelper {
    // Singleton class instance
    private static DatabaseHelper instance = null;
    // Hashtable to hold the DataSource objects
    private Hashtable dsTable = new Hashtable();
}
```

```
public static DatabaseHelper singleton() {
   if(instance == null){
      instance = new DatabaseHelper();
   }
   return instance;
}

public DataSource getDataSource(String dsName) throws NamingException {
    // See if we already have the DataSource
    DataSource ds = (DataSource) dsTable.get(dsName);
    if (ds == null) {
        ds = (DataSource) NamingHelper.singleton().getInitialContext().lookup(
           NamingHelper.singleton().getJNDIName(dsName));
        dsTable.put(dsName, ds);
    }
    return ds;
}
```

Therefore, we model our data as entity beans and our workflow as session beans, but not everything is an EJB. What is and what is not an EJB is determined by individual use cases, keeping performance considerations in mind.

One way to speed up performance of entity beans is to use WebSphere's EJB caching. There are two options:

► Reload the bean for every transaction (default)

► Do not reload the bean for each transaction

  – This option will work only if the EJB Server has exclusive access to the bean

In other words, keep in mind that accessing the EJB, by default, reloads the bean which can be an expensive operation.

5.5 Security concepts

WebSphere security is built on top of the Java 1.2 Security model (Figure 5-13). Secure Web applications require a comprehensive security model. This section describes the key concepts to enable the developer to assign *security roles* to enterprise beans in the deployment descriptor.
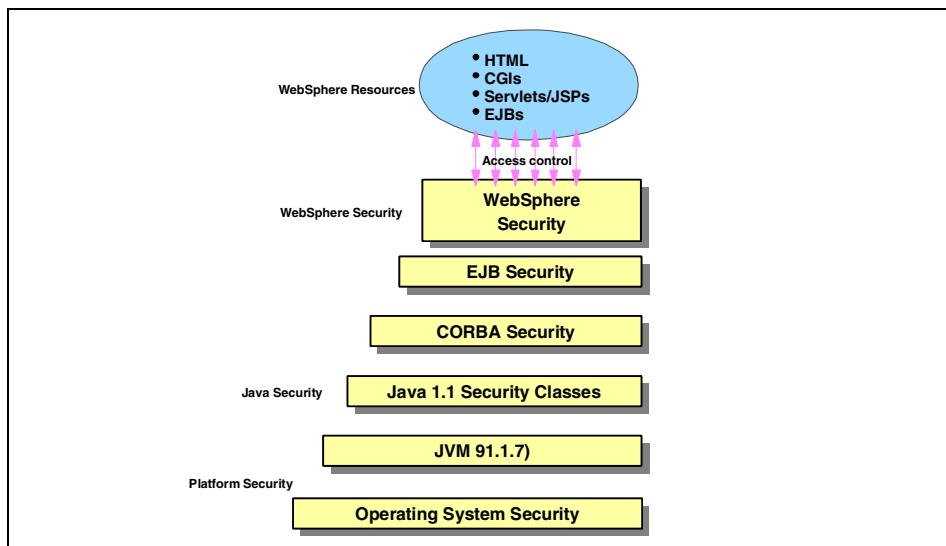
Figure 5-13   WebSphere security model

The EJB 1.1 specification defines how principals (users) can be assigned logical security roles. Security roles further define access permissions for users currently in those roles. For example, to determine the role of a caller of an enterprise bean:

```
boolean authorized = context.isCallerInRole("Administrator");
```

Security roles are declared in the deployment descriptor and require no additional programming on the part of the developer.

5.5.1 Authentication

A user authenticates by logging-on and proving his or her identity to the server. In other words, is the user really who he says he is? A user may belong to a *group* (a category of users) and reside in a *realm*. The J2EE authentication service normally verifies identity by checking the *default realm*. A J2EE user in the default realm may also belong to a J2EE group, however, users in the *certificate realm* may not. Web browser clients that use certificates with the HTTPS protocol for authentication belong in the certificate realm.

After a J2EE server authenticates a client by presenting a logon dialog and requesting the user name and password, a security context is associated with calls from that user to the J2EE server. Clients which do not log on are assigned the anonymous user `guest`. For protected Web components, three types of authentication can be specified: basic, form and certificate.

WebSphere supports standard authentication schemes:

- ► Basic HTTP authentication
- ► X509 client certificates
- ► Custom form based
- ► Digital certificates

The login authentication method is specified at deployment time.

WebSphere supports various repositories for user information (that is, userid and password):

- ► Local operating system (NT or UNIX or Solaris)
- ► LDAP repository (lightweight third party authentication)
- ► Custom registry (WebSphere 4.0)

5.5.2 Authorization

A server authorizes particular users to perform particular actions on an enterprise bean. The server will grant or deny permission to access methods on an enterprise bean. Authorization is defined in three steps: declaring a role, declaring method permissions and mapping roles to J2EE users and groups. Method permissions define which roles are allowed to invoke which methods.

The authorization model is changed from permission based to role based with J2EE1.2 (Figure 5-14). Security roles are defined to allow grouping of method permissions.
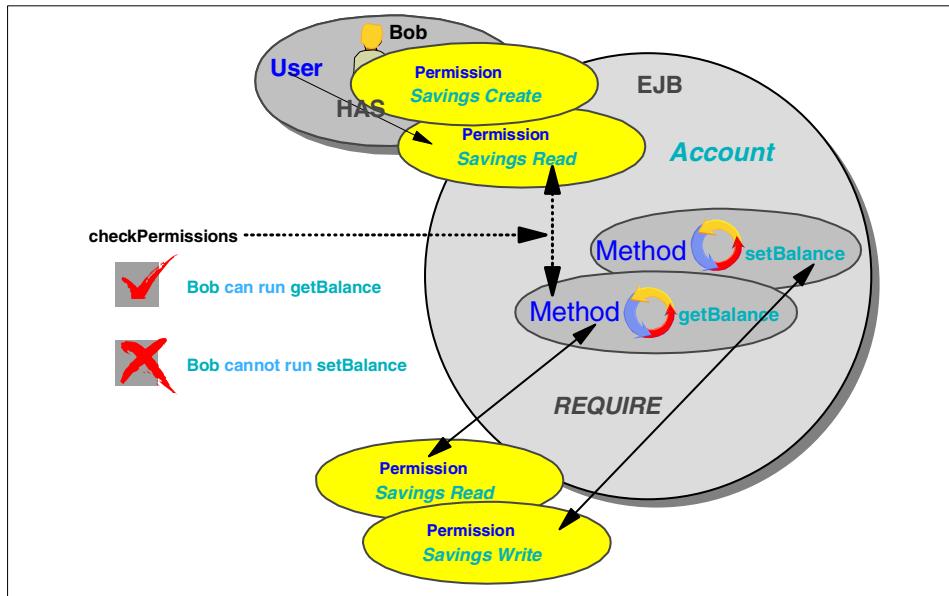
Figure 5-14   WebSphere Application Server 3.5 permission based security

5.5.3 WebSphere security

WebSphere allows you to specify a delegation mode by specifying a `RunAs` mode for each individual enterprise bean. `RunAs` modes can be one of:

► CLIENT IDENTITY

– The identity of the client

► SYSTEM IDENTITY

– The identity of the server

► SPECIFIED IDENTITY

– An identity explicitly identified in the delegation policy

Depending on the active delegation policy, the server invokes methods under a particular identity.

Note: In WebSphere Application Server 4.0, "`Specified Identity`" `RunAs` mode is replaced by "`Specified Role`".

Security server

The security server controls security policies and provides authentication and authorization services. The security server is typically separate and distinct from the application server.

Security plug-in

The security plug-in is a WebSphere runtime component that interacts with the security server. The security plug-in restricts user access to specific Web resources, such as HTML pages, servlets and JSP files; it protects the URL namespace. The security plug-in consults the security server when making authentication and authorization decisions and uses the security policy information acquired from the security application to determine which authentication and authorization services to invoke.

Security collaborator

The security collaborator is a WebSphere runtime component that interacts with the security server. For every remote method invocation of a servlet or EJB, the collaborator:

► Performs authorization checks
► Enforces the delegation policy
► Performs security trace logging

5.5.4 WebSphere 4.0 custom registry

In WebSphere 4.0, support is provided for a custom user registry (Figure 5-15). It works as follows:

► Application developer

  – Implements methods in the `CustomRegistry` interface

► Administrator

  – Selects custom registry option and specifies the class

► Admin/Security server

  – Calls custom registry methods to perform authentication
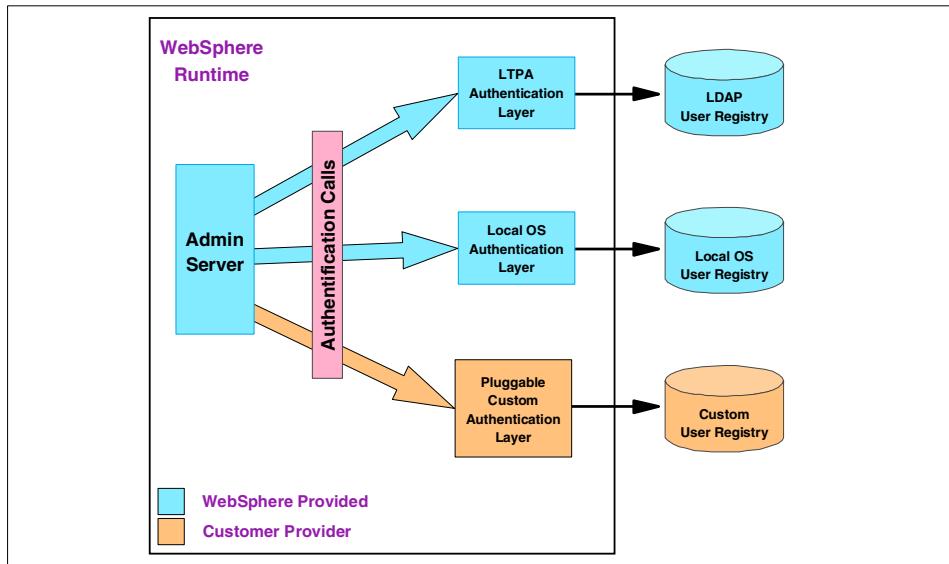
Figure 5-15   WebSphere 4.0 pluggable authentication registry

5.5.5 Bean managed security

The J2EE APIs may be used if the bean manages its own security requirements. Security can be specified at either the method-level or the EJB-level.

In order to determine the caller of an EJB, you invoke the `getCallerPrincipal` method. The `getName` method returns the name of the principal.

```
java.security.Principal principal =
ejbContext.getCallerPrincipal().getName();
```

To determine the caller's role, you invoke the `isCallerInRoleMethod(String roleName)`, where `roleName` is the name of the security role. The role must be one of the security roles defined in the deployment descriptor.

```
boolean isAdmin = ejbContext.isCallerInRole("Administrator");
```

For servlets, the corresponding calls are:

```
getUserPrincipal() and isUserInRole(String roleMame)
```

Figure 5-16 depicts the security flow for a Web client.
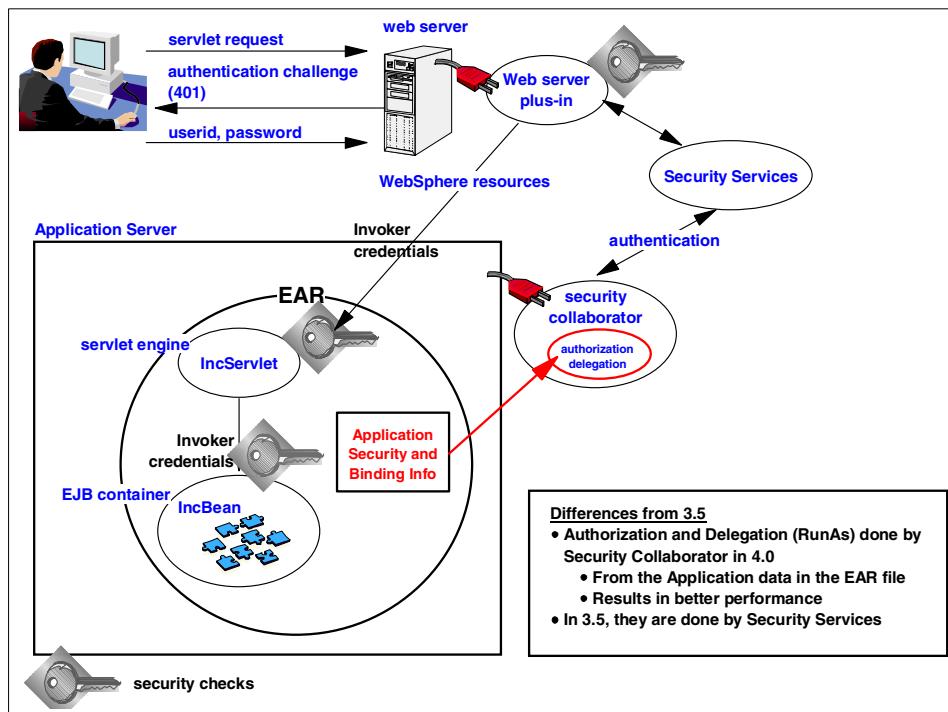
Figure 5-16   Security flow for a Web client

Figure 5-17 depicts the security flow for a Java client.
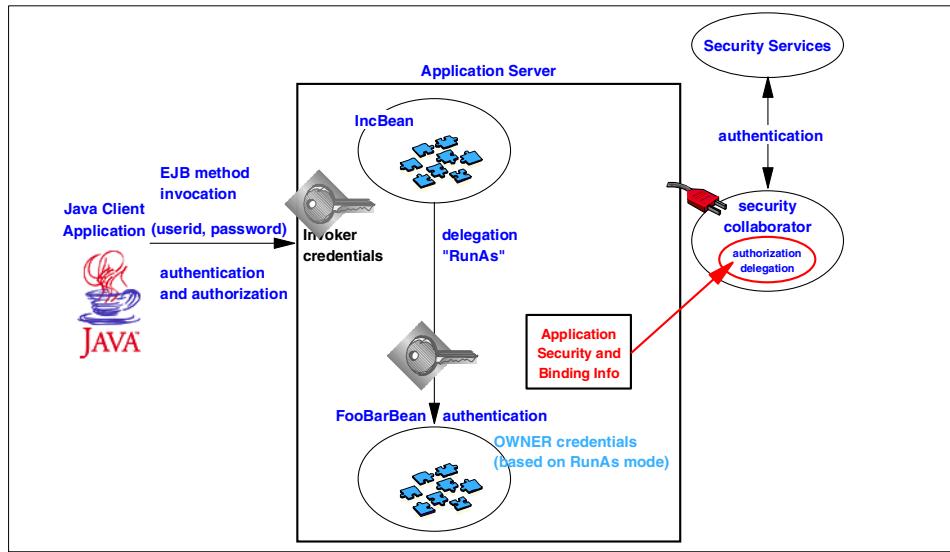
Figure 5-17   Security flow for a Java client

5.6 VA Java and EJBs

VisualAge for Java and Websphere Advanced are the IDE and application server used to build and deploy the EJB respectively (Figure 5-18). The EJBBank application scenario found in the IBM Redbook *EJB Development with VisualAge for Java for Websphere Application Server*, SG24-6144, illustrates the techniques involved in using the tools with WAS 3.x and will not be repeated here. Note that VisualAge generates a deployment descriptor, so this step does not have to be performed manually.
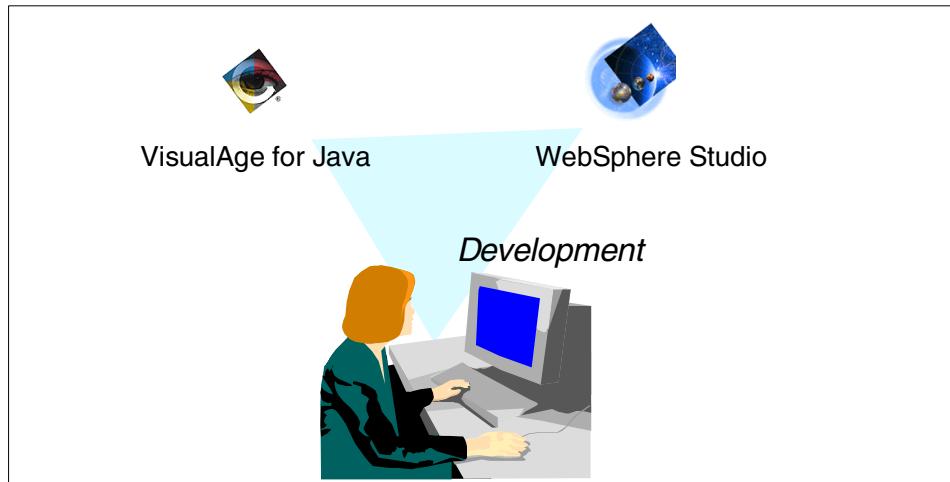
Figure 5-18   VisualAge for Java development and EJBs

The basic steps required to build and deploy EJBs using VisualAge for Java are as follows:

► Create the EJB

 – Use a session facade instead of an AccessBean wrapper, if scalability is desired

► Generate the EJB code

 – Generate AccessBeans, if desired

 – Note that AccessBeans should not be used in a clustered environment with many application servers

► Create the database Schema

► Map the database schema

 – Top down
 – Bottom up
 – Meet in the middle

► Test the EJB

 – Start the persistent name server

 – Start the EJB server

 – Start the servlet engine

 – Select "Run Test Client" from the popup menu on an EJB or EJBGroup

- ► Create the EJB client
  - – Create a servlet
- ► Deploy the EJBs
  - – VisualAge for Java 3.5.3 generates EJB 1.0 code

    (VisualAge for Java 4.0 generates EJB 1.1 JAR)
  - – Create a deployed JAR file
  - – Install the JAR file into the container
- ► Test the EJB client (outside of VAJ)

5.7 XML deployment descriptor

EJB 1.0 used serialized deployment descriptors which are not ideal for purposes of portability. Beginning with EJB 1.1, deployment descriptors are now created using XML. The body of the XML descriptor begins with the `<ejb-jar>` tag. Within the root element, key tags of interest are the `<enterprise-beans>` tag and the `<assembly-descriptor>` tag. The jar file may contain one or more enterprise beans.

The `<enterprise-beans>` tag describes the beans in its associated jar file. Each bean would have either a `<session>` element or an `<entity>` element within the single `<enterprise-beans>` tag. The beans described in the `<enterprise-beans>` tag share the characteristics described in the `<assembly-descriptor>` section.

The complete DTD, along with explanations and a sample, can be found in the EJB 1.1 specification. Here's a sample descriptor:

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC
"-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 1.1//EN"
"http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
<ejb-jar>
  <enterprise-beans>
    <entity>
      <description>
        This is an entity bean
      </description>
      <ejb-name>MyEntityBean</ejb-name>
      <home>MyBeanHome</home>
      <remote>MyBean</remote>
      <ejb-class>MyBean</ejb-class>
      <persistence-type>Container</persistence-type>
```

```
      <prim-key-class>Integer</prim-key-class>
      <reentrant>False</reentrant>
      <cmp-field><field-name>myAddress</field-name></cmp-field>
      <cmp-field><field-name>myName</field-name></cmp-field>
    </entity>
  </enterprise-beans>

  <assembly-descriptor>

    <security-role>
      <description>
        this role represents...
      </description>
      <role-name>everyone</role-name>
    </security-role>

    <method-permission>
      <role-name>everyone</role-name>
      <method>
        <ejb-name>MyBean</ejb-name>
        <method-name>*</method-name>
      </method>
    </method-permission>

    <container-transaction>
      <description>
        all methods require a transaction
      </description>
      <method>
        <ejb-name>MyBean</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>

  </assembly-descriptor>
</ejb-jar>
```

Session and entity beans

The <cmp-field> element can be specified for entity beans with CMP. The required fields include the <ejb-name>, <ejb-class>, <home> and <remote> elements which are placed within the <entity> or <session> tag for entity beans and session beans, respectively.

```
<entity>
  <description>
  This is an entity bean
  </description>
  <ejb-name>MyEntityBean</ejb-name>
  <home>MyBeanHome</home>
  <remote>MyBean</remote>
  <ejb-class>MyBean</ejb-class>
  <persistence-type>Container</persistence-type>
  <prim-key-class>Integer</prim-key-class>
  <reentrant>False</reentrant>
  <cmp-field><field-name>myAddress</field-name></cmp-field>
  <cmp-field><field-name>myName</field-name></cmp-field>
</entity>
```

For an entity bean, the complete set of *required* fields are:

► `<ejb-name>`

► `<home>`

► `<remote>`

► `<ejb-class>` fully qualified class name

► `<prim-key-class>` specifies the fully qualified class name of the primary key for the associated entity bean

► `<persistence-type>` the two allowable values are `Container` for container managed persistence and `Bean` for bean managed persistence

► `<reentrant>` the two allowable values are `True` and `False`. False means an exception will be thrown if the bean calls itself recursively.

For a session bean, the complete set of *required* fields are:

► `<ejb-name>`

► `<home>`

► `<remote>`

► `<ejb-class>`

► `<session-type>` the two allowable values are `Stateful` and `Stateless`

► `<transaction-type>` the two allowable values are `Container` for container managed and `Bean` for bean managed. Only container managed transactions will specify a `<container-transaction>` element in the `<assembly-descriptor>` section.

Primary keys

Primary keys may be specified for entity beans only. Any serializable container managed field can be used as a key field, such as `java.util.Hashtable`. In the simplest case, a single attribute of the bean serves as the primary key.

```
<prim-key-class>java.lang.Integer</prim-key-class>
<primkey-field>id</primkey-field> optional
```

The `<primkey-field>` is used only for entity beans with CMP. Furthermore, it is *not used* if the bean has a custom primary key. If *java.lang.Object* is specified as the type for the primary key, then the primary key definition can be left to the deployer. The developer may wish to defer definition of the primary key if, for example, the key is generated by the database. Container managed persistence would be specified for the `<persistence-type>` when deferring the primary key definition.

```
<persistence-type>Container</persistence-type>
<prim-key-class>java.lang.Object</prim-key-class>
or
<prim-key-class>com.ibm.itso.ejb.AccountPK</prim-key-class>
```

If a single attribute of the bean can serve as a primary key, then it is not necessary to define a custom key class and the primitive type can be specified for the `<prim-key-class>`. For instance, `java.lang.Integer` may be specified for an attribute of type `int`.

Environment entries

Properties of the bean can be specified using environment entries in the deployment descriptor. Each entry would have a name, a type and a value. Property lookup is relative to "`java:comp/env`" in the JNDI Context.

```
<env-entry> optional
  <env-entry-name>checkReorder</env-entry-name>
  <env-entry-type>java.lang.Integer</env-entry-type>
  <env-entry-value>1000</env-entry-value>
</env-entry>
```

Integer checkReorder = (Integer)
jndiContext.lookup("java:comp/env/checkReorder");

The `type` should be specified as a primitive type and the `value` field is optional.

References to other beans

The `<ejb-ref>` element is used to reference other beans using JNDI.

```
<ejb-ref> optional
  <ejb-ref-name>AccountHome</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>com.ibm.itso.ejb.AccountHome</home>
  <remote>com.ibm.itso.ejb.Account</remote>
  <ejb-link>CheckingAccount</ejb-link> optional
</ejb-ref>
```

The `<ejb-ref-name>` is relative to "`java:comp/env`". A client would look up a reference to an enterprise bean using the name specified under the `<ejb-ref-name>` element.

```
Object ref = jndiContext.lookup("java:comp/env/AccountHome");
AccountHome accountHome = (AccountHome)
PortableRemoteObject.narrow(ref, AccountHome.class);
```

For beans that use references to other beans defined under the same ejb-jar element, the `<ejb-link>` element is used to provide a link to their corresponding declaration. The reference in the `<ejb-link>` must match the `<ejb-name>` field of a bean in the same deployment descriptor.

External resources
```
<resource-ref> optional
  <res-ref-name>jdbc/myDB</res-ref-name>
  <res-type>java.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

The `<res-ref-name>` is relative to "`java:comp/env`". External resources should be placed under a subcontext to reflect the resource type. In the case of a JDBC data source we have:

```
DataSource source = jndiContext.lookup("java:comp/env/jdbc/myDB");
```

The `<res-auth>` field can specify `Container` or `Application`. `Container` means any authentication required to access the resource will be handled automatically by the container. `Application` means the bean will have to perform its own authentication by whatever means necessary.

Security roles
The `<security-role-ref>` element is used to define the security roles applicable to each bean.

```
<security-role-ref> optional
  <role-name>JavaGuru</role-name>
  <role-link>everyone</role-link> optional
</security-role-ref>
```

The `<role-name>` must match what is returned by the `EJBContext`. The `<role-link>` field is used to map the security-role-ref to a logical security role defined in the `<assembly-descriptor>` section. If a role-link is not provided then the security-role-ref must map to an actual security role in the environment.

```
EntityContext context;
boolean isJavaGuru = context.isCallerInRole(“JavaGuru“);
```

Bean assembly

Before the bean is deployed, the `<assembly-descriptor>` information is added to the deployment descriptor. The assembly descriptor describes transactional attributes of the bean and security roles used in method permissions along with which roles can call each method.

```
<assembly-descriptor>
<security-role>
 ...
</security-role>
<method-permission>
 ...
</method-permission>
<container-transaction>
 ...
</container-transaction>
</assembly-descriptor>
```

Transaction attributes
```
<container-transaction>
    <description>
      methods that require a transaction
    </description>
    <method> one or more
    ...
    </method>
    <trans-attribute>Required</trans-attribute>
</container-transaction>
```

The `<trans-attribute>` can have any of these values:

- ► `Not supported`
- ► `Supports`
- ► `Required`
- ► `RequiresNew`
- ► `Mandatory`
- ► `Never`

For entity beans, these methods must be assigned transaction attributes in the deployment descriptor:

- ► Business methods in the remote interface
- ► Create and find methods in the home interface
- ► Remove methods

For session beans, these methods must be assigned transaction attributes in the deployment descriptor:

- ► Business methods in the remote interface

Logical security roles

```
<security-role>
  <description>
    this role represents an administrator
  </description>
  <role-name>administrator</role-name>
</security-role>

<security-role>
  <description>
    this role represents everyone
  </description>
  <role-name>everyone</role-name>
</security-role>
```

Actual security roles in the environment, such as `JavaGuru` or `EJBGuru`, may be mapped to the logical role of `administrator` or to a role of `everyone`.

```
<method-permission>
    <role-name>everyone</role-name> one or more
    <method> one or more
      <ejb-name>AccountBean</ejb-name>
      <method-name>deposit</method-name>
    </method>
</method-permission>
```

```
<method-permission>
    <role-name>administrator</role-name> one or more
    <method> one or more
      <ejb-name>AccountBean</ejb-name>
      <method-name>withdraw</method-name>
    </method>
</method-permission>
```

So, everyone can invoke the `deposit` method on `AccountBean`, but only clients with the logical role of `administrator` can make withdrawals. The `<method-permission>` element maps security roles to individual methods in the bean or even to groups of methods.

Method declarations

```
<method-permission>
    <role-name>everyone</role-name>
    <method>
      <ejb-name>MyBean</ejb-name>
      <method-name>*</method-name> Wildcard
      <method-intf>Remote</method-intf>
    </method>
</method-permission>
```

Because a method with the same name can be declared in both the home and remote interfaces, the `<method-intf>` element can be used to distinguish between the two.

```
<method-permission>
    <role-name>everyone</role-name>
    <method>
      <ejb-name>MyBean</ejb-name>
      <method-name>create</method-name>
      <method-params>
        <method-param>int</method-param>
        <method-param>double</method-param>
      </method-params>
    </method>
</method-permission>
```

The wildcard (*) applies to all methods indiscriminately. Named methods such as `create` apply to all overloaded methods with the same name in both the home and remote interfaces. To identify a specific method, the `<method-params>` element is used to list the set of parameters, in order, declared in a method signature.

5.8 EJB-JAR

The EJB-JAR file is the standard format for packaging enterprise beans (Figure 5-19).

WebSphere's application assembly tool (AAT) creates a deployment descriptor for each module in the EAR file. The resulting EJB JAR files can be edited with the AAT. The deployment descriptor information can also be manually edited. In EJB1.1 deployment descriptors are XML files describing a module and its components. The deployment descriptor is stored with the name `META-INF/ejb-jar.xml` in the EJB-JAR file.
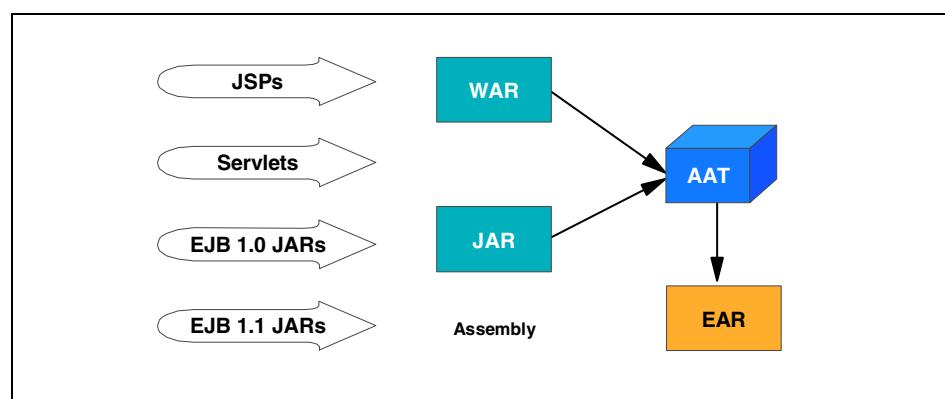
Figure 5-19   EJB 1.0 and 1.1 jars

To summarize, an EJB-JAR contains:

▶ XML deployment descriptor
▶ Enterprise bean classes
▶ Remote interfaces
▶ Home interfaces
▶ Primary key class
▶ Dependent classes and their interfaces
  – Superclasses

Class files of the home and remote interfaces for referenced enterprise beans (defined in other jar files) do not need to be included.

Migrating from EJB 1.0 to 1.1

WebSphere's application assembly tool (AAT) will convert EJB JAR files from 1.0 to 1.1 format.

▶ Open the EJB 1.0 jar file with AAT
▶ Save it using Save As

- ► Resulting jar file contains the EJB 1.1 deployment descriptor and WebSphere extensions
  - – ejb-jar.xml
  - – ibm-ejb-jar-ext.xmi
  - – ibm-ejb-jar-bnd.xmi
  - – Schema files if needed
    - • Schema.rdbxmi
    - • Map.mapxmi
    - • Table.ddl

When migrating from EJB1.0 to EJB1.1 (WebSPhere Application Server 4.0), the following considerations apply:

- ► BMP
  - – Automatic redeployment
- ► CMP deployed with VAJ
  - – Requires manual redeployment
- ► CMP deployed via Admin Console
  - – Automatic redeployment

Also, note that all EJBs will be installed in a single EJB container per server. EJB 1.0 support is deprecated in WebSphere Application Server 4.0.

EJB 1.1 export SmartGuide
VisualAge for Java 4.0 has new EJB1.1 Jar export SmartGuide (Figure 5-20). This function generates XML deployment descriptor.
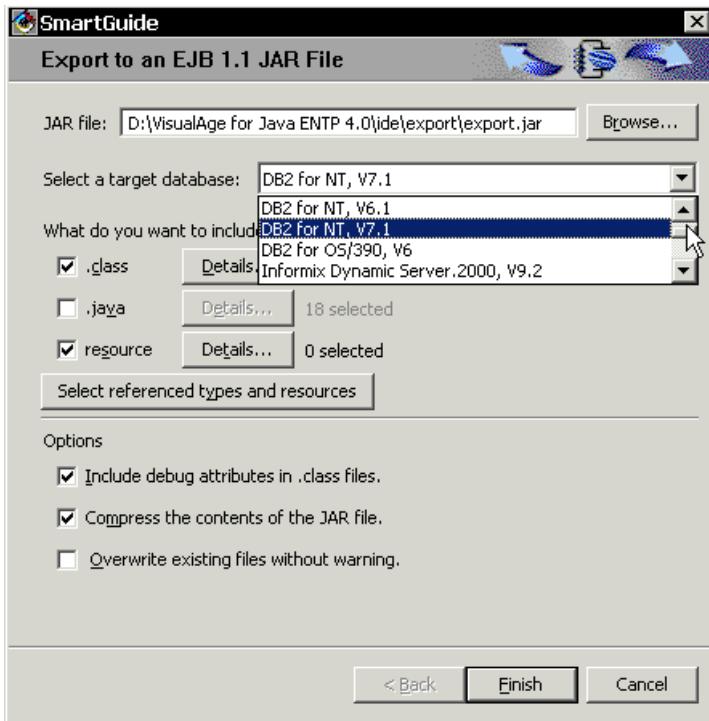
Figure 5-20   EJB 1.1 Jar export SmartGuide

EJB-JAR manifest

The manifest file is deprecated in EJB 1.1, because the enterprise beans are identified in the deployment descriptor.

ejbDeploy tool

An ejbDeploy command line tool is also available to manually deploy EJB archives. There are three ways to deploy an EJB:

► AAT
► During the install of an Enterprise Application
► EJBDeploy command line tool

The parameters to ejbDeploy are:

► input_JAR_name

► working_directory

► output_JAR_name

► [-codegen]

- Only generate code
- Do not compile or run RMIC

► [-cp classpath]

- Additional jar files

► [-noinform]

► [-novalidate]

► [-nowarn]

► [-quiet]

► [-rmic options]

- Additional options

► [-trace]

- Internal tracing

Tools for modifying the WAS repository

The objects stored in the WAS repository for 4.0 are different than 3.5:

► In 3.5 all application data is stored in the WAS repository

► In 4.0 application data is stored in EAR files, not in WAS repository

- The AAT tool is used to edit application data (EAR, EJB Jar, Web WAR)

Therefore, the tools used to modify the WAS repository operate on a different set of objects. Servlets and EJBs could be added to the application server in 3.5, but this must be done in an EJB Jar or WAR file in 4.0.

EJB 1.1 code changes

Code changes that need to be addressed when moving from EJB 1.0 to 1.1. include:

► Check format of JNDI names

► `java.security.Identity` is deprecated

► Cannot throw Remote Exception

- Throw EJB Exception instead

► Cannot use `finalize` method

► Method signature of `ejbCreate` returns primary key type for all entity beans

► Entity bean primary keys can now be `java.lang.String` objects

► Finder methods on entity beans can return `java.util.Collection`

► New `HomeHandle` interface and `getHomeHandle()` method on `EJBHome` interface

– Provides a serializable reference to a home interface for an EJB

Also, if you are using Access Beans, additional code changes may be required for these to work with EJB 1.1. You will need to fix any errors that occur during generation of the deployment code.

6

Transactions and EJBs

This chapter describes how EJBs participate in transactions.

In this chapter, we describe the following:

▶ Container-managed transactions
▶ Bean-managed transactions
▶ Client-initiated transactions
▶ Isolation Levels
▶ JDBC transaction support
▶ JTA transaction support

A transaction involves one or more steps which must be successfully completed as a unit in order for the operation to be successful. Either all steps complete successfully and the transaction is committed or one or more of the steps fail and the transaction is rolled back. Rolling back a transaction reverts to the previous committed state, as if no new changes had occurred. A `begin` and `commit` (or `rollback`) mark the boundaries of a transaction. With container-managed transactions, the transaction boundary is managed by the container. Similarly, with bean-managed transactions, the transaction boundary is managed by the developer.

Note: the J2EE 1.2 specification doesn't actually require that a J2EE server will support access to multiple JDBC databases within a transaction context (using the two-phase commit protocol).

Transaction attributes

Various transaction attributes may be specified either for the bean as a whole or for individual methods on an EJB instance. Transaction time-outs may also be specified, if desired. The default time-out value is set to zero, meaning the corresponding transaction will not time out.

Transaction attributes are specified in the deployment descriptor. The available types for transaction support in EJB 1.1 are:

- ► Required
- ► Mandatory
- ► Supports
- ► BeanManaged
- ► RequiresNew
- ► NotSupported

Required

Transaction is strictly required. This means if the client invokes a method on the bean in a transaction context, that context is passed to the bean, or else a new transaction context will be created by the EJB container and committed before the method returns.

Mandatory

The container throws *javax.jts.TransactionRequired* if a client-initiated transaction does not exist when methods are invoked on the bean.

Supports

The client transaction context, if any, is passed to the bean as the `EJBContext`.

BeanManaged

The client-initiated transaction context is suspended if one exists and transaction management is left to the bean during method invocation.

RequiresNew

A new transaction will always be created by the container. If the client invocation already has a transaction context, it will first be suspended and again resumed before the method returns.

NotSupported

The bean cannot support transactions, so if the client invokes a method on the bean and a transaction context exists, that transaction will be suspended and then resumed after the method completes. The invoked method will not have a transaction context.

Session beans need to specify transaction attributes for business methods. Entity beans require transaction attributes to be specified for business methods as well as the `create`, `remove` and `finder` methods.

6.1 Container managed transactions

Specifying container-managed transactions with the TX_REQUIRED transaction attribute is the easiest way to handle transaction management in the application and will work fine in most cases; the transaction management will be handled by the EJB container.

► Entity beans *must* use container-managed transactions.

► Session beans may use either container-managed transactions or bean-managed transactions.

  – Session beans with container-managed transactions may find it useful to implement the `SessionSynchronization` interface to manage synchronization of instance variables, as desired.

  – Session beans with bean-managed transactions may choose to implement either JDBC transactions or JTA transactions. JDBC transactions involve the use of the `java.sql.Connection` class; whereas, JTA transactions involve the use of the `javax.transaction.UserTransaction` class.

Figure 6-1 summarizes the various transaction options available.
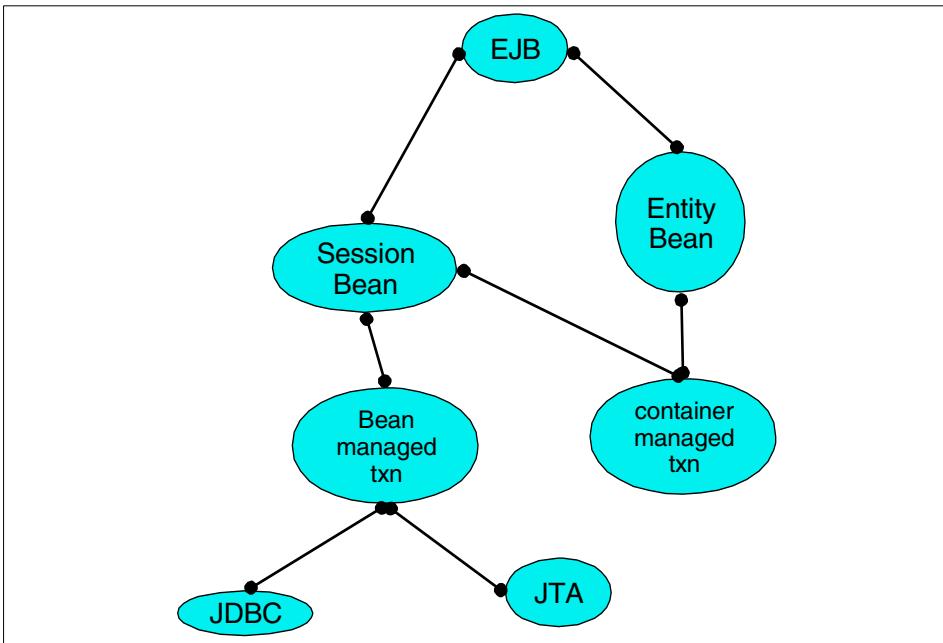
Figure 6-1   Summary of EJB transaction options

Container-managed transaction roll back

System exceptions will *automatically* cause the container to roll back the transaction. Application exceptions do not automatically cause a roll back. In the event of an application exception, the bean developer can call the `setRollbackOnly` method of the `EJBContext` interface to force a roll back.

There are special considerations for entity beans and session beans in the event of a roll back:

► For a session bean, the SessionSynchronization interface can be implemented in order to explicitly reset a bean's instance variables.

   – Session beans with bean-managed transactions would normally not implement the SessionSynchronization interface, since they have full control of the commit.

► For an entity bean, when a rollback occurs the EJB container will invoke `ejbLoad()` which will have the effect of reloading the bean's instance variables from the datastore.

In addition, developers should avoid calling the following methods that might interfere with container-managed transactions:

► *java.sql.Connection* `commit`, `setAutoCommit` and `rollback`

- *javax.ejb.EJBContext* `getUserTransaction`
- *javax.transaction.UserTransaction* - all methods

> **Note:** For more information, see Sun J2EE Developers Guide at:
> `http://java.sun.com/j2ee/j2sdkee/devguide1_2_1.pdf`

Bean-managed transactions do not have these restrictions.

Session beans with container-managed transactions

The EJB specification allows *stateful* session beans to optionally implement the `SessionSynchronization` interface so that they will be notified by the container about the beginning and end of a transaction. Stateful session beans may use this information when accessing data on behalf of the client. Any data that may have been cached by the session bean could be committed when the transaction completes. In addition, the stateful session bean may wish to refresh any required instance variables from the datastore in the case of a roll back, since the container will not do so automatically for entity beans.

The way that the `SessionSynchronization` interface works is for the EJB container to automatically invoke the `afterBegin`, `beforeCompletion` and `afterCompletion` methods on the bean. The `afterBegin` method is called after a new transaction is begun but before the business method is invoked, so it's a good place to synchronize the instance variables of the session bean with the state of the database. The session bean has one last opportunity to force a rollback in the `beforeCompletion` method, if desired. In the `afterCompletion` method, the session bean can check if a roll back occurred and respond accordingly.

Session Synchronization (javax.transaction)

```
public class MyBean implements SessionBean, SessionSynchronization {
  private transient SessionContext ctx;
  private transient boolean tranFailure = false;

  public void afterBegin()
  {}

  public void beforeCompletion() {
    if ( tranFailure ) {
      ctx.setRollbackOnly();
    }
  }

  public void afterCompletion(boolean success)
  {
    if (!success)
      //rollbackOccurred
```

```
            }
```

6.2 Bean managed transactions

Bean-managed transactions involve **Session** beans, since entity beans must use container-managed transactions. (In other words, entity beans may choose to manage their own persistence mechanism, but must leave transaction management to the container). Bean-managed transactions allow fine-grained control on the part of the developer. Session beans with bean-managed transactions may choose to implement either JDBC transactions or JTA transactions.

With JDBC, the developer invokes `setAutoCommit(false)` on the `Connection` object and then demarcates a transaction with the `commit` and `rollback` methods. The beginning of a transaction is normally implicit; a new transaction is begun by the first SQL call following a connect, commit or rollback. For JTA, transactions are demarcated by the `begin`, `commit` and `rollback` methods of the `UserTransaction` interface.

You should avoid calling `getRollbackOnly` and `setRollbackOnly` methods of `EJBContext for` bean-managed transactions. These methods are for container-managed transactions only. For bean-managed, the `getStatus` and `rollback` methods of `UserTransaction` are to be used.

Session beans with bean-managed transactions
Business methods for *stateless* session beans will either commit or roll back the transaction before returning; *stateful* session beans do not have to commit or roll back the transaction before returning. The association of a transaction context with the bean is retained across multiple method calls for JTA transactions but may or may not be retained for JDBC transactions. In the case of JDBC, the association is dropped if the connection is dropped.

6.2.1 JDBC transactions

JDBC transactions are controlled by the transaction manager of a particular DBMS (that is, DB2 or Oracle). They can be used in bean-managed transactions.

The `Connection` interface methods `commit` and `rollback` are used to demarcate JDBC transactions. The setAutoCommit(false) method is invoked to allow the application to manage its own commit for bean-managed transactions. A container-managed session bean can, optionally, include session synchronization to manage the auto commit provided by the container.

Demarcating a JDBC transaction

```
public void transferFunds(int fromAcctID, int toAcctID, java.math.BigDecimal
amt)
throws javax.ejb.CreateException, java.rmi.RemoteException {
   com.ibm.itso.ejb.Acct fromAcct = null;
   com.ibm.itso.ejb.Acct toAcct = null;
   try {
    conn.setAutoCommit(false);
    fromAcct = getAcctHome().findByPrimaryKey(new AcctKey(fromAcctID));
    toAcct = getAcctHome().findByPrimaryKey(new AcctKey(toAcctID));
    fromAcct.withdraw(amt);
    toAcct.deposit(amt);
    conn.commit();
   } catch (Exception e) {
      conn.rollback();
      System.out.println("Transaction failed:"+e);
   }
}
```

6.2.2 JTA transactions

JTA transactions are independent of the type of DBMS used in a particular
implementation and are controlled by the J2EE transaction manager. They can
be used in bean-managed transactions. The J2EE server implements the
`UserTransaction` and `Status` interfaces. The developer invokes methods on
these interfaces through the `EJBContext`.

The benefit of using JTA transactions over the JDBC approach is that the
implementation may transparently access multiple databases within a singe
transaction (two phase commit). The disadvantage is that nested transactions
are not supported.

The `UserTransaction` interface methods `begin`, `commit` and `rollback` are used to
demarcate JTA transactions. Mixing JTA and JDBC is not recommended in a
session bean with bean-managed transactions due to the additional complexity
and associated maintenance issues.

Demarcating a JTA transaction

```
public void transferFunds(int fromAcctID, int toAcctID, java.math.BigDecimal amt)
throws javax.ejb.CreateException, java.rmi.RemoteException {
   com.ibm.itso.ejb.Acct fromAcct = null;
   com.ibm.itso.ejb.Acct toAcct = null;
   UserTransaction userTran = (UserTransaction)ejbcontext.getUserTransaction();
   try {
     userTran.begin();
     fromAcct = getAcctHome().findByPrimaryKey(new AcctKey(fromAcctID));
     toAcct = getAcctHome().findByPrimaryKey(new AcctKey(toAcctID));
```

```
      fromAcct.withdraw(amt);
      toAcct.deposit(amt);
      userTran.commit();
   } catch (Exception e) {
      userTran.rollback();
      System.out.println("Transaction failed:"+e);
   }
}
```

The available methods in the `UserTransaction` interface support demarcation of a transaction, setting transaction time-outs and rolling back a transaction.

UserTransaction interface (javax.transaction)

► void begin()

   Create a new transaction and associate it with the current thread.

► void commit()

   Complete the transaction associated with the current thread.

► int getStatus()

   Obtain the status of the transaction associated with the current thread.

► void rollback()

   Roll back the transaction associated with the current thread.

► void setRollbackOnly()

   Modify the transaction associated with the current thread such that the only possible outcome of the transaction is to roll back the transaction.

► void setTransactionTimeout(int seconds)

   Modify the time-out value associated with transactions started by subsequent invocations of the begin method.

The various states of a transaction can be queried by invoking the `getStatus` method on `UserTransaction`.

Summary of JTA Transaction Status

► static int STATUS_ACTIVE

   Transaction is in the active state.

► static int STATUS_COMMITTED

   Transaction has been committed.

► static int STATUS_COMMITTING

   Transaction is in the process of committing.

- ▶ static int STATUS_MARKED_ROLLBACK

  Transaction has been marked for rollback (maybe as a result of a setRollbackOnly operation)

- ▶ static int STATUS_NO_TRANSACTION

  No transaction is currently associated with the target object.

- ▶ static int STATUS_PREPARED

  Transaction has been prepared

- ▶ static int STATUS_PREPARING

  Transaction is in the process of preparing.

- ▶ static int STATUS_ROLLEDBACK

  Transaction outcome is determined as a rollback.

- ▶ static int STATUS_ROLLING_BACK

  Transaction is in the process of rolling back.

- ▶ static int STATUS_UNKNOWN

  Current status of transaction cannot be determined.

Note: In EJB1.1, you cannot have `UserTransaction` with `SessionSynchronization` at the same time for bean-managed transactions.

6.2.3 Client initiated transactions

Clients may wish to establish their own transaction context using JTA. The EJB server will examine the transaction context of the client and handle the transaction context of the EJB (specified in the deployment descriptor) in a suitable fashion, either suspending the client transaction context or forwarding it to the EJB.

Pseudocode for Client Initiated JTA Transaction

```
public class RegistrationAssistant extends HttpServlet {
  try {
    userTran = (UserTransaction)ctx.lookup("jta/usertransaction");
    userTran.begin();
    // processing...
    userTran.commit();
  } catch (Exception ex) {
    getServletConfig().getServletContext().getRequestDispatcher(errorURL).forwa
rd(request, response);
  }
}
```

6.3 Isolation levels

During deployment, the isolation level for a bean can be specified. This causes the container to enforce certain behavior to ensure that transactions made by multiple client requests do not interfere with one another.

The levels which may be specified for the bean (or for individual methods on the bean) are:

- ► TRANSACTION_READ_UNCOMMITTED

  Dirty reads, nonrepeatable reads and phantom reads are allowed.

- ► TRANSACTION_READ_COMMITTED

  Dirty reads are prevented.

- ► TRANSACTION_REPEATABLE_READ

  Phantom reads are allowed.

- ► TRANSACTION_SERIALIZABLE

  Serializes concurrent transactions.

The application server will normally provide a default if none is specified for the particular database used. Entity beans with container-managed persistence have a default level (normally, READ_COMMITTED) and cannot be modified.

7

Messaging with JMS, WebSphere and MQSeries

In this chapter we introduce messaging, and the Java Message Service API for interaction with messaging systems from Java. We then further the discussion by introducing the support available with WebSphere 3.5.3 and MQSeries 5.2 for performing JMS messaging across JTA transactions. We then provide an example of a simple point to point message scenario by implementing the mortgage payment functionality of our PiggyBank scenario.

Readers should note that this chapter provides a relatively brief overview of JMS, enough to enable us to discuss the WebSphere and MQSeries implementation. Messaging is a complex subject, and a lot of the JMS API is not covered here. Readers wishing to delve deeper into JMS and the MQSeries implementation should refer to the references given in "Related publications" on page 327.

125

7.1 A brief introduction to messaging

Message oriented middleware (MOM) is one of the most established areas of middleware software. Products such as IBM MQSeries supply the capability for applications to communicate across platform boundaries. By using MOM products, applications can be shielded from the complexities of understanding different operating systems, application APIs and the like. What's more, MOM systems offer assured, once only delivery of messages allowing applications to send messages and not worry about whether they will arrive at their destination.

7.1.1 Asynchronous messaging and MQSeries

In a synchronous messaging scenario, an application that wishes to send a message to another application must first set up a communication channel. If the receiving application is unavailable, due for instance to network problems, then the sending application will not be able to send the message, and must block it's processing to retry the send, or return some error condition. If the message does get through, then the sending application must wait for a reply. If the reply takes a long time, or fails to arrive then this can cause problems.

In today's world of heterogeneous networked environments, where applications may be separated by continents, the concept of being able to send a message regardless of the state of the receiving application, and know that the message will be received, is valuable. MQSeries provides the facilities to perform such asynchronous messaging.

MQSeries works on the basis of messages and queues. An application can place a message onto a queue, and that message can then be retrieved by another application. In order to put messages, an application interfaces with a local queue manager (referred to in MQSeries as an MQSeries Queue Manager, or MQM). The queue manager provides services to the application that ensure messages are placed on the correct queue. The message created by the sending application contains header information including the destination queue. When a queue manager receives the message, it places it onto the appropriate queue.

The receiving application can retrieve the message by connecting to the queue manager and 'listening' for messages on a particular queue. For instance by spawning a thread to get messages from a particular queue. If the receiving application is not available, the message is held on the queue.

The sending and receiving applications will normally be on different machines, linked by a network, and connected to different queue managers. Messages can be sent between different queue managers by defining remote queues. This scenario is shown in Figure 7-1.
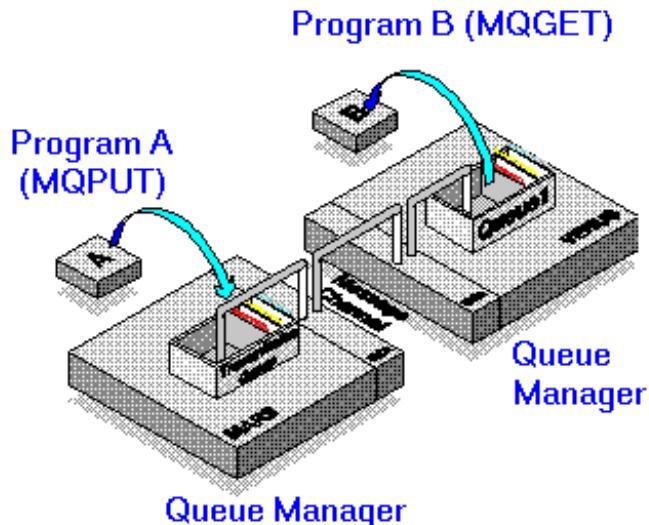
Figure 7-1   Asynchronous messaging example

Here, Program A talks to its local queue manager, which has a remote queue definition for a queue managed by another queue manager. Program A creates a message with the remote queue as the destination and puts it onto the queue. Program B at some point can retrieve the message by communicating with its queue manager to get the message. Program A may include a return address within the message header, which Program B can use to find out which queue it should put any reply message onto. The original message will also contain a unique message ID generated by the queue manager. Program B can include this ID in a reply message to help program A find the reply in the reply queue.

7.2 Messaging architectures

There are two basic messaging architectures in common use. These are point-to-point (PTP), and publish/subscribe (pub/sub). The JMS API supports both, and so we introduce them briefly here.

7.2.1 Point-to-point (PTP)

A PTP messaging architecture is concerned with the delivery of messages from one sending application to a receiving application. The demonstration of asynchronous messaging given in Figure 7-1 is a prime example of PTP, where Program A sends messages indirectly via a queue to program B.

A PTP communication may or may not involve the processing of a reply from the receiving application depending on the needs of the client.

7.2.2 Publish/subscribe

The pub/sub architecture is based around the concept of topics. Publishers can publish messages to a particular topic, and subscribing applications can then read the messages. The publisher and subscribers do not need to have knowledge of each other as all interaction takes place through the topic.

A common pub/sub scenario is that of stock price information. A stock price publisher may publish information to a particular topic, for instance the current price of IBM stock. Any application can then retrieve that information by subscribing to the topic. The publisher and subscriber are decoupled. There may be none, one or many subscribers who consume messages on a particular topic, supplied by one or more publishers.

When publishing information, the publishing application specifies a topic string that identifies the topic for which the message is destined. Topics may be defined within a hierarchy. For instance, Figure 7-2 shows a possible hierarchy for a set of stock quote topics.
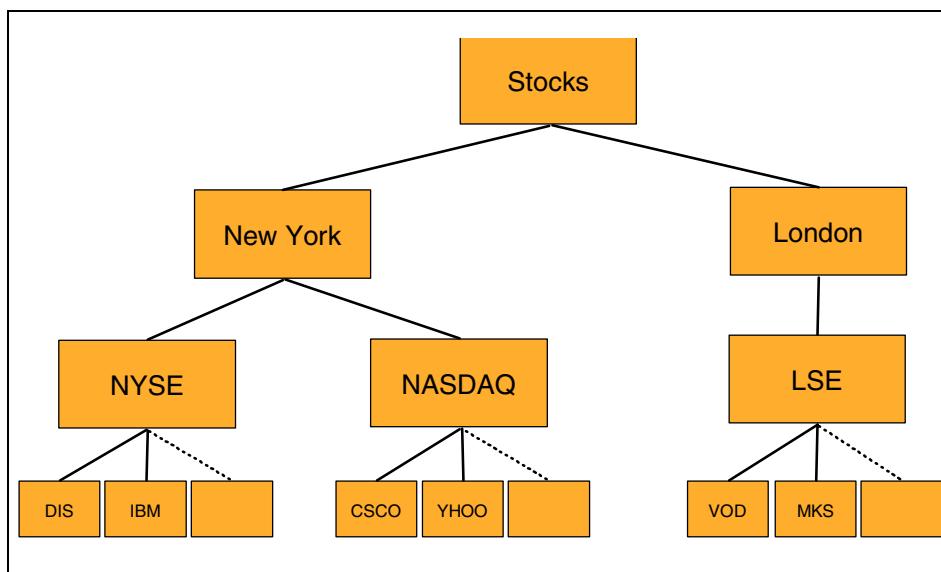
Figure 7-2   Example topic hierarchy

So for example, if we want to publish a message about IBM stock, we can set the topic string to Stocks/New York/NYSE/IBM. Similarly, if we wish to subscribe for messages about Vodafone stock, we can subscribe to Stocks/London/LSE/VOD.

> **Note:** The JMS API does not specify a convention for topic naming. The example shown above is how a topic hierarchy may be built up using MQSeries publish/subscribe.

When an application subscribes to a topic, messages on that topic may be pushed to the application when they arrive, or may be explicitly pulled from the topic by the subscribing application.

> **Note:** MQSeries supports publish/subscribe through the MA0C SupportPac available from `http://www.ibm.com/software/ts/mqseries/txppacs`

7.3 Common messaging patterns

There are four common patterns for message-based communication that are relevant to the PTP architecture, the pub/sub architecture or indeed both. They can be summarized as:

► One-way datagram production
► One-way datagram consumption
► Pseudo-synchronous request production/reply consumption
► Request consumption and reply production

We discuss each of these in more detail, and describe how the patterns are used in the context of J2EE applications.

7.3.1 One-way datagram production

A datagram can be thought of as a message for which no reply is needed, or expected. Commonly, datagrams are used as a way of initiating other business logic. For instance, we will use a datagram within the PiggyBank application to inform the mortgage provider's system that a payment is initiated.

Through datagram usage, the initiating, or sending application can be fully de-coupled from the receiving application allowing for a truly asynchronous relationship between the two. By using MOM software as the link, the ability for the initiating application to continue its processing is not hindered by the availability or otherwise of the receiving application. This ability to kick off remote business processing is particularly useful within web-based applications as it is often unreasonable to delay user response until a reply is received.

Figure 7-3 shows a typical datagram production scenario.
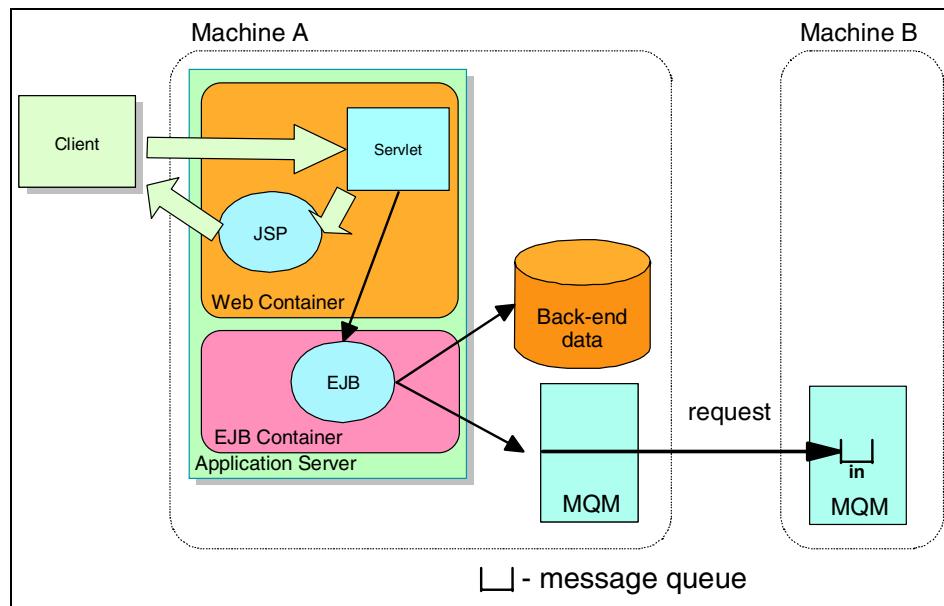
Figure 7-3   Datagram production scenario

By utilizing the assured delivery capabilities of the MOM software, the sending application can be sure that the message will be received by the receiving application. However, it may need to provide compensating logic to handle a business-process failure within the remote application. For instance within our PiggyBank mortgage payment scenario, funds for the payment are transferred from the users account to a central holding account. If the business process kicked off by sending a datagram to the mortgage application fails for whatever reason, then we need a way to credit the funds back to the users account.

In this scenario, a stateless session-bean may send a one-way datagram as part of some business process, often involving a distributed transaction that involves a local database update with the sending of the message. For instance, when making a mortgage payment, the sending application on machine A may create a transaction record. This transaction record may be used to provide the information to handle correctional logic should the remote business process fail.

Within the pub/sub model, the publishing of a message to a particular topic implements this pattern also. However in this case, it is unlikely that the message will be part of a local transaction, and depending on the scenario, it may not be necessary to ensure delivery.

7.3.2 One-way datagram consumption

With reference to Figure 7-3, the datagram sent by machine A is placed on a
queue on machine B. The second pattern deals with the consumption of this
message. We require a way to asynchronously handle the arrival of a message
which will kick-off some business process. This requires a notification
mechanism, and a registered listener to receive such notifications.

The receipt of the message may or may not be included within the scope of a
local transaction. However, we do not need to initiate any reply message to the
sender.

Within the pub/sub architecture this pattern is implemented by subscribers to
topics who will need to consume messages as they are published to the topic.

7.3.3 Pseudo-synchronous request/reply

Sometimes, it is not reasonable to send a message without expecting a reply.
Although there are advantages to de-coupling the sending and receiving
application, there are scenarios in which confirmation of some remote
processing through the receipt of a reply is ideal. A typical example would be a
non-critical business processes in which a request either works or it doesn't.
There is normally no transactional requirement on the request, and it may be
performed on a best-can-do process, but the initiator must know if the request
succeeds or failed.

In this case, the initiator of the request will send the request, and then block
processing to wait for the reply. This is shown in Figure 7-4.
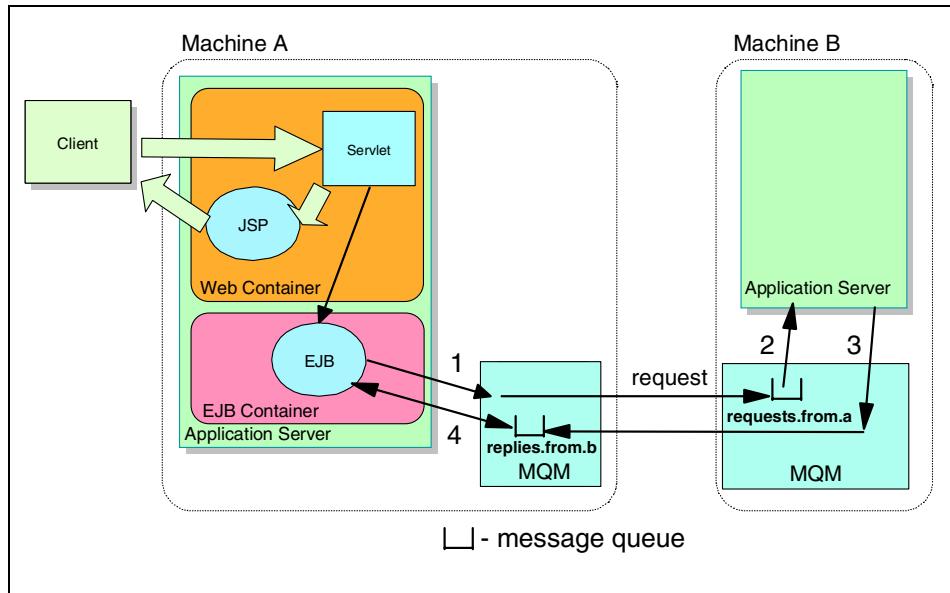
Figure 7-4   Pseudo-synchronous request/reply pattern

At point one, the business logic on machine A sends a message to the requests.from.a queue. It then waits for a reply from the remote application to appear within the replies.from.b queue.

This closer coupling of the two applications leads to extra logic within the sending application to handle the failure to receive a reply. Also if user response time is an issue, as it may be within a Web-based application, then it may be necessary to return a response to the user with enough information for them to then initiate another request at a later time to query the success of the operation.

7.3.4 Request consumption/reply production

This pattern involves the production of a reply message in receipt of a request. This may be as part of an asynchronous or pseudo-synchronous scenario. For instance, this pattern would be implemented by the application running on machine B in Figure 7-4.

As the pattern involves two processes, it can best be thought of as a combination of the second pattern: the asynchronous consumption of a message, and the first: the sending of a one-way datagram in reply. Therefore, we require the same ability to listen for message arrival as discussed for the second pattern.

Depending on the business scenario, the receipt of the request and the sending of the reply may be handled within a local transaction.

7.4 The JMS API

The JMS standard developed by Sun Microsystems aims to provide a messaging provider independent API for performing messaging tasks within Java code. JMS 1.0.2 is defined as a standard service for J2EE 1.2, but there is no reference implementation. Vendors are left to implement the specification on top of their MOM products. For IBM, MQSeries provides a JMS implementation. The J2EE 1.3 platform will contain a reference JMS implementation.

7.4.1 Elements of a JMS system

A JMS system comprises of a number of components. A JMS client may be a Java applet, application, servlet, JavaBean or EJB that uses the JMS API to send and receive messages. Through the JMS API, the client interacts with a JMS provider. The provider has the responsibility of implementing the JMS API on top of an underlying messaging system. The outline of this communication is shown in Figure 7-5.
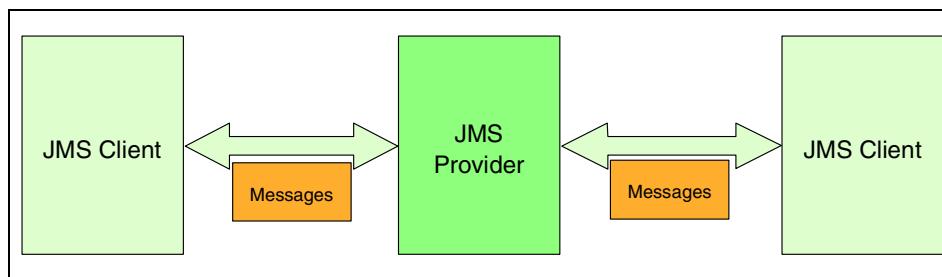
Figure 7-5   JMS elements

A JMS client may therefore communicate with another JMS client via a JMS provider. As the JMS provider is built on top of an existing messaging system, so a JMS client may also communicate with non-JMS clients that use the messaging systems native APIs, as the JMS provider will translate JMS messages into the underlying message format. This is an important feature as it opens up a way to use messaging as an interface to existing message-aware enterprise applications.

JMS messages

The JMS API defines a message API for creating JMS messages. It aims to define a message format generic enough so that JMS providers can map it onto their underlying message format.

JMS messages are composed of three parts:

► **Header**: Contains values used by JMS clients and providers to identify and route messages

► **Properties**: Provides support for application defined properties, useful primarily for message filtering

► **Body**: The content, or payload of the message

JMS defines a set of standard message headers shown in Table 7-1. The values of the headers are normally set after the message is sent.

Table 7-1   JMS message headers

Header	Description
JMSDestination	The destination that the message was sent to.
JMSDeliveryMode	The delivery mode used to send the message. Will be either PERSISTENT or NON-PERSISTENT.
JMSPriority	The priority of the message. An integer from 0 (highest) to 9 (lowest). The default is 4.
JMSExpiration	The time when the message will expire. Default is 0 (never expire.)
JMSMessageID	A unique message identifier. Always begins with 'ID:'
JMSCorrelationID	An identifier used to link replies to the original message. Normally set to the originating message ID by the replying client.
JMSReplyTo	Specifies a destination to which message replies should be sent. Set by the sending client before the message is sent.
JMSTimestamp	The time at which the JMS provider sent the message.

Header	Description
JMSRedelivered	Set by the JMS provider if the message has already been delivered.
JMSType	Allows JMS clients to use JMS provider specific message-types.

JMS providers or clients may not define additional message headers. Instead, the specification allows clients and providers to specify message properties. The specification defines some standard message properties, which can optionally be set for each message. JMS providers can also use message properties to allow clients to set headers in the underlying message definition for communication with non-JMS clients. Additionally clients can set their own message properties. This latter usage then allows clients to perform filtering of messages based on the properties.

The message body may be one of five types, as listed in Table 7-2.

Table 7-2   JMS message body types

Message body type	Description
StreamMessage	Contains a stream of primitive types.
MapMessage	Contains a set of name/value pairs, where name is a String and value is a primitive type.
TextMessage	Contains a String.
ObjectMessage	Contains a serialized Java object.
ByteMessage	Contains a byte stream.

Of these, it is TextMessage that is most commonly used due to its simplicity, and also because of its ability to encapsulate XML data.

JMS objects

Sending and receiving messages within a JMS application requires interaction with a number of objects. There are six such objects, and the point-to-point and publish/subscribe APIs have interfaces for each of them. The objects are listed in Table 7-3.

Table 7-3   JMS objects

Object	Description
ConnectionFactory	Used by the client to obtain a connection to the JMS provider. The ConnectionFactory is an administered object which can be obtained by a JNDI lookup.
Connection	The connection to the JMS provider. This connection may remain open over many message sends and receives. Acts as a factory for creating Session objects.
Session	A single thread for receiving or sending messages to the JMS provider. Used to create messages, and as a factory for MessageProducers and MessageConsumers.
Destination	Normally an administered object obtained through JNDI that refers to a queue or topic.
MessageProducer	Used to send messages to a named Destination.
MessageConsumer	Used to receive messages from a named Destination.

Developing a JMS client to send or receive messages involves creating instances of ConnectionFactory, Connection and Session objects (or as we will see, their point-to-point and pub/sub equivalents.) The Session is then used to create MessageProducers and MessageConsumers. Messages are then created and sent using the MessageProducer, or received using the MessageConsumer.

A JMS client will normally create one Connection object to the JMS provider. From this Connection, one or more Sessions will be created. A Session object may create one or more MessageProducers and MessageConsumers. The Connection object has start() and stop() methods that tell the JMS provider

whether or not to deliver incoming messages to any MessageConsumers. When a Connection object is created it is in stop mode, and no messages will be forwarded by the JMS provider. MessageProducers may still send messages to the JMS provider when the connection is stopped.

Administered objects

In order to interact with a JMS provider, a client must have access to objects that enable it to connect to the JMS provider and also to identify a destination for a message. These are the ConnectionFactory and Destination objects. These are created by an administrator, and made available to clients, usually through JNDI.

Object caching

Of the JMS objects, ConnectionFactories, Connections and Destination objects are thread-safe and therefore may be cached by the JMS client. Caching these objects is desirable to reduce the number of JNDI lookups, and the initiation of connections to the underlying messaging system.

7.4.2 Point-to-point API

The PTP API allows JMS clients to interact with queues provided by the JMS provider. Using the API, JMS clients can send messages to and receive messages from queues.

The API defines a number of interfaces based on the JMS objects for which the JMS Provider must supply implementations. These interfaces are shown in Table 7-4, and each one is described below.

Table 7-4   Point-to-point API interfaces

JMS Object	PTP API interface
ConnectionFactory	QueueConnectionFactory
Connection	QueueConnection
Session	QueueSession
Destination	Queue
MessageProducer	QueueSender
MessageConsumer	QueueReceiver

The QueueConnectionFactory and QueueConnection interfaces provide methods to set up a connection to the JMS provider. Code for using these interfaces is shown in the following code:

`Code to obtain QueueConnectionFactory and QueueConnection objects`

```
// The QueueConnectionFactory is retrieved from JNDI using a defined lookup
// name. There is no standard naming convention for JMS administered objects.
// Using the jms/ prefix is good practice
InitialContext ctx = new InitialContext();
QueueConnectionFactory qcf = (QueueConnectionFactory)ctx.lookup("jms/fooQCF");

// Use the QueueConnectionFactory to obtain a QueueConnection which represents
// a physical connection to the JMS Provider
QueueConnection qConn = qcf.createQueueConnection();
```

To send to or receive messages from a queue, a QueueSession needs to be created. QueueConnection defines a method named createQueueSession() which accepts the following parameters:

► Boolean transacted — Defines whether the session is transacted. If true then all of the messages sent and received from the transaction are treated as a unit of work and may be committed or rolled back.

► int acknowledgeMode — Defines the acknowledge mode. One of:
  – Session.AUTO_ACKNOWLEDGE
  – Session.CLIENT_ACKNOWLEDGE
  – Session.DUPS_OK_ACKNOWLEDGE

In most situations, these can be defined as false, and Session.AUTO_ACKNOWLEDGE, respectively. If the session is transacted then the acknowledgeMode value is ignored. The following code shows the creation of a QueueSession:

Creation of a QueueSession object

```
QueueSession session = qConn.createQueueSession(false,
Session.AUTO_ACKNOWLEDGE);
```

Obtaining a queue
A Queue object represents a queue managed by the JMS provider. In most normal scenarios, the Queue object will be an administered object obtained from a JNDI call. Alternatively, a queue can be created from the QueueConnection object by calling createQueue(String queueName). The argument may be the name of a queue as defined by the JMS provider. However, obtaining a queue in this fashion limits the subsequent portability of the code as queue naming conventions differ between vendors. The following code shows the creation of a Queue object through the JNDI lookup of an administered object:

Creation of a Queue object

```
Queue queue = (Queue)ctx.lookup("jms/fooQueue");
```

Sending a message

To send a message to a queue, we need to create the message itself, and then use a QueueSender object to send it.

To create the message, we use the one of the methods available on the QueueSession object listed in Table 7-5.

Table 7-5   QueueSession methods for message creation

Method	Description
createStreamMessage()	Creates a message of type StreamMessage.
createMapMessage()	Creates a message of type MapMessage.
createTextMessage() createTextMessage(String mess)	Creates a message of type TextMesasge. If an argument is supplied, it is used to initialize the message.
createObjectMessage() createObjectMessage(Serializable obj)	Creates a message of type ObjectMessage. If an argument is supplied, it must be a serializable object which will be used to initialize the message.
createBytesMessage()	Creates a message of type BytesMessage.

If a reply to the message is expected then a reply destination must be set by calling setJMSReplyTo(Destination replyTo) on the message object. The argument specifies the Queue that the receiving application should place its reply on.

The following code shows the creation of a simple TextMessage object:

```
//Creation of a message
TextMessage message = session.createTextMessage("This is a message");
message.setJMSReplyTo((Queue)ctx.lookup('jms/barQueue'));
```

To send the message, we need to create a QueueSender. This is created from the QueueSession object using the createSender(Queue queue) method where the argument supplied is the Queue object we want to put the message on. We can then send the message by simply calling the send(Message mess) method of the QueueSender supplying the message we created.

Once sent, the message will contain extra header information set by the JMS provider which it may be useful to access, especially if a reply is expected. The most useful of these is the getJMSMessageID() method which returns the value of the JMSmessageID header containing the unique identifier assigned by the JMS provider for the message. This can be used later to match up a reply to the sent message.

The following shows code for sending a message:

```
//Sending a message to a queue
QueueSender sender = session.createQueueSender(queue);
sender.sendMessage(message);
String messageID = message.getJMSMessageID();
```

Receiving messages

There are essentially two modes of operation for receiving messages from queues. The first of these used when implementing a pseudo-synchronous request/reply operation. For instance, we may expect a reply to the message we sent in above code, in which case we can perform a block on our processing while we wait for the reply, so we effectively act synchronously in the expectation that a reply will be forthcoming. Alternatively, we may be acting as the receiving application for the message, in which case we need to handle message arrival in an asynchronous fashion. The JMS specification provides ways to handle both of these scenarios.

To receive messages, we must create a QueueConnection object as we did. However, the JMS provider will not start to send messages to us until we call the start() method of the QueueConnection. This informs the JMS provider that we are in a state to accept incoming messages. We should be in a position to receive messages before calling start() by creating a QueueReceiver object from the QueueSession.

The QueueReceiver is a type of MessageConsumer and we obtain it by calling one of the methods of QueueSession defined in Table 7-6.

Table 7-6   Methods available on QueueSession to create QueueReceiver

Method	Description
createReceiver(Queue queue)	Creates a QueueReceiver object which will receive messages for the queue defined as an argument.
createReceiver(Queue queue, String messageSelector)	Creates a QueueReceiver to receive messages from the specified queue whose properties match the messageSelector.

In the latter method, the messageSelector defines a filter against which the properties of a message may be matched. Only messages with properties matching the messageSelector are available from the QueueReceiver. The following code shows the creation of a QueueReceiver:

```
//Creation of a QueueReceiver object
Queue replyQueue = (Queue)ctx.lookup("jms/barQueue");
QueueReceiver replier = session.createReceiver(replyQueue);
```

Receiving messages synchronously
To receive messages from the QueueReceiver synchronously, we can call one of the methods listed in Table 7-7. These methods are defined on the MessageConsumer interface which QueueReceiver extends.

Table 7-7   Synchronous message retrieval methods

Method	Description
receive()	Receives the next message from the QueueReceiver. This method blocks until a message is received.
receive(long timeOut)	Receives the next message from the QueueReceiver. This method blocks until a message is received, or until the timeOut interval specified expires. The timeOut value is specified in milliseconds. A value of 0L represents an indefinite interval.
receiveNoWait()	Returns the next message from the QueueReceiver. If there is no message available, this method returns immediately with null.

In each case, the return type of the method is a Message object containing the first available message on the queue. If the QueueReceiver was specified with a messageSelector then the returned message will be the first message in the queue that matches the filter.

Once a message is received, the headers, properties and body of the message can be acted upon.

Following code shows the code for receiving a message synchronously. When a message is received, it must be cast from the Message interface to one of the five message types. If as the programmer your QueueReceiver may obtain messages of more than one type, then the Java instance of keyword can be used to determine the received message type.

```
//Receiving a message synchronously
// start the QueueConnection to enable message delivery
```

```
qConn.start();

// try to obtain a message for 5 seconds
TextMessage replyMessage = (TextMessage)replier.receive(5000);

// see if this is the reply to the message we sent earlier by comparing the
// message ID of the message we sent to the correlation ID of the message
// we received.
String reply = null;
if(messageID.equals(replyMessage.getJMSCorrelationID()))
    reply = replyMessage.getText();
```

Receiving messages asynchronously

Often applications will need to perform some actions on receipt of a message from another JMS client. For instance, the reply we received may have been sent by another JMS client who was listening for our original message sent.

It is possible to use the receive() method within a loop to perform this function, but this does not allow the application to continue with other work while waiting for a message. The JMS API instead provides a way for an event to be dispatched whenever a QueueReceiver receives a new message.

The API includes a MessageListener interface which can be used to register with the QueueReceiver to receive notification on message arrival by creating a MessageListener, implementing the onMessage() method and registering the listener with the QueueReceiver using the setMessageListener(MessageListener) method. The QueueReceiver will call the onMessage() method of the MessageListener whenever a message arrives.

Once a MessageListener is registered to receive notification from a QueueReceiver, any calls to the synchronous receive methods listed in Table 7-7 on page 141 will fail.

Closing connections

Once you have finished sending and receiving messages, the close() method of each of the objects created should be called.

> **Note:** The MQSeries JMS implementation of QueueConnection close() automatically closes any QueueSessions, QueueReceivers and QueueSenders created from it.

7.4.3 Publish/subscribe API

The JMS API for performing pub/sub messaging is based on the same JMS objects and looks very similar to the PTP API in its usage, however, it includes extra functionality to handle the subscription to topics.

As for the PTP API, the pub/sub API defines interfaces for the JMS objects. These interfaces are listed in Table 7-8.

Table 7-8   Publish/subscribe API interfaces

JMS object	Pub/sub interface
ConnectionFactory	TopicConnectionFactory
Connection	TopicConnection
Session	TopicSession
Destination	Topic
MessageProducer	TopicPublisher
MessageConsumer	TopicSubscriber

The procedures for creating and using the objects are the same as for the PTP API.

Publishing messages to topics

The procedure for publishing a message to a particular topic is very much the same as for sending a message to a queue. The following code is publishing a message.

```
//Example of message publishing using the pub/sub API
// Lookup a TopicConnectionFactory
InitialContext ctx = new InitialContext();
TopicConnectionFactory tcf =
(TopicConnectionFactory)ctx.lookup("jms/stocksTCF");

// Use the TopicConnectionFactory to obtain a TopicConnection which represents
// a physical connection to the JMS Provider
TopicConnection topicConn = tcf.createTopicConnection();

// Create a TopicSession
TopicSession session = topicConn.createTopicSession(false,
Session.AUTO_ACKNOWLEDGE);

// Obtain a Topic
Topic topic = (Topic)ctx.lookup("jms/StocksTopic");
```

```
// create a TopicPublisher
TopicPublisher publisher = session.createPublisher(topic);

// create a message to publish
TextMessage message = session.createTextMessage("IBM 117.80 -1.80");

// publish the message
publisher.publish(message);

// close the TopicConnection. This will close the other resources as well
topicConnection.close();
```

Subscribing to topics

While publishing messages to a topic is analogous to putting a message on a queue, the same is not quite true of subscribing to a topic. A queue only has one client receiving messages from it, so when a message is retrieved from the queue by a client, the JMS provider can consider the message delivered. However, a topic may have many clients subscribed to it. A message posted to the topic is possibly retrieved by all of the subscribing clients.

Durable and non-durable subscriptions

Because our messaging system is asynchronous, so a subscriber may not be permanently attached to receive messages. Therefore, what happens to messages published to a topic while a subscriber is not active? The JMS specification defines two types of topic subscription: non-durable and durable. A non-durable subscription lasts for the duration of the TopicConnection within which it was created. The JMS client only has access to messages published while the non-durable connection is active. A durable subscription is more long-lived. The JMS client may create a durable subscription to a topic and then revive it within a later connection. The JMS provider must return the subscription in the state it was left. Therefore, the JMS client will be able to retrieve any messages published while it was inactive.

Non-durable and durable subscribers are created using one of the methods of TopicSession listed in Table 7-9.

Table 7-9   TopicSubscriber creation methods

Method	Description
createSubscriber(Topic topic)	Creates a non-durable TopicSubscriber object which will receive messages for the specified topic.

Method	Description
createSubscriber(Topic topic, String messageSelector, boolean noLocal)	Creates a non-durable TopicSubscriber object which will receive messages for the specified topic whose properties match the specified messageSelector. If noLocal is true then messages published from the TopicConnection used to create this TopicSession are ignored.
createDurableSubscriber(Topic topic, String name)	Creates a durable TopicSubscriber object which will receive messages for the specified topic. The name argument identifies this durable subscriber.
createDurableSubscriber(Topic topic, String name, String messageSelector, boolean noLocal	Creates a durable TopicSubscriber object which will receive messages for the specified topic whose properties match the specified messageSelector. If noLocal is true then messages published from the TopicConnection used to create this TopicSession are ignored. The name argument identifies this durable subscriber.

When creating a durable subscriber, the name argument is used by the JMS provider to identify the durable subscription and can be used in subsequent TopicConnection lifetimes to pick up the subscription.

Receiving messages

Once a durable or non-durable subscription is obtained, messages can be retrieved using the same methods described in Table 7-7 on page 141 for PTP applications. A MessageListener notification mechanism may also be employed. The TopicConnection start() method must be called first to enable message delivery.

Closing connections and unsubscribing durable topics

As for the PTP API, calls to the close() method of the TopicConnection and any TopicSessions, TopicPublishers and TopicSubscribers created from it should be made.

Durable TopicSubscribers will be maintained by the JMS provider when closed. If the durable TopicSubscriber is no longer required, it can be unsubscribed. This must be done after the TopicSubscriber is closed. The following shows the code required to perform unsubscription:

```
//Unsubscribing a durable TopicSubscriber
```

```
TopicSubscriber ts = session.createTopicSubscriber(topic, "IBMStock");
... // receive some messages

// close the durable TopicSubscriber
ts.close();
// unsubscribe it
session.unsubscribe("IBMStock");
// close the TopicConnection if we are finished with it
topicConn.close();
```

7.4.4 JMS and EJBs

JMS is very likely to be used within session EJBs to allow interaction with message-aware, back-end systems. However, there are some technical issues with the JMS specification and EJB1.1 which are important to understand.

The problems exist when the ability to receive asynchronous messages from a queue or topic subscription is required. As we have seen, the JMS specification allows a MessageListener object to register for notification from the JMS provider when a message is received. However, the EJB 1.1 specification does not allow direct calls to EJB instances, instead, all access to an EJB must be through the home and remote interfaces. Additionally, to handle message arrival in a non-blocking fashion normally involves the creation of threads. The EJB 1.1 specification prevents the creation of new threads within EJBs.

This limitation effectively prevents the ability to use EJBs to act as asynchronous message receivers. The EJB 2.0 specification overcomes these limitations by introducing the concept of message-driven beans. However, for now, there are ways to get around this problem by implementing non-EJB code to register for message notification which also acts as an client to forward the messages to the EJB. An example of such an implementation for the WebSphere Application Server 3.5.3 and MQSeries 5.2 JMS support is detailed in the whitepaper: WebSphere JMS/JTA support for MQSeries Overview, available from `http://www.ibm.com/software/webservers/appserv/whitepapers.html`

> **Note:** WebSphere Application Server Advanced 4.0 will provide message-driven beans and JMS listener support as a technology preview.

7.4.5 JMS exceptions

The JMS API includes an exception class, JMSException. This differs slightly from most normal exception classes in that it provides access to the underlying JMS provider exception produced when an error occurs. JMSException is thrown by most of the JMS API methods and can be handled to extract the underlying exception by using the getLinkedException() method, as shown in the following code.

```
//Handling JMSException
try{
    ... set up the QueueConnection, QueueSession and Queue objects
    QueueSender sender = session.createQueueSender(queue);
    sender.sendMessage(message);
    String messageID = message.getJMSMessageID();
} catch (JMSException jmse){
      jmse.printStackTrace();

      // retrieve the linked exception
      Exception e = jmse.getLinkedException();
      // do something with it
      if (e != null){
          System.err.println("Linked exception:");
          e.printStackTrace();
      }
} finally{
      // close any open objects
}
```

7.5 WebSphere and MQSeries JMS implementation

MQSeries provides support for writing message aware Java applications through the MA88 SupportPac. Java classes are provided for writing applications that can access an MQM locally, using the MQSeries binding transport, or remotely using the MQSeries client transport. The Java classes provide access to the underlying MQSeries Message Queueing Interface (MQI)

In addition, the MA88 SupportPac provides a JMS 1.0.2 implementation based on MQSeries as the JMS provider. This implementation can support the sending and receiving of JMS messages from within JTA transactions. This allows J2EE applications within WebSphere to use distributed two-phase commit transactions which coordinate access to both databases and MQSeries.

> **Note:** The MA88 SupportPac is available from
> `http://www.ibm.com/software/ts/mqseries/txppacs` for Windows NT/2000,
> AIX, Solaris, HP_UX and Linux.

The MQSeries JMS implementation supports both PTP and pub/sub models.

7.5.1 MQSeries JMS administered objects

The MQSeries JMS implementation defines the following ConnectionFactory and
Destination objects:

Table 7-10   MQSeries JMS administered objects

Object	Description
com.ibm.mq.jms. MQQueueConnectionFactory	Non-transactional QueueConnectionFactory.
com.ibm.mq.jms. MQTopicConnectionFactory	Non-transactional TopicConnectionFactory.
com.ibm.ejs.jms.mq. JMSWrapXAQueueConnectionFactory	Transactional QueueConnectionFactory for use in distributed transactions. Only used with the bindings transport.
com.ibm.ejs.jms.mq. JMSWrapXATopicConnectionFactory	Transactional TopicConnectionFactory for use in distributed transactions. Only used with the bindings transport.
com.ibm.mq.jms.MQQueue	Queue object representing an MQSeries queue.
com.ibm.mq.jms.MQTopic	Topic object representing an MQSeries pub/sub topic.

The MA88 SupportPac includes the jmsadmin utility for administration of these
objects within a JNDI namespace, such as that provided by WebSphere. The
jmsadmin utility allows the creation and configuration of these objects. Figure 7-6
shows jmsadmin displaying administered objects created within the WebSphere
JNDI namespace. In this case we have two QueueConnectionFactory objects
defined, non-transactional (MORTQCF) and transactional (MORTTXQCF) and
one Queue object (Q1).

Figure 7-6 The jmsadmin utility

The objects are stored within the "jms/piggy" context. We can implement JMS code to access these objects, as shown in the following code.

```
//Accessing MQ JMS administered objects from WebSphere JNDI
InitialContext ctx = new InitialContext();
QueueConnectionFactory factory =
(QueueConnectionFactory)ctx.lookup("jms/piggy/MORTQCF");
Queue queue = (Queue)ctx.lookup("jms/piggy/Q1");
```

> **Note:** In WebSphere 4.0 the naming ENC has changed and lookups within the WebSphere JNDI namespace should be prefixed with java:comp/env/
> Therefore, the lookups used in the previous example will now be:
>
> ```
> java:comp/env/jms/piggy/MORTQCF
> java:comp/env/jms/piggy/Q1
> ```

Although normally accessed through JNDI, it is possible to use the MQSeries JMS ConnectionFactory objects explicitly to create Connections. The following code illustrates this for an MQQueueConnectionFactory. Creating administered objects in this way requires extra set up information to be supplied which would normally be set for administered objects by the MQSeries administrator through jmsadmin.

```
//Using MQSeries ConnectionFactory objects explicitly
MQQueueConnectionFactory factory = new MQQueueConnectionFactory();

// set up the factory properties. These are MQ specific settings. Refer to
// the MQSeries Using Java manual for more information.
factory.setTransportType(JMSC.MQJMS_TP_CLIENT_MQ_TCPIP);
```

```
factory.setQueueManager("aspender.QM_aspender")
factory.setHostName("aspender");
factory.setChannel("SYSTEM.DEF.SVRCONN");

// Use the factory to create a connection. In this case we supply a
// userid and password. This is only supported when using the client
// bindings.
QueueConnection conn = factory.createQueueConnection("userid", "password");
```

> **Important:** Explicitly using the MQSeries JMS implementation classes in this way makes your JMS code non-portable. You should always aim to place administered objects within a JNDI namespace and access them using the standard JNDI lookup.

7.5.2 Distributed transactions and MQSeries JMS

The JMS specification does not require that JMS messages should act as part of a distributed transaction. However, it does specify that if a vendor implements this functionality then it should use the JTA/JTS transaction model.

The MQSeries JMS implementation can act as a resource manager for distributed transactions coordinated by WebSphere. This includes both container-managed and bean-managed transactions.

Within a distributed transaction, message sends and receives are transacted as far as the local queue manager the connection object is associated with. That is, the JMS part of the transaction will be deemed successful if a message is successfully put onto or got from a queue or topic. From the point where a message is placed onto a queue, the assured messaging capability of MQSeries can be trusted to ensure message delivery.

> **Note:** When using MQSeries JMS within a distributed transaction, the MQSeries MQQueue or MQTopic objects should be configured for persistent messages using jmsadmin. This is the default setting. This ensures that MQSeries will write messages to persistent storage for recovery in the event of queue failure.
>
> Message persistence incurs a performance overhead and may be turned off when performing non-transactional messaging and assured delivery is not required.

To perform JMS operations from within a distributed transaction, the JMS client must have access to a transaction capable connection object. This can be made available by setting up a JMSWrapXAQueueConnectionFactory or JMSWrapXATopicConnectionFactory object within the WebSphere JNDI namespace.

Standard JMS client code within a session bean method which is defined as transactional, for example, with the TX_REQUIRED transaction attribute or which implements bean-managed transactions, and by using a transactional connection factory, will be able to participate within a distributed transaction. Any failure to put or get a message from a queue or topic within the boundary of the transaction will result in the transaction being rolled back.

7.5.3 Implementing transactional request/reply scenarios

Within a single transaction, messages can be placed onto, or received from a queue. In addition, we can receive a message and reply to it. However, if we wish to send a message and then receive a reply, we must implement this within two separate transactions. This is simply because the outbound message put will not be committed until the transaction is committed. Therefore, it is impossible to expect a reply within the same transaction.

In this scenario, the put and get must be implemented in separate methods. Normally, there is no need to transact the receiving of the reply, especially within Web applications as we've noted previously; therefore the get method will normally be marked as TX_NOT_SUPPORTED.

7.5.4 Connection pooling

The MQSeries JMS implementation is built on top of the MQSeries Java classes. These classes provide connection pooling for the underlying connection to the MQSeries queue manager initiated when a QueueSession or TopicSession is created. By utilizing this connection pooling performance can be increased through the saving of connection set-up time.

By default connection pooling is performed if there is more than one connection (JMS Session object) open simultaneously. If so, when a Session object is closed, the connection it uses is kept open and made available to other Session objects. However, if there is only one Session object at any time, connections will never be pooled. In this case it is possible to use non-JMS MQSeries code to configure the connection pool. This is described in *MQSeries Using Java*, SC34-5456.

7.6 PiggyBank scenario — mortgage payment

We will use the JMS API, and the MQSeries JMS implementation to develop the mortgage payment functionality of the PiggyBank application.

PiggyBank has merged with an existing mortgage provider, and wants to enable customers who hold a PiggyBank account and a mortgage with the ability to make mortgage payments straight from their account over the Web. The mortgage provider's systems are not integrated with the PiggyBank EJB based system. To submit a mortgage payment, the PiggyBank application must transfer money from the user's account to a central holding account to which the mortgage provider has access. It must also send a message to the mortgage provider application telling it that a payment is waiting.

The functionality is implemented within a session bean. This session bean must perform a transfer between the user's account and the central holding account, and must also put a message onto a queue to tell the mortgage application to take the money from the holding account. This is a prime example of the need for a distributed transaction across the database and JMS. If the database update fails, then the message to the mortgage application should not be sent. Likewise, if the message send fails, the database update should be rolled back.

The messaging scenario we use will be a datagram message in a point-to-point model, as shown in Figure 7-3 on page 130. This is a fire-and-forget message to which we do not expect a reply. We rely on the MQSeries assured delivery capability to deliver the message to the mortgage application.

To implement the functionality we will create the following objects:

► **MortgagePaymentBean**: A stateless session bean which carries out the payment request within a transaction.

► **JMSHelper**: A utility helper class that provides caching of JMS administered objects.

7.6.1 Developing the JMSHelper utility class

As discussed in 7.4.1, "Elements of a JMS system" on page 133, ConnectionFactory, Connection and Destination objects are thread safe and may be cached within an application for reuse. The aim of the JMSHelper class is to provide caching of these objects.

Normally, you should implement a caching policy for expensive objects, that is, ones which either require a lot of resource to create, or which maintain "expensive to create" resources themselves. In addition, any objects created through JNDI lookups should ideally be cached to avoid the expense of repeated JNDI lookups.

In the case of JMS, the most obvious candidates for caching are the administered objects — ConnectionFactories and Queues/Topics. In most cases these can be safely cached within a singleton class. The case for caching Connection objects is less clear-cut. The JSP specification states that Connection objects are considered heavyweight, and therefore it is desirable to cache at some point. The question that must be considered is where. If Connection objects will be used to receive messages then it is best to cache at a fine grained level, for instance, per session bean to maintain the ability to start and stop message retrieval from individual beans.

In our case, we only ever send messages, and therefore do not wish to maintain control over when the JMS provider will send messages. We create a Connection object and leave it in stopped mode.

The JMSHelper class is therefore implemented as a singleton and we will cache QueueConnectionFactory, QueueConnection and Queue objects across the whole JVM. The code for maintaining the singleton class is shown in the following code.

```
//JMSHelper class
package com.ibm.itso.j2eebook.helpers;

import java.util.*;
import javax.rmi.*;
import javax.naming.*;
import javax.jms.*;

/**
 * Provides access to JMS administered objects.
 *
 * This class is implemented as a singleton.
 *
 * @author: Adrian Spender
 */
public class JMSHelper {

    // singleton instance variable
    private static JMSHelper instance = null;

    private Hashtable qcfs = new Hashtable();
    private Hashtable qconns = new Hashtable();
    private Hashtable queues = new Hashtable();
```

```
/**
 * JMSHelper default constructor.
 */
    private JMSHelper() {
        super();
    }

/**
 * Singleton accessor method.
 *
 * @return Singleton instance of JMSHelper
 */
    public static JMSHelper singleton() {
        if (instance == null) {
            instance = new JMSHelper();
        }
        return instance;
    }
}
```

The class has three hashtables for maintaining references to the cached objects.
To provide access to the cached objects, we implement three methods,
getQCF(), getQueueConnection() and getQueue(). The code for these is
contained within the following examples.

```
//JMSHelper getQCF() method
/**
 * Retrieves a QueueConnectionFactory object for the given name. The
 * object is cached after the first lookup and retrieved from the
 * Hashtable on subsequent invocations.
 *
 * @param String The JNDI name of the administered object.
 *
 * @return The JMS administered object.
 *
 * @exception NamingException Thrown if lookup fails.
 */
    public QueueConnectionFactory getQCF(String qcfName) throws
NamingException{

        QueueConnectionFactory qcf = (QueueConnectionFactory)qcfs.get(qcfName);

        if (qcf == null) {
            // get InitialContext and perform lookup
            qcf = (QueueConnectionFactory)NamingHelper.singleton().
                getInitialContext().lookup(
                NamingHelper.singleton().getJNDIName(qcfName));
            qcfs.put(qcfName, qcf);
```

```
        }
        return qcf;
    }
```

The method accepts a String argument which represents the QueueConnectionFactory we want to look up. The method firstly checks for an already cached copy within the hashtable. If none is found then it performs a JNDI lookup to retrieve the object. The JNDI lookup is performed via another helper class, NamingHelper, which provides access to a cached JNDI context, and also the JNDI name of the object we want to retrieve.

To aid application maintenance, JNDI names are stored in a properties file, PiggyBankJNDINames.properties. This contains keys which can be looked up to provide actual JNDI names, allowing the application code to be independent of JNDI naming conventions.

> **Note:** In WebSphere Advanced 4.0, it is possible to abstract out JNDI names from the code by specifying resource references for JNDI objects including EJB Homes, DataSources and JMS administered objects. This provides the same benefits as the property file used here, with the added benefit of easier maintenance.

The contents of PiggyBankJNDINames.properties relevant to JMS are shown in the following code.

```
# JMS objects defined in PiggyBankJNDINames.properties
#
# JMS administered objects. If you do not have MQSeries installed
# set the value of piggy.MortgageQCF to 'dummy'
#
# No MQ:
#piggy.MortgageQCF = dummy
#
# Non-transactional QCF
#piggy.MortgageQCF = jms/piggy/MORTQCF
#
# Transactional QCF
piggy.MortgageQCF = jms/piggy/MORTTXQCF
#
# Queue
piggy.MortgageQueue = jms/piggy/Q1
```

The values of the properties correspond to the administered objects shown in Figure 7-6 on page 149. Here, the piggy.MortgageQCF property is configured to return the JNDI name of the transactional QueueConnectionFactory object.

Once the QueueConnectionFactory has been obtained, it is stored within the hashtable for later retrieval.

The methods for obtaining QueueConnection and Queue objects are both similar to getQCF(). The getQueueConnection() method is slightly different, in that it does not perform a JNDI lookup. Rather, it uses an already cached QCF to create the QueueConnection. This is shown in the following code.

```
// JMSHelper getQueueConnection() method
/**
 * Retrieves a QueueConnection object for the given QCF name. The
 * object is cached after the first lookup and retrieved from the
 * Hashtable on subsequent invocations. If we don't already have
 * a QCF created then we will create one.
 *
 * @param String The JNDI name of QCF used to create the connection.
 *
 * @return The QueueConnection.
 *
 * @exception NamingException Thrown if lookup fails.
 */
    public QueueConnection getQueueConnection(String qcfName) throws
NamingException, JMSException{

        QueueConnection qconn = (QueueConnection) qconns.get(qcfName);

        if (qconn == null) {
            qconn = getQCF(qcfName).createQueueConnection();
            qconns.put(qcfName, qconn);
        }
        return qconn;
    }
```

The method accepts the QCF name as an argument and uses it to call getQCF().

As we have discussed, it is important to ensure that JMS objects are closed when no longer required to free up the resources they consume. One disadvantage of implementing caching of QueueConnection objects within this singleton is that there is no way to guarantee the closing of the cached objects. The cached objects are available until the JVM is shut down, in which case the resources held by MQSeries on behalf of the QueueConnection object may not be properly released. We implement a finalize() method within JMSHelper to call the close() method of each cached object. However, there is no guarantee that this method will be called.

7.6.2 Developing the mortgage payment session bean

The actual business logic of the mortgage payment use case is implemented in a stateless session bean. The bean performs the following tasks:

► Performs a funds transfer from the users account to the mortgage holding account, "888-8888". This transfer is performed using the BankTransfer stateless session bean described in 5.2, "Session beans" on page 50.

► Puts a message onto a queue to inform the mortgage application of the pending payment.

The bean provides a method, named performPayment() on its remote interface. This method accepts two arguments:

► String accFrom: The account the money is to be paid from
► java.math.BigDecimal amount: The amount of money to pay

The method's transaction attribute is TX_REQUIRED specifying that it is run within a transaction. The attribute for the relevant methods of BankTransferBean, AccountBean and TransferRecordBean must also be set to TX_REQUIRED.

The funds transfer method of performPayment() follows:

```
// MortgagePaymentBean performPayment() method - funds transfer
/**
 * Performs a mortgage payment. This involves performing a transfer from the
 * users bank account to the "holding" account 888-8888 and then sending
 * a JMS message to a queue which tells the mortgage application that
 * a payment is pending. In reality we just send a very simple text message
 * which should really be XML.
 *
 * The TX attribute of this method should be TX_REQUIRED to ensure that the
 * db update and message are dealt with as a unit of work. Alternatively we
 * could have used bean-managed persistence. The TransferAccessBean
 * AccountBean and TransactionRecordBean beans must also be TX_REQUIRED.
 *
 * We return the message ID from the JMS send. In reality we should return
 * an identifier which would allow the user to query the payment.
 *
 * @param accFrom The account from which the payment will be made.
 * @param amount The amount to pay.
 *
 * @return String The message ID of the JMS message send.
 *
 * @exception com.ibm.itso.j2eebook.exceptions.InsufficientFundsException
 *         If the user doesn't have the money!
 * @exception com.ibm.itso.j2eebook.exceptions.MortgagePaymentException
 *         If the payment fails for any reason.
 * @exception JMSException Whoops.
```

```
*/
    public String performPayment(String accFrom, BigDecimal amount)
    throws Exception {

        String result = null;

        // use a call to the BankTransfer session bean to take the desired
        //amount from the users account and place it in the bank's holding
        //account.
        BankTransfer transferBean = getTransferHome().create();
        transferBean.transferFunds(accFrom, "888-8888", amount);
```

The BankTransferBean home interface is accessed through a method called
getTransferHome:

```
// MortgagePaymentBean getTransferHome method
public BankTransferHome getTransferHome() throws NamingException {

        BankTransferHome transferHome =
        (BankTransferHome) javax.rmi.PortableRemoteObject.narrow(
                    HomeHelper.singleton().getHome("piggy.BankTransfer"),
                    BankTransferHome.class);

        return transferHome;
}
```

This uses the HomeHelper utility class to access the home interface.
HomeHelper provides cached access to homes.

Once the home interface is retrieved, the create() method is called to obtain the
remote interface. This is then used to call the transferFunds() method.

After performing the funds transfer, we send the message. The code to do this
has already been discussed in previous sections. It is shown in below with
appropriate comments.

```
        // Obtain the QueueConnection from the JMS helper
        QueueConnection qconn =
        JMSHelper.singleton().getQueueConnection("piggy.MortgageQCF");

        // Create a session. Do not make it transacted. The concept of
        // transacted sessions is different to distributed transactions.
        // We are only sending one message within this session so it doesn't
        // matter.
        QueueSession session =
        qconn.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);

        // Obtain the Destination object from the JMS helper
        Queue queue = JMSHelper.singleton().getQueue("piggy.MortgageQueue");
```

```
            // Create a QueueSender
            QueueSender sender = session.createSender(queue);

            // Create a message to send to the queue...
            String mess = "Pay " + amount + " from account " + accFrom;
            TextMessage message = session.createTextMessage(mess);

            // ...and send it
            sender.send(message);

            // retrieve the message ID which we will send back to prove
            // this worked...
            result = message.getJMSMessageID();

            // Close the sender and session objects. We leave the cached
            // QueueConnection open.
            sender.close();
            session.close();

            return result;
        }
```

7.6.3 Running the example

In this section we will show the example running within WebSphere Application Server. The account and transaction record entity beans are set up to use a JTA enabled JDBC driver, and the JMS ConnectionFactory is transactional.

Figure 7-7 shows the input page for the mortgage payment use case.
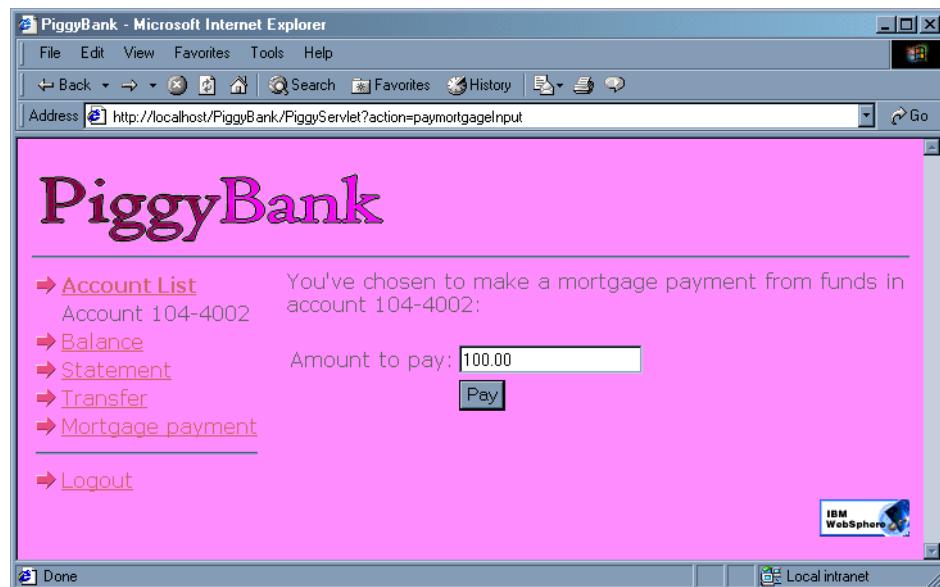
Figure 7-7   Mortgage payment example — input page

We will pay $100.00 from account 104-4002. This amount will be transferred to the bank's 888-8888 holding account and a message sent to MQ. The output of the transaction is shown in Figure 7-8.
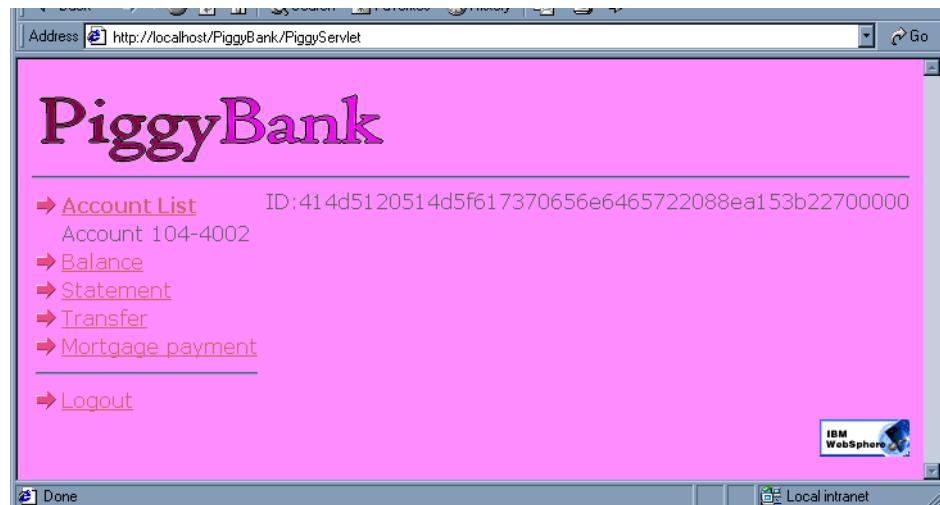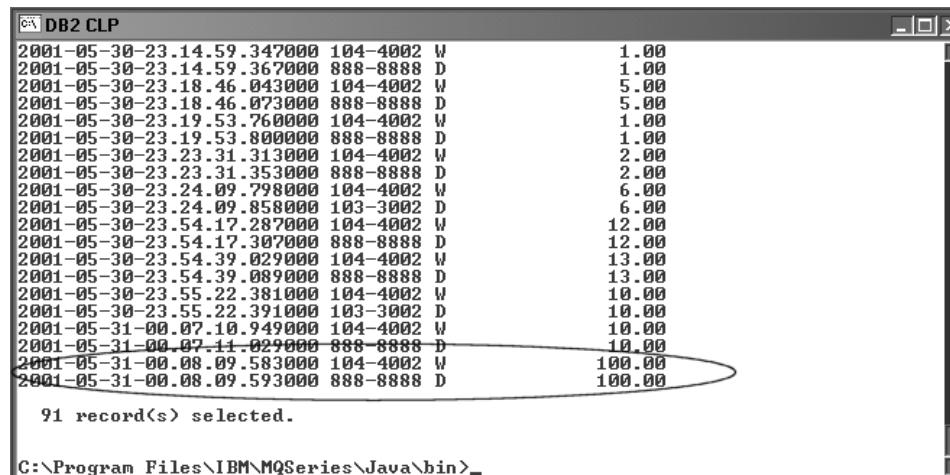
Figure 7-8   Mortgage payment example — output

To investigate the success of the transaction, we can perform an SQL query on the underlying database. The query performed is:

```
select * from itso.transrecord
```

The results of running the query are shown in Figure 7-9. The withdrawal and deposit transactions related to our query are highlighted:

Figure 7-9   Mortgage payment example — database transaction records

The MQSeries message can be viewed through the MQSeries Explorer and is shown in Figure 7-10. Note the message ID highlighted matches that shown in the output screen in Figure 7-8.
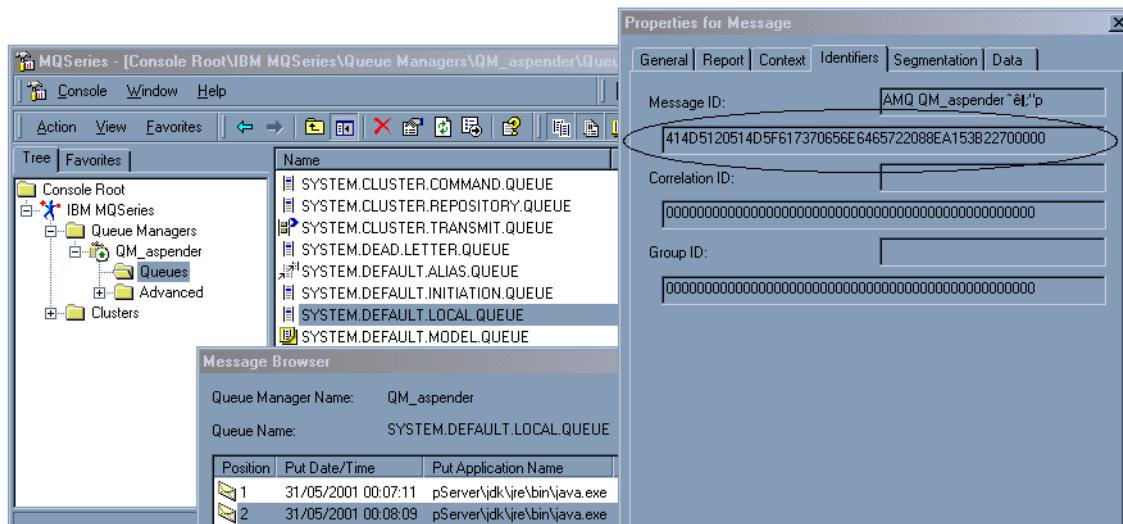
Figure 7-10   Mortgage payment example — MQ message output

A transaction rollback can be forced, for instance, by shutting down the
MQSeries queue manager. In this case, the database is not updated and a
message not sent. The user is given an explanatory message.

The Web container

8

Servlets

In this chapter we introduce the technology of Java servlets and discuss their role in an enterprise application.

Basically, a servlet is a component used to generate dynamic content. Behind that simple definition we will see that many concepts are embedded and must be taken into account.

The following topics are discussed in this chapter:

► The basic concepts of the Java servlet technology.

► The differences between Java Servlet Specifications 2.1 and 2.2. We describe what a Web application is and what it contains.

► How to develop servlets and how to integrate them into an enterprise level application (following the example of the PiggyBank application).

► How WebSphere Application Server provides a runtime environment to the servlets.

8.1 General points

The Servlet API 2.2 is one of the Java standard extensions that must be included in a J2EE platform. A J2EE platform must implement all the functionalities listed in this specification and provide a container to manage all the services required by the APIs.

8.1.1 Definition

A servlet is a Java class and defined as a Web component, which is invoked from a URL, and the output can be dynamically generated HTML, XML, or TEXT.

Moreover, the Java Servlet Specification defines a set of instructions for the components of a J2EE platform to implement. It also deals with all the concepts and the objects associated to the servlet technology.

8.1.2 How it works

Java language was not created just in order to program client-side applications such as applets. A servlet is a good example of how to bring the power of the Java language to the server.

Servlets are server-side components: a servlet is typically a Java component that is deployed, managed and executed on a J2EE compliant server. Servlets are platform independent, they are able to run on the hardware already installed, whatever the operating system is.

The servlet mechanism is based upon the request/response paradigm (see Figure 8-1).
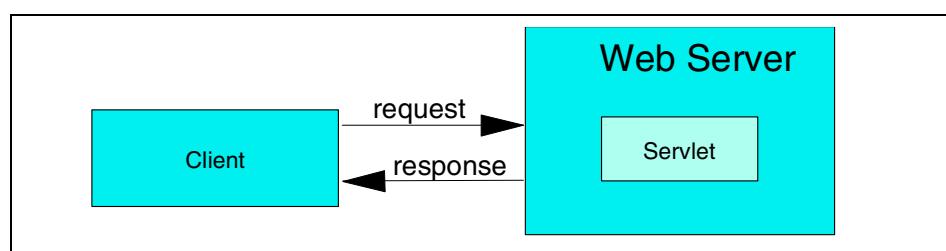
Figure 8-1   Request Response flow for a servlet call

A client makes an HTTP request to a Web server. This Web server manages that request and sends it to the servlet container. Considering its configuration the servlet container (or servlet engine) calls the servlet with objects representing the request and the response.

The servlet uses the request to get the information submitted by the user that makes the request and then executes its business logic. The servlet generates data that is sent back to the client.

Typically, there is only one instance of a servlet class loaded in the container at any time. Clients of the same application asking for the servlet will all deal with the same instance of the object, unless the servlet was reloaded meanwhile.

Servlets are also protocol independent: they are mainly used with the HTTP protocol, but their functionalities can be extended to other protocols. A servlet's output can be XML content or of any other type. However, we will focus on servlets dealing with Web clients in our examples.

8.1.3 A servlet's life cycle

A servlet's life cycle is defined in the Java Servlet Specification. The different states of a servlet are shown in Figure 8-2.
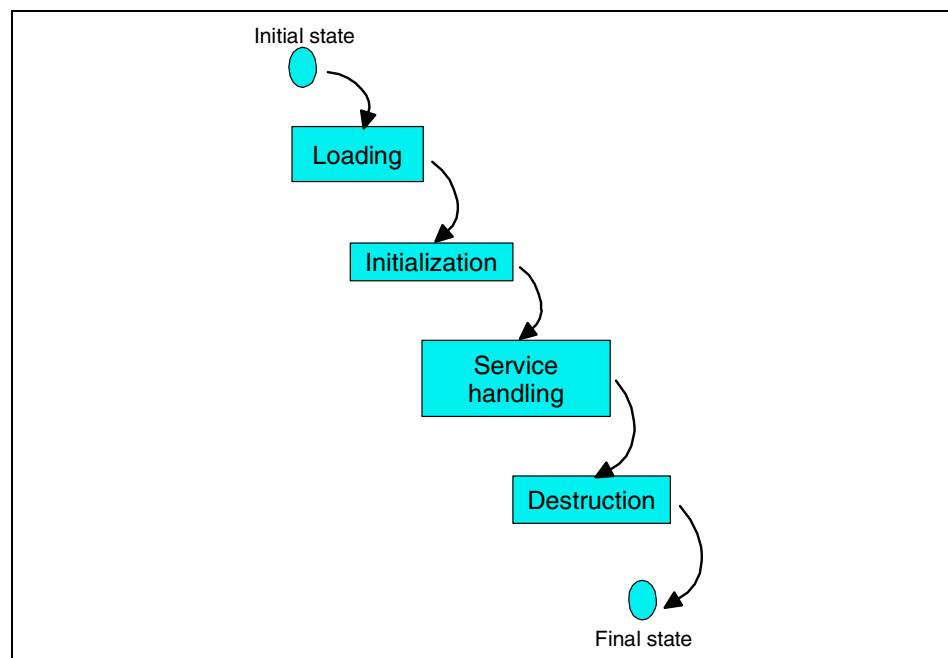
Figure 8-2   A servlet's life cycle

All along its life, a servlet interacts closely with its container. Following the J2EE specification, the container is in charge of providing a runtime execution for the components. It is also responsible for handling a set of services such as concurrency, administration or security.

The servlet container is implied in all the steps through the servlet's life cycle:

1. **Loading**: The servlet container uses a class loader to locate the class file of the servlet and to instantiate it. This operation follows the Java class loading process. An instance of the servlet class (and only one) is then available at the end of this step.

2. **Initialization**: The container activates the servlet by calling one of its method, the *init* method. The *init* method must be implemented by a servlet class. This method is called once by the container and is used to initialize a set of parameters for the servlet instance. The servlet is then ready to handle requests from clients.

3. **Service handling**: Each client request generates a call to the *service* method of the servlet class. This method can either generate the response and transfer it to the Web server (via the servlet container) or dispatch the request to another component within the container. The *service* method is called with two parameters:

   – Request: This object encapsulates all the data coming from the client and must implement the *ServletRequest* interface (the *HttpServletRequest* interface for Web clients).

   – Response: This object encapsulates all the data that must be sent back to the client. It implements the *ServletResponse* interface (the *HttpServletResponse* interface for Web clients).

   Multiple threads may be running the service handling (that is to say the service method) at the same time unless the servlet class implements the SingleThreadModel interface. It could be useful to manage persistency and in that case the container may use multiple instances of the same servlet class.

4. **Destruction**: The servlet container calls the *destroy* method of the servlet to shut down the servlet. The method can be used to release resources. Any request to the servlet then cannot be handled.

Many vendors provide an implementation of the servlet container through their application server. In the IBM WebSphere Application Server, the servlet container is called servlet engine.

8.1.4 The servlet's environment

A servlet executes on a servlet container. This container is responsible for providing a set of support services to its servlets. The environment of a servlet also contains other components. The behavior of all these components (including the container) is described by the Java Servlet Specification.

The servlet container

The servlet container (or servlet engine) acts as a fundamental layer between the Web server and the servlets. It interacts with the Web server from whom it receives requests. Considering its configuration the container has to create an object implementing the *HttpServletRequest* interface and to delegate the request to the servlet instance.

The servlet instance executes its business logic (through a *service* method call). The container then transmits the response to the Web server.

For each request on a same servlet instance the container is responsible for creating and managing a new thread to serve the request.

The servlet container provides the same services whatever the platform is. It allows the servlet developer to focus on the logic of its component, without considering deployment issues. A servlet based application can be deployed and run into any J2EE platform.

A servlet container must at least support the HTTP 1.0 protocol. To provide all its services a servlet container interacts with other Java objects, such as the servlet configuration and the servlet context objects.

The servlet configuration

This object must implement the *ServletConfig* interface and is provided by the servlet container as a parameter of the *init* method of the servlet. The servlet container is responsible for providing a *ServletConfig* object.

The servlet container uses this object to pass configuration information to its servlet instances such as the name and the value of a parameter located in the servlet container. The *ServletConfig* object is accessible for the servlet through its *getServletConfig* method.

Moreover, a *ServletConfig* object allows the access to a *ServletContext* object.

The Servlet context

This object, that must implement the *ServletContext* interface is provided to the servlet within the *ServletConfig* object, passed by the container as a parameter of the *init* method. All along its life cycle a servlet can access its servlet context through the *getServletContext* method.

The servlet container is also responsible for providing an implementation of this interface.

The servlet context provides information about its environment to a servlet. It can be shared by a group of servlets and gives access to resources. A servlet can have access to the following elements through its servlet context:

- ▶ **A set of attributes**: A set of name/object pairs that can be shared between servlets and other components to maintain persistency. A servlet can either add or remove an attribute or simply change its content. A servlet can also access its init parameters which are name/value pairs.

- ▶ **Resources**: Servlets sharing the same context can also share resources such as files. They can access these resources via their servlet context.

- ▶ **A log file**: A servlet can write a set of events to a log file.

- ▶ **A request dispatcher**: This object, created and managed by the servlet container allows a servlet to forward its request to another servlet or to include the content of another servlet into its own response. It allows servlets sharing the same context to communicate with each other.

There can be several servlet context objects in a servlet container at the same time, each managing its pool of servlets. Each servlet instance is running in one of these contexts and all the servlet instances running in the same context will share the same resources.

Following that, a servlet class may have more than one instance in a container, from the moment that each of these instances is attached to a different servlet context. We will get back to that subject when dealing with the Web application concept (8.2, "Differences between APIs 2.1 and 2.2" on page 170).

Finally, the servlet context is the way for the servlet to interact with its environment (its container, but also other resources) without having to reconsider its own logic. Every servlet container has at least a default servlet context.

8.2 Differences between APIs 2.1 and 2.2

The Java Servlet Specification 2.2 was released at the end of the year 1999 and included in the J2EE Specification at the same time. Comparing to the Specification 2.1 it brought in new concepts and introduced several changes on the APIs. It also supplied clarifications on various points.

8.2.1 Web application concept

The most important change in the Java Servlet Specification 2.2 is the introduction of a new concept, the Web application. Here is a definition. A Web application contains Java classes, HTML files, GIF files, and any other files that will be used in the server side application. The format is standardized in the Java Servlet Specification and compatible between multiple vendors.

The Specification also defines the elements a Web application may contain and proposes a hierarchical structure for the contents of a Web application. Many servlet containers (including one from WebSphere Application Server) support this structure, although, it is just a recommendation (see 8.2.2, "Web application archive concept" on page 175 for more details).

Elements contained within a Web application

The following items may be included in a Web application:

► Servlets and JavaServer Pages.

► Utility classes: Standard Java classes may be packaged in a JAR (Java ARchive) file. JAR is a standard-platform-independent file format for aggregating files (mainly Java classes files).

► Static documents: HTML files, images, sounds, videos, etc. This term includes all the documents a Web server is able to handle and to provide to client requests.

► Client side applets, beans, and classes.

► Descriptive meta information which ties all of the above elements together.

Any Web resource can be included in a Web application. This also concerns compressed archives, or other data such as XML files.

Directory structure

The directory structure for a Web application requires the existence of a WEB-INF directory located at the root of the structure (see the following section, "The WEB-INF directory" on page 172, to know more about its content).

There is not any other requirements for the directory structure of a Web application. We recommend that you group the resources in separate directories such as images for all the pictures, html for all the static html pages, and so forth. The directory structure of a basic Web site will fit in for all the static documents.

One of the main interests of the Web application is to group a set of Java components (servlets, JSPs and their utility classes) to allow them to share a set of resources. This resources can be either static resources (files or archives), a set of parameters, and so on.

The WEB-INF directory

This directory contains all the Java classes and support classes that will not be served directly to the client as simple resources such as static files. Located at the root directory of the Web application structure the WEB-INF folder is not part of the public document tree of the application.

A WEB-INF directory may look like the following (Figure 8-3).
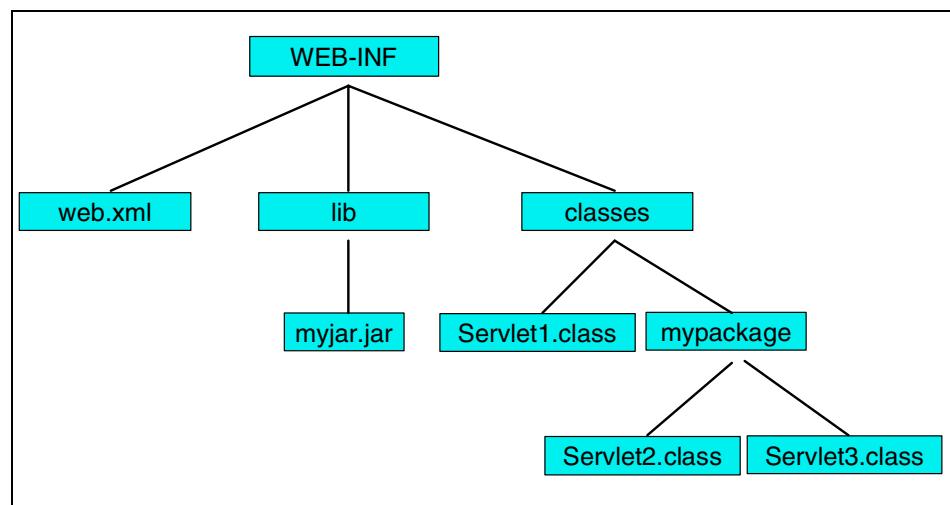
Figure 8-3   The WEB-INF directory: a sample structure

The required elements are:

► **web.xml**: This file is required and is called the deployment descriptor (see the following section, "The deployment descriptor" on page 173).

► **lib**: This directory is required and is used to store all the Java classes used in the application. Mainly, we will have JAR files in this subdirectory.

► **classes**: This directory is also required. Typically, servlet classes and utility classes for the servlets compose this directory. It is possible to keep a package structure in this directory and to put class files under several subdirectories of the classes directory (as it is done for the Servlet2.class file in the subdirectory mypackage in Figure 8-3).

It is important to consider that the files of the WEB-INF directory may contain some of the application logic; that is why they cannot be requested and provided by the Web server as any other static Web resource. The access to these resources must be controlled through the servlet container, within the application server.

The deployment descriptor

A deployment descriptor is an XML-based text file whose elements describe how to assemble and to deploy the unit into a specific environment. In the case of a Web application, the web.xml file located at the root of the WEB-INF directory describes how to deploy the Web application and its contents to the servlet container within the Web server.

The file must conform to a DTD (Document Type Definition) given in the Java Servlet Specification. This file gives configuration information that is useful during the deployment, such as:

► A set of initialization parameters for the servlet context object. All the servlets running in the same Web application have the same servlet context and can access this attributes.

► Servlet information: For each servlet we will find its name, its full qualified class name or the path leading to its file for a JSP, its parameters for initialization (these parameters belong to the servlet and won't be shared with other servlets). We will also find if the servlet must be loaded when the server is starting up and a set of mapping rules, allowing a servlet to respond to requests for a set of URLs.

► A list of default files to send to the user when his request does not specify any file in a directory. Typically index.html, index.htm, and index.jsp may be used here.

► A list of error pages to send to the user when its request generates an error. HTTP errors or Java errors (exceptions) may be handled in that way.

► JSP tag libraries used within the Web application.

► References to external resources and enterprise beans: These references then will be easily located by the servlets. It can include entries in the JNDI lookup table, location of an enterprise bean interfaces or simply a declaration of an application's environment entry.

► Security constraints: the access to some resources can be limited and configured in the deployment descriptor. These resources are grouped into collections and the constraints are on these collections.

► Any other type of information useful during the deployment to the servlet container. It may include the description of the application (including if it is distributable), some timeout settings, additional information about mime mapping, and so on. The complete DTD of the deployment descriptor for a Web application is available at:
`http://java.sun.com/j2ee/dtds/web-app_2_2.dtd`

The main interest with the deployment descriptor file is that the application configuration is specified independently from the server. It clearly simplifies the deployment process because the same application can be deployed into different servers without having to review its content.

The Web application development tools (such as WebSphere Studio) are generally able to generate the deployment descriptor, using a graphical tool.

A Web application in an application server

A Web application is rooted at a specific path within a Web server, called the context path. All the requests from clients beginning with this path will be given to the application.

All the Java components of the same Web application share information and have access to the same pool of resources. This information is encapsulated in the servlet context, provided by the servlet container (see "The Servlet context" on page 169). Following that there is one and only one servlet context per Web application. Servlet instances of a Web application will point to the same *ServletContext* instance.

A servlet container may contain several Web applications: every Web application is rooted to a specific path and handles requests pointing to this path. The servlet container manages a servlet context for each of these Web applications.

The servlet container is responsible for encapsulating the data sent by the client into an *HttpServletRequest* object. This task includes identifying the following elements, obtained from the client's request for a URL (Uniform Resource Locator):

- **Context path**: The path prefix leading to the Web application. This context path is associated with the servlet context of the application.

- **Servlet path**: The section of the requested URL that directly corresponds to the mapping which activated this request. Typically, mapping consists in associating a servlet or a resource with a set of URLs. For each request pointing to one of these URLs, the servlet will be invoked.

- **Path info**: The last part of the URL, not identified as part of the servlet path.

The concatenation of these three elements is the request URI (Uniform Resource Identifier) path or URL path. It represents the URL passed by a request to invoke a servlet.

Typically, with our WEB-INF directory (Figure 8-3 on page 172) deployed on the Web server handling URLs requests, such as http://www.webapp.com at the specific root mywebapp, we can have the following request encapsulation:

```
Requested URL: http://www.webapp.com/mywebapp/servlet/Servlet1
Request URI path: /mywebapp/servlet/Servlet1
Context path: /mywebapp (it represents the path prefix where the Web
application was deployed)
Servlet path: /servlet/Servlet1
```

The servlet will have access to these elements, that can be configured on the deployment descriptor or later on the servlet container for each Web application.

8.2.2 Web application archive concept

Associated to the Web application concept, the Java Servlet Specification 2.2 introduces a new file format named WAR (Web ARchive) to handle a Web application. WAR files can be packaged and signed.

All the elements contained within a Web application can be compressed into a unique file. This file will be used for the deployment.

The Java Servlet Specification 2.2 doesn't give any additional requirements for the packaging of the Web application. The hierarchical directory structure defined in "Directory structure" on page 171 must be supported. Standard tools for compressing and uncompressing directories may be used to create a WAR file.

WAR files must be treated differently than JAR files by the application server as they extend JAR files. A Web application not only contains components, it also contains other resources (see "Elements contained within a Web application" on page 171); that is where it differs from a JAR file.

The main interest of a WAR file is that a Web application can be packaged on a platform-independent way. This gives the opportunity to deploy a Web application in many servers. The use of the deployment descriptor allows for customizing the application on each of these servers.

An application server (such as WebSphere Application Server) is able to handle Web archive files. It is able to build an application and to run it from a WAR file, after deploying it. We recommend for the servlet container to support the structure within the archive as a runtime representation.

WebSphere Application Server Version 3.5.2 supports WAR files.

8.2.3 Other changes

Several changes were made to the Java Servlet Specification between Version 2.1 and Version 2.2. However, the structure of the packages is still the same (Figure 8-4), the content is also the same (the only change is that the interface HttpSessionContext is now deprecated).

The changes concern the methods of several classes and the concepts brought in or clarified. Among these concepts we can talk about the response buffering, the distributable servlets, the request dispatcher invocation, and so forth.

Another thing to notice in this release is that the term servlet engine was replaced by servlet container, as the term container is part of J2EE specification. However, both terms have the same meaning.

For a complete view of the changes, we also consult the Java Servlet Specification 2.2 at `http://java.sun.com/products/servlet/download.html`. Figure 8-4 gives an overview of the Java servlet APIs.

```
☐ 🗎 Servlet API Classes 2.2
  ☐ 🗂 javax.servlet 2.2
    ⊞ C  GenericServlet ᴬ🗎 2.2
    ⊞ I  RequestDispatcher 🗎 2.2
    ⊞ I  Servlet 🗎 2.2
    ⊞ I  ServletConfig 🗎 2.2
    ⊞ I  ServletContext 🗎 2.2
    ⊞ C  ServletException 🗎 2.2
    ⊞ C  ServletInputStream ᴬ🗎 2.2
    ⊞ C  ServletOutputStream ᴬ🗎 2.2
    ⊞ I  ServletRequest 🗎 2.2
    ⊞ I  ServletResponse 🗎 2.2
       I  SingleThreadModel 🗎 2.2
    ⊞ C  UnavailableException 🗎 2.2
  ☐ 🗂 javax.servlet.http 2.2
    ⊞ C  Cookie 🗎 2.2
    ⊞ C  HttpServlet ᴬ🗎 2.2
    ⊞ I  HttpServletRequest 🗎 2.2
    ⊞ I  HttpServletResponse 🗎 2.2
    ⊞ I  HttpSession 🗎 2.2
    ⊞ C  HttpSessionBindingEvent 🗎 2.2
    ⊞ I  HttpSessionBindingListener 🗎 2.2
    ⊞ I  HttpSessionContext 🗎 ⚠ 2.2
    ⊞ C  HttpUtils 🗎 2.2
    ⊞ C  NoBodyOutputStream 🗎 2.2
    ⊞ C  NoBodyResponse 🗎 2.2
```

Figure 8-4   Java servlet APIs overview

The response buffering

A servlet is now able to control the buffer it sends back to the server. Methods for setting and getting the buffer size, resetting, and flushing the buffer were added to the ServletResponse interface. A method giving information about the commit is also available.

The response buffering allows a container to improve efficiency, by controlling the buffer size instead of using a buffer created by the server with a default size.

It also allows a servlet to handle errors. Using a buffer gives to the servlet the opportunity, to write output without committing it. If the servlet finds an error it may change the header of the response and sends another response, such as an error page.

The distributable servlet

A distributed application is an application (a Web application, for example) made up of distinct components running in separate runtime environments, usually on different platforms connected via a network (J2EE specification).

A distributable servlet is a servlet that is part of a distributed application. This servlet will have an instance running on each of these runtime environments. There will be one servlet context per servlet container attached to the application.

Information cannot be shared through the servlet context object for servlet instances running on different servers. Other mechanisms, external to the Web server must be implemented to allow the servlets to have access to shared information.

Another point the Specification is Session management in a distributed environment: all the requests that are part of the same session must be managed by the same container. Moreover, all the objects placed into a session must implement the Serializable interface.

Servlets running on the same container will still have access to resources, through their servlet context.

Request dispatching

Request dispatching consists in forwarding the processing of a request from a servlet to another one or including the output of a servlet in a response. It can't be done without obtaining a *RequestDispatcher* object.

In the APIs 2.1, such a reference was obtained using the *getRequestDispatcher* method of the servlet context. The Specification offers two new ways of getting a request dispatcher:

- ► **In the *ServletContext* interface**: A servlet may obtain a reference to a request dispatcher using the registered name of the component concerned with the dispatching. A servlet can get its registered name using a new method added to the *ServletConfig* interface, *getServletName*. Using this name may be simpler than using the full URI path initially used. Here is an example of how to use these two methods:

```
Get a request dispatcher by name
// get the name of the instance of the servlet
String name = getServletConfig().getServletName();

/* forward to the servlet the request and the response through the
request dispatcher */
getServletContext().getNamedDispatcher(name).forward(request, response);
```

- ► **In the *ServletRequest* interface**: A servlet may use a relative path to obtain a request dispatcher. With the directory structure shown in Figure 8-3 on page 172, we may see the following line of code in the *Servlet1* servlet:

```
request.getRequestDispatcher("/servlet/mypackage.Servlet2");
```

Changes in the APIs

Several other methods were added to the Java servlet APIs. Here are the most useful ones:

- ► **Interface ServletContext**: getInitParameter and getInitParameterNames. This method allow to access parameters that can be shared by all the components of a Web application. These parameters can be set in the deployment descriptor or either in the servlet container.

- ► **Interface HttpServletRequest**:

  - getContextPath: To obtain the context path of a Web application within which the servlet is running (see "A Web application in an application server" on page 174).

  - isUserInRole: Determines if a particular user is in a given security role.

  - getUserPrincipal: Returns a java.security.Principal object for this user.

- ► **Interface HttpServletResponse**: addHeader, addIntHeader, and addDateHeader. This method improves the control over HTTP headers by allowing the creation of various headers with the same header name.

- ► **Interface HttpSession**: Method for accessing the attributes has been replaced to follow the naming conventions of the APIs.

8.2.4 What about Java Servlet APIs 2.3 release?

In May 2001, the proposed final draft for the version 2.3 of the Java Servlet Specification was published. This draft is likely to become Java Servlet Specification Version 2.3. It is available at:
`http://java.sun.com/products/servlet/download.html`

Two important concepts are introduced in this new release:

► **Servlet filtering**: the Specification defines a filter as a reusable piece of code that can transform the content of HTTP requests, responses and header information. It provides a new interface in the API called *Filter* to implement the functionalities of a filter.

   A Filter has access (through its *doFilter* method) to the request object (encapsulating the data provided by the client) before the resource it invokes and can act on it. It has also access to the response object after the resource processing. A filter is attached to a resource, it will be called for each client call on this resource.

   Filters are part of a Web application, they must be configured in the deployment descriptor (a special set of tags is defined) and added to the WAR file. The container is responsible for instantiating the filters and giving them the requests and the responses they are supposed to handle.

► **Application lifecycle event**: the Specification introduces the concept of application level events. It defines event listeners as Java classes following the JavaBean design.

   These events concern the ServletContext and the HttpSession interfaces. When events occur on these objects (for example, the creation of when a Web application is starting up or a change in the attributes) the listener is called and may perform actions.

   Event listeners are part of a Web application, a new set of tags has been created in the deployment descriptor to control their deployment within a container. The class files of the listeners must be added to the Web application.

8.3 Sample servlets

As we explained in the previous sections, a servlet is a Java class implementing an interface. In a Web based application, servlet classes will implement the javax.servlet.http.HttpServlet interface. Now we are going to see how to program sample servlets, before seeing how we can use servlets in an application containing other resources.

This section intends to give examples of basic servlets. People who already know about servlets may skip to 8.4, "Servlets in an enterprise application" on page 186.

8.3.1 A simple servlet

The first example of a servlet code is the SampleServlet1 servlet shown here:

```
//SampleServlet1 servlet
package com.ibm.itso.piggybank.servlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SampleServlet1 extends HttpServlet {
public void service(HttpServletRequest request, HttpServletResponse response)
                                throws IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML><TITLE>Sample Servlet: a basic Servlet</TITLE>");
    out.println("<BODY><H1>My first Servlet:</H1>");
    out.println("This is a very simple Servlet, isn't it ?");
    out.println("</BODY></HTML>");
    out.close();
    }
}
```

A few comments about the code

This previous servlet is quite simple, it has no special functionalities. Let's look at the code step by step.

```
//SampleServlet1 package declaration
package com.ibm.itso.piggybank.servlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

The servlet is part of the package com.ibm.itso.piggybank.servlets and uses some classes of the packages listed in the lines beginning with the keyword import.

```
//SampleServlet1 class and service method declaration
public class SampleServlet1 extends HttpServlet {
public void service(HttpServletRequest request, HttpServletResponse response)
                                throws IOException {
```

The preceding list shows the class declaration and the service method declaration. In all our Web examples, servlets will be Java classes extending the HttpServlet class. The service method is the method called on each client request to the servlet (see 8.1.3, "A servlet's life cycle" on page 167).

```
//SampleServlet1 service method code
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("<HTML><TITLE>Sample Servlet: a basic Servlet</TITLE>");
out.println("<BODY><H1>My first Servlet:</H1>");
out.println("This is a very simple Servlet, isn't it ?");
out.println("</BODY></HTML>");
out.close();
```

The preceding code shows the content of the service method. We handle the response object, responsible for sending the response back to the client. In this example we generate a simple HTML page.

Running and invoking our servlet

Servlets need a runtime environment (see 8.1.4, "The servlet's environment" on page 168) to execute. They need to be loaded, instantiated and initialized into a servlet container.

On each client request to our servlet, the service method will execute and send a response. To invoke our servlet, we need to enter a URL in a browser, such as:

```
http://host/servlet/com.ibm.itso.piggybank.servlets.SampleServlet1
http://host/mywebapp/servlet/com.ibm.itso.piggybank.servlets.SampleServlet1
```

The second form invokes the servlet if it has been added to a Web application rooted at the /mywebapp context within the Web server.

Our servlet generates the following output (Figure 8-5).

Figure 8-5   SampleServlet1 output

8.3.2 Servlets processing an HTML form

For the second example we will deal with a servlet generating an HTML form. A second servlet will process the parameters sent by the client in this form to display them on a page. Once again, this example intends to demonstrate the power of servlets through a simple example.

SampleForm servlet

The SampleForm servlet displays an HTML form to the user requesting the servlet. This example shows the code for the SampleForm class.

```
//SampleForm servlet code
package com.ibm.itso.piggybank.servlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SampleForm extends HttpServlet {

public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    performTask(request, response, "POST",
"com.ibm.itso.piggybank.servlets.SampleFormHandler");
```

```
}

public void performTask(HttpServletRequest request,
                        HttpServletResponse response,
                        String method,
                        String url) throws IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML><TITLE>Sample Servlet: an HTML form</TITLE>");
    out.println("<BODY><H1>Let's talk about you:</H1><HR>");
    out.println("<H4>Would you please fill in the following form ?</H4>");
    out.println("<FORM METHOD=\"" + method + "\" ACTION=\"" + url + "\">");
    out.println("Your name ");
    out.println("<INPUT TYPE=TEXT NAME=clientName><BR>");
    out.println("Your favorite sport: <BR>");
    out.println("<INPUT TYPE=RADIO NAME=clientSport"+
                "VALUE=\"football\">Football<BR>");
    out.println("<INPUT TYPE=RADIO NAME=clientSport"+
                "VALUE=\"tennis\">Tennis <BR>");
    out.println("<INPUT TYPE=RADIO NAME=clientSport+
                "VALUE=\"other\" CHECKED=\"TRUE\">Other<BR>");
    out.println("<INPUT TYPE=\"SUBMIT\" NAME=\"SENDPOST\" NAME=\"Send\">");
    out.println("<INPUT TYPE=\"RESET\" NAME=\"RESET\" NAME=\"Reset\">");
    out.println("</FORM></BODY></HTML>");
    out.close();
}
}
```

This servlet may look like the SampleServlet1 servlet: the packages included are the same, it also extends the HttpServlet class and it handles a response object, generating an HTML page.

However, the SampleForm class has several differences:

► It handles a doGet method: this method is part of the HttpServlet class. It is called by the server through the service method to handle a GET request. Typically when clients request a URL in their browser, they make a GET request to the server. In our example, the doGet method is called by the service method of the servlet (the service method called will be the one of the superclass HttpServlet). It then forwards its work to the performTask method.

► It displays an HTML form: the HTML code generated by this servlet is a form. It invites the user to enter values in its browser. A form is composed of fields, each field has a name and a value (a field acts as a variable in a program). This values can be changed by the user and sent to the server when clicking on the Submit button. On this click, the user sends a request to the servlet com.ibm.itso.piggybank.servlets.SampleFormHandler.

The servlet may be called using the following URL:

```
http://host/servlet/com.ibm.itso.piggybank.servlets.SampleForm
```

After completion the screen may look like the one shown in Figure 8-6.
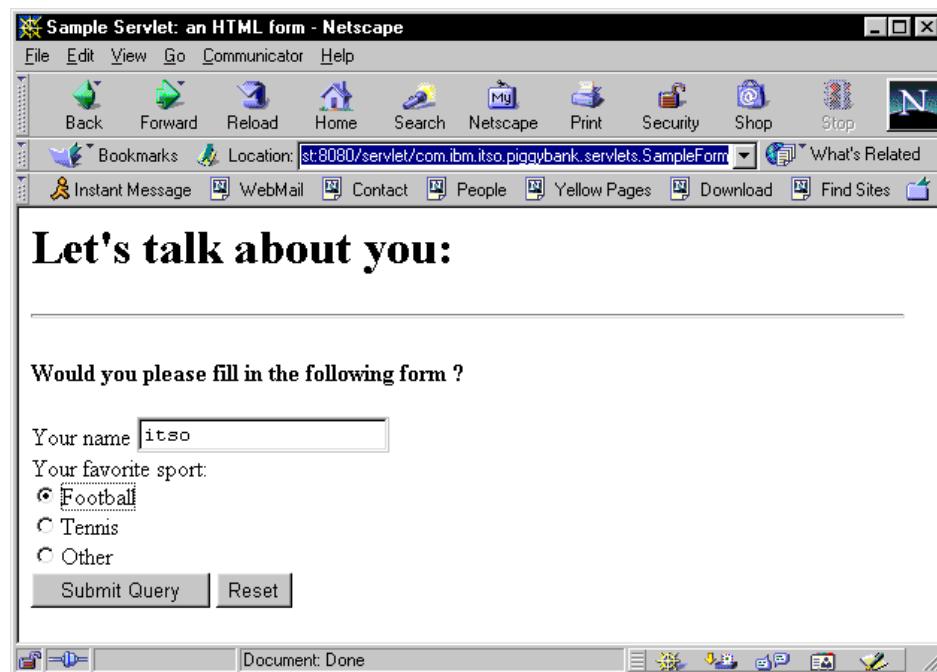
Figure 8-6   SampleForm servlet output

SampleFormHandler servlet

The SampleFormHandler servlet is called when the user clicks on the Submit Query button. Its role is to process the data sent by the client and to display it. The following shows the code for the SampleFormHandler class:

```
//SampleFormHandler servlet code
package com.ibm.itso.piggybank.servlets;

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

public class SampleFormHandler extends HttpServlet {
    private int count = 0;

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

```
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    // Retrieving the parameters
    String clientName = new String(request.getParameter("clientName"));
    String clientSport = new String(request.getParameter("clientSport"));

    // Displaying the output
    out.println("<HTML><TITLE>Sample Servlet: the results of the "+
                                            "form</TITLE>");
    out.println("<BODY><H1>Hello " + clientName + "</H1>");
    out.println("<H4>I see that you like " + clientSport + "</H4>");
    out.println("<H4>This Servlet has been called " + ++count +
                                            " time(s)</H4>");
    out.println("</BODY></HTML>");
    out.close();
}
}
```

The main characteristics of the SampleFormHandler class are:

► It handles a doPost method: this method handles POST requests coming from clients. It is also called via the service method of the servlet.

   The main difference between a GET and a POST request is that a POST method is designed to send data to a server. Both requests are used to get a resource from the server, but with a GET request, the parameters sent to the server must be contained in the requested URL.

► It retrieves data from the HttpServletRequest object: the method getParameter is used to retrieve the data sent by the client on its request to the server. A parameter is given, the name of the variable (this name is the same as the one indicated in the HTML form displayed to the user).

► It increments a counter: the servlet has a variable, called count incremented on each call to the component. This instance variable will remain active during the life of the servlet.

Figure 8-7 shows the output of the SampleFormHandler servlet.
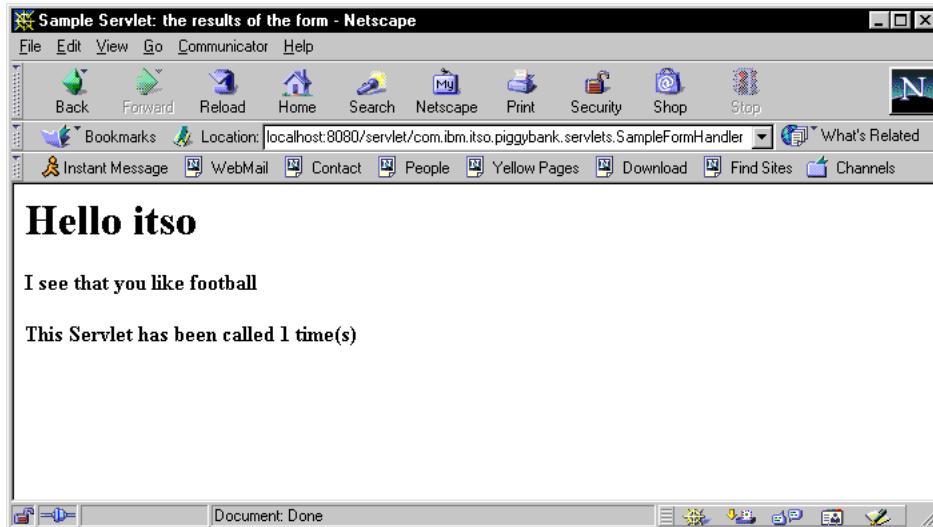
Figure 8-7 SampleFormHandler servlet output

We have seen in these first examples how to program simple servlets to generate dynamic content. A servlet is a Java class, extending the superclass HttpServlet. In its service method, we can handle client requests and generate simple output. Now we are going to see how servlets can be used in the enterprise applications and how they interact with other components such as EJBs, JSPs. Also, we are going to see the role they must play in such applications.

8.4 Servlets in an enterprise application

An enterprise application contains many components: servlets, JSPs, EJBs. It also manages a pool of static resources. It is important to specify the role of each of the components to understand how to use them in an application.

8.4.1 The role of servlets

We saw in Chapter 4 the layered architecture of an application. Servlets are part of the controller layer.

Servlets in the controller layer

The role of a servlet is to handle a request from a client and to generate the output. To do these tasks, a servlet will communicate with other components such as EJBs for the business logic and JSPs for the presentation of the output.

Servlets are the link between the client request and the model. They must interface with the components performing the business logic of the application.

Typically, a servlet will have to transform a client request in an object that is understandable for the model. It also determines which components are required to perform the request and call these components.

Once the business logic is performed, the servlet will have to interface with the presentation layer, providing an object used to generate the output. So the servlet also manages the link between the model and the presentation and may decide to delegate the output to an HTML page.

So the first role of a servlet is to act as a controller: a servlet is responsible for taking parameters from the client, it then gives these parameters to the appropriate components which handle the business logic. Finally, the servlet takes the result back and uses them to give a response to the user. The servlet can either return an html page to the user or forward the presentation task to a JSP.

Servlets manage sessions

The servlets will also have to manage other functionalities such as providing session management and authentication. In a layered architecture, a servlet represents the entry point of the application.

As the HTTP protocol is stateless, a server does not automatically track the requests coming from the same user. In a Web application, where a user may perform a logical series of requests, it is essential to identify the user at each request. Identifying a user means the application must recognize the user every time he or she makes a request.

Session management is the responsibility of the controller in an application. According to the previous requests made by the user, the servlet will process its request to the appropriate business logic and responds, contacting the appropriate presentation components.

Session management deals with maintaining the data collected in the trips to the application. It must be ensured by the servlets and managed outside the servlet, so that every servlet of the application may have access to this information. This information, called "session data", will help the controller in its tasks.

The second main role of a servlet is to manage user sessions, creating them and maintaining them. A servlet will typically put data in a session and retrieve it on request.

8.4.2 Controller design

Implementing a controller may uncover several issues. How many servlets do we need? How must they relate to each other?

General points

Considering the complexity of the application and the use cases, that is to say the various functionalities from a user point of view, the common answer is to implement one servlet per use case, and a controller servlet.

However, there can be alternate solutions, and even within this solution various design options can be chosen.

For the PiggyBank application example, we have decided to implement just one servlet, the *ControllerServlet*. This servlet handles all the client requests.

According to the request of the client (login, display a balance for an account, make a transfer, and so on) the servlet will check the session state for this user and forward its work to a method. The controller will handle errors, redirecting the user to other pages within the application.

This choice of implementing a method per functionality may be discussed. However, we mainly intend to show how a servlet can act as a controller in an application and how it can manage user sessions, and how a servlet handles data within the session.

The ControllerServlet overview

The signature of our servlet is shown in the following code:

```
//ControllerServlet definition and methods
package com.ibm.itso.j2eebook.servlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import com.ibm.itso.j2eebook.dataaccess.*;
import com.ibm.itso.j2eebook.databeans.*;
import com.ibm.itso.j2eebook.servlets.util.*;
import com.ibm.itso.j2eebook.exceptions.*;

public class ControllerServlet extends HttpServlet {

public String getServletInfo () {}

public void init () {}
```

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

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

public void performTask (HttpServletRequest request, HttpServletResponse
response) throws IOException {}

public void performLogin (HttpServletRequest request, HttpServletResponse
response, HttpSession session) throws IOException {}

public void performAccountList (HttpServletRequest request, HttpServletResponse
response, HttpSession session) throws IOException {}

public void performAccountDisplay (HttpServletRequest request,
HttpServletResponse response, HttpSession session) throws IOException {}

public void performOperationInput (HttpServletRequest request,
HttpServletResponse response, HttpSession session) throws IOException {}

public void performStatementDisplay (HttpServletRequest request,
HttpServletResponse response, HttpSession session)throws IOException {}

public void performMortgagePayment (HttpServletRequest request,
HttpServletResponse response, HttpSession session) throws IOException {}

public void performTransfer (HttpServletRequest request, HttpServletResponse
response, HttpSession session) throws IOException {}

public void performLogout (HttpServletRequest request, HttpServletResponse
response, HttpSession session) throws IOException{}

}
```

As explained in Chapter 4, our servlet handles all the requests coming to the application. We will see the body of the methods and the code included in it. Figure 8-8 shows the overview of how the controller handles requests.
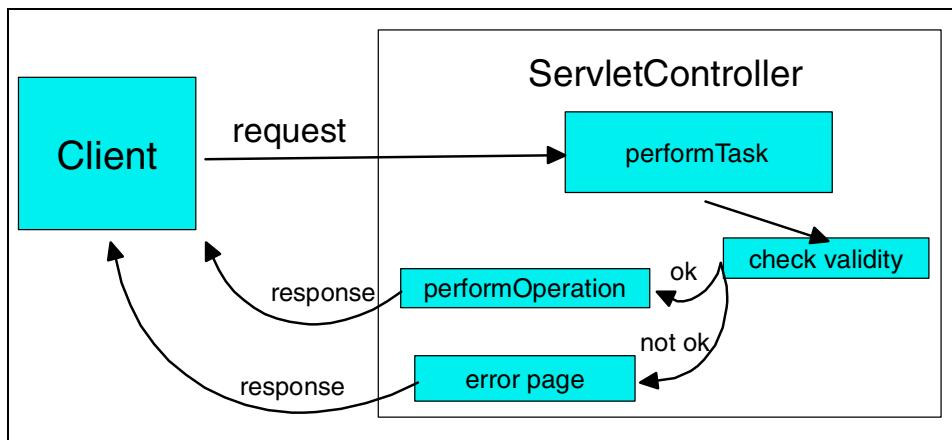
Figure 8-8   ControllerServlet overview

When the client makes a request to the application, the *performTask* method of the *ControllerServlet* is called. This method checks if the user is allowed to perform this request, considering the state of its session and the request itself.

If the user is allowed, the *performtask* method calls another method which handles the request and communicates with the appropriate components to generate the response. Each method performs a functionality of the application.

This method will use Helper objects to communicate with the EJBs to get the data concerning the client. This data may be stored in a database or anywhere else. The controller does not need to know about it, it just interfaces these components via the Helper objects. The controller then transfers the data to the client by putting them in the response object or in the session object. The presentation task is given to a JSP file which will generate a dynamic page, using the objects provided by the controller.

The methods are (we will see their code later; the interest here is to understand the action they perform):

► *performLogin*: This method logs on a user and obtains information about him or her from the enterprise data.

► *performAccountList*: This method is called to display a list of the accounts for a current user.

► *performAccountDisplay*: This method is provided for displaying information concerning a specific account.

► *performOperationInput*: This method is called to obtain a form for either, transferring funds from an account to another, or paying a bill with the money from a specified account.

- ▶ performStatementDisplay: This method is called to display all the last transactions for a specified account.

- ▶ *performMortgagePayment*: This method invokes the component to pay an amount from a specified account.

- ▶ *performTransfer*: This method calls the components to perform the transfer of funds between two accounts belonging to the same client.

- ▶ *performLogout*: Logs out the user.

All of these methods take three parameters. Request and response are the objects encapsulating the data provided by the client and the data that will be sent back to the client. The session encapsulates the data contained in the user session.

8.4.3 Session management issues

Managing the user session is essential for an application. We saw in a previous section ("Servlets manage sessions" on page 187) the interest for an application to trace the users request.

We give an overview of the technologies that may be used to get information from a client. Obtaining information from a client request is the basis to manage sessions.

Then we deal with the *HttpSession* interface provided by the Java Servlet API, because this interface helps to manage sessions.

The approaches listed below are not independent, they may be used together to build a reliable application and some typically relate to other.

Cookies

The simplest technology to recognize a user is the *cookie*. A cookie is composed of a name and a value, it is stored in the client browser and can be retrieved on the server when the client comes back.

The Java Servlet APIs 2.2 provide a class in the package *javax.servlet.http* to implement a cookie. The following two examples show how to simply use these APIs to send a cookie to a browser.

```
//ServletCookie servlet class
package com.ibm.itso.j2eebook.servlets.util;

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

```
public class ServletCookie extends javax.servlet.http.HttpServlet {
    public static String URL_SERVLET_COOKIE =
"/servlet/com.ibm.itso.j2eebook.servlets.util.ServletCookie2";

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

public void performTask(javax.servlet.http.HttpServletRequest request,
javax.servlet.http.HttpServletResponse response)
        throws IOException {

    // How to create a cookie
    Cookie cookie = new Cookie ("userName","user1");

    // How to add it to the response
    response.addCookie(cookie);

    // Displays simple output
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML><TITLE>Servlet Cookie</TITLE>");
    out.println("<BODY><H1>Welcome to the servlet cookie :</H1>");
    out.println("Click here to see your cookie : ");
    out.println("<A HREF=\"" + URL_SERVLET_COOKIE + "\">here </A>");
    out.println("</BODY></HTML>");
    out.close();
}
}
```

The *ServletCookie* class simply handles a request by displaying standard HTML output and adding a cookie in its response. The HTML page contains a link so the user can call *ServletCookie2*.

```
//ServletCookie2 servlet class
package com.ibm.itso.j2eebook.servlets.util;

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

public class ServletCookie2 extends javax.servlet.http.HttpServlet {
    public static String COOKIE_NAME = "userName";

public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    performTask(request, response);
}
```

```
public void performTask(javax.servlet.http.HttpServletRequest request,
javax.servlet.http.HttpServletResponse response)
      throws IOException {

   // How to retrieve cookies from a request
   Cookie [] cookies = request.getCookies();


   // Displays simple output
   response.setContentType("text/html");
   PrintWriter out = response.getWriter();
   out.println("<HTML><TITLE>Servlet Cookie 2</TITLE>");
   out.println("<BODY><H1>Welcome to the servlet cookie :</H1>");
   out.println("<H3>Here is your cookie  : </H3><BR>");

   // Displays all the cookies
   for (int i=0; i < cookies.length; i++) {
      if (cookies[i].getName().equalsIgnoreCase(COOKIE_NAME)) {
         out.println(" Cookie name : <B>" + cookies[i].getName() +
                                              "</B><BR>");
         out.println(" Cookie value : <B>" + cookies[i].getValue() +
                                              "</B><BR>");
      }
   }
   out.println("</BODY></HTML>");
   out.close();
}
}
```

The *ServletCookie2* class uses the API methods to retrieve an array of *Cookie* from the request object. It then displays standard HTML output to show the name and the value of the cookie sent by the user in its request.

We can see through this example that the cookie technology is very simple to use. The Java servlet APIs provide a simple way to create cookies and to retrieve them from a user request. Moreover, JavaScript and CGI programs can also use cookies as they are passed within every client request. Cookies are accessible for the Web server that sends them, and not for the others.

However, cookies must be used with parsimony as they present several disadvantages. Among them, cookies are limited in terms of size, to 4 Kb generally (20 cookies per Web server and 300 total).

Cookies can't be used to store important data. Cookies also represent a client-side approach to an application problem, so we do not recommend for you to use them to store session data.

Finally, Web browsers can refuse them. In these cases an alternate solution must be provided.

HTML hidden field

An alternative to the use of a cookie may be to handle a special type of field in an HTML form, the *hidden* field. As its name indicates, this field is transparent to the user and sent with the request as any other parameter of the form.

It has a name and a value that can be specified by a servlet when responding to a client request. If we go back to the example of the servlets handling an HTML form ("Servlets processing an HTML form" on page 182) we could add the following lines in the performTask method of the SampleForm servlet ("SampleForm servlet" on page 182 shows the code of the servlet):

```
out.println("<INPUT TYPE=\"HIDDEN\" ");
out.println("NAME=\"hiddenField\" VALUE=\"value1\">");
```

These lines define a hidden field in the HTML form of which the name is hiddenField and the value is value1. This parameter would be passed as the other parameters of the form to the servlet that processes the request.

For example, the SampleFormHandler servlet (see "SampleFormHandler servlet" on page 184 for its code) could access the parameter with the following line:

```
String hiddenField = request.getParameter("hiddenField");
```

The hidden field mechanism is useful, because it is transparent to the user; however, as cookies, it can only contain String values and not objects.

We do not recommended that you use hidden fields to store session data, as it is also a client-side approach. Indeed, all the hidden fields will cross the network on each client request.

HttpSession

The Java Servlet API provides an interface to identify users and to link the different requests they make, the *HttpSession* interface.

The Web container within the servlets and the JSPs of an application that are running provide an implementation of this interface, and the Web components have access to the methods of this interface.

The container recognizes a user by assigning him or her a unique identifier, the session identifier. An object is associated with this identifier, and the servlet may store, retrieve and remove objects from this user session. The objects belonging to a particular session represent the session data for a particular user.

Every time the client makes a request, the servlet gets the session object associated to him or her through the identifier (a new session is created if the user does not have any) and performs operations on it.

The session identifiers are managed by the container which ensures that there is one session for each user. The API provides methods to destroy a session, typically when a user logs out.

The following shows an example of handling the *HttpSession* objects; this code can be included in the *service* method of a servlet.

```
//Using the HttpSession interface
    HttpSession session = request.getSession(true);
    if (session.isNew()) {

        // perform actions for a new user
        ...

        // set an attribute in the user session
        session.setAttribute("att1", o);
        ...

        }
    else {
        // perform actions for a user identified
        ...

        // retrieve an attribute from the session
        Object o = session.getAttribute("att1");

        ...
        }
```

There are two *getSession* methods in the *HttpServletRequest* interface to provide the session to the servlet:

► public HttpSession getSession(): This method returns the session associated with the request and creates one if there is not one already.

► public HttpSession getSession(boolean create): This method does the same thing unless create is set to false. In that case, the method returns null if there is no session associated with the request.

Here are the most useful methods of the interface HttpSession (we will use it in our application):

► public boolean isNew (): This method returns whether the session was just created or not.

- ▶ public void setAttribute (String name, Object value): This method is used to add the object specified to the session, it will be retrieved by its name.

- ▶ public Object getAttribute (String name): This method is called to retrieve an object by its name.

- ▶ public void invalidate (): This method "destroys" a session. All the objects stored in the session are unbound.

Session objects are stored on the server and provide the standard interface to manage sessions in applications. Session data objects (that is to say, the objects stored in the sessions) must implement the *Serializable* interface, and they are also stored on the server (within the *HttpSession* instances).

We will see in 8.5.2, "Session management in WAS" on page 209 how the application server can manage persistency for the session objects.

The Web container recognizes a user with the identifier attached to its requests. This identifier is stored in the user's browser as a cookie and is the only piece of data stored in the client-side.

However, as we saw in "Cookies" on page 191, some browsers do not accept cookies, and the server must find a solution to recognize the user.

URL rewriting

URL rewriting can be an alternate solution to cookies. It is a technical method that allows you to append data at the end of any link generated by a servlet. The session identifier can be assigned to the URL, so that when a user makes another request to the application, the server is able to get back its session object.

The Java servlet API provide two methods in the HttpResponse interface to encode the URLs. The following code shows an example of how URL rewriting may be used.

```
//URL rewriting example
    // retrieving the user session
    HttpSession session = request.getSession(true);

    // Display HTML content
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    ...

    // Rewriting
    String url = response.encodeURL("/servlet/Servlet2");

    // Make a link in the HTML page displayed to the URL rewritten
```

```
out.println("<A HREF=\"" + url + "\">here </A>");
...
```

The methods used to rewrite a URL are:

▶ **public String encodeURL (String url)**: This method adds the session identifier associated with the client request at the end of the parameter given. For example, it will return something like:

    "`/servlet/Servlet2;jsessionid=Q5L3VDIAAAAAACIBS32XLDI`**".**

We made a link to that URL in our HTML page, if the user requests this resource, the server will be able to retrieve the session object associated to the identifier given in the URL and so on for all the objects within the session.

▶ **public String encodeRedirectURL (String url)**: This method does the same thing as *encodeURL*: it adds the session identifier at the end of the parameter given. It must be used before using the *sendRedirect* method of the *HttpServletResponse* interface. This *sendRedirect* method is used to send a temporary redirect response to a client.

URL rewriting may be a good alternate to the use of cookies. Web containers must provide the classes to implement the encoding methods.

However, using URL rewriting implies for you to rewrite all the links of an application and this can be cumbersome. It also implies for you to avoid linking some parts of the application to static HTML pages, because the session identifier will be lost.

8.4.4 ServletController detail

Now we look at the servlet of our application, the *ServletController*. We explained in 8.4, "Servlets in an enterprise application" on page 186, the role of our servlet and gave an overview of its functionality and its methods (see 8.4.2, "Controller design" on page 188).

Our servlet intends to act as a controller and manages the user sessions. It receives all the requests from the client. The servlet uses *HttpSession* objects to store objects associated to the client session (see "HttpSession" on page 194).

The *performTask* method

The method performing the client request is the performTask method. The following list shows the code for this method.

```
//ControllerServlet servlet: the performTask method
public void performTask(HttpServletRequest request,
                        HttpServletResponse response)
    throws IOException {
```

```
            try {

                HttpSession session = request.getSession(true);
                if (session.isNew()) performLogin(request, response, session);
                else {
                    /* retrieving the parameters of the session
                    current is the current state
                    action is the required state
                    */
                    String current = session.getAttribute("state").toString();
                    String action = request.getParameter("action");

                    if (action.equals("logout")) performLogout(request, response,
                                                                     session);
                    else {
                        /* call the external navigation handler to checck the validity
                        of the operation */
                        if (! BrowseControl.isAllowed(current, action)) {
                            session.invalidate();
                            response.sendRedirect("/index.html");
                        }
                        else {
                        /* The operation is allowed, call to the appropriate method */
                            if (action.equals("login")) performLogin(request, response,
                                                                        session);
                            else if (action.equals("accountList"))
                                        performAccountList(request, response, session);
                            else if (action.equals("accountDisplay"))
                                        performAccountDisplay(request, response, session);
                            else if (action.equals("transferInput") ||
                                        action.equals("paymortgageInput"))
                                        performOperationInput(request, response, session);
                            else if (action.equals("transfer"))
                                        performTransfer(request, response, session);
                            else if (action.equals("paymortgage"))
                                        performMortgagePayment(request, response, session);
                        }
                    }
                }
            }
            catch (Exception e) {
                response.sendRedirect("/index.html");
            }
        }
```

The request process is (see also Figure 8-8 on page 190):

1. The request tries to get the session object associated with the request of the client. If this session is a new session, then it is the first call the client makes to the application, and the servlet calls the performLogin method to login the user.

2. If the session object associated with the user is not new, the servlet retrieves two parameters:

   – From the session: a String value called state. The servlet stores in the session object an attribute corresponding to the last page the client has requested and received. It allows it to know where the client is coming from.

     This is done through the `session.getAttribute("state").toString()` command.

   – From the request: a parameter called action. It represents the action the user wants to perform or where he wants to go.

     This is done through the `request.getParameter("action")` command.

     The action parameter needs to be transferred within the request, it can be done using a hidden field (see "HTML hidden field" on page 194) in the JSP.

3. The servlet then calls the navigation handler to see if, according to the current state of the user session, he is allowed to perform the operation he requested. `BrowseControl.isAllowed(current, action)` calls this external function which returns true or false (see "The navigation handler" on page 199 and Example 8-1 on page 200, for the code of this method). If the user is not allowed, the servlet redirects him to an error page.

4. If the operation is allowed, the servlet calls the method for performing the operation the user requested (see "The ControllerServlet overview" on page 188 for an overview of these methods). The details for these methods are provided in "The operation methods" on page 201).

The navigation handler

The navigation handler is an external object that determines if a user is allowed to perform an action on a particular resource, according to the state of his or her session.

Typically in an application, it answers the question "Can we display page 2 to a user coming from page 1?". In an enterprise application where a user can perform operations such as transferring funds, it is very important to establish navigation rules.

For example, if a user is transferring funds between accounts and uses the "Back" button of his browser, it may perform the operation another time. Session management is built to prevent an application from this kind of problem.

In our application, we can see each page displayed to the user as a node of a directed graph. Some links between the pages are valid while others are invalid.

The navigation handler is implemented to tell us if the link from the current state of the session to the requested action is valid. In case it is not, an error page must be displayed to the user.

Implementations of the navigation handling may differ, according to the applications. We have decided to implement a static class, containing a boolean method which checks if the user requested action is valid according to the state of his or her session. The following list shows the code for this class.

Example 8-1

```
//BrowseControl class code
package com.ibm.itso.j2eebook.servlets.util;

public class BrowseControl {

    public static String LOGIN = "login";
    public static String ACCOUNT_LIST = "accountList";
    public static String ACCOUNT_DISPLAY = "accountDisplay";
    public static String TRANSFER_INPUT = "transferInput";
    public static String TRANSFER = "transfer";
    public static String MORTGAGE_PAYMENT = "paymortgage";
    public static String MORTGAGE_PAYMENT_INPUT = "paymortgageInput";
    public static String STATEMENT = "statement";

    public BrowseControl() {
        super();
}
    public static boolean compare (String string1, String string2) {
        return string1.equalsIgnoreCase (string2);
        }
    public static boolean isAllowed (String state, String action) {
        if (action.equalsIgnoreCase(LOGIN))
            return false;
        if (state.equalsIgnoreCase(LOGIN))
            return compare (action, ACCOUNT_LIST);
        else if (state.equalsIgnoreCase(ACCOUNT_LIST))
            return (compare(action, ACCOUNT_DISPLAY) ||
            compare (action, ACCOUNT_LIST));
        else if (state.equalsIgnoreCase(ACCOUNT_DISPLAY))
            return ! (compare(action, TRANSFER) ||
            compare (action, MORTGAGE_PAYMENT));
```

```
            else if (state.equalsIgnoreCase(TRANSFER_INPUT))
                return (! compare (action, MORTGAGE_PAYMENT));
            else if (state.equalsIgnoreCase(MORTGAGE_PAYMENT_INPUT))
                return (! compare (action, TRANSFER));
            else if (state.equalsIgnoreCase(TRANSFER))
                return (compare (action, ACCOUNT_DISPLAY) ||
                    compare (action, ACCOUNT_LIST) ||
                    compare (action, MORTGAGE_PAYMENT_INPUT) ||
                    compare (action, TRANSFER_INPUT) ||
                    compare (action, STATEMENT));
            else if (state.equalsIgnoreCase(STATEMENT))
                return (! compare(action, ACCOUNT_LIST));
            else if (state.equalsIgnoreCase(MORTGAGE_PAYMENT))
                return (compare (action, ACCOUNT_DISPLAY) ||
                    compare (action, ACCOUNT_LIST) ||
                    compare (action, TRANSFER_INPUT) ||
                    compare (action, MORTGAGE_PAYMENT_INPUT) ||
                    compare (action, STATEMENT));
            return false;
            }
}
```

We store in static variables the different possible state names of the session. The *isAllowed* method performs our navigation logic: it decides if the action required is allowed according to the state. The method is a simple sequence of conditions and comparisons between String identifiers that return a boolean value used by the *ControllerServlet* servlet (see "The performTask method" on page 197).

The *equalsIgnoreCase* method, from the *java.lang.String* class just compares two String values, returning either true if the values are the same (ignoring the case) or false.

The operation methods

The process for the operation methods is (common to all the methods as the servlet acts as a controller):

1. The servlet retrieves the parameters passed within the request.

2. The servlet contacts the appropriate business logic with these parameters. The servlet does not need to know the content of the business logic, it just interacts with Helper components. The business logic components perform their task and send the data back to the servlet.

3. The servlet updates the session object associated with the user who made the request, inserting attributes (concerning the state of the session, but also session data).

4. The servlet sends the data back to a JSP. This JSP handles the presentation logic.

The performLogin method

This method logs a user in the application. The following list shows the code of the method.

```
//ControllerServlet servlet: the performLogin method
public void performLogin(HttpServletRequest request, HttpServletResponse
response, HttpSession session)
      throws IOException {

   try {
      // retrieve parameters
      String clientName = request.getParameter("userid");
      String clientPass = request.getParameter("password");

      // contact business components
      CustomerDataBean cdb =
                CustomerHelper.singleton().getCustomer(clientName);
      Vector accounts = DB2Helper.singleton.getAccounts(clientName);

      // update session object
      session.setAttribute("customer", cdb);
      session.setAttribute("accounts", accounts);
      session.setAttribute("state", "login");

      // forward to a JSP
      getServletContext().getRequestDispatcher(
                "/customerMenu.jsp").forward(request, response);
   }
   catch (Exception e) {
   response.sendRedirect("/index.html");
   }
}
```

The method follows the flow shown in the introduction of the section. The parameters, userid and password, are retrieved from the client request and business components are called with these parameters.

These business components interface with the servlet through the following objects:

► **CustomerHelper**: This component provides to the servlet a CustomerDataBean, object that the JSP will use for displaying dynamic content. This object encapsulates all the data concerning a particular client, its name, address, and so on. This data is retrieved from the database and stored in the session object.

- ▶ **DB2Helper**: This component provides a direct access to the database to get a list of the account numbers for a specified user. The list is returned as a vector object.

Once again, we notice that the servlet just calls these components and transfers them to the session. It does not need to know anything about the implementation of these components.

The servlet then updates the session state and forwards the presentation logic to a JSP, using a request dispatcher.

The AccountList method
The following list shows the code of this method.

```
//ControllerServlet servlet: the performAccountList method
public void performAccountList(HttpServletRequest request,
   HttpServletResponse response, HttpSession session) throws IOException {
   try {
      // Set the current state of the session
      session.setAttribute("state","accountList");

      // Forward to the appropriate JSP
      getServletContext().getRequestDispatcher(
                     "/accountList.jsp").forward(request, response);
      }
   catch (Exception e) {
      response.sendRedirect("/index.html");
   }
}
```

This method just needs to update the state attribute of the session and to send the presentation task to the appropriate JSP. Any other operation is not necessary.

The performAccountDisplay method
The code of this method is performing the operation of displaying the balance for a specified account.

```
//ControllerServlet servlet: the performAccountDisplay method
public void performAccountDisplay(HttpServletRequest request,
   HttpServletResponse response, HttpSession session) throws IOException {
   try {
      // Retrieving the account number
      String accountId = null;
      if (request.getParameter("account") != null)
         accountId = request.getParameter("account");
      else
         accountId = session.getAttribute("currentAcc").toString();
```

```
        // Retrieving a dataBean for the specified account
        AccountDataBean adb = AccountHelper.singleton().getAccount(accountId);

        // setting the current account attribute of the session
        session.setAttribute("currentAcc", accountId);

        // giving the account data bean to the request
        request.setAttribute("account", adb);

        // changing the state of the session
        session.setAttribute("state","accountDisplay");

        // Forwards the presentation to the JSP
        getServletContext().getRequestDispatcher(
                   "/accountDisplay.jsp").forward(request, response);
        }
    catch (Exception e) {
        response.sendRedirect("/index.html");
    }
}
```

The account is retrieved from either the request or the session attribute currentAcc (as the user may already have selected an account and performed and displayed some other pages). The servlet gets a data bean from an AccountHelper object and sends it to the JSP.

The AccountHelper is a business component providing a data bean to the servlet. This data bean encapsulates all the data for a particular account number.

The data bean is given to the request object and not to the session object. Attributes can be set at both levels, the difference is that a request attribute is valid only for one request. Session attributes can be maintained between several requests coming from the same client.

As long as we just need the data concerning this account for this particular request, it is not necessary to clutter up the session.

The session state is then updated and the request forwarded to the JSP to display the dynamic content concerning this particular account.

The performOperationInput method
The following code shows how to retrieve the action from the client request, update the state attribute of the session object and forward the request to a JSP.

```
//ControllerServlet servlet: the performOperationInput method
public void performOperationInput(HttpServletRequest request,
    HttpServletResponse response, HttpSession session) throws IOException {
    try {
```

```
    // retrieve the action
    String action = request.getParameter("action");

    // set the session attribute with the value of the action
    session.setAttribute("state", action);

    // forward to the appropriate jsp
    getServletContext().getRequestDispatcher(
                    "/"+ action + ".jsp").forward(request, response);
    }
    catch (Exception e) {
        response.sendRedirect("/index.html");
        }
}
```

The performStatementDisplay method

This method retrieves from the session the account selected by the user and calls a Helper object to retrieve the last transactions concerning this account. It obtains a *Vector* object that is transmitted to the request. The method then calls the appropriate JSP. The following list shows the code for this method.

```
//ControllerServlet servlet: the performStatementDisplay method
public void performStatementDisplay(HttpServletRequest request,
    HttpServletResponse response,HttpSession session) throws IOException {
    try {

        // Retrieve parameters from the request
        String accountId = session.getAttribute("currentAcc").toString();

        // retrieve the list of transactions
        TransactionListHelper tlh = new TransactionListHelper();
        Vector txns = tlh.getTransactionList(accountId);

        // setting the attribute of the request
        request.setAttribute("txns", txns);

        // changing the state of the session
        session.setAttribute("state", "accountDisplay");

        // Forwards the presentation logic to the JSP
        getServletContext().getRequestDispatcher(
                        "/statement.jsp").forward( request, response);
    } catch (Exception e) {
        response.sendRedirect("/error.html");
    }
}
```

The performTransfer method

The performTransfer method calls the business components to transfer money between two accounts belonging to the same user. The following list shows the code for this method.

```
//ControllerServlet servlet: the performTransfer method
public void performTransfer(HttpServletRequest request,
    HttpServletResponse response, HttpSession session) throws IOException {
    try {

        // retrieve parameters
        String accFrom = request.getParameter("accFrom");
        String accTo = request.getParameter("accTo");
        String amount = request.getParameter("amount");

        // call the business method with the helper
        String message =
            TransferHelper.singleton().makeTransfer(accFrom, accTo, amount);

        // gives the message session object
        session.setAttribute("message", message);
        // set the session state
        session.setAttribute("state", "transfer");

        // forward to the appropriate JSP
        getServletContext().getRequestDispatcher(
                    "/transferOutput.jsp").forward(request, response);
    }
    catch (Exception e) {
        response.sendRedirect("/index.html");
        }
}
```

The servlet retrieves the parameters from the client request, it corresponds to values entered in the form displayed to him or her by the *TransferInput.jsp* file. It then calls a TransferHelper, a business component performing the transfer of funds.

The role of the servlet is to control the operation not to perform it. The helper object returns a message that the servlet puts in the session. The state of the session is updated and the request forwarded to a JSP.

The performMortgagePayment method

The following list shows the code for the method.

```
//ControllerServlet servlet: the performMortgagePayment method
public void performMortgagePayment(HttpServletRequest request,
        HttpServletResponse response, HttpSession session) throws IOException {
    try {
```

```
    // retrieve parameters
    String accountId = session.getAttribute("currentAcc").toString();
    String amount = request.getParameter("amount");

    // call the helper object
    String message =
        MortgagePaymentHelper.singleton().makePayment(accountId, amount);

    // set the message attribute in the session
    session.setAttribute("message", message);
    // update session state
    session.setAttribute("state", "paymortgage");

    // forward the work to the appropriate JSP
    getServletContext().getRequestDispatcher(
                "/paymortgageOutput.jsp").forward(request, response);
    }
    catch (Exception e) {
        response.sendRedirect("/index.html");
        }
}
```

Parameters are retrieved from the form and sent to a Helper, which performs the
business logic. The servlet then gets the message back and forwards the
presentation to the JSP.

The performLogout method
The logout operation is used to release all the objects within the session. The
following list shows the code for this method.

```
//ControllerServlet servlet: the performLogout method
public void performLogout(HttpServletRequest request,
    HttpServletResponse response, HttpSession session) throws IOException {
    try {
        session.invalidate();
        response.sendRedirect("/logout.html");
    }
    catch (Exception e) {
        response.sendRedirect("/logout.html");
    }
}
```

The *invalidate* method called on the session object unbinds all the objects of the
session and ends the user connection to the application. A static HTML page is
then displayed.

8.5 Servlets in WebSphere Application Server

WebSphere Application Server (WAS) provides a servlet container to run the servlets. The container handles the requests for the Web components (see 2.2.3, "Servlets and JavaServer Pages" on page 10) and provides a set of services to the servlets, for a servlet's life cycle.

8.5.1 The servlet container in WAS

In WebSphere Application Server, there is typically one servlet container per application server. The container provides all the services required by the specification to the servlets.

The implementation of the servlet container in WebSphere provides a set of APIs that allow some extensions to the specification, mainly in the session state management.

Web server and application server

In WebSphere Application Server, an application server communicates with the Web server through a plug-in. The client requests are sent to the Web server and, according to its configuration, it decides which requests it is able to handle and which ones must be handled by the servlet container.

Typically, a Web server can handle requests for static resources such as HTML pages. It will forward the requests to the application server for requests to servlets or JSPs.

One of the major changes in WebSphere Application Server 4.0 will be that the communication between the Web server and the application server will use the HTTP protocol or the HTTPS protocol (for secure transports). The OSE transport and the servlet redirector will not be used anymore. The configuration information for the plug-in is stored in XML format.

Servlet information

WebSphere supports the Web ARchive format to deploy Web applications. In such a file, information regarding a servlet can be set in a deployment descriptor (see "The deployment descriptor" on page 173). The following list shows a section containing servlet information in a deployment descriptor.

A servlet declaration in a deployment descriptor of a Web application
```
<servlet>
  <servlet-name>Servlet1</servlet-name>
  <servlet-class>com.ibm.itso.j2eebook.servlets.Servlet1</servlet-class>
  <init-param>
      <param-name>param1</param-name>
```

```
    <param-value>value1</param-value>
  <init-param>
</servlet>
<servlet-mapping>
<servlet-name>Servlet1<servlet-name>
<url-mapping>/servlet/*</url-mapping>
</servlet-mapping>
```

The name of the servlet (this name will be used at the application level), the class file of the servlet and initialization parameters can be specified here.

The servlet mapping allows the servlet container to handle client requests for a set of URLs with the same servlet. In our example, Servlet1 will be called to match all the following requests (considering our Web application is deployed at the root context of the host):

```
http://www.mywebapp.com/servlet/Servlet1
http://www.mywebapp.com/servlet/Servlet2
http://www.mywebapp.com/servlet/dir1/Servlet3
```

Moreover, it is possible to configure servlets directly in WebSphere Application Server and to have access to all these parameters directly.

8.5.2 Session management in WAS

WebSphere Application Server provides session support through a graphical user interface, the session manager. A session manager is associated with each servlet container.

We will also consult Chapter 7 "Session Support" of the *WebSphere V3.5 Handbook*, SG24-6161, to know more about session management in WebSphere Application Server.

Technique options

The session manager allows you to choose some administrative options, such as enabling the use of cookies and enabling the use of URL rewriting for tracking user sessions. WebSphere supports both of these techniques. In the case where these practices are used together, the servlet container will use the URL rewriting to store the session identifier on the client request even if his or her browser accepts cookies.

If the cookies approach is chosen, the application server will use a couple name/value to store on the client browser the session identifier (see "HttpSession" on page 194). With this identifier, the container will be able to be bound to a particular object stored within its JVM when the user makes another request to the application.

A set of parameters can be set when the cookies are used, such as the maximum number of milliseconds it can stay alive in a client browser (the -1 default value means they stay alive until the browser is closed) or a path on the server to which it will be sent (it may be used to prevent some resources from receiving the cookie).

Persistency options

Sessions are usually placed in the server memory, but this practice may be changed to store sessions in a database. It allows you to manage persistency: in case of a failover, a session can be retrieved.

Storing sessions in a database is also used when an application is distributed among several application servers and that each of these application servers need to access the same session objects.

This practice is called *session cluster*: it allows any servlet container of the configuration to access a common pool of sessions. Moreover, it allows load balancing: a session object can be accessed by any servlet container of the configuration and does not need to be attached to a particular one.

In WebSphere Application Server, it is possible (through the session manager) to configure a database to store session objects. When calling the *getSession* method of the interface *HttpSession*, a servlet container will make a request to this database to get the object associated with a particular user identifier.

That is also why all the objects put in a session must be serializable: the servlet container serializes and de-serializes the session objects to store them in the database.

Best practices options

Using persistency to manage sessions implies that you will consider several issues. Sessions are stored in a database and each request may have an impact on the performances of an application, and other problems such as concurrent access must be regarded.

WebSphere Application Server allows you to control the session management, through the session manager interface and to set some parameters. However, these parameters must be regarded closely as they are used to improve the performances of an application.

Single row or multi-row

Sessions can be stored in the database using a single row or using a row per object. For sessions using the single row schema, the *HttpSession* object is serialized and de-serialized every time a request is made on the object.

When an application only needs to update a field in a particular session object, all the other objects are retrieved from the database. Moreover, a hard limit has to be considered, as a single row can't generally take more than 2 MB of space (using the BLOB data type in DB2).

Using multi-row allows you to store each object of the session in one row in the database. A particular session object has several rows in the database, one per object. When accessing a particular object within a session, the application server does not need to serialize and deserialize all the objects of the session; it only accesses this particular row of the database.

Multi-row storage allows you to extend the size of the session objects as the limit of 2 MB applies to each row of the database and so on to each object within a particular session. But multi-row storage requires more space in the database and more read and write operations.

Session caching

Sessions are basically stored in the application server memory. When using persistency for managing sessions, they are stored in a database. However, WebSphere Application Server allows you to use both practices together.

The application server can hold in memory a list of the most recent sessions accessed, in order to avoid database access. When a session is accessed, the application server adds it to the list and ensures the coherency between the database version and its in-memory version.

WebSphere Application Server allows you to set the session pool size, that is to say, the number of sessions currently available in the memory. It also allows you to overflow this number, in the case of using non-persistent sessions.

When using non-persistent sessions, the application server holds in memory all the sessions. If the limit is reached, all the sessions generated will be invalid, that is to say, it will not be possible for the application to use them to store and to retrieve session data. Using the overflow option will allow you to expand the number of sessions, increasing the memory used to store sessions without limits.

When using persistent sessions, the session pool size represents the number of sessions contained in the cache memory of the application server. When this number is reached, sessions are sent directly back to the database and the application server doesn't verify their presence in the cache.

Using the session caching and the overflow option must be strictly controlled as the performances and the availability of a server are implied in this processes.

Session writing

In case of persistent sessions, WebSphere Application Server allows you to control the frequency for writing sessions to the database.

Typically, the application server updates the database with the session object at the end of the process of the request. The developer can use a special function within its servlet code to commit the information to the database whenever he or she wants. However, it implies that you use an IBM extension to the Java Servlet APIs.

For an application, updating session objects frequently is not recommended.

8.5.3 Other IBM extensions

IBM WebSphere Application Server provides an implementation of the Java Servlet APIs and also extensions to these APIs. These extensions allow better control of the application server, but must be reconsidered when an application is deployed within another application server.

If this application server just implements the standard APIs, the application may not run correctly.

IBMSession

The *IBMSession* interface, included in the package com.ibm.websphere.servlet.session provides an extension to the HttpSession interface defined in the Java Servlet APIs.

Typically, the following methods are available:

- **public void sync()**: This method is used in the servlet code to update the session object in the database. It acts as a commit order on a transaction to the session.

- **public boolean isOverflow()**: This method is used when non-persistent sessions are used. It allows you to know if sessions held in memory have exceeded the limit allowed.

- **public String getUserName()**: This function allows you to control security. A session is associated with a particular user and could be retrieved only by this authenticated user.

This interface may be used by developers in their servlet code, but they must be aware that it is an implementation designed for WebSphere Application Server.

Other extensions

Several extensions to the Java Servlet APIs are given in the following list; also we refer you to the documentation of IBM WebSphere Application Server for an exhaustive list of these add-ins and their functionalities:

– **Request and response support**: Two objects are provided to overload the request and response objects of the APIs. These objects can be used to be forwarded to other servlets for processing.

– **Filter and event support**: WebSphere Application Server provides support for the two major features introduced by the latest Java Specification (see 8.2.4, "What about Java Servlet APIs 2.3 release?" on page 179). Classes allowing the filtering of responses are available in the com.ibm.websphere.filter package, while the package com.ibm.websphere.servlet.event provides support for the application event handling.

– **Error handling**: The com.ibm.websphere.servlet.error.ServletErrorClass class can be used to handle exceptions in an application.

9

JavaServer Pages

This chapter discusses how JavaServer Pages (JSPs) can act as the presentation mechanism for dynamic data in your Web application. We will start with a brief introduction to the technology, and then concentrate on the elements that make a JSP. We will then see how JSPs can be used to implement the display of account data in our sample application.

The required JSP level mandated by J2EE 1.2 is JSP 1.1. The major difference between JSP 1.0 and JSP 1.1 is the introduction of custom tag libraries. Readers familiar with the basic elements of JSPs may want to skip directly to Chapter 10, "JSPs extended: custom tags" on page 251, which explains custom tag libraries in detail. However, the intention for this chapter is not to explain at a high level how to use JSP tags — our intended readership is Java developers, not page designers — but rather to delve a little deeper into their workings in the hope that this understanding makes tasks such as debugging easier.

9.1 Motivation for JSPs

In the early days of Java servlet development, it soon became apparent that, while servlets provide an excellent mechanism for the handling of requests for dynamic content, they do not provide a useful way of displaying the response. The hard-coding of presentation within the servlet code presented a number of problems:

► The coding of presentation detail using println statements made the actual servlet code large and unwieldy.

► Changing the "look and feel" of the presentation layer needed changes to Java code, and resultant recompiling, testing and deploying

► The roles of Web designer and Java programmer were not distinct, leading to development process problems.

Solutions to these and other problems were quickly created. An initial, home spun approach adopted by many applications was to insert markers, or tags, within HTML files. Servlets could then read in the HTML from the file system, and parse it looking for the tags. The servlet would replace any tags it found with the relevant dynamic data. Various vendors implemented solutions for performing such parsing within their application server products. Meanwhile, other non-Java technologies such as Microsoft's Active Server Pages tackled the same problem.

The approach was picked up and formalized into a standard under Sun Microsystem's Jeeves project, and became JavaServer Pages. The initial public specification, 0.91, became available in June 1998, but it was not until the final release of the 1.0 specification in September 1999 that the technology matured. Unfortunately, many vendor implementations of JSP 0.91 existed, and the marked differences between the two specifications caused migration difficulties with many applications built using the earlier release.

Almost as soon as JSP 1.0 was released, Sun released the 1.1 specification. The aim of 1.1 was mainly to update the specification to make use of the facilities available in the Servlet 2.2 specification, and also to introduce the notion of custom tag libraries, as discussed in Chapter 10, "JSPs extended: custom tags" on page 251. In addition, Sun handed responsibility for providing a reference implementation of both Servlet and JSP specifications to the Apache group's Jakarta project. The Tomcat server, Jakarta's Servlet and JSP implementation is rolled into the Sun J2EE reference implementation. Sun still maintains control over the development of Servlet and JSP specifications under the Java Community Process.

9.2 JSP execution model

In order to effectively use JSPs, it is first important to understand how they work. Essentially, a JSP file can be thought of as an HTML page with embedded segments of Java code which are executed at runtime. As such, a JSP file looks mostly like normal HTML, and can be edited by a Web designer who maintains responsibility for the look and feel of the site. Many Web page development tools, including WebSphere Studio's PageDesigner, have built-in support for editing JSP files.

However, more interesting to the reader of this redbook is what actually happens when a JSP page is invoked. Figure 9-1 shows the invocation routine.
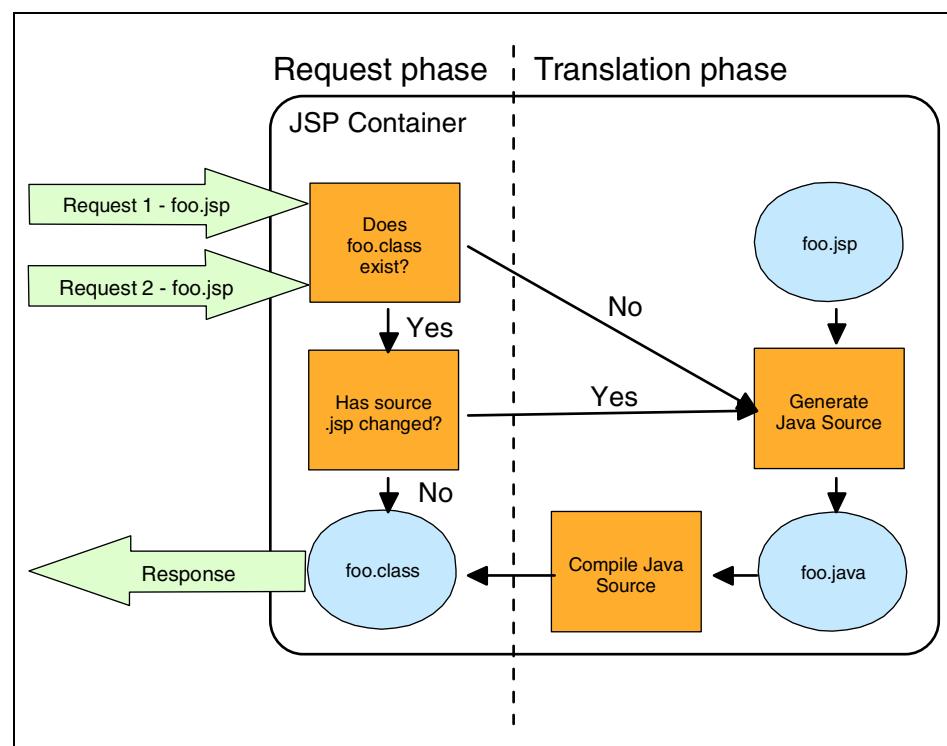

Figure 9-1   JSP invocation

There are two distinct phases in the invocation routine, the translation and the request phase. Central to this is that fact that within the JSP container, requests for individual JSPs result in the creation of a Java class. This class is actually a servlet. That is, it implements the javax.servlet.Servlet interface. The two phases represent the creation of the servlet class, and its subsequent invocation.

9.2.1 Phase one — translation

In Figure 9-1, request one represents the first request for foo.jsp since the JSP was deployed to the server. In this case, there is no existing servlet class for the JSP. The first step in this phase is to translate the JSP source file into a Java source file which can then be compiled into a servlet, known as the JSP Page implementation class. In fact, in WebSphere's JSP implementation, the foo.jsp file is translated into two files: a .java file containing the Java code for the servlet, and a .dat file containing the static elements of the JSP. The .dat file actually contains a serialized array of byte arrays, with each byte array containing the static page elements before the next JSP tag. The .java file is then compiled to create a .class file which represents the compiled servlet. The .class and .dat files are stored on the file system. The .java file may or may not be retained after the translation stage. This translation phase is performed once per JSP. If a compiled class file already exists on the file system then the request for the JSP proceeds directly to phase two.

> **Note:** For a detailed discussion of the way that WebSphere Application Server handles the compilation of JSPs, refer to Chapter 6, "JSP Support", in WebSphere V3.5 Handbook, SG24-6161.

9.2.2 Phase two — request processing

Phase two represents the actual invocation of the compiled servlet — the JSP page implementation class — to provide a response to the initial request. For request one, this is phase is entered immediately after phase one. If the servlet is not already loaded, it is brought into memory and the request serviced. For request two, the JSP container knows of the existence of an already compiled servlet class loaded in memory, and directly services the request. By default, the JSP implementation in WebSphere also checks the source .jsp file for changes. If it has been modified it will enter the translation phase. Each subsequent request is handled by the servlet until the servlet is unloaded from memory, for instance when the application server is stopped. This load, service requests, unload life cycle is exactly the same as for servlets.

9.2.3 HttpJspPage interface

The JSP container invokes the JSP page implementation class in the same way as any other servlet. We stated before that the class file generated for the JSP implements the javax.servlet.Servlet interface. In fact, JSPs actually implement the javax.servlet.jsp.JspPage interface; a subclass of javax.servlet.Servlet. As with servlets, there is nothing inherent within the JSP specification regarding the

HTTP protocol, JSPs can be written to provide request/response processing for any protocol. However, as with servlets, an HTTP specific interface: javax.servlet.jsp.HttpJspPage is provided for convenience. This interface hierarchy is shown in Figure 9-2.
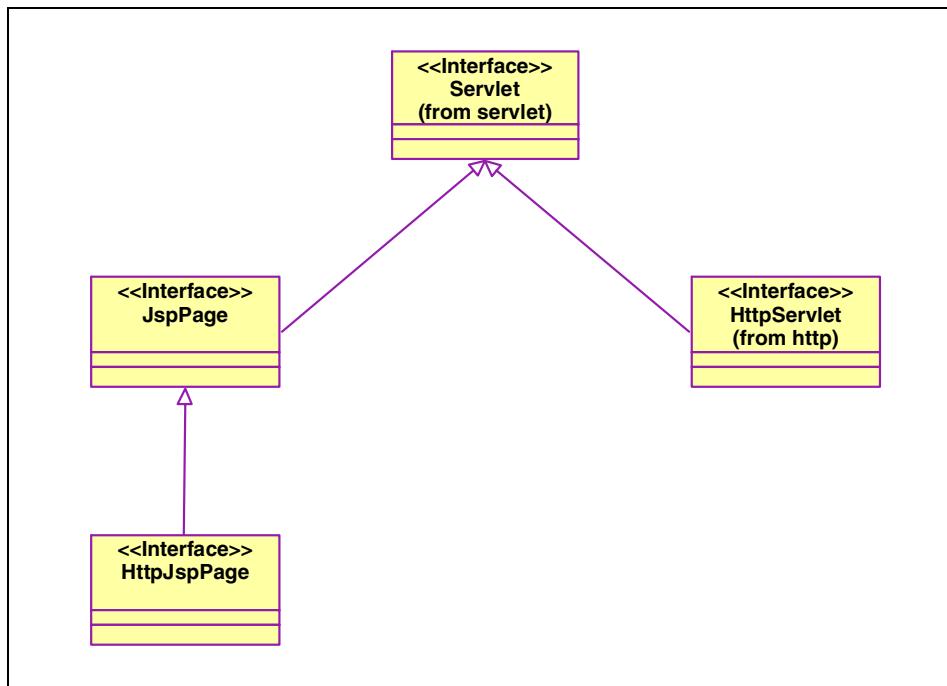

Figure 9-2   Interface hierarchy for javax.servlet.jsp.HttpJspPage

The javax.servlet.jsp.HttpJspPage interface defines the following abstract methods, either directly, or through the javax.servlet.jsp.JspPage interface (Table 9-1).

Table 9-1   javax.servlet.jsp.HttpJspPage interface methods

Method	Description
`public void jspInit()`	This method can be defined within the JSP page using a directive, and is invoked when the JSP page implementation class is first loaded. Analogous to the init method of a servlet.

Method	Description
`public void jspDestroy()`	This method can be defined within the JSP page using a directive and is invoked before the JSP page implementation class is unloaded. Analogous to the destroy method of a servlet.
`public void _jspService (HttpServletRequest, HttpServletResponse) throws ServletException`	Automatically generated by JSP container based on the embedded JSP tags in the JSP page. This method is invoked for each client request. Analogous to the service method of a servlet.

9.2.4 The WebSphere JSP 1.1 implementation

WebSphere's implementation of JSP 1.1 is provided by the Apache group's Jasper engine. Jasper is the JSP engine for Tomcat, which you will recall is the reference implementation for Servlet 2.2 and JSP 1.1. IBM developers contribute to the Jakarta project in the spirit of open source development.

The implementation includes a servlet: org.apache.jasper.runtime.JspServlet which handles every request for a JSP by deciding what processing needs to happen. It maintains information on the JSPs currently compiled and loaded in memory, and handles the compilation of JSP source files. The Jasper engine also provides a concrete implementation of the javax.servlet.jsp.HttpPageServet interface called org.apache.jasper.runtime.HttpJspBase which all compiled JSPs extend.

> **Important:** WebSphere 3.5. provides JSP implementations for JSP 0.91, 1.0 and 1.1 (with FixPack 2 or higher.) The 1.1 implementation is provided by the Jasper engine, while the 0.91 and 1.0 implementations are IBM produced. WebSphere Advanced 4.0 only provides the 1.1 implementation. JSP 1.0 files will work correctly, but users with JSPs built around the 0.91 standard will need to migrate their code to 1.1.

9.2.5 Example of JSP page implementation class generation

The following code contains the source for a simple JSP page, date1.jsp, which simply displays the current date and time (Example 9-1).

Example 9-1

```
<!date1.jsp: JSP source>
<HTML>
<HEAD><TITLE>Date and time</TITLE></HEAD>
```

```
<BODY>
<P>The current date and time is: <%= new java.util.Date() %></P>
</BODY>
</HTML>
```

This file contains one JSP expression tag to display the String representation of a Java Date object. When this JSP is invoked, the JSP processor creates the Java source for the JSP page implementation class. This is shown in following list (Example 9-2). Notable sections of code are highlighted in bold, and comments are added to aid the understanding.

Example 9-2

```
//date1.jsp: JSP page implementation class source
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;
import java.io.ByteArrayOutputStream;
import org.apache.jasper.compiler.ibmtsx.*;
import org.apache.jasper.compiler.ibmdb.*;
import java.sql.SQLException;

// Here our class is declared. Note that it extends
// org.apache.jasper.runtime.HttpJspBase
public class date1_jsp_0 extends HttpJspBase {

    static char[][] _jspx_html_data = null;

    static {
    }
    public date1_jsp_0( ) {
    }

    private static boolean _jspx_inited = false;

    public final void _jspx_init() throws JasperException {
        ObjectInputStream oin = null;
        int numStrings = 0;
        try {
            // Here the static content of the JSP is read in from a file called
```

```
                       // date1_jsp_0.dat stored on the filesystem as a result of the
                       // JSP compilation phase
                       InputStream fin =
                               this.getClass().getClassLoader().getResourceAsStream(
                                                               "date1_jsp_0.dat");
                       oin = new ObjectInputStream(fin);
                       _jspx_html_data = (char[][]) oin.readObject();
                   } catch (Exception ex) {
                       throw new JasperException("Unable to open data file");
                   } finally {
                       if (oin != null)
                           try {
                               oin.close();
                           } catch (IOException ignore) {
                           }
                   }
               }

               // This method is called for each JSP page request
               public void _jspService(HttpServletRequest request,
                  HttpServletResponse  response) throws IOException, ServletException {

                   JspFactory _jspxFactory = null;

                   // Here, the implicit objects are declared...
                   PageContext pageContext = null;
                   HttpSession session = null;
                   ServletContext application = null;
                   ServletConfig config = null;
                   JspWriter out = null;
                   Object page = this;
                   String  _value = null;
                   try {

                       if (_jspx_inited == false) {
                           _jspx_init();
                           _jspx_inited = true;
                       }
                       _jspxFactory = JspFactory.getDefaultFactory();
                       response.setContentType("text/html;charset=Cp1252");

                       // ...and then initialized
                       pageContext = _jspxFactory.getPageContext(this, request, response,
                                                          "", true, 8192, true);
                       application = pageContext.getServletContext();
                       config = pageContext.getServletConfig();
                       session = pageContext.getSession();

                       // one of the implicit objects is the OutputStream used to write
```

```
            // the response back to the caller. It is an object of type
            // JspWriter.
            out = pageContext.getOut();

            // The static content before the JSP tag is written to the Response
            // OutputStream.
            out.print(_jspx_html_data[0]);

            // This line of code represents our single JSP expression tag
            out.print( new java.util.Date() );

            // The static content after the JSP tag is written out.
            out.print(_jspx_html_data[1]);

    } catch (Exception ex) {
        if (out.getBufferSize() != 0)
            out.clearBuffer();
        pageContext.handlePageException(ex);
    } catch ( Throwable t) {
        if (out.getBufferSize() != 0)
            out.clearBuffer();
        pageContext.handlePageException(new Exception(t.getMessage()));
    } finally {
        out.flush();
        _jspxFactory.releasePageContext(pageContext);
    }
  }
}
```

9.3 JSP 1.1 elements

The JSP 1.1 specification defines the same set of basic page elements as
JSP1.0. These are:

- ► Directives
- ► Scripting elements
- ► Actions

Each of these elements are described below. But first we introduce some
common objects available to every JSP page.

9.3.1 Implicit object variables

A number of implicit objects are available to every JSP page. You can see the declaration and initialization of these objects within Example 9-2 on page 221. Table 9-2 summarizes the role of these objects, and the variable used to access these from the page.

Table 9-2   JSP implicit objects

Object variable	Role
request	The request that initiated the invocation of the JSP. For an HTTP request, it is a subtype of javax.servlet.HttpServletRequest.
response	The response to the request. For an HTTP request, it is a subtype of javax.servlet.HttpServletResponse.
pageContext	A platform independent utility object that provides access to the vendor-specific implementations of other implicit objects. Also provides accessor methods to objects stored within the page scope.
session	Provides access to any available javax.servlet.http.HttpSession object associated with the requestor.
application	Provides accessor methods to objects stored within the javax.servlet.ServletContext object for the Web application.
out	Provides the mechanism for writing to the Response OutputStream, an instance of javax.servlet.jsp.JspWriter.
config	Provides access to the javax.servlet.ServletConfig object for the JSP page.
page	Analogous to the Java implicit object.
exception	Available when the JSP is acting as an error page. Allows access to the exception object thrown from the calling page.

Note: The session and exception implicit object variables may not be available if certain directives are specified, see "The page directive" on page 226 for more details.

9.3.2 Object scope

Each implicit object belongs to a particular scope within the Web application. Indeed, the same scopes also become important when using JavaBeans within JSPs, as described in 9.4, "Using JavaBeans within JSPs" on page 236. Table 9-3 lists the available scopes.

Table 9-3   JSP object scopes

Scope	Description	Applies to
page	Objects are available from the javax.servlet.jsp.PageContext for the current JSP page. References to the objects are discarded upon completion of the _jspService() method.	response pageContext out config page exception
request	Objects are available from the javax.servlet.http.HttpServletRequest object for the current request. Objects are discarded at the completion of the current request.	request
session	Objects are available from the javax.servlet.http.HttpSession object associated with the current client. Objects are discarded when the HttpSession is invalidated.	session
application	Objects are available from the javax.servlet.ServletContext object associated with the Web application. Objects are discarded when the ServletContext object is reclaimed.	application

The implicit objects available through the pageContext, request, session and application variables each provide getAttribute() and setAttribute() methods which allow objects to be stored and retrieved within the particular scope they are associated with. As such, objects stored using the pageContext.setAttribute() method have the most limited scope: they exist only for the duration of that page invocation. Conversely, objects stored using the application.setAttribute() method have the widest scope: they exist for as long as the Web application is running.

9.3.3 XML syntax for JSP elements

The JSP 1.0 specification introduced an XML syntax for the composition of JSP pages. The initial motivation was the thought that JSP authoring tools could generate JSPs as valid XML documents while allowing users to use a user-friendly graphical interface to build the page. In JSP 1.1, the authors of the specification did not complete their work on the XML mapping, and therefore JSP

1.1 does not mandate that JSP containers accept JSP pages in XML format. The XML mapping will be developed further in future JSP specifications. Moreover, there should be no expectation that the future XML mapping will be backwards compatible with the one defined in JSP 1.1

For completeness, where JSP syntax is described in the sections below, we have included the XML notation for reference. WebSphere's JSP container will correctly interpret the XML notation for the directives, scripting elements and actions described below. However, you should bear in mind that tools such as WebSphere Studio do not generate XML style tags (although again, Studio will understand tags hand-coded using the XML syntax) and that the traditional syntax is currently more user-friendly to us humans!

9.3.4 Directives

Directives provide a way to provide information to the JSP container which can be used to determine the behavior and composition of the JSP page. JSP 1.1 defines three types of directive: page, include and taglib. The taglib directive is covered in detail in Chapter 10, "JSPs extended: custom tags" on page 251, but we shall cover the page and include directives below.

All directive tags have the same syntax:

```
<%@ directive {attr="value"}* %>
```

Or, in XML syntax:

```
<jsp:directive.directive {atttr="value"}* />
```

In both cases, *directive*, represents either page, include or taglib.

The page directive

The page directive tag allows for the definition of a number of attributes which are passed to the JSP container. As the name suggests, page directives apply to the JSP page as a whole. Multiple attributes can be specified within a single page directive tag, or attributes can be specified in individual page directive tags. However, each attribute can only be specified once within a JSP. The exception to this rule is the import attribute. Multiple import attributes can be declared on a single JSP.

A full list of valid attributes is defined in Table 9-4.

Table 9-4   Valid page directive attributes

Attribute	Description
language	Defines the scripting language to be used by scripting elements. JSP 1.1 defines the value "java" as the only supported scripting language. If no language attribute is specified, "java" is defaulted to. WebSphere 3.5.2 onwards also supports scripting languages that in turn support IBMs Bean Scripting Framework (BSF). Currently this is limited to Netscape's JavaScript implementation. For more details on BSF, refer to the Web site at: `http://oss.software.ibm.com/developerworks/projects/bsf`
extends	Allows the JSP page implementation class to extend a class other than the JSP container's implementation of javax.servlet.jsp.JspPage.
import	Allows the definition of classes and packages which will be available to scripting elements within the JSP. Classes within the following packages are available by default: ► java.lang.* ► javax.servlet.* ► javax.servlet.jsp.* ► javax.servlet.http.*
session	If true, the implicit object variable `session` will refer to a session object either already existing for the requesting user, or explicitly created for the page invocation. If false, then any reference to the session implicit object within the JSP page results in a translation-time error. The default is true.
buffer	Defines the buffering model for the JspWriter underlying the out implicit object variable. The default is an 8kb buffer. It can be set to a value of <x>kb, or none.
autoFlush	If true, the buffered output is flushed when full. If false, an exception is thrown should the buffer fill up. A value of false if the value of buffer is set to none will result in a translation-time error. The default is true.
isThreadSafe	If true, the JSP page implementation class will implement the javax.servlet.SingleThreadModel interface and requests for the JSP will be handled sequentially in order they were received. If false, then requests will be dispatched to the page on arrival and handled by individual threads. The default value is false.
info	Allows a String object to be defined which can be returned by calling the getServletInfo() method of the JSP page implementation class. This will override any description provided at deployment time.

Attribute	Description
isErrorPage	Specifies if the page acts as an error page for other JSPs. If true, then the exception implicit object variable is available for use. If false, any reference to exception will result in a translation-time error. The default is false.
errorPage	Defines a URL to which any uncaught exceptions thrown during page execution can be directed. The URL must specify a JSP page that has isErrorPage set to true. The exception class is placed in the request object sent to the JSP acting as the error page where it is made available through the exception implicit object variable.
contentType	Defines the encoding of the JSP page and the response content.

There are two of points to note regarding individual attributes. First, great care should be taken when considering using the extends attribute to allow your JSP to subclass another class. If used, the parent class must implement the javax.servlet.jsp.HttpJspPage interface. However, remember that in Table 9-2 on page 224 we introduced the pageContext implicit object variable and explained it has the role of providing access to vendor-specific implementations of the other implicit objects in an implementation neutral way. For example, a vendor may implement a high performance JspWriter class and provide access to it through their subclass of the abstract javax.servlet.jsp.PageContext class. If you choose to use the extends attribute of the page directive to subclass your own implementation of the HttpJspPage interface, then you will not have access to any vendor-specific performance enhancements.

The second point of note concerns the session attribute. As explained, if the session attribute is set to true, or not specified at all, then the session implicit object variable will reference the HttpSession object for the requesting client. If there is no HttpSession object available, then one will be created explicitly. This can create a performance overhead if your JSP page is not intended to interact with session objects. If this is the case, then you should ensure that the `<%@ page session="false" %>` directive tag is included within your JSP.

The following list shows a snippet of JSP source code containing example page directives:

```
<!Example page directive tags>

<HTML>
<HEAD><TITLE>Date and time</TITLE>
<%@ page import="java.util.*" session="false" isErrorPage="false" %>
<%@ page errorPage="/error.jsp" %>
</HEAD>
<BODY>
<P>The current date and time is: <%= new Date() %></P>
```

```
</BODY>
</HTML>
```

The include directive

The include directive allows content held in other textual files on the filesystem to be included within the JSP page source at **translation time**. That is, the contents of the included file will be included in the JSP source at the point where the tag is defined and therefore processed by the JSP page compilation procedure. The included file may contain both static content and other JSP tags. The include page directive takes one attribute: file.

The file attribute defines a relative URL, or URI, to the file to be included within the JSP page. This URI may be either page or context relative. For more information, refer to "A Web application in an application server" on page 174.

The included file may contain any valid JSP tags. These tags will be translated to Java source and included in the JSP page compilation class. The rules regarding page directives stated in "The page directive" on page 226 still apply to page directive tags within the included file. A page directive tag contained within the included file will apply to the whole JSP at translation time. Any page directives within the included file that conflict with tags contained in the "including" JSP source will cause translation time errors. For instance, if a JSP file contains a page directive specifying session="false" and then includes another file containing a JSP scripting element referencing the session attribute, a translation error will occur.

One consequence of the fact that the file referenced within the include directive is included during the translation phase is that subsequent changes to the included file will not be picked up by the JSP container on subsequent invocations of the JSP. The changes will only be visible when the JSP containing the include directive is itself updated, or its JSP page compilation class is deleted, forcing the translation phase to be carried out. If this behavior is not desirable, the include action tag provides another include mechanism, and is explained in "The include action tag" on page 234.

The following list shows an example of an include tag:

```
<!Example include directive tag: includeexample.jsp>
<HTML>
<HEAD><TITLE>Some document.</TITLE>
<%@ page session="false" %>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<P>This page includes the contents of another jsp which shows a timestamp:</P>
<%@ include file="includedate.jsp" %>
</BODY>
</HTML>
```

The contents of the file to be included, includedate.jsp are shown below:

```
<%@ page import="java.util.*" %>
<P>The current date and time on the server is: <%= new Date() %></P>
```

The results of calling the example are shown in Figure 9-3.
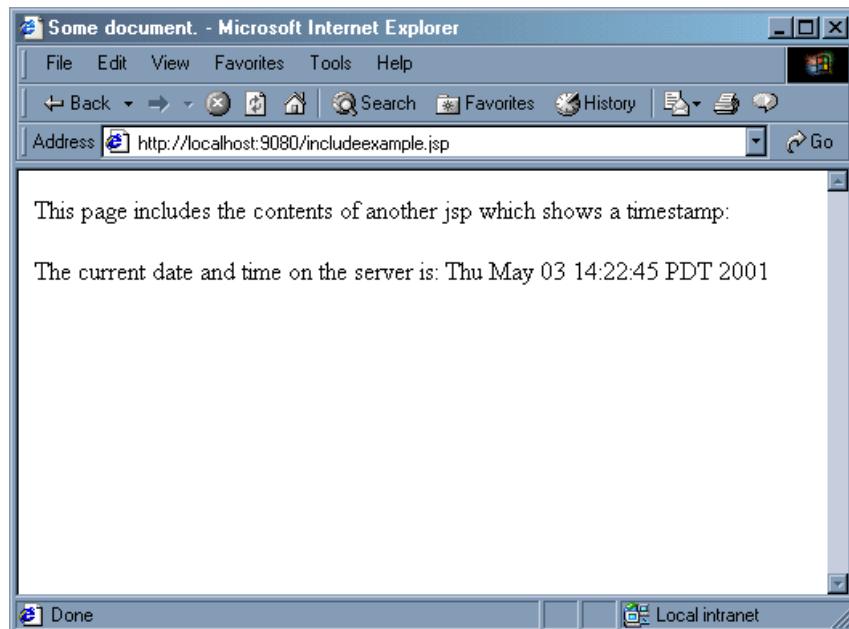
Figure 9-3   Results of invoking includeexample.jsp

9.3.5 Scripting elements

Scripting elements allow the insertion of code into the JSP. Technically the actual code is defined by the scripting language. As mentioned in Table 9-4 on page 227, the only scripting language required by the JSP 1.1 specification is Java. Therefore, we shall look at how the scripting elements can be used to insert segments of Java code into your JSPs.

There are three forms of scripting element:

- ▶ Declarations
- ▶ Scriptlets
- ▶ Expressions

Each of the types is described below.

Declarations

Declarations allow us to perform the declaration of variables and methods within our JSP source. We can then use these variables and methods within other scripting elements on our page. The syntax for a declaration element is:

```
<%! declaration(s) %>
```

Or, in XML syntax:

```
<jsp:declaration>
    declaration(s)
</jsp:declaration>
```

For example, to declare a variable, we can use the tag:

```
<%! int a = 1; %>
```

Or, to declare a method:

```
<%! public boolean isPositive(int x){
        if (x<0)
          return false;
      else
          return true;
} %>
```

> **Note:** For an example of why the XML syntax for JSP elements is considered less user-friendly, we need to look no further than the scripting elements. In XML, all document elements are parsed. And some common programming elements such as the < operator in Java are significant in XML. The XML DTD defined by the JSP 1.0 specification requires any such elements to be within a CDATA statement, so our isPositive() method would look something like this:
>
> ```
> <jsp:declaration> <![CDATA[ public boolean isPositive(int i){
>     if (i < 0)
>         return false;
>     else
>         return true;
> } ]]> </jsp:declaration>
> ```

Note that all variables declared within declaration elements will become class instance variables within the JSP page implementation class. Therefore they should not be used to store any invocation specific data, for example, counters or data relating to a specific client.

Scriptlets

Scriptlets are the most general purpose JSP element, but also the element to use with most caution. They can contain any valid Java code that you could normally place within the body of a Java method, including variable declarations, method calls, and so on. The contents of any scriptlet expressions within a JSP will be included within the _jspService() method of the JSP page compilation class. The syntax of a scriptlet tag is:

```
<% scriptlet %>
```

Or, in XML syntax:

```
<jsp:scriptlet> scriptlet </jsp:scriptlet>
```

Individual scriptlets do not have to contain a complete piece of Java syntax, they can interleave themselves with other page elements allowing you to build up conditional display of static page elements. However, if the combination of the scriptlets at translation time does not yield valid Java syntax, then a translation-time error will occur. An example scriptlet usage is shown below:

```
<P>The number 1 is <% if(!isPositive(1)) { %> not <% } %> a positive
number.</P>
```

The output of this piece of code will be:

```
The number 1 is a positive number.
```

On first appearance, it can be confusing as to why the word *not* fails to appear in the output. It doesn't appear within the Java syntax, and is not a valid Java type or object. Remember though, that within the JSP page compilation class, all of the static page elements are contained within byte arrays and are written to the output stream with statements, such as `out.print(_jspx_html_data[1]);`. In our example above, the out.print() statement for the byte array containing the word *not* will appear within the conditional block we open in the first scriptlet, and will only be written out if the conditional returns true. The second scriptlet then closes the conditional statement.

As you can imagine, excessive use of scripting elements in this way can make JSP source hard to read. If you refer back to the start of this chapter you'll remember that one of the problems that led to the evolution of JSPs was the presence of presentation elements making Java logic hard to understand. Now with JSPs containing excessive scriptlet use we have the opposite: our presentation logic is hard to understand and develop, because it is cluttered with Java code!

Expressions

We already have seen an example of an expression in Example 9-1 on page 220 when we used one to display the string representation of a java.util.Date object. Expressions attempt to convert the result of the expression evaluation to a String. They take advantage of the fact that the object within Java can be represented as a String, either through implementing a toString() method, or inheriting one from a parent class or ultimately, java.lang.Object. Primitive types can also be directly output. The syntax for expression tags is:

```
<%= expression %>
```

Or, in XML syntax:

```
<jsp:expression> expression </jsp:expression>
```

Use of implicit object variables in scripting elements

All of the available implicit object variables, as described in Table 9-2 on page 224, are available for use from within scriptlets and expressions. For example, objects placed within the HttpServletRequest object, or within the HttpSession object, can be retrieved and used within scriptlets and expressions via the request and session variables, respectively.

9.3.6 Actions

The third type of JSP element is the set of action tags. We will only briefly introduce the standard action tags defined within the JSP 1.1 specification here.

The standard action tags are listed in Table 9-5.

Table 9-5   Standard action tags

Action tag	Description
<jsp:useBean>	Allows access to a Java object, usually a JavaBean, retrieved from a given scope or newly instantiated, through a named identifier.
<jsp:setProperty>	Sets the value of a Bean property.
<jsp:getProperty>	Outputs a Bean property, converted to a String.
<jsp:include>	Allows inclusion of static and dynamic content within the current JSP page.
<jsp:forward>	Forwards the responsibility for request handling to another static or dynamic resource.

Action tag	Description
<jsp:plugin>	Enables generation of browser-specific HTML to enable use of the Java Plugin with Applets or other Java components.
<jsp:param>	Used in connection with the include, forward and plugin action tags to supply parameters in key/value pairs.

The first three tags will be discussed in more detail in 9.4, "Using JavaBeans within JSPs" on page 236. Action tag syntax is also used extensively within Chapter 10, "JSPs extended: custom tags" on page 251. For now we will concentrate on the param, include and forward tags. We will not discuss the plugin tag.

All action tags use the same tag syntax, which unlike the previous tags, is XML compatible. Action tags have two forms, either an open or closed body tag. The exception is the param action, which only appears in the form of a closed body tag.

The param action tag

The param tag usually appears within the body of one of the other tags, and allows the inclusion of a key/value type attribute. For example, to set a parameter named *foo* with the value bar, the following tag may be used:

```
<jsp:param name="foo" value="bar"/>
```

The value attribute may be an expression tag which is evaluated at runtime, for example:

```
<jsp:param name="surname" value="<%= customer.getSurname() %>"/>
```

The include action tag

The include action tag provides a way to include static and dynamic content within the output produced by the JSP page containing the tag. The content included by an include action tag is parsed at **request-time.** This contrasts with the include directive explained in "The include directive" on page 229.

The syntax of the tag is:

```
<jsp:include page="urlSpec" flush="true"/>
```

Or, as an open bodied tag:

```
<jsp:include page="urlSpec" flush="true">
    {<jsp:param ... />}*
</jsp:include>
```

The page attribute specifies the relative URL, or URI, for the resource to include. The resource may be a static file, JSP or servlet. The value of the page attribute may be specified via a JSP expression tag. The flush attribute specifies a boolean indicating if the page buffer should be flushed before the inclusion. JSP 1.1 does not allow a false value for buffer. The attribute must always be present, and always set to true.

The include action tag works in equivalent fashion to the include() method of the javax.servlet.RequestDispatcher class defined in the Servlet 2.2 specification. The request object, and any parameters defined are passed to the resource specified in the page attribute. The included resource then takes over responsibility for producing output. Once finished, the responsibility returns to the calling JSP. The included page can only write output using the output stream provided. It cannot set headers.

If the resource specified in the page attribute is a JSP, then that JSP undergoes the normal life cycle of translation processing/request processing. As it is being included in a runtime context, any change to the included JSP will cause it to go through the translation phase again. If we refer back to our example of the include directive, and re-code it to use an include action tag instead, then any changes we make to includedate.jsp between invocations of includeexample.jsp will be reflected on the next invocation.

The forward action tag

Just as the include action tag works like the include() method of javax.servlet.RequestDispatcher, so the forward action tag works in the same way as the forward() method. The syntax of the tag is:

```
<jsp:forward page="urlSpec"/>
```

Or, as an open bodied tag:

```
<jsp:forward page="urlSpec">
    {<jsp:param .../>}*
</jsp:forward>
```

When a JSP page uses a forward action tag, it terminates its own execution and hands responsibility to the resource specified in the page attribute. If the output buffer contains any output written before the forward action tag is processed, then the buffer is cleared before the forward takes place. The resource specified in the page attribute will provide all of the output, including headers.

The JSP 1.1 specification states that any output written to the page buffer will be cleared when the forward tag is called. Therefore, that output will not appear. However, if output has already been flushed from the buffer, or a page directive specifies a buffer size of none and output has been written, the attempt to clear the buffer will result in a request-time exception being thrown.

9.4 Using JavaBeans within JSPs

So far we have introduced JSPs and looked at the syntax of the three basic elements. We have not looked at the context in which they may be used. In this section we will briefly review the two common models for JSP usage before examining how we can make use of JavaBeans within our JSP pages.

9.4.1 The page-centric model of JSP usage

In the page-centric model, clients interact directly with JSP pages. The Java code embedded within the JSP interacts with business logic, and returns the results directly back to the user. The flow of a typical request is shown in Figure 9-4.
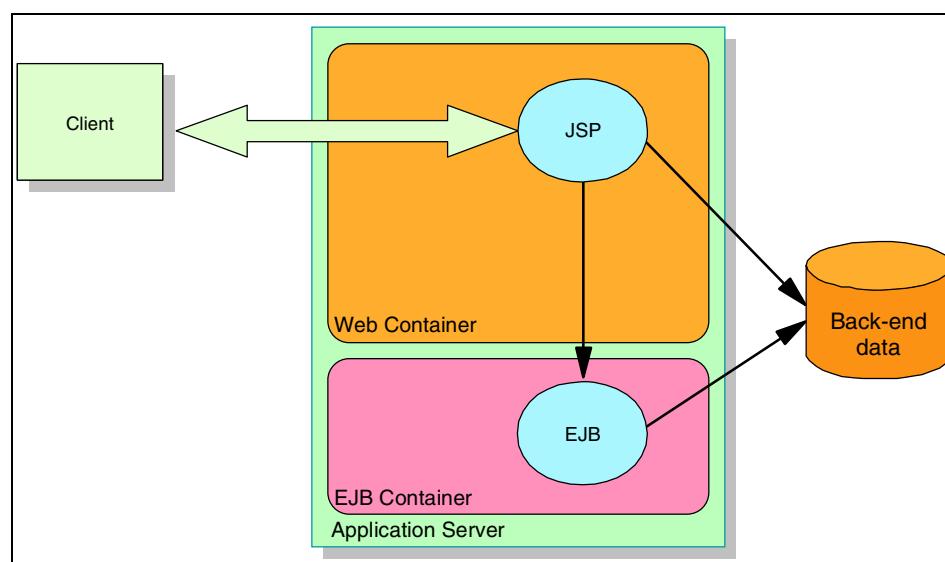
Figure 9-4   Page-centric JSP usage

This model is simple to implement and may be useful for small sites with little complexity, especially if they do not require access to any form of enterprise data. However, it introduces complexity within the JSPs which not only require code to manage the presentation of data, but also to control other aspects of the application such as page navigation, session management and database access. Therefore, the JSPs are likely to contain lots of scriptlets and other JSP tags. This in turn leads to a problem in our desire to separate the roles of Web page designer from Java coder.

9.4.2 The MVC model of JSP usage

The second model works on the Model View Controller paradigm with servlets acting as the front end to client requests. The servlets acts as controllers, with responsibility for page navigation, session management, enterprise data access and other such tasks. The servlets defer the responsibility for presentation handling to JSPs. This model is shown in Figure 9-5.
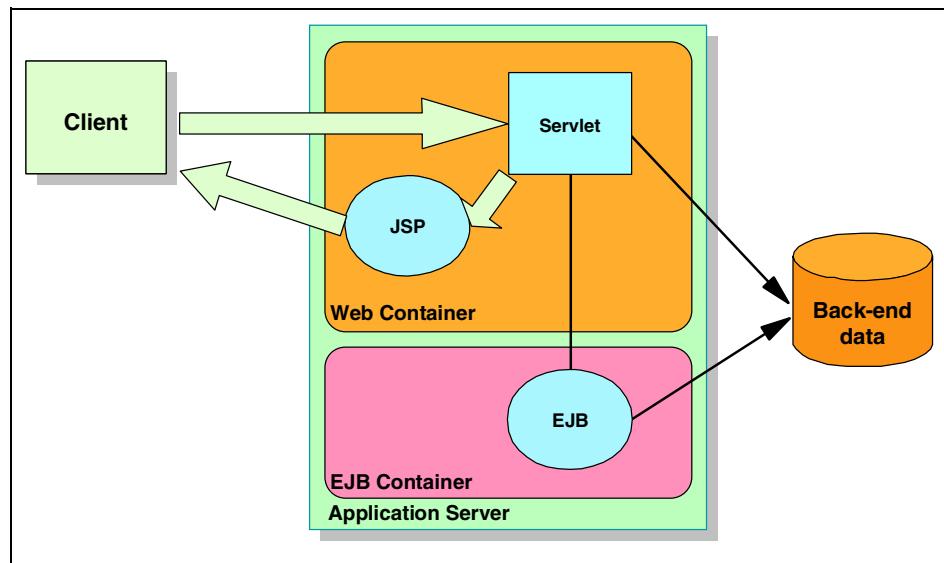
Figure 9-5   MVC oriented JSP usage

Freed by servlets from the code to handle such issues as enterprise access, our JSPs can concentrate on presentation, with JSP tags used only to insert dynamic data where needed. As such, our JSP pages may contain less JSP scripting. Our page designers can concentrate on building JSPs with minimal need for Java and application domain knowledge, while our Java developers can build the controlling logic within servlets.

Note: This is only the briefest of introductions to Web application models, intended only to show the role of JSPs. For more information, refer to various redbooks, such as "Design and Implement Servlets, JSPs, and EJBs for IBM WebSphere Application Server", SG24-5754, and the IBM Patterns for e-business Web site at `http://www.ibm.com/framework/patterns`.

9.4.3 JavaBeans as a contract

Within both models, but especially the MVC oriented Servlet/JSP model, we need a way to supply JSPs with data to be presented. We can do this using JavaBeans to encapsulate the data to be displayed. The JavaBean can act as a contract between the servlet and JSP which both have certain needs:

- ► The servlet requires a mechanism to display the results of invoking business logic.
- ► The JSP requires access to data which will enable it to process its JSP tags.

Servlets can populate JavaBeans with data that JSPs can then access through the properties of the bean. One pattern for such a contract is the display page access bean. An instance of the bean is created by a servlet from the results of some business logic processing. The bean contains no business logic of its own, but simply provides access to the results. The design of the bean interface can be defined between the Java coder and the page designer. Moreover, because JavaBeans can be introspected to discover their properties, methods and events, tooling such as WebSphere Studio's PageDesigner can provide visual tools for handling the beans.

As we introduced in 9.3.6, "Actions" on page 233, the JSP specification provides three action tags for managing beans within a JSP page. We will now look at each of these in more depth.

9.4.4 The useBean tag

The useBean tag allows a JavaBean instance, either existing or explicitly instantiated, to be accessed within a JSP page with reference to a named variable, or ID. There are two syntaxes for the useBean tag. The first, closed body form is:

```
<jsp:useBean attributes/>
```

The open bodied form is:

```
<jsp:useBean attributes>
      body
</jsp:useBean>
```

We will see later on what significance the contents of the body within the open form have.

JavaBean scope

Before discussing the attributes of the useBean tag, it is useful to point out that the instance of the JavaBean referred to, or created by the use of a useBean tag, has to be stored somewhere. In 9.3.2, "Object scope" on page 225, we hinted that the scopes defined in Table 9-3 on page 225 would also be important when accessing JavaBeans. When using the useBean tag, our bean will either be retrieved from a named scope, or created and added to the object related to that scope. For convenience, Table 9-6 lists the four scopes and their associated "bucket" objects.

Table 9-6   Object scopes

Scope	Object
page	javax.servlet.jsp.PageContext
request	javax.servlet.http.HttpServletRequest
session	javax.servlet.http.HttpSession
application	javax.servlet.ServletContext

UseBean tag attributes

The attributes of the useBean tag are defined in Table 9-7.

Table 9-7   useBean tag attributes

Attribute	Description
id	The identifier with which the bean can be accessed within the given scope, and also the variable by which the bean will be accessed from scripting elements.
scope	The scope within which an instance of the bean is available, or within which the new bean instance will be stored. The default is page scope.
typespec	Not an attribute as such, but the definition of valid combinations of attributes defining how the bean is instantiated.

Of these, the typespec attribute is the most intriguing as it defines the behavior of the tag. The typespec defines valid combinations of the three attributes listed in Table 9-8.

Table 9-8   Typespec attributes

Attribute	Description
class	Fully qualified class name for the bean implementation class.
beanName	The name of the bean, as you would supply to the instantiate() method of java.beans.Beans. This usually refers to the a fully qualified class name, or the filename of a serialized bean.
type	The class that the bean will be cast to. The class defined here is the type for the local instance variable id, which will be defined within the _jspService() method to hold the bean instance. If not specified, the value will default to the value of the class attribute.

As mentioned, the attributes may only be specified in specific combinations, namely:

- ▶ `class="classname"`
- ▶ `class="classname" type="typename"`
- ▶ `type="typename" beanName="beanFileName"`
- ▶ `type="typename"`

Note: In each combination, at least one **class** or **type** attribute is present, otherwise a translation-time error occurs. These attributes will define the type of the variable used to hold a reference to the bean. If no **type** attribute is defined the variable will be defined as an object of the type classname. If the **type** attribute is defined, then the JSP Container will cast the object to an object of the type typename. Therefore, it should be clear that typename should always be equal to classname, a superclass that classname extends, or an interface that classname implements.

The typespec is mandatory, and therefore one of the above combinations should be present within the useBean tag. Each combination defines a unique path through the bean creation road map shown in Figure 9-6.
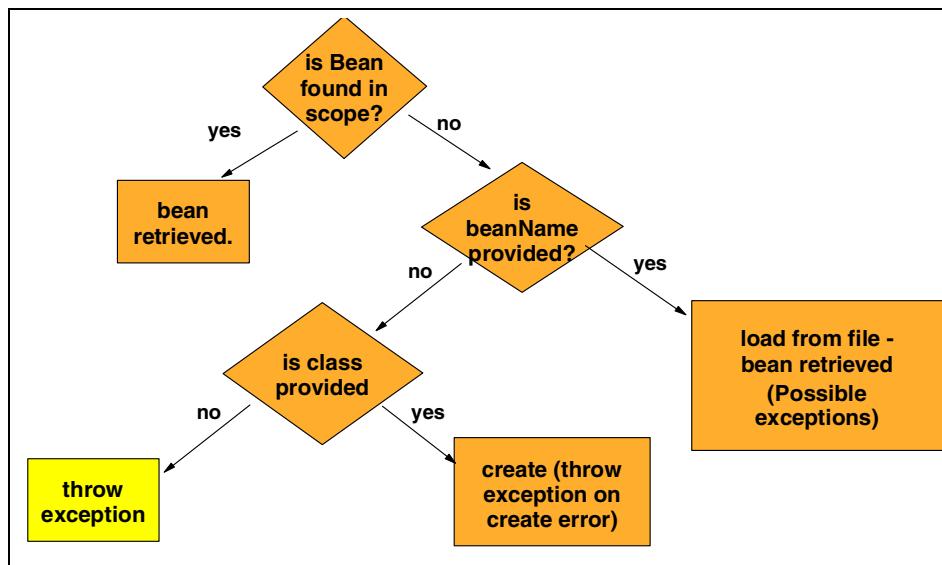
Figure 9-6   Bean creation road map

The initial step undertaken by the JSP container is to attempt to find an existing bean instance with the identifier given in the ID attribute within the scope defined with the scope attribute. If a bean instance is found, then a local variable is created with the given ID and initialized with a reference to the bean. The variable will be of the class type defined by the class attribute, or if present, the type attribute. If **only** the type attribute is defined and the object cannot be found in the specified scope, an InstantiationException is thrown and the useBean tag terminates.

> **Note:** Just because a bean instance already may exist within a given scope, this does not mean it is available to the JSP. Only when a local variable referring to the bean is set up using the useBean tag, does the bean instance become available to other scripting elements within the JSP.

If no bean instance was found within the given scope, but the beanName attribute was supplied within the typespec along with the type attribute, then the tag attempts to instantiate an instance of the bean using java.beans.Bean.instantiate() using the value of beanName as an argument. The result is then cast to the class type defined by the type attribute, and is stored within the scope given. The instantiate method may throw an InstantiationException, in which case the useBean tag terminates. If the returned class cannot be cast to the class defined by type, then a ClassCastException is thrown.

If no beanName attribute is supplied, then the value of the class attribute is used to create a new bean instance which is stored within the given scope. If the type attribute is defined, then the new instance is stored in a variable of the class type defined in the type attribute. If the classes defined in the class and type attributes are not compatible, then the JSP compiler picks up on this at translation time and throws an error.

Examples of useBean tags are shown in the following code (Example 9-3).

Example 9-3

```
<!Examples of useBean tag usage>
<%--This tag expects to find a bean with ID customer in the request scope. If
not it will throw an InstantiationException --%>
<jsp:useBean id="customer"
type="com.ibm.itso.j2eebook.helpers.CustomerDataBean" scope="request"/>

<%-- This tag attempts to find a bean with ID customer in the request scope. If
not it will attempt to de-serialize the bean from a file defined in the
declaration and store it in the request scope. This may throw an
InstantiationException if the de-serialization fails. If the de-serialized
bean cannot be cast to com.itso.j2eebook.helpers.CustomerDataBean, a
ClassCastException is thrown. --%>
<%! String beanName = "beanFile.ser" %>
<jsp:useBean id="customer"
type="com.ibm.itso.j2eebook.helpers.CustomerDataBean" beanName="<%= beanFile
%>"/>

<%-- This tag attempts to find a bean with ID customer in the request scope. If
not it will create a new instance of
com.itso.j2eebook.helpers.CustomerDataBean, and store it in the request
scope. The new bean may be populated with the contents of the tag's body. If
the class cannot be instantiated, an InstantiationException is thrown. --%>
<jsp:useBean id="customer"
class="com.ibm.itso.j2eebook.helpers.CustomerDataBean">
    ...
</jsp>useBean>

<%-- This tag attempts to find a bean with ID customer in the request scope. If
not it will create a new instance of
com.itso.j2eebook.helpers.CustomerDataBean, and store it in the request
scope in a variable of type com.ibm.itso.j2eebook.helpers.Person. If the
class cannot be instantiated, an InstantiationException is thrown. The
correctness of the casting was checked at translation-time. --%.
<jsp:useBean id="customer"
class="com.ibm.itso.j2eebook.helpers.CustomerDataBean"
type="com.ibm.itso.j2eebook.helpers.Person">
    ...
</jsp>useBean>
```

9.4.5 Accessing bean properties

Once a bean has been made available via a variable with the name specified in ID, that bean can be used in other JSP tags within the page. Methods on the bean can be accessed simply by calling them on the ID. For instance, if our CustomerDataBean bean used in Example 9-3 has a getFirstName() method that returns a string, we can call that method within an expression tag as follows:

```
<p>Hello <%= customer.getFirstName() %>.</p>
```

If our bean provides setter methods then these could be called within scriptlets:

```
<% customer.setFirstName("Adrian"); %>
```

However, the JSP specification provides two special tags for getting and setting bean properties, which make certain types of operation simple.

The getProperty tag

The getProperty tag allows access to a specific property of a named bean. It is constructed using the following syntax:

```
<jsp:getProperty name="id" property="property"/>
```

Where, *id* is the identifier of bean reference created with a useBean tag, and *property* is the name of a valid property on the bean. Tools such as WebSphere Studio PageDesigner can allow building of such a tag graphically by introspecting the bean to discover its properties.

The getProperty tag works in the same way as an expression tag. The value of the property is converted to a String, and put into the output stream. For example, we could recode our example of retrieving a customer name as follows:

```
<p>Hello <jsp:getProperty name="customer" property="firstName"/>
```

The setProperty tag

On initial inspection, the setProperty tag could be thought of as simply an alternative syntax for setting bean properties compared to using a setter method within a scriptlet as we did above. However, it offers richer functionality than that by allowing the matching up of parameters in the request object, submitted from an HTML form, for example, with those in the bean. The syntax of the tag is as follows:

```
<jsp:setProperty name="id" prop_expr/>
```

Where *id* is the identifier of a bean reference created with a useBean tag. The *prop_expr* is actually a combination of other attributes, as listed in Table 9-9.

Table 9-9   setProperty tag prop_expr attributes

Attribute	Description
property	The name of a bean property to set
param	The name of the request parameter whose value the bean property will be set to
value	A specific value to set the property to

As familiar from the useBean tag, only certain combinations of these attributes are allowed. The name attribute is always specified, followed by one of the following *prop_expr* combinations:

► `property="`*property*`" value="`*value*`"`
► `property="`*property*`"`
► `property="`*property*`" param="`*param*`"`
► `property="*"`

The first of these allows the explicit setting of a bean property with a specified value, for example:

`<jsp:setProperty name="customer" property="firstName" value="Adrian"/>`

Here we specify a string as value, but the value attribute may also contain a JSP expression tag. The object within the expression is passed through to the bean **without conversion** to a String.

If we only specify a property attribute, then the bean property will be matched up with any parameter within the request object having the same name. For instance, if we specify:

`<jsp:setProperty name="customer" property="firstName"/>`

Then the JSP page implementation class will attempt to assign the value of `request.getParameter("firstName")` to the bean property.

Of course, we may know that the parameter names in the request do not match up with the properties of our bean. In fact this may be the likelier scenario. In this case, we can specify the name of the request parameter value to assign to the bean as such:

`<jsp:setProperty name="customer" property="firstName" param="prenom"/>`

The final option open to us is to attempt to match all of the properties of our bean with parameters of the same name from the request object. We can do this by specifying:

```
<jsp:setProperty name="customer" property="*"/>
```

Therefore, the firstName property will be matched with any firstName parameter in the request object and so on. If there is no matching request parameter for a bean property then it remains unset. You should always make sure that your bean's default constructor sets appropriate default values.

Type conversions

In all cases, when the object to be assigned to the bean property is of type String (the only case where it may not be is when using the value attribute with an expression) then the JSP container will automatically attempt to convert the String into the type of the bean parameter. This conversion will work for the following types:

► Boolean or boolean
► Byte or byte
► Char or char
► Double or double
► Integer or int
► Float or float
► Long or long

Any attempt to set a bean property of a type not listed (or of course not of type String) to a value that is not of a compatible type will result in a translation-time error.

Indexed properties

It is possible to set the values of indexed properties within a bean using the setProperty tag by specifying a value attribute and setting it to an expression that will produce an array of values, for instance, by calling the `request.getParameterValues("`*name*`")` method to obtain the values of an indexed parameter within an HTML form.

9.5 Piggybank scenario — building the account list JSP

To demonstrate the JSP elements and also the use of JavaBeans within JSPs, we will look at the JSP used within the PiggyBank application to display a list of the users account. This JSP is called from the ControllerServlet when a user selects the account list option of the main menu. The typical output of the JSP is shown in Figure 9-7.

Figure 9-7   Output of the accountList JSP

The dynamically generated parts of the page are circled. The data required to generate them is obtained from the objects in Table 9-10.

Table 9-10   Objects containing data for page content

Name	Scope	Description
customer	session	A CustomerDataBean JavaBean for the customer.
accounts	session	A Vector of Strings. Each string contains an account ID.

The customer data bean was created when the user logged into the application and is available through the users HttpSession object. The accounts Vector was also created at login through a direct JDBC call to the underlying database to obtain a list of accounts the user holds and is also held in the session.

The actual JSP page was built using WebSphere Studio PageDesigner. Here, we concentrate on the source code generated by the tool to highlight uses of JSP syntax.

The complete appropriate parts of the JSP page source are as follows:

```
<!accountList.jsp source>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"><!-- Sample HTML
file -->

<HTML>
<HEAD>
<META name="GENERATOR" content="IBM WebSphere Page Designer V3.5.3 for
Windows">
<META http-equiv="Content-Style-Type" content="text/css">
<TITLE>
PiggyBank
<%@ page import="java.util.*"%>
</TITLE>
<LINK href="file:///H:/SG246124/studio/J2EEbook/theme/Master.css"
rel="stylesheet" type="text/css">
</HEAD>

<BODY bgcolor="#FFFFFF"
background="file:///H:/SG246124/studio/J2EEbook/images/hp_bg.gif"
link="#ffff00" vlink="#cccc00">
<jsp:useBean id="customer"
type="com.ibm.itso.j2eebook.databeans.CustomerDataBean" scope="session"/>
...
    <TD rowspan="2" valign="top">
      <P><jsp:getProperty name="customer" property="firstName" />, here are all
of your accounts, select one to work with it:</P>
<TABLE border="1">
     <% Vector accounts = (Vector)session.getAttribute("accounts");
        for(int count=0; count < accounts.size(); count++){%>
          <TR><TD>Account:</TD>
              <TD><A
href="file:///H:/SG246124/studio/J2EEbook/PiggyBank/PiggyServlet?action=account
Display&account=<%= accounts.elementAt(count)%>">
      <%= accounts.elementAt(count)%></A></TD></TR>
     <%}%>
      </TABLE>
...
</BODY>
</HTML>
```

First, a page declaration tag is used to import the java.util package into the JSP so that we can use the Vector object later on:

```
<%@ page import="java.util.*"%>
```

Next, the CustomerDataBean is made available to the JSP page by using a useBean action tag:

```
<jsp:useBean id="customer"
type="com.ibm.itso.j2eebook.databeans.CustomerDataBean" scope="session">
```

We know that the bean will be available within session scope and therefore define the useBean tag with a typespec that only defines the type attribute. If for any reason this page is called when a CustomerDataBean is not in session scope (for instance, by directly invoking the JSP from a browser outside of the Web application), then an InstantiationException will occur. The useBean tag will not attempt to create a new instance of CustomerDataBean.

Next, we use a getProperty action tag to display the users name:

```
<jsp:getProperty name="customer" property="firstName" />
```

Finally, we come to the main part of the page, the table of accounts. The table is built up from the Vector of account IDs. The accounts are hyperlinks that result in a request to the ControllerServlet. The table is built up using a scriptlet and expression tags:

```
<% Vector accounts = (Vector)session.getAttribute("accounts");
        for(int count=0; count < accounts.size(); count++){%>
```

Here we use a scriptlet to get the vector of account from the session. We use the session implicit object variable to access the users HttpSession object. We then create a for loop structure which will allow us to iterate through the list of accounts.

```
<TR><TD>Account:</TD>
<TD><A
href="file:///H:/SG246124/studio/J2EEbook/PiggyBank/PiggyServlet?action=account
Display&account=<%= accounts.elementAt(count)%>">
        <%= accounts.elementAt(count)%></A></TD></TR>
```

Here we use expression tags to obtain the account ID from the Vector. One is used to build the hyperlink URL, and the other as the display text. Note that through using an expression tag we don't cast the return type of the Vector.elementAt() call. The expression tag automatically converts the result of the expression to a String by calling toString().

```
    <%}%>
```

Finally, we must close the for loop with a small scriptlet.

9.6 Further resources

For further information on JavaServer Pages, refer to the following resources:

- ▶ *Design and Implement Servlets, JSPs and EJBs for IBM WebSphere Application Server*, SG24-5754
- ▶ *Servlet and JSP Programming with IBM WebSphere Studio and VisualAge for Java*, SG24-5755
- ▶ Chapter 6, "JSP Support", in the *WebSphere V3.5 Handbook*, SG24-6161

10

JSPs extended: custom tags

In this chapter we extend our discussion of JSPs by introducing the major new part of the JSP 1.1 specification: custom tags.

We will describe what custom tags are, and why you may wish to use them. We also discuss common usages and tag patterns. We then detail the elements that make up a tag and show how tag libraries can be developed with examples.

10.1 Motivation for custom tags

In Chapter 9, we stated that one of the aims of JSPs is to provide a clean separation of roles between the Java developer, responsible for coding the controller and business logic of a Web application, from the page designer who holds responsibility for displaying the application's data. JSPs are borne out of the desire to leave application logic free of presentation logic. However, we have seen that although JSPs provide such separation, if not used sensibly they can introduce the opposite problem, where the JSP becomes too cluttered with Java code, especially scriptlets. Such intrusion of Java code into the JSP makes the development and maintenance difficult for a number of reasons, among them are:

- ► The use of scriptlets requires Java programming experience; they cannot be made much easier to create by tooling, such as WebSphere Studio PageDesigner.

- ► Scriplets are useful for applying conditional logic to page element display, but doing so makes the markup language hard to understand.

- ► More Java code within a JSP means that there is increased likelihood of errors occurring in the JSP page compilation class. Although IBM VisualAge for Java provides JSP debugging facilities, instrumenting and debugging production problems within JSPs is harder than for normal Java classes.

- ► Although we can access JavaBeans within our JSPs, if changes need to be made to that bean's interface, we may need to make coding changes in every JSP that uses it.

With JSP 1.1 we have the ability to hide much of this complexity and reduce these problems by abstracting out much of the Java code within our JSPs into a custom tag.

10.1.1 A simple custom tag

As custom tag developers, it is first important to understand what a custom tag looks like and how it may be used. To do this, lets look at an example of a custom tag used to provide our table of accounts, previously generated by a scriptlet in 9.5, "Piggybank scenario — building the account list JSP" on page 245. The following code uses a custom tag (Example 10-1).

Example 10-1

```
<!Example of custom tag usage>
...
<%@taglib uri="http://jakarta.apache.org/taglibs/utility" prefix="utils"%>
...
<TABLE border="1">
```

```
<% Vector accounts = (Vector)session.getAttribute("accounts");%>
<utils:for varName="count" iterations="<%= accounts.size()%>" begin="0">
  <TR><TD>Account:</TD>
  <TD><A href="/PiggyBank/AccountDisplayServlet&account=<%=
accounts.elementAt(count.intValue())%>">
  <%= accounts.elementAt(count.intValue())%></A></TD></TR>
</utils:for>
</TABLE>
...
```

The parts of the example related to the custom tags are shown in bold. First, a taglib directive is used to create a reference to a file containing information that the JSP container will need to handle the tag. The directive also specifies a prefix for all tags in the library, in this case: utils.

The tag we used is called for, and not surprisingly mimics the actions of a for loop. The contents of the tag body are iterated over a number of times, determined by the attributes of the tag. In this case, we iterate over the size of the Vector. The varName attribute specifies an identifier we can use to obtain the iteration and is an instance of Integer.

You'll notice that the syntax of a custom tag is exactly the same as for the action tags as discussed in 9.3.6, "Actions" on page 233, and indeed, custom tags can be either closed-bodied or open-bodied.

> **Note:** The tag used above is actually contained within an open-source Utility tag library available from the Apache Group's Jakarta project. It was developed by Mandar Raje and Justyna Horwat and is available from `http://jakarta.apache.org/taglibs`.

10.1.2 Advantages of custom tags

By abstracting out Java code from JSPs into custom tags we can overcome a lot of the problems mentioned above:

► Our JSP authors may find it easier to work with a custom tag that looks like other JSP action tags, rather than to program Java code directly within a scriptlet. Tooling like WebSphere Studio's PageDesigner can automate a lot of the work involved in using the tag.

► The reduced Java code within the JSP makes it much easier to understand. This may be important when trying to maintain other people's JSPs. A JSP author with no Java skills will find it much easier to pick up and maintain JSPs authored by somebody else if they use custom tags as the interface should be cleaner and simpler.

► The custom tags can include much more exception and error handling logic without exposing this to the JSP author or clogging up the JSP page. Our custom tags can be developed and tested in isolation from the actual JSPs they will be used in.

► We can wrapper access to JavaBeans within custom tags. The properties and methods of the bean can be accessed through use of custom tags. Therefore, we can maintain and develop the bean without changing the interface the JSP uses. We can also add new features to our tags while maintaining backwards compatibility, something that is not always easy to do when accessing a Java API directly.

By using custom tags, we can harness the possibility of making our JSP pages quicker and easier to develop, and also more maintainable. What's more, by using them to wrapper common tasks, we can also take advantage of portability by reusing custom tags across Web applications. We have already seen an example of reuse in Example 10-1 on page 252. Whole libraries of third-party custom tags are freely available.

There are also advantages possible only by using custom tags. The most notable is the ability to post-process JSP output before it is written, as described in "BodyTag interface and BodyTagSupport" on page 268. This is a potentially powerful piece of functionality that is not achievable using scriptlets.

10.1.3 Disadvantages of custom tags

Despite the obvious potential advantages to their use, custom tags should not be viewed as a panacea for the problems associated with JSP development. Overuse of custom tags can lead to other problems:

► **Creation of meta-languages**: By forcing a multitude of custom tags onto the JSP authors we also force them to learn how the syntax of all the tags. Effectively we create another language, which may have a simpler interface than Java, but still needs learning. In any case, custom tags should always be supplied with sufficient documentation and examples.

► **Controller logic**: Many custom tag libraries are available to access APIs, such as JDBC, JNDI and so forth. Although the tags provide a simple and powerful way to use such services, for example, to access a database, so their use leads to the presence of controller and business logic within the JSP. This goes against the goals of the Model View Controller (MVC) architecture.

► **Performance**: The runtime work involved in handling custom tags within the JSP container does have an overhead. If performance of the Web application is critical then custom tags may not be appropriate.

10.1.4 When to use

Given the advantages and disadvantages listed above, the question of when custom tag usage is appropriate arises. The simple answer to this is that the primary consideration when deciding should be: Will the tag remove the need to put Java code within the JSP? The critical word in that question is **need**. If you need to implement some functionality within a JSP that will require some non-trivial Java code, then it is most probably a good candidate for a custom tag. This is especially true when the same Java code is present within more than one JSP for the reasons of maintainability.

By basing the decision to implement a custom tag on the need to have the functionality you reduce the temptation to implement some function that really should go elsewhere, for example, within the servlet controller logic. The custom tags you implement should stay true to the overall application architecture and design. For some designs, for example, if they are based on the page-centric model of JSP usage, then custom tags for database access and the like may well be necessary. However, if a database tag is used within an MVC based design, then this suggests that the functionality is implemented in the wrong place.

If we decide that a piece of JSP functionality is a suitable candidate for a custom tag, then there are other questions that arise. The nature of the functionality should be considered. Custom tags can generally be thought of as either generic or domain specific. Generic tags, for example, our for tag in Example 10-1 on page 252, have wide ranging uses. Can you pick up an existing tag for this purpose? Domain tags are more specific, and therefore their scope for re-use may be limited and the effort in creating the functionality within a custom tag may not be paid back, if the function is only used on one or two JSPs within your application.

You should also give consideration to the factors surrounding the authoring of the JSPs. If the JSP author(s) are confident with Java, then they may prefer scripting to using custom tags. Conversely, if the JSP authors are non-Java programmers, then a custom tag interface will be better suited. Custom tags are also useful when the dialog between the JSP author and Java programmer is limited by physical distance or communication factors.

In practice, it is often best to provide the JSP author with choice. By implementing the logic of the tag within a bean and providing the custom tag as a wrapper to the bean's interface, then you may be able to provide the author with the choice of scripting or using custom tags.

10.1.5 Common custom tag usages

When custom tags are used, they often perform a function that falls into one of the following categories:

- ► **Markup generation**: Generation of markup language output, for example, to create an HTML form.
- ► **Scripting generation**: Generation of client-side scripting code such as JavaScript, for example, to perform client-side data validation.
- ► **Environment access**: Providing access to and modification of implicit objects.
- ► **API access**: Simplifying access to APIs such as JDBC, JNDI and JavaMail.
- ► **Content manipulation**: Manipulating content before output, and caching of generated content.

10.2 Custom tag elements

In order for the JSP container to understand and execute a custom tag, the code implementing the tag and information on how to translate the tag and find and invoke the code must be available to the container. Tags for a particular function are normally aggregated into a tag library.

Tag libraries, or taglibs, are normally packaged as JAR files. The taglib is made up of the following elements:

- ► **Tag library definition (TLD)**: An XML file containing elements that describe each tag in the library. The TLD will be used by the JSP container during translation time to interpret each tag, and at request time to find and invoke the tag code.
- ► **Tag handler**: A Java class that provides the JSP container with access to the functionality of the tag. However, this does not necessarily mean that the tag handler implements the tag functionality. As we hinted, this may be contained within other classes/JavaBeans. The taglib will contain a tag handler class for each custom tag.
- ► **Supplemental classes**: The implementation of a custom tag may include some optional files which if used, need to be included within the taglib. These are:
  - – TagExtraInfo: Provides additional information, over and above that contained within the TLD, about a custom tag.
  - – BeanInfo class: Tag handlers are themselves JavaBeans and the JSP container will introspect them to obtain properties and methods. This can be aided by a BeanInfo class for each Tag handler.

On first impression it appears that building a custom tag is a complex procedure. First, we must define the syntax of our tag. Then we must develop the code that will implement our tag functionality, and provide access to it by creating a tag handler class. Depending on our tag we may then need to create a TagExtraInfo class to provide the JSP container with extra information to translate our tag. We then need to create a TLD to help the container find and use our tag handler.

Fortunately, once the syntax of the tag is defined and the functionality implemented, the steps to create all of the other elements are straightforward.

10.2.1 A simple example

To illustrate the various elements, and the steps taken to create them we will develop a very simple tag that displays a hello message. Our hello tag is a closed body tag that accepts one attribute, called name. The syntax of the tag will be:

```
<book:hello name="name"/>
```

The actual tag implementation will be contained within a JavaBean, the code of which is shown in below. The bean supplies one method, hello(), which accepts a String argument. The method outputs a message.

```java
//Simple custom tag example: HelloBean.java
package com.ibm.itso.j2eebook.tagexamples.beans;

/**
 * JavaBean which displays a Hello message.
 *
 * @author: Adrian Spender
 */
public class HelloBean {

    /**
     * HelloBean default constructor.
     */
    public HelloBean() {
      super();
    }

    /**
     * Returns a string saying hello to the name argument.
     *
     * @return A hello message
     * @param name java.lang.String The name to say hello to
     */
    public String hello(String name) {
        return "Hello " + name;
    }
```

```
}
```

Now we have the syntax for our tag defined, and the bean that provides the implementation. The rest of the steps are best described as cookbook code. That is, the code is largely the same whatever tag you are implementing. The classes and methods you need to define are always the same. The first few times you go through these steps will perhaps seem tedious, but creating them soon becomes relatively trivial.

First, we will create the tag handler class. We discuss this further in 10.3.1, "The tag handler" on page 261, but for our simple tag, we just need to create a class that extends the javax.servlet.jsp.tagext.TagSupport class, and provide an endTag() method which will be executed when our tag has been parsed by the JSP container. We also need to create an instance variable called name that will hold the value of our tag's attribute. Our tag handler class is shown in Example 10-2. Important sections of code are in bold, and comments are provided to explain what is happening.

Example 10-2

```
//Simple custom tag example: HelloTag.java
package com.ibm.itso.j2eebook.tagexamples;

import java.io.IOException;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import com.ibm.itso.j2eebook.tagexamples.beans.*;

/**
 * Handles the hello tag.
 *
 * @author: Adrian Spender
 */
public class HelloTag extends javax.servlet.jsp.tagext.TagSupport {

    // This variable will store the name attribute of the tag
    private java.lang.String name;

/**
 * HelloTag default constructor.
 */
    public HelloTag() {
        super();
    }

/**
 * Called when the end of the tag is reached. Here we simply
 * call our bean to get our output and then write it out.
 *
```

```
 * Note that we have access to the pageContext implicit object.
 *
 * At the end of the method we return an integer that tells the
 * JSP container whether the rest of the JSP page should be
 * executed. If set to SKIP_PAGE then the JSP execution will
 * stop. If set to EVAL_PAGE, execution will continue.
 *
 * @return indication to continue page execution
 * @exception javax.servlet.jsp.JspException
 */
    public int doEndTag() throws JspException {

        int result = EVAL_PAGE;

        // Here we call our bean to get our message
        HelloBean hb = new HelloBean();
        String mess = hb.hello(name);

        try {
            // and here we write the message back out
            pageContext.getOut().write(mess);
        } catch (IOException e) {
            // whoops
            e.printStackTrace();
        }

        release();
        return result;
    }

... as our tag is a bean, we also provide accessor and setter methods
}
```

Once the tag handler class has been built, we just need to create the TLD to
provide the JSP container with the information it needs. We explain the TLD in
10.4.1, "The tag library descriptor" on page 280. Our hello tag will be packaged
with the other example custom tags from this chapter in a taglib. The TLD
includes information on each tag. The relevant parts for hello are shown in
Example 10-3.

Example 10-3

```
<!Simple custom tag example: j2eebook.tld>
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
  "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<taglib>
```

```
<tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>j2eebook</shortname>
  <info>Tag examples from ITSO J2EE Redbook</info>

  <tag>
   <name>hello</name>
   <tagclass>com.ibm.itso.j2eebook.tagexamples.HelloTag</tagclass>
   <bodycontent>empty</bodycontent>
   <info>
     Takes a name and says hello
   </info>
   <attribute>
     <name>name</name>
     <required>true</required>
     </attribute>
   </tag>
</taglib>
```

We are now ready to use our tag. (Well not quite. It has to be deployed, but we'll cover that later.) The HTML to invoke is shown in Example 10-4. We first declare the taglib that the tag is in, giving it a prefix, book, which we will use to identify the tags.

Example 10-4

```
<!Simple custom tag example: helloSample.html>
<%@ taglib uri="WEB-INF/j2eebook.tld" prefix="book" %>
<HTML>
<HEAD>
<TITLE>Simple custom tag example</TITLE>
<%@ page session="false" %>
</HEAD>
<BODY>
<H2>Simple custom tag example.</h2>
<P><book:hello name="Adrian"/></P>
</BODY>
</HTML>
```

The result of executing the HTML is shown in Figure 10-1.
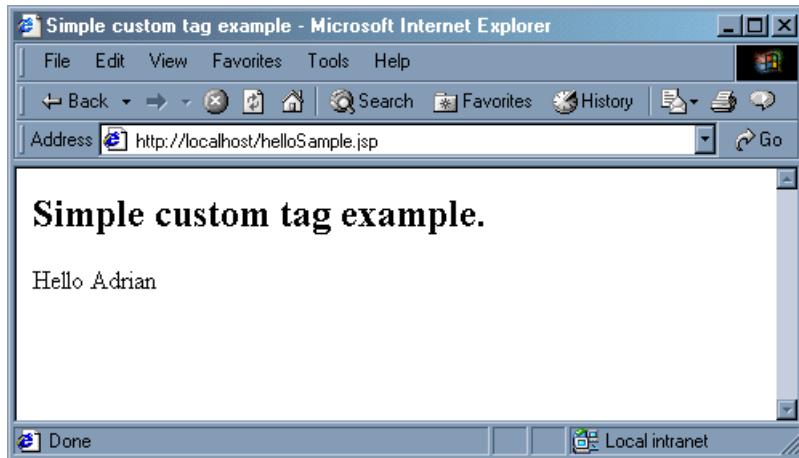
Figure 10-1   Simple custom tag example: results

10.3 Tag development

Understanding the steps needed to implement the tag handler and associated classes is simply a case of understanding the interfaces and required methods defined within the JSP 1.1 specification. In this section we will attempt to provide a clear indication of how to do this by first defining the interfaces available to us. We then look at some common tag patterns to provide examples of how these interfaces are used.

10.3.1 The tag handler

The tag handler as we've discussed, provides the interface between our tag logic and the JSP container. There is nothing stopping you from implementing the actual logic of the tag within the tag handler class. For simple tags, such as our example in 10.2.1, "A simple example" on page 257, this may be appropriate (we implemented a HelloBean just to show the concept of abstracting the tag logic from the tag handler), but in most cases you will want to provide the logic separately for the purpose of reuse.

The JSP 1.1 specification defines two interfaces to help us build custom tags: the Tag and the BodyTag interfaces. Remember that our custom tags may be either closed bodied or open bodied. Our example in 10.2.1, "A simple example" on page 257 is a closed bodied tag, whereas the for tag used in 10.1.1, "A simple custom tag" on page 252 is an open-bodied tag. With an open bodied tag we

may or may not wish to perform some processing on the body contents. The Tag interface allows us to build closed or open bodied tags that perform simple pass-through of the tag's body. By using the BodyTag interface we can perform more complex processing of the body content.

In every case, our tag handler is actually a JavaBean. We simply specialize it by requiring the implementation of one of the interfaces described below.

Tag interface and TagSupport

The Tag interface defines a basic set of properties and methods through which the logic of our custom tag may be accessed. Through the Tag interface we can build tags that either have no body content or have body content which we will simply want to pass through and not process. The specification provides a default implementation of the Tag interface in javax.servlet.jsp.TagSupport which supplies additional useful methods. Normally when developing a tag using the Tag interface, you should extend TagSupport.

Tag interface properties

The Tag interface defines two properties which must be made available to every custom tag. The first and most crucial of these is the pageContext property. Through this we gain access to the javax.servlet.jsp.PageContext object of the JSP in which our tag is embedded. Why is this crucial? Well first it gives us access to the output stream for the JSP. If we want our tag to write anything at all to the output then we need this. You can see an example of how we use the pageContext property to get the output stream to write our hello message in Example 10-2 on page 258 by calling `pageContext.getOut()`. The second important aspect of the pageContext property is that, as we described in 9.3.1, "Implicit object variables" on page 224, the javax.servlet.jsp.PageContext interface gives us access to all of the JSP implicit objects. Therefore, through the pageContext property we can access and set attributes within the page, request, session and application scopes of our JSP. Table 10-1 lists some of the methods that we can call on the pageContext property to gain access to the various implicit objects.

Table 10-1   Useful methods available from pageContext

Method	Returns	Usage
getOut()	javax.servlet.JspWriter	Used to gain access to the output stream in order to write output from the tag.
getPage()	java.lang.Object	Returns the equivalent to this.
getAttribute(String key)	java.lang.Object	Returns an object from page scope.

Method	Returns	Usage
setAttribute(String, key, Object ebj)	void	Places an object within page scope.
getRequest()	javax.servlet. ServletRequest	Used to gain access to the request object. For instance, to place objects within request scope using setAttribute().
getResponse()	javax.servlet.ServletRespo nse	Used to gain access to the response object.
getSession()	javax.servlet.http. ServletResponse	Used to gain access to the clients session object. For instance to place objects within session scope using setAttribute().
getServletConfig()	javax.servlet.ServletConfig	Used to gain access to the ServletConfig object.
getServletContext()	javax.servlet. ServletContext	Used to gain access to the Web applications ServletContext. For instance, to place objects within application scope using setAttribute().

The second property defined in the Tag interface is called parent. It allows us to access the properties of another custom tag when nesting tags. We describe this in further detail when discussing the ancestor pattern in 10.3.3, "Common tag patterns" on page 277.

Setting additional properties
As well as the pageContext and parent properties, we can define attributes on our custom tag interface which map to properties within the tag handler. We did this with the name attribute of our hello tag. By defining a property within the tag handler, the JSP container will map the tag attribute of the same name to the property. So in our example, our custom tag attribute called name:

```
<book:hello name="Adrian"/>
```

maps to the property defined within our tag handler bean:

```
private java.lang.String name;
```

Of course, as a bean we must also provide suitable getName() and setName() methods for the property. The JSP container will use the setter methods to populate the properties when invoking the tag handler.

If we want to make properties within our tag handler, or even our tag handler available to other tags, and JSP scripting elements, then we specifically need to place them within the pageContext. We can do this by using the pageContext.setAttribute() method. We will also need to create a TagExtraInfo class, and this is described in 10.3.2, "TagExtraInfo class" on page 273.

Tag interface methods

In addition to the properties, the Tag interface defines a number of methods which must be implemented. These are described in Table 10-2.

Table 10-2   Methods defined on the Tag interface

Method	Description
void setPageContext(PageContext context)	Sets the pageContext object for the tag handler. This method is generally only called by the JSP container.
void setParent(Tag tag)	Sets a reference to the parent tag. This method is generally only called by the JSP container.
int doStartTag()	Called when the opening tag is encountered.
int doEndTag()	Called when the ending tag is encountered.
Tag getParent()	Returns a reference to an enclosing tag. Explained in 10.3.3, "Common tag patterns" on page 277.
void release()	Called when the JSP container decides to stop using a tag handler instance. Can be coded to release any resources used by that tag.

The TagSupport class provides default implementations of these methods, but it is the doStartTag() and doEndTag() methods that are of real interest to us. The doStartTag() method is called when the JSP container first encounters our tag. Within this method we can access the pageContext and property methods, as well as any attributes defined within the tag itself. The return type of the doStartTag() method is an integer. This will be one of two values defined within the Tag interface and listed in Table 10-3.

Table 10-3   Valid return values for doStartTag() when implementing Tag interface

Return value	Description
EVAL_BODY_INCLUDE	Indicates that any body content between an opening and closing tag should be parsed by the JSP container.
SKIP_BODY	Indicates that any body content between an opening and closing tag should be skipped and not parsed by the JSP container.

Therefore, within the doStartTag() we can implement logic to decide if any content contained within the body of the tag, for example HTML markup or JSP expressions, should be parsed by the JSP container and written to the output stream. If we are implementing a closed-body tag as in our hello example, then the returned value from the doStartTag() method is irrelevant.

Regardless of the return value of doStartTag(), the JSP container always calls the doEndTag() method. In the case of a closed body tag, it will be called immediately after the doStartTag() method. For an open bodied tag it will be called after the JSP container evaluates tag's body content.

The doEndTag() method allows us to interact with the tag properties, and write output as we can in the doStartTag(). The return value of the doEndTag() method may take two values, as shown in Table 10-4.

Table 10-4   Valid return values for doEndTag() when implementing Tag interface

Return value	Description
EVAL_PAGE	Tells the JSP container to continue processing the rest of the JSP page.
SKIP_PAGE	Tells the JSP container to stop processing the JSP page and complete the request processing.

Therefore, within the doEndTag() method we can make the decision whether the rest of our JSP page should be processed.

TagSupport class defaults

Normally when developing a custom tag implementing the Tag interface, you will want to have a tag handler class extend the TagSupport class and then override either the doStartTag() or doEndTag() methods to provide your tag logic. If not overridden, the default implementations return the values shown in Table 10-5.

Table 10-5   Default return values of the javax.servlet.jsp.tagext.TagSupport class

Method	Default return value
doStartTag()	SKIP_BODY
doEndTag()	EVAL_PAGE

Flow of a tag invocation when implementing the Tag interface

The tag handler class is instantiated by the JSP container when a tag using that class is first encountered. The flow that occurs during the invocation of a tag handler class that is extending TagSupport is shown in Figure 10-2.
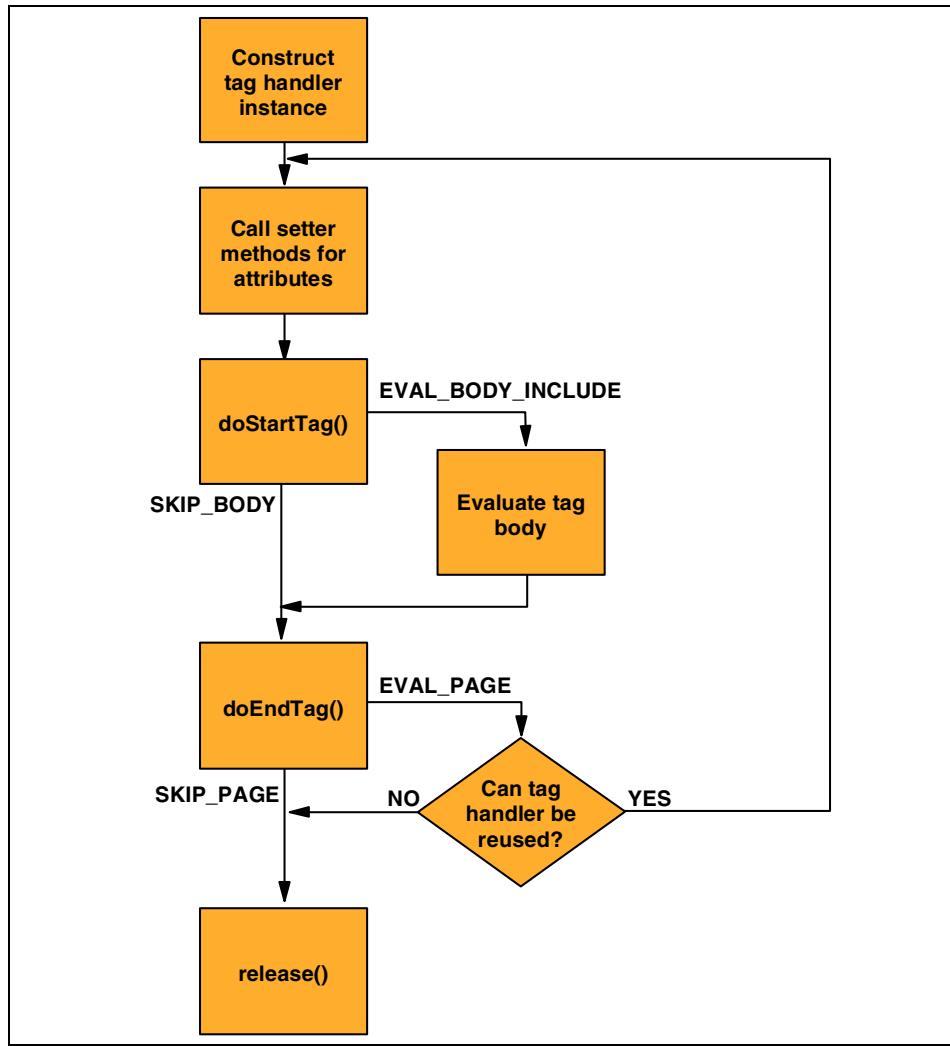
Figure 10-2   Flow of a custom tag implementing the Tag interface

In Figure 10-2, note that the JSP container can decide to reuse tag handler instances within the same JSP page if it wants. You should therefore be careful when setting and using class instance variables within the tag handler.

BodyTag interface and BodyTagSupport

Although the Tag interface allows us to tell the JSP container whether or not to process the contents of a tag's body, we cannot perform any processing of our own on the contents. By implementing the BodyTag interface in our tag handler, this possibility becomes open to us.

The BodyTag interface extends the Tag interface, and as such all of the properties and methods described above are available to us. However, BodyTag defines additional properties and methods.

The specification defines a default implementation of the BodyTag interface in javax.servlet.jsp.BodyTagSupport. You should extend this class when developing a tag implementing the BodyTag interface.

The bodyContent property

The property is defined by the BodyTag interface, named bodyContent. It provides access to an instance of class BodyContent, a subclass of JspWriter.

In order to perform processing on the contents of our tag, we need to have access to the contents before they are written out to the output stream. The BodyContent class acts as a container for our body contents before they are written out to the stream. In order to understand how this can happen, it is important to realize that when we make a call to pageContext.getOut() to obtain a JspWriter instance, we actually obtain a new JspWriter instance for our tag. Each tag within the JSP has its own writer and these are contained within a stack. When the contents of a JspWriter's buffer are flushed, they are written to the next JspWriter in the stack. Eventually, we will reach the JspWriter available through the JSP page's out implicit object. Only when the buffer for this object is written will our contents be sent to the client.

The BodyContent class provides an unbounded buffer that can only be flushed explicitly. Therefore, we can write as much as we want to it, and be sure that none of our tag's contents are pushed up to the next JspWriter until we want to. Second, the BodyContent class provides us with a number of methods that allow us to act on the contents before we push them up the stack. These methods are listed in Table 10-6.

Table 10-6   Methods available on BodyContent class

Method	Return type	Description
clearBody()	void	Allows us to clear the contents of the buffer. Unlike the clear() method of JspWriter, clearBody() will always be guaranteed not to have pushed content up the stack and will not throw any exceptions.
getReader()	java.io.Reader	Provides us with a Reader for the contents of the buffer.
getString()	java.lang.String	Provides us with a String object containing the contents of the buffer.
writeOut(java.io.Writer writer)	void	Write out the contents of the buffer to the provided Writer.
getEnclosingWriter()	javax.servlet.jsp.JspWriter	Provides us with a reference to the next JspWriter in the stack.

Through the getReader() or getString() methods we can access the contents of the buffer, and perform any processing we want on it. For instance, we could search for any instances of the string "foo" and replace it with "bar".

BodyTag interface methods
Although the bodyContent property gives us the ability to change the contents of the output, we need a hook into the flow of the tag processing to allow us to use it. Moreover, it would be nice if we could conditionally decide if the tag contents should be processed more than once. The BodyTag interface defines two additional methods that provide these hooks. These are listed in Table 10-7.

Table 10-7   Additional methods available on BodyTag interface

Method	Description
void doInitBody()	Called before the body contents are evaluated by the JSP container.
void doAfterBody()	Called when the body contents have been evaluated by the JSP container.

Remember that as BodyTag extends Tag, we still have the doStartTag() and doEndTag() methods defined. However, there is a difference in the valid return values for doStartTag() when we are implementing the BodyTag interface. The valid return values are listed in Table 10-8.

Table 10-8   Valid return values for doStartTag() when implementing BodyTag

Return value	Description
EVAL_BODY_TAG	Indicates that the JSP container should evaluate the body content of the tag.
SKIP_BODY	Indicates that any body content between an opening and closing tag should be skipped and not evaluated by the JSP container.

The EVAL_BODY_INCLUDE return value is not permitted when implementing BodyTag, instead to evaluate the body of the tag we must return EVAL_BODY_TAG.

When doStartTag() returns EVAL_BODY_TAG, the JSP container will first call the doInitBody() method before evaluating the body content. Once evaluated, the doAfterBody() method is then invoked.

Within doAfterBody() we can access the bodyContent property to get the BodyContent object which contains the output of the content evaluation. It is from within this method that we can perform any required processing of the body content before it is finally written back to the client.

The return type of doAfterBody() is an integer. The valid return values are shown in Table 10-9.

Table 10-9   Valid return values from doAfterBody()

Return value	Description
EVAL_BODY_TAG	Indicates that the JSP container should re-evaluate the body content.
SKIP_BODY	Indicates that the JSP container should stop the evaluation of the body content, and that the buffer should be pushed up to the next JspWriter in the stack.

Therefore, this gives us the ability to decide to make the JSP container evaluate the body content any number of times. Because doInitBody() is only called the first time the body content is evaluated, we can set variables here that will control our iterations. Moreover, we have options about how we deal with the buffer. We

may clear it before every body evaluation, and keep evaluating until we get the results we want. However, more usefully, we may append the output of an evaluation onto the buffer allowing us to iteratively build up the output we want to display to the user.

Note: Iterations over the body content may sometime produce unexpected results. Because the content may contain JSP expressions, scriptlets, actions or indeed other custom tags that access data in a shared context, such data may be updated by external sources between evaluations.

We can decide to stop iterating over the body content by specifying the return value SKIP_BODY. At this point, we must make the decision to output the body contents or not. If we do nothing, the contents of the output will be discarded. To output the contents we must send them to the next JspWriter in the chain. We can do this by using the getPreviousOut() method available from BodyTagSupport to obtain a reference to the previous JspWriter on the stack. We can then call the various print() methods of that writer to output our content.

The BodyTagSupport class

Just as the specification provides us with the TagSupport default implementation of the Tag interface, we also have a BodyTagSupport class. This extends TagSupport and adds default implementations of doInitBody() and doAfterBody(). The default return values of the various methods are shown in Table 10-10.

Table 10-10   Default return values of the javax.servlet.jsp.tagext.BodyTagSupport

Return value	Description
doStartTag()	EVAL_BODY_TAG
doAfterBody()	SKIP_BODY
doEndTag()	EVAL_PAGE

It is worthwhile to note that the default return value of doStartTag() has changed from that returned in the TagSupport class. BodyTagSupport overrides the method to specify that the JSP should evaluate the body of the tag. If we are implementing a tag handler using BodyTagSupport, it is safe to assume that we have some body content which we will want to evaluate!

Note: It is the varying uses and allowable combinations of the various return values that provides much of the initial confusion when trying to understand how custom tags are implemented. Once you understand the way they control the flow of the tag invocation things become a lot simpler.

Flow of a tag invocation when implementing the BodyTag interface

Figure 10-3 shows the flow of method calls when using BodyTag. The calls and decisions before and after the doStartTag() and doEndTag() methods are the same as for Figure 10-2 on page 267. Tag handlers implementing the BodyTag interface may also be reused by the JSP container.
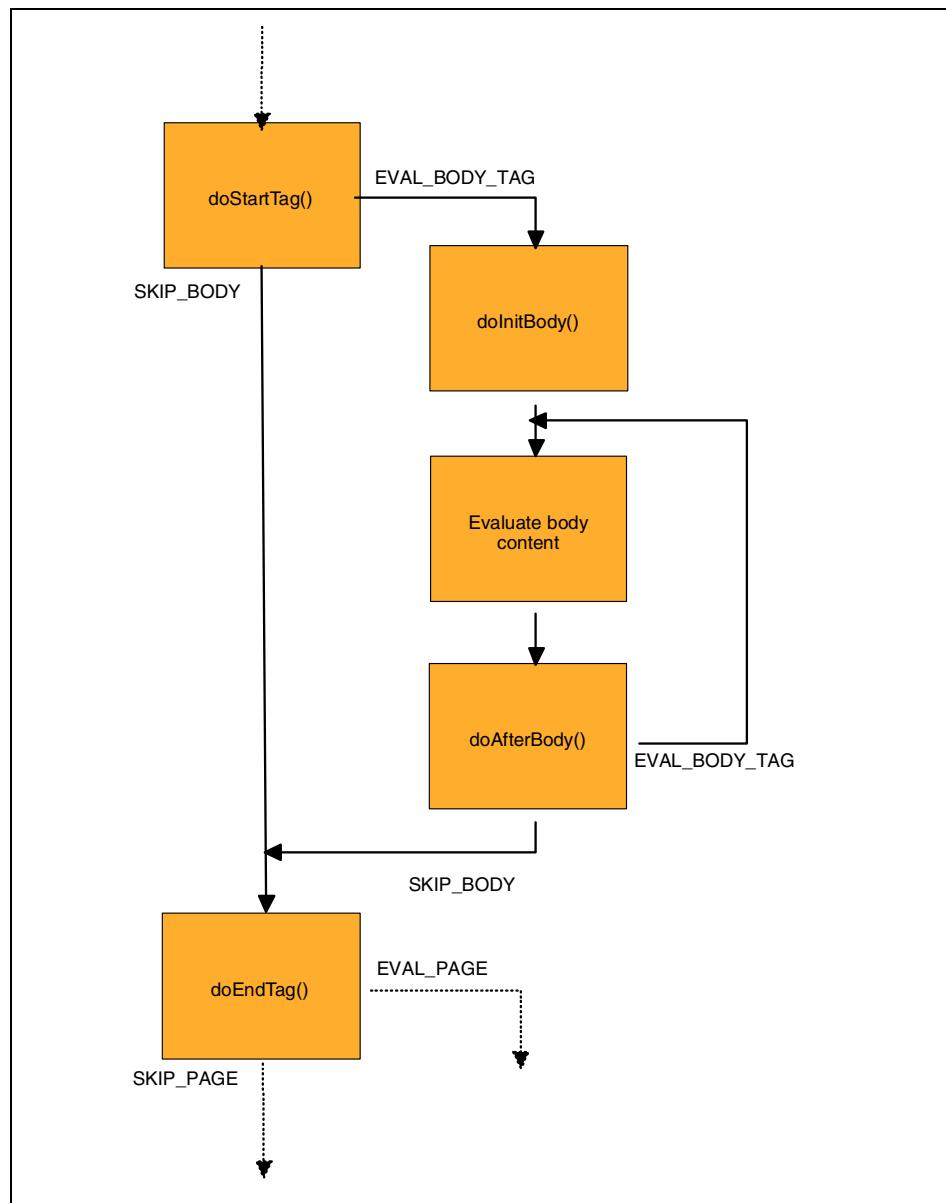
Figure 10-3   Flow of a custom tag implementing the BodyTag interface

10.3.2 TagExtraInfo class

We have already hinted at one use of the TagExtraInfo (TEI) class to make attributes of our tag handler available as scripting variables within the rest of the JSP page. In fact, the TEI does not do this explicitly. It acts as a tool for the JSP container to interrogate during the JSP translation phase to determine the correct behavior and interface of the custom tag. As such, the TEI class performs the following roles:

▶ Indicates to the JSP container translation mechanism what scripting variables the tag exposes to the rest of the JSP.

▶ Provides a way of validating a set of attributes for the custom tag.

The JSP container uses the TEI at translation time to provide information over and above that provided by the TLD described in 10.4.1, "The tag library descriptor" on page 280.

Specifying scripting variables

If we wish to make scripting variables available from our custom tag to other JSP elements within the page, then we need to place the object within the pageContext using the setAttribute() method. For instance, we may use the following tag within a page:

```
<foo:createHashtable id="bar" initSize="10"/>
```

Within our tag handler we define the following instance variables:

```
private Hashtable fooTable;
private String id = null;
private int initSize = 0;
```

Within our doStartTag() method, we create the Hashtable and set it within the pageContext:

```
public int doStartTag(){
    if(initSize != 0)
        fooTable = new Hashtable(initSize);
    else
        fooTable = new Hashtable();

    pageContext.setAttribute(id, fooTable);
    ...
}
```

Elsewhere within the method we may populate the hashtable with some objects. Now we have put the object within the pageContext we want, to be able to access it from other JSP tags using the value we specified for the id attribute, for example:

```
<% Enumeration fooEnum = bar.elements();
    while(fooEnum.hasMoreElements()){
        out.println(fooEnum.nextElement);
    }%>
```

However, at translation time, the JSP container will not know anything about bar. It is not an implicit object, and has not been introduced to the page through a useBean action. Therefore, we supply an interface through the TEI that will tell the translation engine all it needs to know to successfully compile the JSP page compilation class.

In order to allow the JSP container to validate any scripting variables we want to make available from the custom tag, we just need to define them within the TEI for the tag. We do this by implementing the getVariableInfo() method of the TagExtraInfo class. The signature of the method is:

```
public VariableInfo[] getVariableInfo(TagData data)
```

The getVariableInfo() method will be called by the JSP container during translation time for every instance of the JSP tag within the page. The method accepts one argument, an object of type TagData. The TagData object contains an array of key/value pairs that represent the attributes specified within the particular tag instance being translated. The only parts of the TagData class we need to know are the methods for getting the attribute name/value information for the particular tag instance. These are listed in Table 10-11.

Table 10-11   Useful methods on TagData

Method	Returns	Description
getAttribute(String attr)	Object	Returns the value for the given attribute. Needs to be cast to the known object class.
getAttributeString(String attr)	String	Returns a String containing the value of the given attribute.
getAttributes()	Enumeration	Returns an enumeration of all the specified attribute names. The names can be used to retrieve the attribute values using the other two methods.

For instance, we can access the value of the id attribute of our tag as such:

```
String id = data.getAttributeString("id");
```

In our example, the String returned will be "bar".

By using the TagData object within the getVariableInfo() method we can tell the JSP container about any of the attributes to be used as scripting variables. To do this, we create a VariableInfo object for each attribute. The VariableInfo object is created with four elements listed in Table 10-12.

Table 10-12   Information contained within a VariableInfo object

Property	Description
varName	A key string by which this object may be retrieved from the pageContext.
className	The name of the class the variable contains an instance of. May be a short class name or a fully qualified class name.
declare	A boolean indicating if the instance of className will be created by this tag instance.
scope	The scope of the object with respect to the tag.

This information is provided to a constructor when creating a new VariableInfo object. The value of varName will normally be the result of a call to one of the getAttribute methods of the TagData class.

The scope variable is of the most interest. It defines at what level the scripting variable will be available within the tag creating it, and the JSP page as a whole. There are three possible values, as shown in Table 10-13.

Table 10-13   Possible values of scope for VariableInfo object

Value	Description
NESTED	The variable will only be available between the opening and closing tags of this tag instance.
AT_BEGIN	The variable will be available from the opening tag of this tag instance until the end of the JSP.
AT_END	The variable will be available from the closing tag of this tag instance until the end of the JSP.

The NESTED scope is the most restrictive in that the scripting variable will only be available to JSP tags contained within the body of or custom tag. If we want it to be available to any tags after our custom tag then we would use AT_END. AT_BEGIN is the most open scope.

We create a VariableInfo object for each attribute of the tag which will become a
scripting variable and return them from the method in an array. For our
createHashtable tag, we can therefore implement the following getVariableInfo
method as shown below.

```
//Example getVariableInfo() method
public VariableInfo[] getVariableInfo(TagData data){
    return new VariableInfo[] {
        new VariableInfo(data.getAttributeString("id"),
                         "java.util.Hashtable",
                         true,
                         VariableInfo.AT_BEGIN)};
}
```

This tells the JSP container that the tag will create a scripting variable within
pageContext called (in the context of our id="bar" example) bar that will be of
type hashtable and will be valid from the beginning tag until the end of the JSP
page.

Validating attribute combinations

The other function of the TEI class for a custom tag is to provide a way for the
JSP container to validate combinations of attributes. The TLD as we will describe
in 10.4.1, "The tag library descriptor" on page 280 allows us to specify that an
attribute is required or not required. However, the TEI allows us to specify more
complex requirements, such as:

► Valid attribute combinations. For example, if attribute foo is specified then
   attribute bar must also be specified.

► Validating attribute values. For example, attribute foo must be either "Adrian",
   "Julien" or "Anthony".

► Validating attribute value types. For example, attribute foo must be of type
   java.util.Hashtable.

The TEI class has an isValid() method available for us to implement to perform
this function. The signature of the method is:

```
public boolean isValid(TagData data)
```

We can use the methods of TagData defined in Table 10-11 on page 274 to help
us perform this validation. The method should return true if the attributes are all
valid, or false otherwise.

10.3.3 Common tag patterns

Most custom tags fall into one of a set of common implementation patterns. By identifying the type of pattern a tag will implement, and therefore the behavior it will exhibit, we can then ascertain information, such as the probable tag interface, the convenience class it will extend, and the need to provide a TEI.

> **Note:** The patterns do not categorize tags into the type of function they provide, as we introduced in 10.1.5, "Common custom tag usages" on page 256. The functionality aspect refers more to tag libraries than individual tags. For instance, a tag library that wrappers access to JDBC may contain a number of individual tags, so that each conforms to a particular pattern.

A custom tag can be loosely categorized into one or more of six patterns:

- ► Macro
- ► Conditional
- ► Ancestor
- ► Content capture
- ► Iteration
- ► Declaration

Each of the patterns is described below.

Macro

A macro tag will ultimately result in the output of some markup language to the client. It may be closed or open bodied and may accept various parameters that provide input to the markup language, or control the way the output is created. The body contents of the tag are simply included within the markup output and are not processed. An example of a macro tag may be a tag which outputs its body contents as a hyperlink.

A macro tag handler extends the TagSupport convenience class.

The doStartTag() method accesses the tag properties and may write output using pageContent.getOut(). It returns EVAL_BODY_INCLUDE.

The doEndTag() may write more output after the body contents, and will return EVAL_PAGE.

Conditional

A conditional tag is one that makes decisions on whether to process JSP page content within the tag body, and also beyond the tag. The decision may be based on the values of tag attributes, or on objects accessed from the pageContext. An example conditional tag could be a tag that only allows the JSP container to process the body contents if a boolean attribute is true.

A conditional tag will extend the TagSupport convenience class.

The doStartTag() method accesses tag properties and may make a decision on whether to allow evaluation of the body content by returning either EVAL_BODY_INCLUDE or SKIP_BODY.

The doEndTag() method may make a decision on whether to allow evaluation of the rest of the JSP page by returning either EVAL_PAGE or SKIP_PAGE.

Ancestor

An ancestor tag is one that accesses information contained in another tag within which it is nested. In order to do this, it uses a couple of methods we haven't introduced yet, getParent() and findAncestorWithClass().

A nested tag is one that appears within the body content of another tag, for example:

```
<footags:foo>
    <%-- The bar tag is nested within the foo tag --%>
    <footags:bar/>
</footags:foo>
```

The getParent() method returns a reference to the parent tag handler. The JSP container will place a reference to the parent tag within a tag handler when it is initialized. By using getParent() to obtain that reference we then have access to the methods of the parent tag handler. In practice the method we want to call on a tag within which we are nested may be even further up the tree of nested tags. Therefore, we can use the findAncestorWithClass() method to traverse up the tree until we find the tag we need.

The findAncestorWithClass() method is defined as a static method within TagSupport. It takes two arguments, a reference to the class it is being called from, and the class of the tag handler it wishes to obtain a reference to. It then recursively calls getParent() until the desired tag handler is found. For example, If our foo and bar tags are implemented by FooHandler and BarHandler respectively, then we can use the method within the doStartTag() of BarHandler:

```
SomeObject obj = this.findAncestorWithClass(this,
                            somepackage.FooHandler).getSomeObject();
```

A tag that implements the ancestor pattern will use findAncestorWithClass() in this way. However, this facility in isolation is of little use. Therefore, the ancestor pattern will normally be used in conjunction with another pattern. One example of this is implementing a set of tags, which implement the patterns we have seen so far. Consider the following example:

```
<%Example if, then and else custom tags>
<book:if conditional="true">
    <book:then>
        <%!-- some content displayed if the conditional is true--%>
    </book:then>
    <book:else>
        <%!-- some other content displayed if the conditional is false --%>
    </book:else>
</book:if>
```

The if tag is a simple macro tag. It has an attribute, and the body content is always evaluated.

The then and else tags are examples of the conditional and ancestor patterns. They use the ancestor pattern to access the conditional attribute of the if tag. They then use this to decide whether to process their body contents.

Content capture

The content capture pattern describes a tag that performs some actions on its own content by accessing the bodyContent object. The body content may be manipulated in some way and then output, or even not output at all. Examples of content capture tags include:

► A log tag which outputs the contents of the tag body to a server log and not to the client

► A filter tag that filters body contents depending on some attributes, for example, word replacement

A content capture tag will extend the BodyTagSupport class to allow it to use the doInitBody() and doAfterBody() methods to access the bodyContent property. It will process the bodyContent only once, returning SKIP_BODY from the doAfterBody() method.

Iteration

The iteration pattern is similar to the content capture pattern in that it accesses the body content; however, its main function is to repeatedly evaluate the body content in order to build up the output to the client. A typical example of usage may be to iterate over a collection class displaying each element as a row in an HTML table.

Iteration tags extend the BodyTagSupport convenience class and will initialize some variable to control the number of iterations within the doInitBody() method. The doAfterBody() method will return EVAL_BODY_TAG until a certain condition of the iteration variable is reached when SKIP_BODY is called. The bodyContent.clear() method is not called between body evaluations.

Declaration

The declaration pattern uses the ability to interact with the pageContext property to set scripting variables as described in "Specifying scripting variables" on page 273, or to find or place objects within a scope of the JSP by accessing the page, request, session or application implicit objects through the pageContext.

The declaration pattern is normally used in conjunction with another pattern. A TEI is normally produced to aid the JSP container at translation time.

10.4 Tag deployment

Once we have developed the tag function, tag handler and a TEI if necessary then we need to package it all together into a tag library. As stated, tags that perform a similar function, or that combine to perform a function are normally packaged together. In order to aid the JSP container at translation and request time, we also need to develop the tag library descriptor, or TLD.

In this section we will look at the TLD in detail, and then look at how we package all of the elements for deployment.

10.4.1 The tag library descriptor

The TLD is an XML document that describes the contents of a tag library to the JSP container. It helps the container identify custom tags within a JSP page, find the tag handler, recognize the attributes of the tag and in combination with any TEI, validate the tag.

The structure of the TLD is pretty self-explanatory. As an XML file, the TLD contains a standard XML root element and DTD statement.

The TLD contains two main elements, taglib and tag. The taglib element defines details of the tag library as a whole. It also contains a tag element for each custom tag within the library. Table 10-14 defines possible sub elements within a taglib element.

Table 10-14   Sub elements of the <taglib> TLD element

Sub-element	Description
tlibversion	Defines the version of this tag library.
jspversion	Defines the version of the JSP specification that the tags within the library need.
shortname	Defines a name for the tag library.
uri	Defines a unique reference to the tag library.
info	A string describing the contents of the tag library.
tag	A tag element defines the details for a particular custom tag in the library.

Of these, the uri element is the most important. It ties the taglib directive defined within a JSP page with a specific tag library. The URI may not be specified within the TLD, but rather within the web.xml file for the Web application we are deploying as described in 10.4.2, "Packaging tag libraries" on page 282.

The tag element defines most of the information needed by the JSP container for a particular tag. The sub elements of the tag element are listed in Table 10-15.

Table 10-15   Sub elements of the <tag> TLD element

Sub element	Description
name	The name by which the tag will be referred to in a JSP page.
tagclass	The tag handler class .
teiclass	If the tag has a TEI, then its class is referred here.
bodycontent	Attempts to define the contents of the tag body.
info	A sting describing the function of the tag.
attribute	The attribute element defines the details of a tag attribute.

There is one tag element for each custom tag within the library. The bodycontent element is, according to the JSP specification, primarily for use by authoring tools. It may take one of three values: tagdependent, JSP or empty. In practice, it is most often used to define tags which will have no body content by specifying "empty".

The tag element will include an attribute element for each attribute of the tag. The attribute element has three sub elements: name is a required element and defines the name of the attribute. The required element, if specified as true, tells the JSP container that a translation-time error should occur if the attribute is not specified within a given tag instance. The rtexprvalue, if true, specifies that JSP container should allow the attribute value to be expressed using a JSP expression tag.

10.4.2 Packaging tag libraries

The classes and TLD that make up a tag library should be packed within a JAR file for deployment to the application server. The tag handler classes and TEI classes should be placed in the root of the JAR file under their correct package hierarchy. The location of the TLD file depends on how it is to be referenced within JSP pages.

In order to enable the JSP container to tie a taglib declaration within a JSP page to a particular tag library, we must provide a section within the web.xml deployment descriptor for the web application. An example entry is shown below:

```
<%Example taglib definition in web.xml>
<taglib>
    <taglib-uri>
        uri
    </taglib-uri>
    <taglib-location>
        location
    </taglib-location>
</taglib>
```

The URI refers to the uri attribute of a taglib directive on the JSP page on which the tag library is used. The JSP container uses the information provided in web.xml to map this uri to a physical TLD defined by the location attribute.

Therefore, the URI we specify can be abstracted from the physical location of the TLD allowing us more flexibility and portability when deploying our Web applications. For example, in the for tag used in Example 10-1 on page 252 the taglib directive is:

```
<%@taglib uri="http://jakarta.apache.org/taglibs/utility" prefix="utils"%>
```

The uri specified here is absolute, that is, it specifies a protocol, host and uri. Within the web.xml file for our Web application, we must provide a taglib element to map this to a particular location. The location specifies a relative URI for the location of the TLD.

It is possible to avoid the need to specify a mapping within web.xml by making the uri attribute of taglib directive a relative URI to the TLD itself. For example, in Example 10-4 on page 260, we specified:

```
<%@ taglib uri="/WEB-INF/j2eebook.tld" prefix="book" %>
```

In this case, the JSP container still looks for such a URI within the taglib elements of the web.xml descriptor for the Web application. If we do not specify one then it will use the relative URI to attempt to find the TLD directly.

Finally, the taglib directive may specify a jar file for the URI, for example:

```
<%@ taglib uri="booktags.jar" prefix="book" %>
```

In this case, the TLD must be located within the META-INF directory of the booktags.jar file.

Although these latter two techniques can be useful in taglib development, in production absolute URIs ideally should be used to aid portability and maintenance.

10.5 PiggyBank scenario — menu builder

In this section we will develop a simple set of custom tags to help assemble the menu a customer sees when using the PiggyBank application. The menu consists of a number of options, and is shown in Figure 10-4.
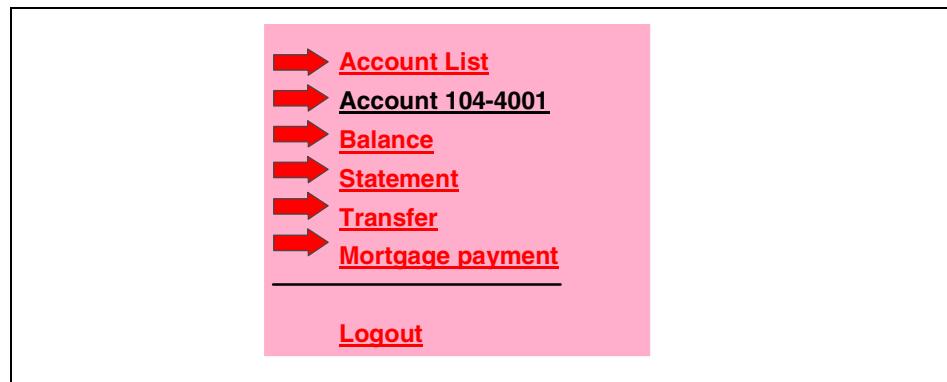
Figure 10-4   The PiggyBank application menu

The same menu appears on each of the JSPs within the application. However, not all menu items should be available from every page. Table 10-16 lists the menu items and when they should appear.

Table 10-16   PiggyBank application menu items

Menu item	Validity
Account List	Whenever a user is logged into the application
Account Identifier Balance Statement Transfer Mortgage payment	Whenever a user has selected an account to work with from the account list
Logout	Whenever a user is logged into the application

One way to control the display of the menu items is to build the menu HTML for each JSP individually. However, this is time consuming and hard to maintain. If we want to change the menu in any way, the changes will need to be made in each JSP.

We can take out the menu HTML into another file and then use a JSP include directive to place it into each JSP; this way our changes can be reflected by just editing the menu JSP file. However, some of our menu items are conditional on the state of the users interaction with the application. We would need to create menu JSPs for each of these menu states.

The ControllerServlet code described in Chapter 8, "Servlets" on page 165 controls page navigation by using an attribute named state which is stored in the users HttpSession. Whenever a user performs some action within the application, the state attribute is updated. This attribute can be used to control the display of the menu items. The list of menu items, and the valid states for their display are shown in Table 10-17.

Table 10-17   Valid states for menu item display

Menu items	ControllerServlet state
AccountList	login accountList accountDisplay statement transferInput transfer paymortgageInput paymortgage

Menu items	ControllerServlet state
Account Identifier Balance Statement Transfer Mortgage payment	accountDisplay statement transferInput transfer paymortgageInput paymortgage
Logout	login accountList accountDisplay statement transferInput transfer paymortgageInput paymortgage

Therefore, we could write Java code within the menu JSP to control the display of individual menu items according to the current state of the user. However, this would be a fairly complex scriptlet, so perhaps the use of custom tags would be better.

The solution we develop is intended to show the powerful way that custom tags can be used to develop page logic through a simple use of the features and patterns discussed in this chapter. There are plenty of other examples available elsewhere which show performing tasks, such as database queries from tags. Such tags go against the MVC model on which our application is based.

10.5.1 Definition of the tags

The goal of our tags is to allow the buildup of an HTML menu by deciding if an item should be displayed through checking an attribute within the users HttpSession. As such, our tags will need to perform the following tasks:

- ► Access the session attribute holding the users current navigation state.
- ► Define menu items.
- ► Define the conditions under which a menu item should be displayed.
- ► Match navigation state to menu item conditions.
- ► Display the menu item.

To provide this functionality, we will define three tags:

- ► **menu**: Defines a menu and provides access to the session attribute holding the navigation state.

- ► **menuitem**: Defines a menu item and controls its display based upon current navigation state

- ► **state**: Defines a valid state under which a menuitem should be displayed.

The menu tag will have one argument, stateKey, which defines the HttpSession attribute under which the users navigation state is held. The tag is open-bodied and contains one or more menuitem tags.

The menuitem tag is an open-bodied tag with no attributes which defines a single item to display. The body of the tag contains one or more state tags, and the HTML that makes up the menu item.

The state tag is a closed-body tag which contains one attribute, name, which defines a valid state under which the menuitem it is enclosed in will be displayed.

The usage of the tags is shown below which contains the source code for the PiggyBank menu.jsp file:

```
menu.jsp
<%@ taglib uri="WEB-INF/menu.tld" prefix="menus"%>

<!-- First menu item to display account list. Always displayed. -->
<menus:menu stateKey="state">
  <TABLE>
  <TBODY>
    <menus:menuitem>
      <menus:state name="login"/>
      <menus:state name="accountList"/>
      <menus:state name="accountDisplay"/>
      <menus:state name="statement"/>
      <menus:state name="transferInput"/>
      <menus:state name="transfer"/>
      <menus:state name="paymortgageInput"/>
      <menus:state name="paymortgage"/>
      <TR>
          <TD><IMG src="/images/b_lis019.gif" width="16" height="15"
             border="0"></TD>
          <TD><B><A href="/PiggyBank/PiggyServlet?action=accountList">
             Account List</A></B></TD>
       </TR>
    </menus:menuitem>

    <!-- Second menu item to display individual account options. Displayed if
         a user has selected an account to work with. -->
    <menus:menuitem>
      <menus:state name="accountDisplay"/>
      <menus:state name="statement"/>
      <menus:state name="transferInput"/>
```

```
    <menus:state name="transfer"/>
    <menus:state name="paymortgageInput"/>
    <menus:state name="paymortgage"/>
        <TR>
          <TD></TD>
          <TD>Account <%= session.getAttribute("currentAcc") %></TD>
        </TR>
        <TR>
          <TD><IMG src="/images/b_lis019.gif" width="16" height="15"
              border="0"></TD>
          <TD><A href="/PiggyBank/PiggyServlet?action=accountDisplay">Balance
              </A></TD>
        </TR>
        <TR>
          <TD><IMG src="/images/b_lis019.gif" width="16" height="15"
              border="0"></TD>
          <TD><A href="/PiggyBank/PiggyServlet?action=statement">Statement</A>
            </TD>
        </TR>
        <TR>
          <TD><IMG src="/images/b_lis019.gif" width="16" height="15"
              border="0"></TD>
          <TD><A href="/PiggyBank/PiggyServlet?action=transferInput">Transfer
              </A></TD>
        </TR>
        <TR>
          <TD><IMG src="/images/b_lis019.gif" width="16" height="15"
              border="0"></TD>
          <TD><A href="/PiggyBank/PiggyServlet?action=paymortgageInput">
              Mortgage payment</A></TD>
        </TR>
</menus:menuitem>

<!-- third menu item to display the logout menu. Always displayed if a user
       is logged in -->
<menus:menuitem>
  <menus:state name="login"/>
  <menus:state name="accountList"/>
  <menus:state name="accountDisplay"/>
  <menus:state name="statement"/>
  <menus:state name="transferInput"/>
  <menus:state name="transfer"/>
  <menus:state name="paymortgageInput"/>
  <menus:state name="paymortgage"/>
        <TR>
          <TD colspan="2"><HR></TD>
        </TR>
        <TR>
          <TD><IMG src="/images/b_lis019.gif" width="16" height="15"
```

```
                 border="0"></TD>
            <TD><A href="/PiggyBank/PiggyServlet?action=logout">Logout</A></TD>
        </TR>
    </menus:menuitem>
  </TBODY>
  </TABLE>
</menus:menu>
```

This JSP defines one menu tag, which contains three menuitem tags. Each menuitem defines a number of state tags. The state tags define the valid states for display of the menu item, as listed in Table 10-17 on page 284.

10.5.2 The menu tag handler

The job of the menu tag is simply to wrap a number of menuitem tags, and to provide them with access to the users navigation state. As such, the menu tag is not strictly necessary, we could implement the functionality we desire without it. However, from a usability point of view it makes sense to nest menuitem tags within a parent tag.

Another major advantage of defining the menu tag is that we can allow for the JSP author to define the name of the attribute that holds the users navigation state. To explain this further, remember that in the PiggyBank application, the attribute that holds this information is named *state*. We could code the tag handler for the menuitem tag to retrieve this attribute directly from the session, but this would limit the re-use potential of the tag library. If we change the name of the attribute, we must change and recompile the tag code.

Instead, we can make the value of the session attribute available as a variable of the menu tag handler. Other nested tags can then access this variable by a defined name through using the ancestor pattern, as described in "Ancestor" on page 278.

The menu tag is an implementation of the macro pattern. It can also be considered as an implementation of the declaration pattern, even though it doesn't make an attribute explicitly available through a scripting variable.

The following list is the relevant parts of the menu tag handler code.

```
//MenuTag.java
package com.ibm.itso.j2eebook.tags;

import java.io.IOException;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

/**
```

```
 * Handles the menu tag. This tag implements the macro pattern
 * and does not perform any conditional checking. Therefore
 * it extends TagSupport, and always return EVAL_BODY_INCLUDE
 * from doStartTag() and EVAL_PAGE from doEndTag().
 *
 * @author Adrian Spender
 */
public class MenuTag extends javax.servlet.jsp.tagext.TagSupport {
    // defines the id supplied as an attribute to the tag which
    // represents the HttpSession attribute containing the users
    // page navigation state.
    String stateKey = null;

    // This variable will make the value of the HttpSession attribute
    // defined by stateKey available to nested tags through the use
    // of findAncestorWithClass()
    String userState = null;

...

/**
 * doEndTag() is invoked by the JSP container when the end tag
 * is encountered.
 *
 * return EVAL_PAGE
 *
 * @exception javax.servlet.jsp.JspException
 */
    public int doEndTag() throws JspException {

        int result = EVAL_PAGE;
        release();
        return result;
    }

/**
 * doStartTag() is invoked by the JSP container when the begin tag
 * is encountered.
 *
 * By this point, the JSP container has initialized the stateKey
 * variable for us, so we use it to access its value from the
 * session context. If nothing is returned then we throw an
 * exception.
 *
 * @return EVAL_BODY_INCLUDE
 *
 * @exception javax.servlet.jsp.JspException If the session attribute
 * is null or non-existent.
 */
```

```
    public int doStartTag() throws JspException {

        int result = EVAL_BODY_INCLUDE;

        // retrieve the userState from the session context
        userState = (String)pageContext.getSession().getAttribute(stateKey);

        // if the user state is null then throw an exception
        if (userState == null)
            throw new JspException("Cannot access " + stateKey + " in session
scope.");

        // else, evaluate the body
        return result;
    }

... accessor methods for the bean properties
}
```

The most notable section of this code is the doStartTag() method which sets the userState instance variable by accessing the users HttpSession object through the pageContext object.

The doStartTag() method always returns EVAL_BODY_INCLUDE to tell the JSP container to evaluate the body contents. The doEndTag() method always returns EVAL_PAGE to continue page processing after the tag has been evaluated.

The userState variable can be accessed through accessor methods available, because we implement the tag handler as a JavaBean. The getUserState() accessor will be used by the menuitem tag handler.

10.5.3 The menuitem tag handler

The menuitem tag is the most involved of the three tags. It performs the following tasks:

►   Maintains a Vector object containing valid navigation states.

►   Determines if the tag body contents should be output.

The tag handler extends the javax.servlet.jsp.BodyTagSupport class to enable it to determine if the body content output should be displayed. It does not modify the body contents through post-processing and does not iterate over the body contents more than once.

The most notable parts of the tag handler are listed below.

```
//MenuitemTag.java
package com.ibm.itso.j2eebook.tags;
```

```
import java.io.IOException;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.util.*;

/**
 * Handles the menuitem tag. This tag implements the
 * ancestor and content capture patterns.
 *
 * It only outputs its body contents if the userState variable
 * of the parent menu tag is one of the valid states defined
 * in the validStates Vector.
 *
 * This tag extends BodyTagSupport, and always return EVAL_BODY_TAG
 * from doStartTag() and EVAL_PAGE from doEndTag().
 *
 * @author Adrian Spender
 */
public class MenuitemTag extends javax.servlet.jsp.tagext.BodyTagSupport {

    // This vector will contain the list of valid states
    // for the display of this menu item.
    Vector validStates = new Vector();

...

/**
 * doAfterBody() is invoked by the JSP container for BodyTagSupport classes
 * after each time the contents of the tag have been processed.
 *
 * To get the results of processing the tag contents,
 * use getBodyContent().getString()
 *
 * Within this method we call isValidState() to determine if the
 * body content should be processed. If true is returned, we
 * write the body content to the parent JspWriter.
 *
 * @return SKIP_BODY because we only want to process the body once.
 *
 * @exception javax.servlet.jsp.JspException
 *
 */
    public int doAfterBody() throws JspException {

        BodyContent body = getBodyContent();
        String content = body.getString();

        try {
```

```
                // if the userstate is not one of the valid states for this
                // menuitem then do not display it.
                if (isValidState())
                    getPreviousOut().write(content);
        } catch (IOException e) {
            e.printStackTrace();
            throw new JspException("IO error");
        }
        int result = SKIP_BODY;

        return result;
    }

... default doStartTag() doInitBody() and doEndTag() methods

/**
 * Compares the valid states for this menu item with the current page
 * navigation state made available through the parent menu tag. If
 * the current navigation state is contained within the valid states
 * we return true.
 *
 * @return true if we are in a valid state to display this menu item.
 */
    boolean isValidState() {

        // retrieve the current user state from the parent menu tag
        // using the ancestor pattern.
        String userState =
        ((MenuTag) findAncestorWithClass(this,
com.ibm.itso.j2eebook.tags.MenuTag.class))
        .getUserState();

        return validStates.contains(userState);
    }
}
```

This isValidState() method performs the main business logic of the tag. For this relatively simple example, this is contained within the tag handler itself, but for anything more complex, it would be abstracted into a JavaBean. The method uses the ancestor pattern to invoke the getUserState() method of the menu tag handler. This is then compared to the vector of valid states which the menuitem tag maintains.

The doAfterBody() method is called by the JSP container when the tag's body contents have been processed. The bodyContent property is used to retrieve the evaluated body contents. If the call to isValidState() returns true, then the contents are written to the parent JSP writer. If false is returned, indicating that the user is not in a valid state for this menu item, then the contents will not be written.

10.5.4 The state tag handler

The state tag handler requires one attribute, name, which defines a valid state for the display of the menu item it is nested within. The tag handler simply adds the value of name to the vector maintained by the parent menuitem tag through use of the ancestor pattern. The relevant parts of the state tag handler are shown below.

```
//stateTag.java
package com.ibm.itso.j2eebook.tags;

import java.io.IOException;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
/**
 * Handles the state tag. This tag implements the macro and
 * ancestor pattern.
 *
 * It extends TagSupport and always returns EVAL_BODY_INCLUDE from
 * doStartTag() and EVAL_PAGE from doEndTag().
 *
 * @author Adrian Spender
 */
public class StateTag extends javax.servlet.jsp.tagext.TagSupport {

    // defines the name supplied as an attribute to the tag which
    // represents a valid state
    private String name;

}

/**
 * doEndTag() is invoked by the JSP container when the end tag
 * is encountered.
 *
 * return EVAL_PAGE
 *
 * @exception javax.servlet.jsp.JspException
 */
    public int doEndTag() throws JspException {
```

```
              int result = EVAL_PAGE;
              release();
              return result;
       }

/**
 * doStartTag() is invoked by the JSP container when the begin tag
 * is encountered.
 *
 * Within this method, we use the ancestor pattern to add the defined
 * state to the list of valid states maintained by our parent menuitem
 * tag.
 *
 * @return EVAL_BODY_INCLUDE
 *
 * @exception javax.servlet.jsp.JspException
 */
    public int doStartTag() throws JspException {

        int result = EVAL_BODY_INCLUDE;

        // add the state specified to the menuitem vector
        if (name != null) {
            ((MenuitemTag)findAncestorWithClass(this,
com.ibm.itso.j2eebook.tags.MenuitemTag.class)).getValidStates().addElement(name
);
        }

        return result;
    }
}
```

The doStartTag() method accesses the menuitem validStates Vector through a
call to findAncestorWithClass().

10.5.5 Building the TLD

As relatively simple tags which neither define scripting variables, or have
complex combinations of attributes, TEI classes are not required. We can supply
enough information to enable the JSP container to translate the tags by
supplying a TLD file. This is shown in example below.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
 PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
 "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<taglib>
```

```
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>menu</shortname>
  <info>
Piggybank menu taglib
  </info>


  <tag>
    <name>state</name>
    <tagclass>com.ibm.itso.j2eebook.tags.StateTag</tagclass>
    <bodycontent>empty</bodycontent>
    <info>
      Defines a valid state for the display of the menu item
    </info>
    <attribute>
      <name>name</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>

  <tag>
    <name>menuitem</name>
    <tagclass>com.ibm.itso.j2eebook.tags.MenuitemTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <info>
      Defines a menu item
    </info>
  </tag>

  <tag>
    <name>menu</name>
    <tagclass>com.ibm.itso.j2eebook.tags.MenuTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <info>
      Defines a menu
    </info>
    <attribute>
      <name>stateKey</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>
</taglib>
```

10.6 Further resources

For further information on JSP custom tags, refer to the following resources:

- ► Chapter 6, "JSP Support", in the WebSphere V3.5 Handbook, SG24-6161
- ► The Apache taglib library, found at: `http://jakarta.apache.org/taglibs`

Part 4

Additional discussions

11

Application clients and J2EE communications

This chapter discusses the application clients, an application component type that must be supported by a J2EE platform as well as by Web applications or EJBs.

This chapter contains the following topics:

▶ Application clients

▶ Java Mail API

▶ JNDI

  – Naming Services
  – Directory Services

11.1 Application clients

Application clients are Java programming language that executes on a desktop computer. They are often referred as standalone applications, because they run in their own Java Virtual Machine (JVM). But as J2EE components, there are several differences. Graphical user interfaces are the common appearance of what the J2EE Specification refers to as application client.

A J2EE product must support this type of application component and provides a container to run application clients (see Chapter 2, "J2EE overview" on page 7). However, an application client does not run on the J2EE server, but on the client machine.

The application client container is required to provide the runtime support for the standalone application. An application client, as a Web application or as EJBs is able to use the set of standard services provided by the J2EE Specification and has access to the APIs listed in 2.3, "Standard services" on page 11.

Figure 11-1 shows the relationship between an application client and other J2EE components.
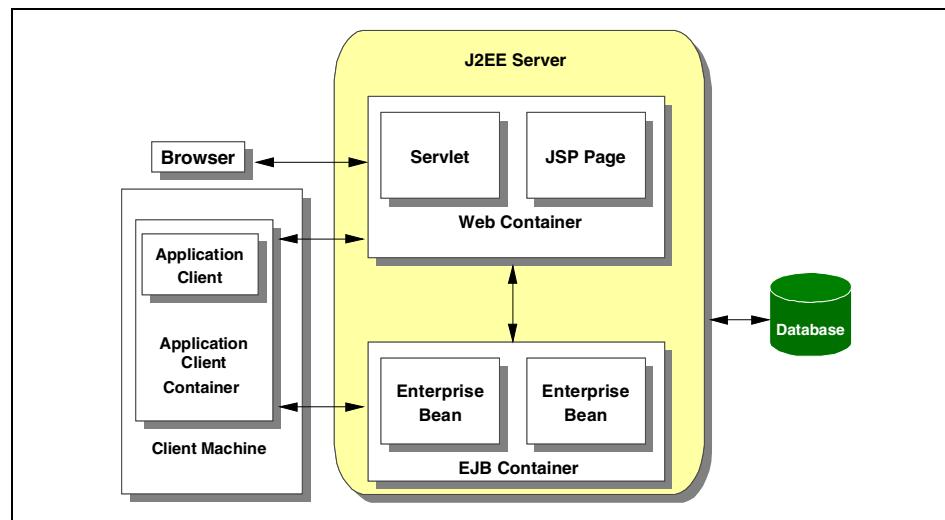
Figure 11-1   Application client in a J2EE environment

Enterprise applications mainly use server-side technologies, but standalone applications can be used for handling tasks such as system or application administration.

Finally, an application client can be packaged in a JAR file, with a deployment descriptor.

► **Security**: The application client container must authenticate the application user to satisfy the authentication and authorization constraints enforced by the enterprise bean containers and Web containers.

► **Transaction**: There is no requirement for the application client container to provide transaction management support for application clients.

► **Naming**: All the J2EE components application clients use the JNDI API to lookup enterprise beans, get access to resource managers, access configurable parameters set at deployment time, and so forth.

The deployment descriptor for an application client describes the enterprise beans and external resources referenced by the application. Access to resources must be configured at deployment time.

11.2 JNDI

Java Naming and Directory Interface. The purpose of a naming service is to associate a name with an object and provide a way of accessing the actual object given only its name. The directory services piece of JNDI adds the ability to associate attributes with the objects and look up the attributes of an object and search for objects based on their attributes. JNDI is used to access EJBs. JNDI can also be used to access remote objects in an RMI registry. You can also use JNDI to access LDAP directory services (not limited to LDAP).

Naming Services (javax.naming)
The `javax.naming` and `javax.naming.directory` packages encompass the set of JNDI APIs.

Looking up an Object
Use the `Context.lookup()` API to look up an object bound to a given name. For example:

```
Object anObject = ctx.lookup("textfile.txt");
```

The object returned can be cast to the object type:

```
File aFileObject = (java.io.File)ctx.lookup("textfile.txt");
```

Here's another example:

```
Printer myPrinter = (Printer)aNamespace.lookup("myPrinterName");
myPrinter.print("textfile.txt");
```

Listing the contents of a Context

To see what objects are available in a naming system, use the `Context.list()` API. The API returns an enumeration of `NameClassPair`. Each element of `NameClassPair` consists of the object name and class name. For example:

```
NamingEnumeration list = ctx.list("pictures");
while (list.hasMore())
{
NameClassPair pair = (NameClassPair)list.next();
System.out.println(pair);
}
```

This lists the files and directories in the "pictures" directory.

Adding a Binding

The `Context.bind()` API adds a binding to a context.

```
// Create the object to be bound

Cookie bigCookie = new Cookie("choclateChip");

// Perform the bind

ctx.bind("gottaHave", bigCookie);
```

This example creates an object of class Cookie and binds it to the name "gottaHave" in the context ctx. If you subsequently looked up the name "gottaHave" in ctx, then you would get the Cookie object.

Directory Services (javax.naming.directory)

The `javax.naming.directory.DirContext` interface provides methods to support operations involving `Attributes`. Similar to `Context`, a `DirContext` contains a set of name-object bindings.

The `getAttributes()` method of `DirContext` returns an `Attributes` object for the bound object with the given name. The `getAll()` method of the `Attributes` object returns a `NamingEnumeration` of `javax.naming.Attribute` objects, one for each attribute. The `get()` method of the `Attribute` object returns a single attribute value of type `java.lang.Object`.

DirContext Interface

```
Hashtable env = new Hashtable();

env.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.ldap.LdapCtxFactory");

env.put(Context.PROVIDER_URL,
"ldap://localhost:80/o=IceCreamFlavors");
```

```
DirContext ctx = new InitialDirContext(env);
```

Attributes Interface

```
String name = "";

name ="cn=Ice Cream, ou=Dessert";

Attributes flavors =
((DirContext)ctx).getAttributes(name);
```

The Attributes interface also contains a getIDs() method that returns an enumeration of the attribute names.

Attribute Interface

To obtain an enumeration of the Attribute objects:

```
NamingEnumeration allFlavors = flavors.getAll();

while (allFlavors.hasMore()) {
  Attribute flavor = (Attribute)allFlavors.next();
  NamingEnumeration allKinds = flavor.getAll();
  while(allKinds.hasMore())
    System.out.println("Kind of IceCream: "+allKinds.next());
}
```

To obtain a single Attribute:

```
Attribute flavor = flavors.get("vanilla");

IceCream vanillaIceCream = (IceCream)flavor.get();
```

Note that if the attribute is multi-valued then what the get() method returns is service-provider specific.

For multi-valued attributes:

```
Attribute flavor = flavors.get("vanilla");

NamingEnumeration kindsOfVanilla =
flavor.getAll();//enum of multi-valued attribute; ie french vanilla,
natural vanilla bean, country vanilla, etc...

while (kindsOfVanilla.hasMore())
  System.out.println(kindsOfVanilla.next());
```

Other useful methods on the Attribute interface include the contains() and size() methods.

```
boolean hasVanillaRumRaisin = attr.contains("rum raisin");

int numValues = attr.size();//how many values this attr has
```

11.2.1 JNDI properties

Resource bundles can be used to lookup properties. The sample code uses a
`NamingHelper` to do the lookup.

```
// Naming helper (com.ibm.itso.j2eebook.helpers)
import java.util.*;
import javax.naming.*;
/**
 * Provides access to JNDI properties for lookups. The properties
 * are stored in two resource files that are read when this class is
 * instantiated. The files contain the following:
 * <UL><LI>JNDI_LOOKUP_PROPERTIES:<BR>
 *      <UL><LI>The name of the InitialContextFactory class to use</LI>
 *          <LI>The IIOP URL for the Persistant name server</LI></UL></LI>
 *      <LI>JNDI_NAMES:<BR>
 *      <UL><LI>The JNDI names of any resources.</LI></UL></LI></UL>
 *
 * This class is implemented as a singleton.
 *
 * @author: Adrian Spender
 */
public class NamingHelper {

    // singleton instance variable
    private static NamingHelper instance = null;

    // InitialContext object
    private static InitialContext ctx = null;

    // default names which are used if the files are not present, or
    // cannot be read. These are for WAS 3.5 on the same server
    private static final String defaultContextFactoryName =
"com.ibm.ejs.ns.jndi.CNInitialContextFactory";
    private static final String defaultLookupURL = "iiop:///";

    // names of the .properties files
    private static final String JNDI_LOOKUP_PROPERTIES = "PiggyBankJNDILookup";
    private static final String JNDI_NAMES = "PiggyBankJNDINames";

    // Resource bundles to hold the file contents
    private static PropertyResourceBundle JNDIProperties;
    private static PropertyResourceBundle JNDINames;
}
```

The `getJNDIName()` method takes a key an an argument and returns the
associated resource name:

```
// getJNDIName method
/**
```

```
 * Retrieves the JNDI name for a specified resource. If we cannot find the
 * name, for instance if the property file is corrupt, then we return
 * the name provided to us.
 *
 * @param String Fully qualified class name of the EJB we want to look up
 *
 * @return The JNDI name to use
 */
public String getJNDIName(String name) {
   String dynamicJNDIName = null;

   try{
       dynamicJNDIName = JNDINames.getString(name);
   } catch (MissingResourceException mre){}

    if (dynamicJNDIName != null)
        return dynamicJNDIName;
    else
        return name;
}
```

11.3 JavaMail

J2EE provides RMI/IIOP, JMS and JavaMail for communications. A messaging
API is loosely coupled versus a tightly coupled API, such as RMI, since the
messaging does not wait for a response. Before looking at the JavaMail API, let
us discuss what the Java Mail API is and is not.

What JavaMail is:

- ► A mail user agent
- ► Used to read and write mail
- ► Independent of the provider protocol

What JavaMail is **not**:

- ► Not a mail transfer agent
- ► Not used to deliver mail
- ► Not for forwarding mail
- ► Not for transporting mail

SMTP

This defines the mechanism for delivery of mail. A JavaMail application would
communicate with the SMTP server for a particular ISP. The SMTP server relays
the message to the SMTP server at the destination, and the recipient gets the
message through the use of POP, for instance.

POP3, IMAP

A mechanism used to store and retrieve mail to and from an SMTP server. POP supports a single mailbox for each user. Other functionality like seeing how many new messages there are or showing a list of unread messages is a function of the mail program (that is, Microsoft Outlook). IMAP is similar to POP, but additionally, it is able to access messages from more than one computer which has become extremely important as reliance on electronic messaging and use of multiple computers increases.

MIME

This defines the content of the message, for instance, the format and the attachments.

To handle non-plain text mail content (MIME, URL and file attachments) the JavaBeans activation framework is required. J2EE 1.2 requires support of JAF 1.0.

11.3.1 JavaMail API

These are the core classes:

- ► Session
- ► Message
- ► Address
- ► Authenticator
- ► Transport
- ► Store
- ► Folder

Core classes

Here is a brief summary of the core classes.

Session (javax.mail.session)

This defines a mail session. The `getDefaultInstance` and `getInstance` methods are used to obtain a default session and a unique session, respectively.

```
Session session = Session.getDefaultInstance(properties, null);

Session session = Session.getInstance(properties, null);
```

The `null` argument can be replaced with a valid `Authenticator` object.

Message (javax.mail.message)

A subclass of this abstract class can be used to create the message to send (that is, `javax.mail.internet.MimeMessage`).

You can set the message:

```
MimeMessage theMessage = new MimeMessage(theSession);
```

You can set the content of the message:

```
message.setContent("Not feeling very sleepy", "text/plain");
```

You can set the content of the message for plain text:

```
message.setText("Had too much Java");
```

You can set the subject for the message:

```
message.setSubject("Sleepless in Seattle");
```

Address (javax.mail.Address)

A subclass of this abstract class can be used to address the mail item (that is, `javax.mail.internet.InternetAddress`).

```
Address anAddress = new InternetAddress("anthony@floridabeach.com");
```

The sender is identified with the `setFrom` and `replyTo` methods:

```
message.setFrom(anAddress);
```

For multiple senders:

```
message.addFrom(addresses);
```

To specify recipients:

```
message.addRecipient(Message.RecipientType.TO, toAddress);
message.addRecipient(Message.RecipientType.CC, ccAddress);
```

Authenticator (javax.mail.Authenticator)

An abstract class used to protect mail resources on the mail server.

```
Authenticator anAuthenticator = new MyAuthenticator();
Session theSession =
Session.getDefaultInstance(properties, anAuthenticator);
```

Transport (javax.mail.Service)

This is protocol specific. The transport understands the language used to send the message (that is, SMTP). It is an abstract class.

The basic send() makes a connection to the server for each method call:

```
Transport.send(theMessage);
```

To keep the connection active (for multiple messages):

```
theMessage.saveChanges();
Transport aTransport = theSession.getTransport("SMTP");
aTransport.connect(host, user, pswd);
aTransport.sendMessage(theMessage, theMessage.getAllRecipients());
aTransport.close();
```

Store and Folder (javax.mail.Store & javax.mail.Folder)

The process goes something like this: get the session, connect to a store, tell the store the protocol to use, get the folder, open the folder, get the message, get the message content, close the folder and close the store.

```
Store aStore = session.getStore("POP3");

aStore.connect(host, user, pswd);

Folder aFolder = aStore.getFolder("INBOX");

aFolder.open(Folder.READ_ONLY);

Message theMessage[] = aFolder.getMessages();
System.out.println( ((MimeMessage)theMessage).getContent() );

aFolder.close(aBoolean);

aStore.close();
```

Using the APIs

Next, we describe how an application might use these APIs.

Sending

Sending messages involves: getting a session, creating a message, specifying the message content, and sending the message.

```
Properties properties = System.getProperties();
properties.put("mail.smtp.host", host);

// theSession
Session theSession = Session.getDefaultInstance(properties, null);

// theMessage
MimeMessage theMessage = new MimeMessage(theSession);

// from and to
theMessage.setFrom(new InternetAddress(from));
theMessage.addRecipient(Message.RecipientType.TO,
                        new InternetAddress(to));
```

```
theMessage.setSubject("About PiggyBank");
theMessage.setText("Welcome to PiggyBank");
Transport.send(theMessage);
```

Retrieving

Retrieving a message involves: getting a session, getting a store, connecting to a store, opening a folder, and getting the message.

```
Store aStore = session.getStore("POP3");

aStore.connect(host, user, pswd);

Folder aFolder = store.getFolder("INBOX");

aFolder.open(Folder.READ_ONLY);

Message message[] = aFolder.getMessages();

...process each message

aFolder.close(false);

aStore.close();
```

Deleting

Messages are deleted through the use of Flags:

- ► `Flags.Flag.ANSWERED`
- ► `Flags.Flag.DELETED`
- ► `Flags.Flag.DRAFT`
- ► `Flags.Flag.FLAGGED`
- ► `Flags.Flag.RECENT`
- ► `Flags.Flag.SEEN`
- ► `Flags.Flag.USER`

To delete a message:

```
Message theMessage = aFolder.getMessage(1);
theMessage.setFlag(Flags.Flag.DELETED, true); //sets the DELETED flag

// Checks if DELETED flag is set for theMessage
if (theMessage.isSet(Flags.Flag.DELETED))
  System.out.println("DELETED message");
```

Replying

The `reply()` method is used to configure a new message.

```
MimeMessage aReply = (MimeMessage)theMessage.reply(false);
   //where false means reply only to the sender
   //and true means reply to all

aReply.setFrom(new InternetAddress("anthony@floridabeach.com"));

aReply.setText("take me off your mailing list!");

Transport.send(aReply);
```

Forwarding

Forwarding messages evolves like this:

► Messages contain one or more **BodyParts**.

► BodyParts can be combined into a container called **Multipart**.

► So, create the BodyParts:

– The 1st BodyPart is the message to forward.

– The 2nd BodyPart is any new text to be appended.

► Combine the BodyParts into a Multipart.

► Add the Multipart to a properly addressed message.

► Send the message.

For forwarding BodyParts:

```
Message forward = new MimeMessage(theSession);

// Header, from and to
forward.setSubject("Fwd: "+existingMessage.getSubject());
forward.setFrom(new InternetAddress(from));
forward.addRecipient(Message.RecipientType.TO,
                     new InternetAddress(to));

// BodyParts
BodyPart bodyPart1 = new MimeBodyPart();
BodyPart bodyPart2 = new MimeBodyPart();

// specify new content
bodyPart1.setText("Parts is Parts:");

// forward the existing content
bodyPart2.setDataHandler(existingMessage.getDataHandler());
```

```
// Multipart
Multipart multipart = new MimeMultipart();
multipart.addBodyPart(bodyPart1);
multipart.addBodyPart(bodyPart2);

forward.setContent(multipart);
Transport.send(forward);
```

Attachments

Resources can be attached to and detached from messages using the JavaMail API.

Sending attachments involves:

► Specify the attachment datasource:

    – FileDataSource
    – URLDataSource

► Use the datasource to set the DataHandler.

► Use the DataHandler to attach the BodyPart.

► Set the file name associated with the attachment.

```
// Part1 contains the message

multipart.addBodyPart(bodyPart1);

// Part two is the attachment

bodyPart2 = new MimeBodyPart();

DataSource source = new FileDataSource(filename);

bodyPart2.setDataHandler(new DataHandler(source));

bodyPart2.setFileName(filename);

multipart.addBodyPart(bodyPart2);

theMessage.setContent(multipart);

Transport.send(theMessage);
```

Receiving attachments involves:

► Get the content.

► Process each part:

    – Get the content
    – Get the attachments

HTML and Images

HTML and Image content can also be processed. To send an HTML file:

```
String html = "<H1>PiggyBank</H1>";

theMessage.setContent(html, "text/html") );
```

When receiving the HTML message, the JavaMail API sees the incoming message as a stream of bytes.

Instead, to treat the image as an attachment, you must:

▶ Tell the Multipart that the parts are related.

▶ Reference the image with a "cid URL".

```
BodyPart bodyPart1 = new MimeBodyPart();

String html = "<H1>PiggyBank</H1>"+"<img src=\"cid:yadayada\">";

bodyPart1.setContent(html, "text/html");

// a "related" Multipart

MimeMultipart multipart = new MimeMultipart("related");

multipart.addBodyPart(bodyPart1);

// Fetch the image and associate to part

DataSource source = new FileDataSource(theImageFile);

BodyPart bodyPart2 = new MimeBodyPart();

bodyPart2.setDataHandler(new DataHandler(source));

bodyPart2.setHeader("Content-ID","yadayada");the cid is a reference
to the Content-ID header of the attachment
```

12

Deploying J2EE applications to WebSphere

This chapter discusses deployment issues as the J2EE Specification provides a complete set of instructions to package application components together to make a J2EE application.

WebSphere Application Server 4.0 fully supports the J2EE model, providing tools to package and deploy applications. WebSphere Studio for the Web application and the Application Assembly Tool for creating all the J2EE modules can be used.

WebSphere 4.0 introduces changes to the runtime support and application packaging.

12.1 J2EE deployment model

The J2EE Specification gives special instructions on how to assemble application components in order to deploy them. How components are packaged together is discussed in section 2.6, "Deployment" on 17. Components are packaged into modules, each module containing a deployment descriptor. A J2EE application is a file with an EAR extension containing several J2EE modules.

Figure 12-1 summarizes the different modules a J2EE application may contain.
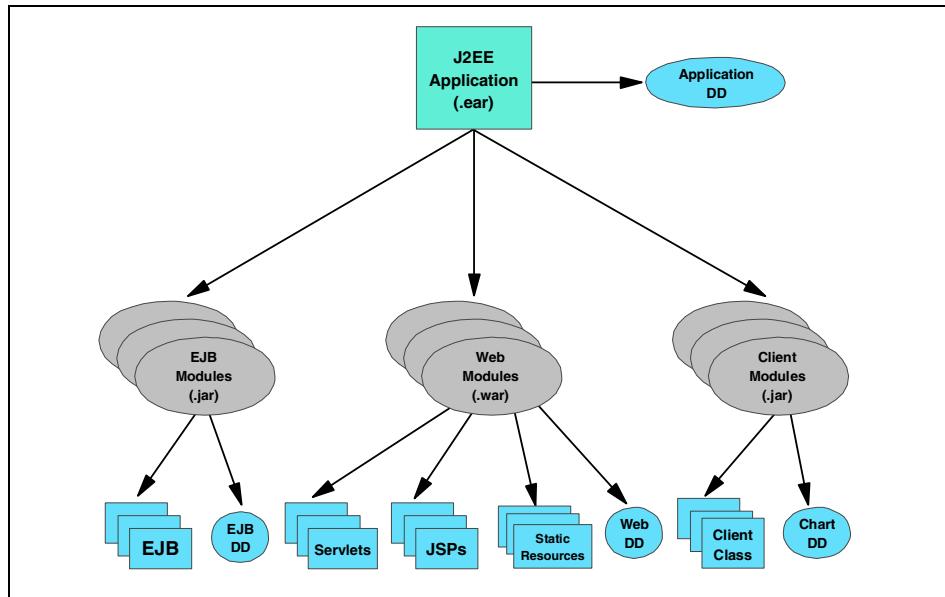
Figure 12-1   J2EE deployment model overview

There are basically three kinds of modules that can be used in a J2EE application:

▶ **EJB module**: This module contains the class files for the Enterprise JavaBeans components. It must conform to the EJB 1.1 Specification. The EJB module is discussed in Chapter 5.

▶ **Web module**: This module contains the Web components of an application, that is to say servlets, JSPs and other Web resources. These Web components are discussed in Chapters 8 and 9.

▶ **Application client module**: This module contains all the Java stand-alone applications. It is discussed in Chapter 11.

Each of these modules contain its own deployment descriptor (we will refer to the appropriate chapter to see the content of this file for each kind of J2EE module). This modules are packaged in an EAR file which also contains a deployment descriptor.

The deployment descriptor of an EAR file indicates the modules contained in the application and their place in the archive.

WebSphere Application Server supports the EAR format. Through the Application Assembly Tool, it provides a graphical interface to create these modules and to assemble them into an EAR file.

12.2 Assembly and deployment of PiggyBank

To deploy the PiggyBank application to WebSphere Application Server 4.0, we need to build 2 J2EE modules: a Web application module and an EJB module.

12.2.1 The Web application creation

Creating a Web application requires to place the Web components of our application into an archive with a specific directory structure. The archive must also have a deployment descriptor and a WAR extension.

Creating the file

Websphere Studio allows a developer to create a Web archive easily. It provides a tool to generate the deployment descriptor for a particular project. Under WebSphere application Server 4.0, the Application Assembly Tool also provides a graphical interface to package servlets, JSPs, HTML pages and other content.

Figure 12-2 shows the main menu of the Application Assembly Tool for a Web application.
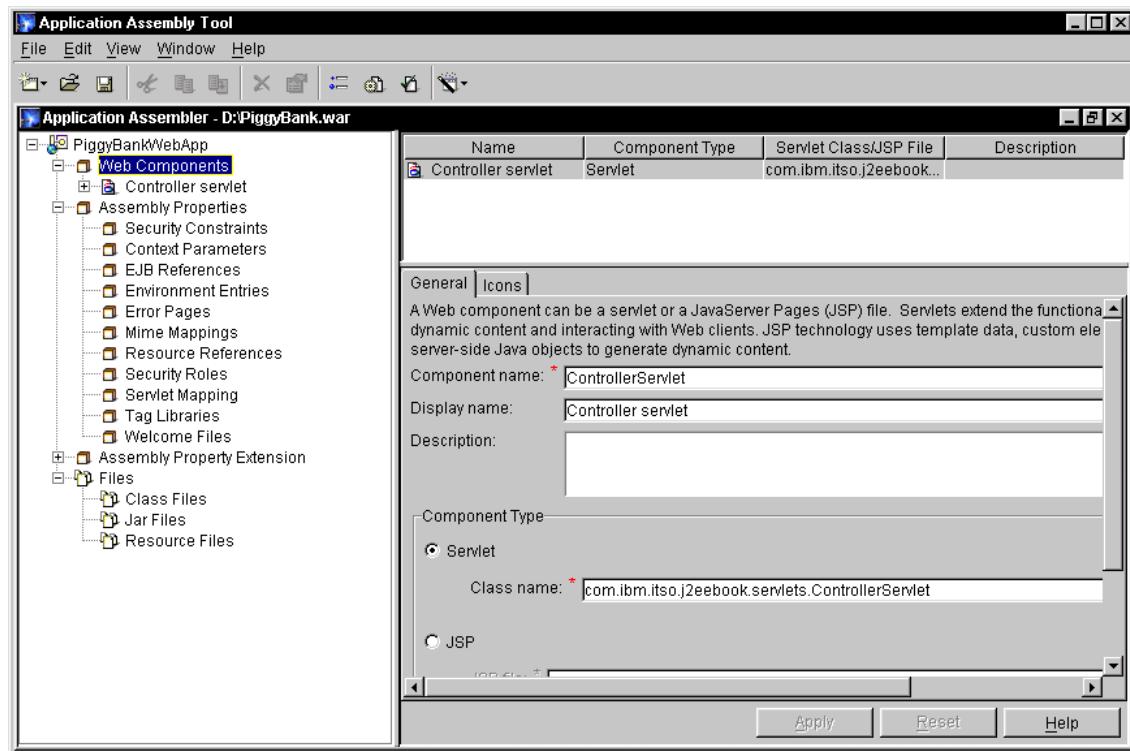
Figure 12-2   The Application Assembly Tool view for a Web application

The Application Assembly Tool allows you to specify all the information needed to generate the deployment descriptor for a Web application (see "The deployment descriptor" on page 173 to have an overview of the parameters that can be set in this file).

Content of the Web application

For the PiggyBank scenario, whatever is the tool used to create the WAR file, the archive must contain the following files in the appropriate directory (Figure 12-3).
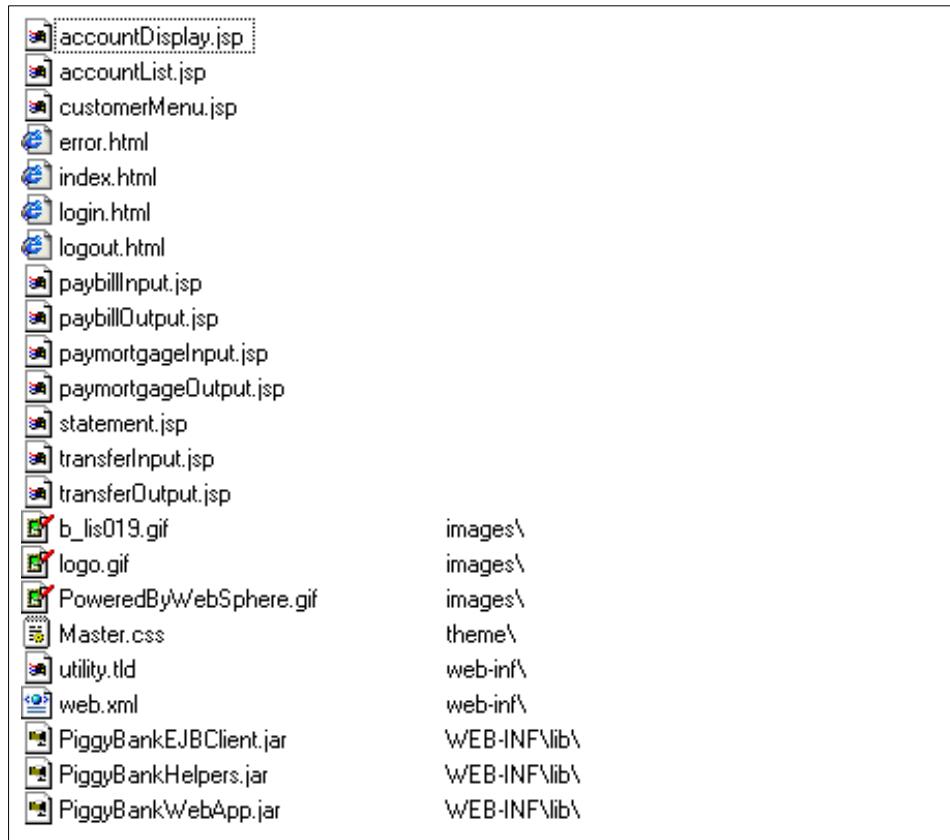
Figure 12-3   PiggyBank Web application archive content

The directories of the archive contain the following elements:

▶ **Root directory**: The root directory contains all the html files and the JSPs used by the application.

▶ **WEB-INF directory**: This directory contains the deployment descriptor and the JAR files of the application (in the subdirectory lib):

   – PiggyBankEJBClient.jar: This archive contains all the EJB client classes required by the application.

   – PiggyBankHelpers.jar: This archive contains the class files used to invoke the EJBs. Helper objects act as a layer between the servlet and the EJBs.

   – PiggyBankWebApp.jar: This archive contains the servlet class file and all the classes required by the application (the data beans are included in this JAR file).

The directory also contains the tag library used by the application.

WebSphere Application Server also adds a binding file and an extension file in the directory to determine how objects with local names are referenced at runtime. They are configured in the AAT and stored in an archive in files called `ibm-web-bnd.xmi` for the binding and `ibm-web-ext.xmi` for a specific type of module and any extension for WebSphere.

▶ **Other directories**: Images and theme contain other static resources belonging to the application.

The deployment descriptor of our Web application

The deployment descriptor contains information used by the servlet container to configure the runtime environment for servlets and JSPs.

The following code shows the deployment descriptor for the PiggyBank Web application, that is to say, the content of the web.xml file located in the Web-Inf directory (Example 12-1).

Example 12-1

```
//PiggyBank Web application deployment descriptor
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.2//EN" "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>

    <display-name>PiggyBankWebApp</display-name>
    <servlet>
        <servlet-name>ControllerServlet</servlet-name>
<servlet-class>com.ibm.itso.j2eebook.servlets.ControllerServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>ControllerServlet</servlet-name>
        <url-pattern>/PiggyServlet</url-pattern>
    </servlet-mapping>

    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
    </welcome-file-list>

    <error-page>
        <error-code>404</error-code>
        <location>/error.html</location>
    </error-page>

    <taglib>
        <taglib-uri>http://jakarta.apache.org/taglibs/utility</taglib-uri>
        <taglib-location>/WEB-INF/utility.tld</taglib-location>
```

```
        </taglib>

</web-app>
```

The deployment descriptor is a set of tags with values. We also refer to "The deployment descriptor" on page 173 for an overview of the tags. The deployment descriptor shown in Example 12-1 specifies the following information for the Web application:

- ► **tag <servlet>**: This tag gives specific information for the servlets contained within the application. Here the name of the servlet (ControllerServlet will be the name used by the application) and the full path leading to its class file are specified.

- ► **tag <servlet-mapping>**: This tag associates a particular servlet (given its name) to a set of URIs. In our example all the requests to the application finishing with the syntax /PiggyServlet will be handled by the ControllerServlet. If our application is deployed with the /PiggyBank context in the server ourhost.com, a request for the following URL, http://ourhost.com/PiggyBank/PiggyServlet, will be handled by the ControllerServlet servlet.

- ► **tag <welcome-file>**: This tag maps a particular resource to a request to the application within which no resource is specified. If our application is deployed within the context /PiggyBank, requests, such as http://ourhost.com/PiggyBank, will be served by the index.html file.

- ► **tag <error-page>**: This tag allows you to handle particular application errors by responding to the request with a specified file in case of a specified error.

- ► **tag <taglib>**: This tag is used to describe a particular library that the JSPs are using in the application.

12.2.2 The EJB 1.1 JAR file creation

VisualAge for Java 4.0 has a tool to export EJB with XML deployment descriptor. We discuss this in 5.6, "VA Java and EJBs" on page 100.

12.2.3 Assembling the application in an EAR file

J2EE modules can be packaged together in a J2EE application. A J2EE application is typically represented by an archive file with the extension EAR (for Enterprise ARchive).

For the PiggyBank application, the Application Assembly Tool in Websphere Application Server 4.0 can be used to create the EAR file. It is required to create an archive and to assemble the two modules in this archive with a deployment descriptor.

The following code shows the deployment descriptor for the entire application.

```
PiggyBank J2EE application deployment descriptor
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application
1.2//EN" "http://java.sun.com/j2ee/dtds/application_1_2.dtd">
<application>

    <display-name>PiggyBank</display-name>
    <description>EAR file for PiggyBank application, contains the web
application and the EJB module </description>

    <module>
        <web>
            <web-uri>PiggyBank.war</web-uri>
            <context-root>/PiggyBank</context-root>
        </web>
    </module>

    <module>
        <ejb>PiggyBank.jar</ejb>
    </module>

</application>
```

The deployment descriptor contains the place of the modules contained within the application and their relative location in the archive. For the Web application the context-root tag allows you to specify the context in which the application will be deployed on the server.

In our example, the EAR file contains:

- The deployment descriptor, named application.xml
- The Web module, PiggyBank.war
- The EJB module, PiggyBank.jar

12.2.4 Deployment of the J2EE application

Once the archive is created, it can be deployed to WebSphere Application Server (see 12.2.3, "Assembling the application in an EAR file" on page 319).

Because WebSphere fully supports the J2EE Specification, an enterprise application can be deployed using the EAR file.

WebSphere Application Server will generate a binding file and also an extension file when deploying the application to determine how objects with local names are referenced at runtime. These bindings and extensions may be configured in the Console, they are stored in files called `ibm-application-bnd.xmi` and `ibm-application-ext.xmi`.

<div style="text-align: right">

A

</div>

Additional material

This redbook refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

`ftp://www.redbooks.ibm.com/redbooks/SG246124`

Alternatively, you can go to the IBM Redbooks Web site at:

`ibm.com``/redbooks`

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG24-6124.

Using the Web material

The additional Web material that accompanies this redbook includes the following files:

File name *Description*
SG246124-J2EE.zip Sample Programs

323

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space: 10MB for sample codes
Operating System: Windows NT/2000
Processor: Pentium III 450 or higher
Memory: 256MB or above

How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material zip file into this folder.

Files will be created into following directories.

► DB

Required files to create sample database which is used in this book.

► deploy

Deployed codes to WebSphere Application Server 3.5.3.

► MQSeries

Required files to use samples related MQSeries.

► src

All source codes in this book.

► Studio

WebSphere Studio related codes.

► VAJ

Repository files for VisualAge for Java 3.5.3

How to use the sample

Here is how to use the sample.

Creating database tables

EJBBank database must be created prior to run PiggyBank sample in this book. To create the database:

► Make sure DB2 is running.

► Open IBM DB2, Command Window.

► Go into db directory of our sample codes.

- ▶ Run Createbank.bat which uses ejbbank.ddl described in "The DDL file used for the code fragments" on page 83.
- ▶ Then run Loadbank.bat which creates all records to tables created above.
- ▶ You can confirm all records are created running by Listbank.bat.

Deploying samples

Please refer to 12.2, "Assembly and deployment of PiggyBank" on page 315.

Source codes

Source codes are separated into two directories under the src directory.

- ▶ Java

  All source codes which have a *.java extension. All codes are created under com.ibm.itso.j2eebook package and divided into following subpackages.
  - – dataaccess
  - – databeans
  - – ejb
  - – exceptions
  - – helpers
  - – servlets
  - – tags
- ▶ Web

  All Web materials exclude Java source codes (*.html, *.gif, *.jsp, *.css and etc.) and are divided into following subdirectories.
  - – \(root)

    html and JSP files
  - – images

    gif files
  - – theme

    css (cascade style sheet) files
  - – WEB-INF

    tld (tag library descriptor) files

Importing repository

Select PiggyBank.dat in the vaj directory to import to VisualAge for Java Enterprise Edition Version 3.5 and above.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see "How to get IBM Redbooks" on page 328.

► *EJB Development with VisualAge for Java for Websphere Application Server,* SG24-6144

► *WebSphere V3.5 Handbook,* SG24-6161

► *Design and Implement Servlets, JSPs and EJBs for IBM WebSphere Application Server,* SG24-5754

► *Programming with VisualAge for Java Version 3.5,* SG24-5264

► *Servlet and JSP Programming with IBM WebSphere Studio and VisualAge for Java,* SG24-5755

Other resources

These publications are also relevant as further information sources:

► Kyle Brown, et al. *Enterprise Java Programming with IBM WebSphere.* Addison Wesley Professional, 2001, ISBN 0201616173.

► David Flanagan, et al. *Java Enterprise in a Nutshell: A Desktop Quick Reference.* O'Reilly and Associates, 1999, ISBN 1565924835.

► *Professional JMS*. Wrox Press Inc., 2001, ISBN 1861004931.

► *MQSeries Using Java*, SC34-5456

Referenced Web sites

These Web sites are also relevant as further information sources:

► http://www.ibm.com/software/webservers/appserv/whitepapers.html WebSphere Application Server — White Papers

- http://www-4.ibm.com/software/webservers/appserv/whitepapers/was_mqseries.pdf WebSphere JMS/JTA support for MQSeries Overview

- http://www.ibm.com/software/webservers/appserv/support.html
WebSphere Application Server — Support

- http://www.ibm.com/websphere
WebSphere Software Platform

- http://www.ibm.com/developerworks/patterns
IBM Patterns for e-business

- http://www.ibm.com/software/ts/mqseries/txppacs
IBM MQSeries Family SupportPacs

- http://www.ibm.com/framework/patterns
IBM Patterns for e-business

- http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd
Tag Library Descriptor

- http://java.sun.com/j2ee/dtds/web-app_2_2.dtd
Elements

- http://java.sun.com/j2ee
The J2EE Specification

- http://java.sun.com/j2ee/j2sdkee/devguide1_2_1.pdf
Java 2 Enterprise Edition Developer's Guide

- http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd
Method element syntax

- http://java.sun.com/products/servlet/download.html
Implementations and Specifications

- http://java.sun.com/j2ee/dtds/application_1_2.dtd
Deployment descriptor

- http://oss.software.ibm.com/developerworks/projects/bsf
bsf Project

- http://jakarta.apache.org/taglibs
Taglibs

How to get IBM Redbooks

Search for additional Redbooks or Redpieces, view, download, or order hardcopy from the Redbooks Web site:

ibm.com/redbooks

Also download additional materials (code samples or diskette/CD-ROM images) from this Redbooks site.

Redpieces are Redbooks in progress; not all Redbooks become Redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

IBM Redbooks collections

Redbooks are also available on CD-ROMs. Click the CD-ROMs button on the Redbooks Web site for information about all the CD-ROMs offered, as well as updates and formats.

Special notices

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

The following terms are trademarks of other companies:

Tivoli, Manage. Anything. Anywhere.,The Power To Manage., Anything. Anywhere.,TME, NetView, Cross-Site, Tivoli Ready, Tivoli Certified, Planet Tivoli, and Tivoli Enterprise are trademarks or registered trademarks of Tivoli Systems Inc., an IBM company, in the United States, other countries, or both. In Denmark, Tivoli is a trademark licensed from Kjøbenhavns Sommer - Tivoli A/S.

C-bus is a trademark of Corollary, Inc. in the United States and/or other countries.

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

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

PC Direct is a trademark of Ziff Communications Company in the United States and/or other countries and is used by IBM Corporation under license.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States and/or other countries.

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

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

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

Glossary

A

Application Assembly Tool (AAT). WebSphere's Application Assembly Tool creates a deployment descriptor for each module in EAR.

Abstract class. A class that provides common behavior across a set of subclasses but is not itself designed to have instances. An abstract class represents a concept; classes derived from it represent implementations of the concept. See also *base class*.

Access beans. Access beans are Java Bean Wrappers for enterprise beans that are typically used by client programs. Access beans adapt enterprise beans to the JavaBeans programming model and hide the complexity associated with the home and remote interfaces.

Accessor methods. Methods that an object provides to define the interface to its instance variables. The accessor method to return the value of an instance variable is called a *get* method or *getter* method, and the mutator method to assign a value to an instance variable is called a *set* method or *setter* method.

Ancestor tag. An ancestor tag is one that accesses information contained in another tag within which it is nested.

Applet. A Java program designed to run within a Web browser. Contrast with application.

Application. In Java programming, a self-contained, stand-alone Java program that includes a main() method. Contrast with applet.

Application Programming Interface (API). A software interface that enables applications to communicate with each other. An API is the set of programming language constructs or statements that can be coded in an application program to obtain the specific functions and services provided by an underlying operating system or service program.

B

Business-to-business (B2B). Patterns for e-business define reusable assets that can help speed the process of developing applications. Business-to-business between parties who do not belong to the same company.

Base class. A class from which other classes or beans are derived. A base class may itself be derived from another base class. See also *abstract class*.

Bean. A definition or instance of a Java-Beans component. See also *JavaBeans*.

BeanInfo. (1) A companion class for a bean that defines a set of methods that can be accessed to retrieve information on the bean's properties, events, and methods. (2) In the VisualAge for Java IDE, a page in the Class Browser that provides bean information.

Bean Managed Persistence (BMP). One of the style of the Entity Bean. It is requires the application developer to implement the methods necessary to persist the state of the bean. The BMP implementation could handle arbitrarily complex relationships and table joins.

C

Class. A template that defines properties, operations, and behavior for all instances of that template.

CLASSPATH. (1) In VisualAge for Java the lists of pathnames which will be searched for dynamically loaded classes, BeanInfo information and external source for debugging. (2) In your deployment environment, the environment variable that specifies the directories in which to look for class and resource files.

Container managed persistence. Container-managed persistence implies that the EJB Container on the application server will manage the persistence of the entity bean. The code to persist the bean is generated by a vendor-specific tool.

Connection pooling. Pool and reuse the pre-connected connections to DBMS or Message queue to reduce the time to connect.

Cookie. A cookie is composed of a name and a value, it is stored in the client browser and can be retrieved on the server when the client comes back.

Data Base Management System (DBMS). A complex set of programs that control the organization, storage and retrieval of data for many users; extensively used in business environments.

Data Definition Language (DDL, SQL). A language that describes data and their relationships in a database. It is composed of data definition statements that create, alter, or destroy database objects such as tables, aliases, views, and indexes.

Deployment. Release the codes to the runtime environment. The deployment task generally consists of three steps: installation of the components on the server (or container), followed by configuration and execution of the application. Deployments descriptors assist in this deployment task.

Deployment descriptor. A deployment descriptor is an XML file describing how to assemble and deploy a unit into a specific environment. It is provided with the application and specifies the required external resources, as well as the security requirements and environment parameters.

Digital certificates. An electronic counterparts to driver licenses, passports and membership cards. You can present a Digital Certificate electronically to prove your identity or your right to access information or services online.

Directory services. Part of JNDI. This adds the ability to associate attributes with the objects and look up the attributes of an object and search for objects based on their attributes.

Distributed transactions. Within a distributed transaction, message sends and receives are transacted as far as the local queue manager the connection object is associated with.

Document Type Definition (DTD). A way of describing the structure of an XML or SGML document and how the document relates to other objects.

E

Enterprise ARchive (EAR). EAR file has the .ear extension, it contains a deployment descriptor and one or more J2EE modules, each of these contains its own deployment descriptor.

Enterprise Java Beans(EJB). An EJB is a server-side component representing an object in the business model of the enterprise application.

EJB client. Servlets, JSPs, Java applications and any other application access to an EJB can be an EJB Client.

EJB container. A container which can maintain and make available multiple EJB objects and object types for remote use.

G

Graphical User Interface (GUI). A type of interface that enables users to communicate with a program by manipulating graphical features, rather than by entering commands. Typically, a GUI includes a combination of graphics, pointing devices, menu bars and other menus, overlapping windows, and icons.

H

Hypertext Markup Language (HTML). The basic language that is used to build hypertext documents on the World Wide Web. It is used in basic, plain ASCII-text documents, but when those documents are interpreted, or *rendered*, by a Web browser such as Netscape, the document can display formatted text, color, a variety of fonts, graphical images, special effects, hypertext jumps to other Internet locations, and information forms.

Hypertext Transfer Protocol (HTTP). The protocol for moving hypertext files across the Internet. Requires an HTTP client program on one end, and an HTTP server program on the other end. HTTP is the most important protocol used in the World Wide Web.

I

Inheritance. (1) A mechanism by which an object class can use the attributes, relationships, and methods defined in classes related to it (its base classes). (2) An object-oriented programming technique that allows you to use existing classes as bases for creating other classes.

Instance. Synonym for *object*, a particular instantiation of a data type.

InstantDB. Simple files based DBMS. Bundled with WebSphere Application Server package as an administrative DB. Instant DB does not support cloning of application servers, redirection of application requests from Web servers on one machine to application servers on another, access from remote machines, distributed transactions, multinode function, workload manager persistence, or full CMP function on EJBs.

Interface. A named set of method declarations that is implemented by a class.

Interface Definition Language (IDL). This is a language provided by OMG to define the interface for an object in a distributed CORBA environment.

Internet Message Access Protocol (IMAP). A method of accessing electronic mail or bulletin board messages that are kept on a mail server.

Internet Protocol (IP). The protocol that provides basic Internet functions.

Isolation level. JDBC or EJB supports several type of isolation level which allow dirty read, phantom red, etc.

J

Java archive (JAR). A platform-independent file format that groups many files into one. JAR files are used for compression, reduced download time, and security.

Java message service. A standard mechanism for the components to send and receive messages.

JavaBeans. The specification that defines the platform-neutral component model used to represent parts. Instances of JavaBeans (often called beans) may have methods, properties, and events.

JavaMail. A standard API which allows an application to send email notifications.

JavaServer Pages. A technology which enables to mix static HTML with dynamically generated content from servlets.

JDBC. The specification that defines an API that enables programs to access databases that comply with this standard.

Java Naming and Directory Interface. A standard API which allows the J2EE components to look up other remote objects that they may need to access.

Java Transaction API (JTA). A standard API which allows applications and J2EE Servers to access transactions.

Java Transaction Service (JTS). The implementation of a transaction manager which supports JTA.

L

Lightweight Directory Access Protocol (LDAP). One of the very few Internet protocols that has become associated with a well-documented, well-known, and easy-to-use API.

M

Message Driven Beans (MDB). A message listener that can reliably consume messages from a queue or subscription (EJB 2.0 specification).

Message Queueing Interface (MQI). An MQI client/server is a component of the MQSeries product. You can run an MQSeries application on an MQI client and it can interact, by means of a communications protocol, with one or more MQI servers and connect to their queue managers.

Multipurpose Internet Mail Extension (MIME). MIME is what allows users to send information over e-mail in forms other than simple text. MIME allows for electronic transmission of audio, video, applications, images, and so forth. Many MIME types, such as GIFs and PostScript files, are predefined. You can also define your own MIME types. MIME types allow Web browsers to output files which are not in an HTML format.

Message Oriented Middleware (MOM). A middleware which facilitate data connectivity across networks and Internet, providing message queuing, dynamic formatting, rules engine, and data replication.

O

Object. (1) A computer representation of something that a user can work with to perform a task. An object can appear as text or an

icon. (2) A collection of data and methods that operate on that data, which together represent a logical entity in the system. In object-oriented programming, objects are grouped into classes that share common data definitions and methods. Each object in the class is said to be an instance of the class. (3) An instance of an object class consisting of attributes, a data structure, and operational methods. It can represent a person, place, thing, event, or concept. Each instance has the same properties, attributes, and methods as other instances of the object class, although it has unique values assigned to its attributes.

Object Level Trace (OLT). An extension of the IBM Distributed Debugger that enables you to trace and debug multilingual, distributed applications from a single workstation

Object-oriented programming (OOP). A programming approach based on the concepts of data abstraction and inheritance. Unlike procedural programming techniques, object-oriented programming concentrates on those data objects that constitute the problem and how they are manipulated, not on how something is accomplished.

ODBC driver. An ODBC driver is a dynamic link library that implements ODBC function calls and interacts with a data source.

Open Database Connectivity (ODBC). A Microsoft-developed C database API that allows access to database management systems calling callable SQL, which does not require the use of an SQL preprocessor. In addition, ODBC provides an architecture that allows users to add modules (database drivers) that link the application to their choice of database management systems at run time. Applications no longer need to be directly linked to the modules of all the database management systems that are supported.

Open Servlet Engine (OSE). A lightweight, proprietary protocol for transporting data. It is used by WebSphere to forward requests from the Web server to an application server for processing.

P

Project. In VisualAge for Java, the topmost kind of program element. A project contains Java packages.

Property. An initial setting or characteristic of a bean; for example, a name, font, text, or positional characteristic.

R

Remote method invocation. The API that enables you to write distributed Java programs, allowing methods of remote Java objects to be accessed from other Java virtual machines.

Repository. In VisualAge for Java, the storage area, separate from the workspace, that contains all editions (both open and versioned) of all program elements that have ever been in the workspace, including the current editions that are in the workspace. You can add editions of program elements to the workspace from the repository.

Repository explorer. In VisualAge for Java, the window from which you can view and compare editions of program elements that are in the repository.

Repository file. A file that you can export from VisualAge for Java that contains information about selected projects or packages. This file can then be imported into any VisualAge for Java session.

RMI over IIOP. A version of RMI implemented to use the CORBA IIOP protocol.

S

Scriptlet. Scripting elements allow the insertion of code into the JSP.

Servlet. A server-side class that respond to HTTP requests.

Session bean. A relatively short-lived enterprise bran. There are two types of session bean: stateful and stateless.

SmartGuide. In IBM software products, an interface that guides you through performing common tasks.

SMTP. Defines the mechanism for delivery of mail. A JavaMail application would communicate with the SMTP server for a particular ISP.

T

Tag Library Definition (TLD). An XML file containing elements that describe each tag in the library. The TLD will be used by the JSP container during translation time to interpret each tag, and at request time to find and invoke the tag code.

Two-phase commit. Two phase commits are done to maintain data integrity and accuracy within the distributed databases through synchronized locking of all pieces of a transaction.

U

Uniform Resource Identifier (URI). URIs are defined as sequences of characters chosen from a limited subset of the repertoire of ASCII characters both for transmission in network protocols and representation in spoken and written human communication.

Uniform Resource Locator (URL). A standard identifier for a resource on the World Wide Web, used by Web browsers to initiate a connection. The URL includes the communications protocol to use, the name of the server, and path information identifying the objects to be retrieved on the server.

URL rewriting. A technique that allows to append data at the end of any link generated by a servlet. The session identifier can be assigned to the URL so that when a user makes another request to the application the server is able to get back its session object.

User-to-business. A general case of users (internal to the enterprise or external) interacting with enterprise transactions and data.

W

Web ARchive (WAR). A JAR file which contains a web module.

Web container. An entity that implements the web component contract of the J2EE architecture.

Web services. A programmable application logic that is accessible using standard Internet protocols. Web Services provide well-defined interfaces, or contracts, that describe the services provided.

Workbench. In VisualAge for Java, the main window from which you can manage the work-space, create and modify code, and open browsers and other tools.

Workspace. The work area that contains all the code you are currently working on (that is, current editions). The workspace also contains the standard Java class libraries and other class libraries.

X

Xtensive Markup Language (XML). A markup language that allows you to define the tags need to identify the data and text in the documents.

Abbreviations and acronyms

AAT	Application Assembly Tool		**MQI**	Message Queueing Interface
API	Application Programming Interface		**MIME**	Multipurpose Internet Mail Extension
B2B	Business to Business		**MOM**	Message Oriented Middleware
BMP	Bean Managed Persistence			
DBMS	Data Base Management System		**MVC**	Model View Controller
DDL	Data Definition Language		**OLT**	Object Level Trace
DTD	Document Type Definition		**OOP**	Object Oriented Programming
EAR	Enterprise ARchive			
EJB	Enterprise Java Beans		**ODBC**	Open DataBase Connectivity
GUI	Graphical User Interface		**OSE**	Open Servlet Engine
HTML	Hypertext Markup Language		**RMI**	Remote Method Invocation
HTTP	HyperText Transfer Protocol		**TLD**	Tag Library Definition
IBM	International Business Machines Corporation		**URI**	Uniform Resource Identifier
			URL	Uniform Resource Locator
IDE	Integrated Development Environment		**WAR**	Web ARchive
IDL	Interface Definition Language		**WAS**	WebSphere Application Server
IMAP	Internet Message Access Protocol		**WTE**	WebSphere Test Environment
IP	Internet Protocol		**XML**	Xtensive Markup Language
ITSO	International Technical Support Organization			
JAR	Java ARchive			
JDBC	Java DataBase Connectivity			
JMS	Java Message Service			
JSP	JavaServer Pages			
JTA	Java Transaction API			
JTS	Java Transaction Service			
LDAP	Lightweight Directory Access Protocol			
MDB	Message Driven Beans			

Index

Numerics

3-tier programming   39

A

AAT   113
AccessBean wrapper   101
Actions   223
Adding a Binding   302
AIX   25
Ancestor   278
Applet   10, 171
Application (object variable)   224
Application (scope)   225
Application client   18, 300
Associations   83
AT_BEGIN   275
AT_END   275
Attachments (JavaMail)   311
Attribute   260
Authentication   94
Authorization   95
AutoFlush (attribute)   227

B

B2B   39
Basic HTTP authentication   95
Bean   171
Bean managed security   98
Bean managed transactions   120
BMP   11, 42
BodyContent   260, 268, 281
BodyTag   268
BodyTagSupport   268
Buffer (attribute)   227
ByteMessage   135

C

Class
    com.ibm
        ejs.jms.mq
            JMSWrapXA...   148
        mq.jms

MQQueue   148
MQQueueConnectionFactory   148
MQTopic   148
MQTopicConnectionFactory   148
vap
    finders
        VapEJBJDBCFinderObject   71
Websphere
    servlet
        error
            ServletErrorClass   213
        session
            IBMSession   212
Java
    beans
        Bean.instantiate   241
    lang
        ClassCastException   241
        InstantiationException   241
    rmi
        RemoteException   45, 46
    security
        Identity (deprecated)   113
        Principal   98
    sql
        Connection   117, 118, 120
javax
    ejb

        BeanFinderHelper   70
        CreateException   45, 75
        DuplicateKeyException   62, 75
        EJBContext   73, 119
        EJBException   75
        EJBHome   45
        EJBObect   46
        EJBObject   64
        EntityBean   11, 59, 64
        EntityContext   61, 68
        FinderException   75
        HomeHandle   113
        ObjectNotFoundException   73, 75
        RemoveException   75
        SessionSynchronization   54, 118
    jms

        JMSException   147
        MessageDrivenBean   76
        MessageListener   142
        messageListener   75
        QueueReceiver   142

343

Programming J2EE APIs with WebSphere Advanced

IBM

Redbooks

Programming J2EE APIs with WebSphere Advanced

Feel the power of the J2EE-compliant environment

Learn J2EE APIs by practical application development

Deploy J2EE applications to WebSphere

This IBM Redbook has examples of programming the new J2EE APIs using VisualAge for Java and deployment on WebSphere Advanced.

Part 1 introduces J2EE and the PiggyBank application scenario, which is an integrated application used to illustrate various principles and techniques for enterprise software development.

In Part 2, learn the depth of EJB container of the J2EE specification, which includes transactional EJBs, transactions, messaging with JMS, WebSphere and MQSeries.

In Part 3, learn the latest servlet and JSP specification with Web application concepts.