EJB 3 Developer Guide

A practical guide for developers and architects to the Enterprise Java Beans Standard

Michael Sikora

EJB 3 Developer Guide

A Practical Guide for developers and architects to the Enterprise Java Beans Standard

Michael Sikora

EJB 3 Developer Guide

Copyright © 2008 Packt Publishing

Credits

Author

Michael Sikora

Reviewers

Meenakshi Verma

Chris Mawata

Lawrence Bean

Senior Acquisition Editor

Douglas Paterson

Development Editor

Sarah Cullington

Technical Editor

Bhupali Khule

Editorial Team Leader

Mithil Kulkarni

Project Manager

Abhijeet Deobhakta

Project Coordinator

Brinell Catherine Lewis

Indexer

Monica Ajmera

Proofreader

Angie Butcher

Production Coordinator

Shantanu Zagade

Cover Work

Shantanu Zagade

About the Author

Michael Sikora is an enterprise architect with a background in the Unified
Process and Java EE. He has a particular interest in object oriented and database
technology. He has worked for many large UK clients such as ICL Fujitsu, Mercury
Communications, Vodafone, and BUPA. He has used Java since 2000. Before
that he spent a decade designing and developing database and data warehouse
systems. He has experience in Oracle, PL/SQL, and C. In the 1980s he worked for
Shell developing exploration software. He graduated in Mathematics from Essex
University and has Masters degrees from London University and Imperial College.

Michael currently resides in London, England and enjoys mountaineering
and still hopes to complete the seven summits. His website is
`http://www.ejbconsultants.com.`

About the Reviewers

Meenakshi Verma has been part of the IT industry since 1998. She is experienced in putting up solutions across multiple industry segments using Java/J2EE technologies. She is currently based in Toronto, Canada and is working with Sapient, a leading North American Consultancy organization. Sapient is a global organization which has offices across North America, Europe and India.

Meenakshi has been helping with technical reviews for books published by Packt Publishing across varied enterprise solutions. Her earlier work includes *JasperReports for Java Developers*, *Java EE 5 Development using GlassFish Application Server*, *Practical Data Analysis and Reporting with BIRT*.

> I'd like to thank my husband (Atul Verma) for his encouragement and support throughout the review of this book and many others, and my three year old son (Prieyaansh Verma) for giving me the warmth of his love despite my hectic schedules. I also owe thanks & gratitude to my Father (Mr Bhopal Singh) and Mother (Mrs Raj Bala) for laying a strong foundation in me and giving me their unconditional love and support.

Chris Mawata earned his Masters degree at the University of Western Australia and his Ph.D at the University of Hawaii. He was a professor at the University of Tennessee at Chattanooga for fifteen years after which he went into Java consulting and technical training. He currently trains Java developers and writes course materials for Learning Tree International. He is also sponsored by a National Science Foundation grant to be a Java consultant for Phenotype Screening Corporation.

Lawrence Bean fell out of Computer Science and into Music Education in his sophomore year of college. He graduated with a BA in Music Education from the University of Maine in 1986 and had a ten year career as a Choral Music Educator in the Kennebunk, Maine school system, while continuing his technology pursuits as a hobby. His large non-audition groups won silver at the Events America Choral Festival and his select group was featured on Good Morning America and in Yankee Magazine for their annual performances of traditional christmas carols at the highly acclaimed Kennebunkport Christmas Prelude.

He fell back into Computer Science with the offer of a position as Technology Coordinator at SU#47 in greater Bath, Maine and completed his MS in Computer Science in 2006. For the past ten years he has literally built the school system's technology program, from pulling CAT-5 through the walls and constructing PCs out of spare parts to writing data analysis modules and a custom SIF-compliant Student Information System, while continuing his musical pursuits as a hobby.

Throughout he has been highly involved in the Maine Music Educators Association, American Choral Directors Association of Maine, Association of Computer Technology Educators in Maine, and the Barbershop Harmony Society, holding various positions of responsibility and bringing the Maine All-State Auditions into the 21st century with on-line applications, judging, and results processing.

He has been a strong advocate for free, open-source solutions for public schools and currently presents workshops on IPCop Firewalls and the Koha ILS.

Larry lives with his very patient wife Betsy in Saco, Maine, USA.

Table of Contents

Preface

Enterprise JavaBeans (EJB) technology is a core part of the Java EE 5 specification. EJB is a framework for building enterprise-scale object-oriented, distributed, component-based business applications. EJB business applications are written in Java, are scalable and can be deployed on any platform that supports the EJB specification.

EJB applications are deployed to and execute under the control of an EJB container. The EJB container provides services typically required by enterprise applications such as security, transaction management, resource pooling, and systems management.

The EJB 3 specification, released in May 2006, is a radical change from previous versions of the technology. Developing business applications is considerably easier with EJB 3. The handling of persistence in particular has radically changed in EJB 3. Persistence is no longer a service provided by an EJB container but rather by a persistence provider conforming to the Java Persistence API (JPA) specification. Java applications which need to be persisted but which do not require the services provided by an EJB container can be persisted outside an EJB container by a JPA persistence provider. In this book we cover JPA as well as the core EJB 3 services.

This book is a concise, example-driven introduction to EJB 3. The best way to learn a new software technology is by studying and trying out programming examples. In this book you will see a lot of code and one example after another. We do not assume any prior knowledge of EJB. However this book does assume at least a couple of years' experience of Java and some knowledge of relational database technology. The examples in this book have been deployed on the GlassFish application server. GlassFish was chosen primarily because this is the Java EE 5 reference implementation.

What This Book Covers

Chapter 1 Introduction to EJB 3—A general overview of the Java EE architecture including EJB container services, the JPA persistence engine, and initial installation of the GlassFish application server.

Chapter 2 Session Beans—Creation of a session bean and its client and examples of running it from the application client container. Exploring the program directory structure. Packaging and deploying a session bean. A look at the stateless and stateful session beans lifecycle.

Chapter 3 Entities—Exploring EJB 3 entities. How to package and deploy entities and map an entity to a database table. Metadata defaults and how to override them. Generating primary keys.

Chapter 4 Object/Relational Mapping—One-to-one, one-to-many and many-to-many associations. Default object/relational mapping values and how to override them. Object/relational inheritance mapping and additional annotations.

Chapter 5 JPQL (Java Persistence Query Language)—Looking at different groups of queries including aggregate functions, queries with relationships, subqueries, queries with parameters and named queries. JPQL joins and functions are also explained.

Chapter 6 Entity Manager—Looks in detail at the entity manager. Covers both the EJB 3 container-managed and application-managed entity managers.

Chapter 7 Transactions—ACID properties and isolation levels. Container-managed transactions. Bean-managed transactions.

Chapter 8 Messaging—Looks at the JMS (Java Messaging Service) API. Examples of queue producer and queue consumers. Topic producer and consumers. Message driven beans: their activation configuration properties, lifecycles and transactions.

Chapter 9 EJB Timer Service—Examples of single and interval events. Timer interface methods. Timers and transactions.

Chapter 10 Interceptors—Covers interceptor methods, classes and interceptor communication as well as default interceptors.

Chapter 11 Implementing EJB 3 Web Services—An overview of web service concepts and the web service definition language (WSDL). Creating and deploying a Java application as a web service. Creating and deploying an EJB session bean as a web service. Creating a web service client.

Chapter 12 EJB 3 Security—A look at security, GlassFish authentication, declarative and programmatic EJB authorization and Web Tier authentication and authorization.

The *Appendix* shows EJB 3 annotations described in this book with their corresponding packages.

What You Need for This Book

First you must have version 5 or higher of the Java Development Kit (JDK). This can be downloaded from `http://java.sun.com/javase/downloads/index.jsp`.

You also need a version of **Ant** equal to or higher than 1.6.5. Ant is a tool for building Java programs, similar in concept to the **Unix Make** tool, except that it is platform independent. Ant is part of Apache's Jakarta project. If you are unfamiliar with Ant you should look at the Ant web site (`http://ant.apache.org/`). If you do not have Ant there is no need to download it as the GlassFish application server includes a copy of Ant.

Finally you need a version of GlassFish equal to or higher than V2b41d. This can be downloaded from `https://glassfish.dev.java.net//`.

Instructions on setting up environment variables and installing GlassFish are given in the "Getting Started" section in Chapter 1.

Who Is This Book For

Previous experience of working with Java is essential and knowledge of relational databases is desirable.

This book is primarily aimed at professional developers who already have a working knowledge of Java. Enterprise architects and designers with a background in Java would also find this book of use.

As this book is an introduction to EJB 3, it is aimed at those who are new to EJB 3. However, as the new version of EJB is so radically different from the previous version (EJB 2.x), the book would also be suitable and of interest to those who have had experience working with EJB 2.x. The text makes it clear where the differences between the versions of EJB lie, although it is not explored in detail.

Approach of this Book

This book has been written for developers new to EJB 3 who want to use the technology. Such readers usually want to see examples and program code. In this book we work through one example after another and we show lots of program code. If you are new to a technology and have looked at your first `HelloWorld` example, the next thing you want to do is to code and run the program yourself. In the case of EJBs this also means packaging and deploying the EJB to a container. Although we promise no `HelloWorld` examples in this book, we look at packaging and deployment straight after coding our first EJB, rather than ignoring these topics or leaving them to the end of the book.

All the source code together with packaging and deployment scripts is available from the book's web site.

This book is not a reference book, so we don't attempt to cover all EJB 3 features. We've kept this book concise to help you quickly get up and running with EJB 3.

Although an EJB will run in any EJB container, the deployment process is container-dependent. This means that we need to pick a specific container for our examples. Sun's open-source, free GlassFish container was chosen primarily because this was the Java EE reference container implementation and also because GlassFish has Toplink embedded. Toplink in turn is the reference JPA persistence engine.

We also chose not to use an IDE (Integrated Development Environment) partly because they are all quite different. A book based on IDE A would be of little use to a reader using IDE B. As IDEs are screen-based navigational tools, any resulting book would contain a large number of screenshots and would be at least double in length. More importantly is the author's view that an IDE is not an ideal learning tool. Apart from having to learn to navigate through a large number of screens, often an IDE will hide the underlying EJB technology. Of course once you have learnt EJB an IDE will make you much more productive in your work.

Instead we use the **Ant** build tool for compiling, packaging, deploying, and running our EJBs.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We name our interface `TimeService.java`, which contains a single method definition `getTime()`."

A block of code will be set as follows:

```
@Stateless
public class TimeServiceBean implements TimeService {
    public String getTime() {
        Formatter fmt = new Formatter();
        Calendar cal = Calendar.getInstance();
        fmt.format("%tr", cal);
        return fmt.toString();
    }
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items will be made bold:

```
@Stateless
public class TimeServiceBean implements TimeService {
    private @EJB NiceDayService niceDay;
    public String getTime() {
        Formatter fmt = new Formatter();
        Calendar cal = Calendar.getInstance();
        fmt.format("%tr", cal);
        return fmt.toString() +
                        niceDay.getMessage();
    }
}
```

Any command-line input and output is written as follows:

```
C:\EJB3Chapter02\glassfish\lab1\build>jar  -tvf  TimeService.jar
```

New terms and **important words** are introduced in a bold-type font. Words that you see on the screen, in menus or dialog boxes for example, appear in our text like this: " Under the **Web Services** heading in the **Common Tasks** pane on the left of the screen, click on **Arithmetic**."

Important notes appear in a box like this.

Tips and tricks appear like this.

Reader Feedback

Feedback from our readers is always welcome. Let us know what you think about this book, what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply drop an email to `feedback@packtpub.com`, making sure to mention the book title in the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or email `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer Support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the Example Code for the Book

Visit `http://www.packtpub.com/files/code/5609_Code.zip` to directly download the example code.

The downloadable files contain instructions on how to use them.

Errata

Although we have taken every care to ensure the accuracy of our contents, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in text or code—we would be grateful if you would report this to us. By doing this you can save other readers from frustration, and help to improve subsequent versions of this book. If you find any errata, report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **let us know** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata added to the list of existing errata. The existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with some aspect of the book, and we will do our best to address it.

1
Introduction to the EJB 3 Architecture

In this chapter we will give an overview of EJB 3 and how it fits into the Java EE multi-layer framework. We will also look at installing and getting started with the GlassFish container. The topics covered are:

- An overview of Java EE
- The EJB 3 Architecture
- Getting Started with GlassFish

Introduction to the Java EE Architecture

Modern enterprise applications have their responsibilities divided over a number of layers. A common architecture is the 3-layer model consisting of presentation, business, and database layers. The presentation layer is responsible for presenting a user interface and handling interactions with the end user. The business layer is responsible for executing business logic. The database layer is responsible for storage of business data; typically a relational database management system is used for this layer. Layering is used throughout computer science for managing complexity where each layer serves a distinct purpose.

Java Platform Enterprise Edition (Java EE) technology provides services to enterprise applications using a multi-layer architecture. Java EE applications are web-enabled and Java–based, which means they may be written once and deployed on any container supporting the Java EE standard. An application server is the environment in which the container resides. However, in practice we don't need to distinguish between an application server and a container, so we will use the terms interchangeably. The Java EE specification is supported by commercial vendors such as Sun, IBM, Oracle, BEA Systems as well as open-source ventures such as JBoss.

Java EE presentation layer technologies include servlets, JSP pages, and JSF components. These are developed for a business application then subsequently deployed and run in a web container. A client would interact with the web container either from a browser or an applet. In either case the http or https internet protocol would be used for communication.

Enterprise JavaBeans version 3 (EJB 3) is the technology Java EE version 5 (Java EE 5) provides for the business layer. In Java EE 5 we subdivide the business layer into one layer which is concerned with business processing and a second layer which deals with persistence. In EJB 3 the business processing artifacts are **session** and **message-driven beans**. These are developed for a business application and deployed and run in an EJB container. The persistence layer artifact is an **entity**; this is persisted to the database layer using a **persistence provider** or **persistence engine**. The persistence engine implements another specification, the **Java Persistence API (JPA)**. Both EJB 3 and the JPA are specifications for which a number of organizations provide implementations. Both specifications can be downloaded from `http://www.jcp.org/en/jsr/detail?id=220`. The figure below summarizes Java EE 5 architecture:
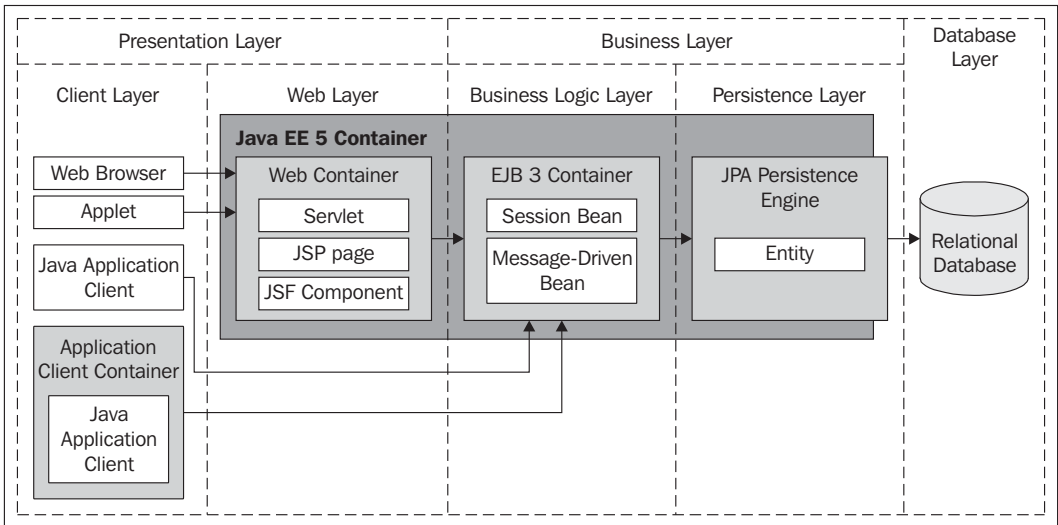
Note that our 3-layer model has become 5-layers. The distinction between client/web and business logic/persistence layers is not always made. Consequently we refer to Java EE architecture simply as n-layer or multi-layer. A Java EE container offers many other services such as web services, the Java Messaging Service (JMS), and resource adapters.

Note from the diagram that we can access an EJB directly from a Java SE application, such as Swing, without going through a web container. The Java application can be stand-alone, or can be run from an Application Client Container (ACC). An ACC enables a client executing in its own Java Virtual Machine (JVM) outside the EJB container to access a limited number of Java EE services.

The EJB 3 Architecture

The EJB 3 architecture offers a standard for developing distributed, object-oriented, component-based business applications. The components developed in an EJB framework are session and message-driven beans. Collectively these are known as EJBs. These are usually relatively coarse-grained objects encapsulating a business process. They are components in the sense that EJBs can be combined to create a business application. Furthermore if the EJBs have been well designed they can be reused by another application. EJBs are distributed in the sense that they can reside on different computer servers and can be invoked by a remote client from a different system on the network.

A session bean must have a business interface, which can be either remote or local. A remote client invokes the remote interface of a session bean as shown in the following diagram:
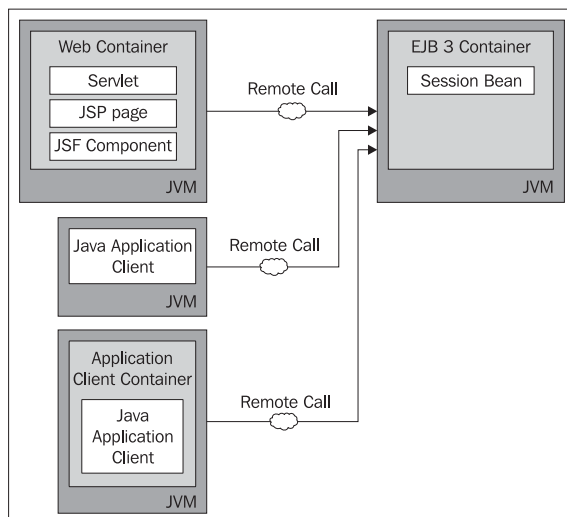
However a session bean and its client may reside in the same JVM instance. In such cases the client invokes the local interface of the session bean. The following diagram shows a web container client invoking the session beans local interface:
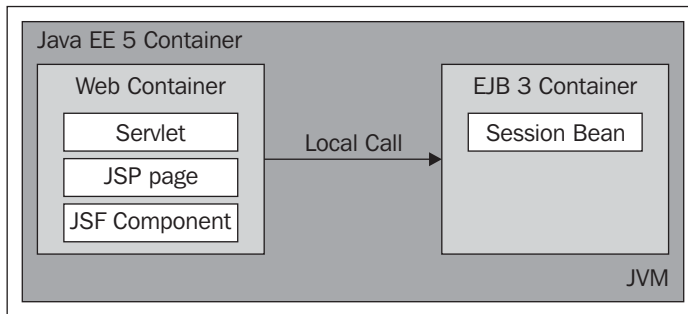
A message-driven bean is an asynchronous recipient of a JMS message. The client, which can be a Java application or Java EE component such as a session bean, sends a JMS message to a message queue or topic. The message queue or topic may be managed by a Java EE container or alternatively by a dedicated JMS sever. The following diagram shows a client sending a JMS message which is received by a message-driven bean:
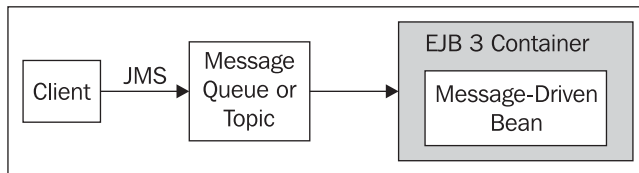
EJBs are deployed and run in a container which is designed to make applications scalable, multi-user, and thread-safe. An EJB container also provides a number of services that an enterprise scale business application is likely to need. We will list these services in the next section.

In contrast to session and message-driven beans, entities are relatively fine-grained objects which have a relatively long life and need to be persisted. Prior to EJB 3, entity beans played the role of entities and were defined as remotely accessible components, like session and message-driven beans. In EJB 3 entities are Java objects and so can utilize object-oriented features such as inheritance and polymorphism, which entity beans could not. In EJB 3, entities are persisted by a persistence provider or persistence engine implementing the JPA specification. This persistence engine can run within an EJB container or outside a container where a business application does not require other EJB services.

Strictly speaking EJBs, being remotely accessible components, include only session and message-driven beans and not entities. However, whenever we refer to EJBs we will in general include entities, unless the specific context requires us to make a distinction. When we refer to EJB *components*, we mean session and message-driven beans and not entities.

EJBs being Java-based may be written once and deployed on any application server supporting the EJB standard.

EJB Container Services

An EJB container provides a large number of services and we will list a few of these here. Much of this book describes some of these services in detail, in particular those which a business application is likely to invoke.

EJB containers support concurrency and all EJB components are thread-safe. EJB containers provide pooling for EJB component instances. Pooling, in particular, contributes to the scalability of the EJB architecture. We will discuss pooling for session beans in Chapter 2 and for message-driven beans in Chapter 8. Load balancing and clustering are EJB container services which also contribute to the scalability of EJB.

EJB containers provide a naming service, the Java Naming and Directory Interface (JNDI), for accessing EJBs or any other container-managed resource such as JMS queue connections. In EJB 3 a simpler annotation-based dependency injection facility is available which in many cases provides an alternative to JNDI. All EJB 3 containers support Java RMI-IIOP (Remote Method Invocation run over Internet Inter-Orb Protocol), which enables a session to be remotely accessed by a client. A client does not need to know whether the invoked EJB is remote or local, residing in the same JVM. This feature is known as location transparency.

Business systems are often transactional and EJB provides a container-managed transaction service. This is described in Chapter 7.

EJB supports messaging by providing JMS-based message-driven beans. We will discuss message-driven beans in Chapter 8.

EJB provides a basic scheduling capability: the **Timer** service, which is described in Chapter 9.

A new feature of EJB 3 is the **Interceptor** service. This allows common, tangential aspects of EJB components to be separated from any business logic. This concept is based on **AOP** (Aspect Oriented Programming) and is described in Chapter 10.

EJB allows you to convert a stateless session bean into a **web service**; this is covered in Chapter 11.

EJB provides standards for both the authentication and authorization aspects of security. **Authentication** is concerned with validating the identity of a user. **Authorization** is concerned with controlling a user's access to an application, or part of an application. We have covered security in Chapter 12.

Last, but certainly not the least, most business applications need a service for persisting entities. In EJB 3 this service is delegated by the container to a Java Persistence API (JPA) persistence engine.

The JPA Persistence Engine

Many applications do not require the services provided by an EJB container but still need persistence services. For this reason JPA has been issued as a separate specification and applications running outside an EJB container can also make use of JPA services. The main services include:

- Entity Manager
- Object/Relational Mapping
- The Java Persistence Query Language (JPQL)

The Entity Manager provides services for persistence, transaction management, and managing the lifecycle of entities. Object/Relational metadata annotations are provided for mapping entities onto relational database tables. **JPQL** is used for retrieving persisted entities. We will look at these in more detail in the forthcoming chapters.

Although the JPA specification is recent, it leverages object/relational mapping technology associated with products such as Hibernate and Oracle Toplink. These products have been available for many years; in the case of Toplink for over a decade. The JPA specification drew heavily on these two products in particular. Furthermore, Toplink and Hibernate are the actual default persistence engines for a number of EJB 3 containers. For example, both Sun's GlassFish container and Oracle Application Server 11g use Toplink as the embedded persistence engine. The JBoss EJB 3 container uses Hibernate as the embedded persistence engine. These are pluggable defaults however, so it is possible to use Hibernate with GlassFish for example.

EJB 3 Compared with Earlier Versions

The main features introduced in EJB 3 can be summarized as:

- Simplified Persistence API
- Metadata Annotations
- Improved Query Language
- Use of Defaulting
- Dependency Injection
- Simplification of Session Beans

The first two features are probably the most important, but we will expand on each of the above features in this section.

The main difference between EJB 3 and EJB 2.x is the handling of persistence which we have already outlined. Prior to EJB 3 there was rather limited object/relational mapping between entity beans and relational tables. Inheritance and polymorphism were not possible prior to EJB 3. An EJB 3 entity is truly a Java object; this could not be said of an entity bean.

The other main EJB 3 innovation is the introduction of **metadata annotations**. Metadata annotations were first introduced in Java SE 5, so this version of Java or higher must be used when developing EJB 3 applications. Metadata annotations can be used as an alternative to XML deployment descriptors both for configuring EJB components and specifying object/relational mappings with entities. However, deployment descriptors can be used in both cases. We will look at annotation versus deployment descriptor aspects in Chapter 2.

The EJB Query language (EJB QL) available in earlier versions was rather limited in comparison with JPA's JPQL. In particular JPQL provides the following enhancements:

- Projections
- GROUP BY and HAVING clauses
- Joins
- Subqueries
- Dynamic Queries
- Queries with parameters
- Bulk update and delete operations

Extensive use of defaults is made in EJB 3. So, for example, most metadata annotations do not require elements or parameters to be specified, the default is usually common, expected behavior. Annotation elements are usually needed only when we want to configure exceptional behavior.

Dependency injection, first featured in the **Spring** framework, has been introduced in EJB 3 as an alternative to JNDI for looking up container-managed resources.

Session beans have been simplified. We no longer need to specify component and home interfaces. Furthermore the session bean class no longer has to implement a number of callback interfaces even when these are not required by the application. In EJB 3 these lifecycle callback methods are implemented by session beans only when required.

Getting Started

For this book GlassFish was downloaded on Windows XP and all the examples were run on that platform. The instructions that follow assume a Windows platform is being used, however the installation instructions for other platforms such as Linux/Unix are almost the same.

Installing GlassFish

Before installing GlassFish check that version 5 or higher of the Java Development Kit (JDK) is present on your workstation. This can be downloaded from

```
http://java.sun.com/javase/downloads/index.jsp
```

For this book we used JDK 5.0 Update 12.

Set the environment variable `JAVA_HOME` to point to the directory in which the JDK is installed. Add **%JAVA_HOME%\bin**, to the **PATH** environment variable. This is done by clicking **Control Panel** from the **Start** menu. Then double-click the system icon. Select the **Advanced** tab on the **System Properties** box. Finally click the **Environment Variables** button.

Next download a version of GlassFish equal to or higher than V2b41d from:

```
https://glassfish.dev.java.net//
```

into the directory in which you want to install it. All the code examples in this book were tested with GlassFish version V2b41d. The actual name of the downloaded jar file will depend on the version and platform selected. For this book `glassfish-installer-v2-b41d.jar` was used. In the same directory as the downloaded jar file, run the jar file as follows:

```
java -Xmx256m -jar glassfish-installer-v2-b41d.jar
```

This will unzip the file and create the `glassfish` subdirectory.

Set the `GLASSFISH_HOME` environment variable to the directory where GlassFish was installed. Add `%GLASSFISH_HOME%\bin` to the `PATH` environment variable.

Set the environment variable `ANT_HOME` to the directory in which Ant is installed. GlassFish comes bundled with the Ant build tool and the installed Ant directory will be `glassfish\lib\ant`.

If you already have a version of Ant equal to higher than 1.6.5 then set `ANT_HOME` accordingly.

Add `%ANT_HOME%\bin` to the PATH environment variable.

Change the directory to the `glassfish` directory and complete the installation by running the Ant setup script:

```
ant -f setup.xml
```

The setup process may fail because of port conflicts with existing software you may have. If so you will need to edit the `setup.xml` file. Within the file you should see the following fragment setting up default properties:

```
        .....
   <property name="domain.name" value="domain1"/>
   <property name="instance.name" value="server"/>
   <property name="admin.user" value="admin"/>
   <property name="admin.password" value="adminadmin"/>
   <property name="admin.port" value="4848"/>
   <property name="instance.port" value="8080"/>
   <property name="orb.port" value="3700"/>
   <property name="imq.port" value="7676"/>
   <property name="https.port" value="8181"/>
   <property name="glassfish.license" value="LICENSE.txt"/>
      ....
```
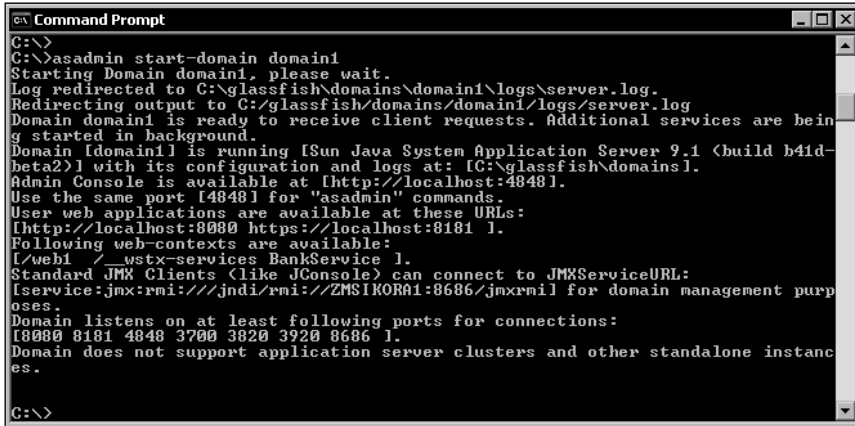
Change the value for the conflicting port, and run `setup` again.

Testing the Installation

GlassFish is started with the following command:

```
asadmin start-domain domain1
```

You should see the following messages:

In particular this shows the admin console URL and the URL for web applications. So enter the URL http://localhost:8080 from a browser. If GlassFish has started up correctly you should get the following page:
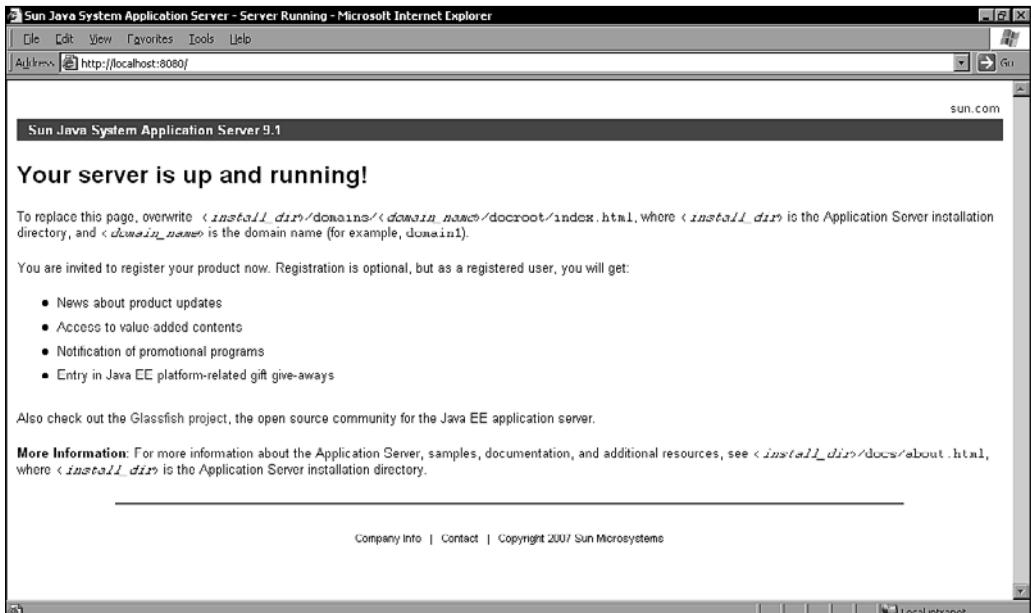
Accessing the Administrator Console

Enter the URL `http://localhost:4848` from a browser. You should get the following page:

The default user name is `admin` and the default password is `adminadmin`. After you have entered these you should get the following page:
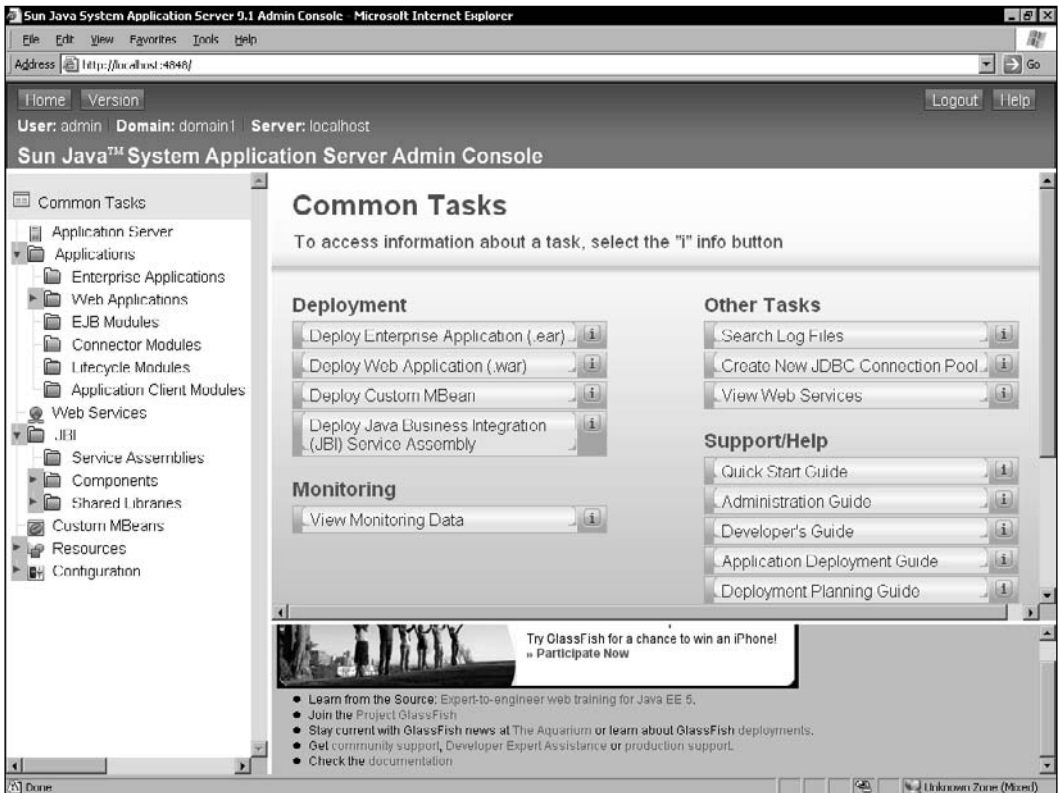
The administrator console is used for numerous tasks such as creating and configuring domains, monitoring performance, and deploying applications. An alternative for deploying applications is using the **asadmin** tool from the command line.

In the remainder of this book we have used the **asadmin** tool for deployment. By including it as an Ant target the whole process of compiling, packaging, and deploying is automated.

Shutting Down GlassFish

To stop GlassFish use the following command:

```
asadmin stop-domain domain1
```

Downloading Example Source Code

The source code for the examples can be found as a zip file on the book's web site. Download the file and unzip it into a drive and directory of your choice. The examples in the book assume the source code has been downloaded into the `C:\` directory. Subdirectories correspond to individual chapters: `C:\EJB3Chapter02`, `C:\EJB3Chapter03`, and so on. You should consult the `readme` file before running any scripts.

Summary

We gave an overview of the EJB 3 architecture and its relationship to the multi-layer Java EE architecture. EJB technology applies to the business processing and persistence layers of the multi-layer model. There are three main artifacts a business application using EJB technology will develop: session-beans, message-driven beans, and entities.

These artifacts are deployed to an EJB container which provides a number of services. We had a brief look at some of these services. In particular we saw that a separate service, the Java Persistence API, is provided for persisting entities.

Finally we saw how to get started with Sun's open-source GlassFish EJB 3 container. All the examples which follow in this book have been deployed and run on GlassFish.

In the following chapter we will look at developing the EJB component which handles business processing, namely session beans.

2
Session Beans

Session Beans are an EJB technology for encapsulating business processes or workflow. In this chapter we will cover the following topics:

- Stateless session beans
- Stateful session beans
- Annotations
- Packaging and Deploying a session bean
- Running a session bean client from the application client container
- Stateless and Stateful session bean lifecycles

Introduction

In object-oriented analysis and design a control class encapsulates business logic for a use case. Session beans are used to implement such control classes. Check credit card details, transfer funds, and book reservation are examples of potential session bean candidates. Session beans are transient and relatively short lived. In particular, session beans are not persistent; they are not stored in a database or other permanent file system. Session beans can create and update entities, which are persistent, as we shall see in the following chapters.

A client interacts with a session bean by invoking one or more methods defined in the bean. This sequence of method calls we call a session, hence the name session beans. The client can be a web-tier client such as a servlet or JSP page, or a standalone Java application program.

Like EJB 2.x, EJB 3 session beans are a component technology. In EJB 3 a session bean component consists of a bean interface and a bean class. The bean interface provides all the bean method definitions, the bean class provides the method implementations. We no longer have to produce a home interface as was required in EJB 2.x.

It is important to stress that the client does not interact directly with a bean. Instead the client interacts with a proxy generated by the EJB container. This proxy intercepts method calls and provides the container services such as transaction management and security.

Session beans come in two flavors: **stateless** and **stateful**. We will start with an example of a stateless session bean.

Stateless Session Beans

A stateless session bean's state spans a single method call. We cannot have method A updating an instance variable x say, and expect the updated value of x to be available for any subsequent method invocation. This holds true both for a new method B which accesses x, or indeed, if method A is invoked a second or subsequent time. We have no guarantee that the same stateless session bean instance will be invoked between method calls. For this reason, stateless session beans should not use instance variables to maintain conversational state.

As a simple example we will develop a stateless session bean which simply returns the current time to the client. This is almost as minimal as the traditional HelloWorld example but is more useful. Recall a session bean component consists of a bean interface and a bean class. We name our interface `TimeService.java`, which contains a single method definition `getTime()`:

```
package ejb30.session;
import javax.ejb.Remote;

@Remote
public interface TimeService {
    public String getTime();
}
```

Note that we have prefixed the interface definition with the `@Remote` annotation. This indicates to the EJB container that this bean may be invoked by a remote client. By remote, we mean a client that does not reside in the same Java Virtual Machine (JVM) as the container. An interface can also be local as we shall see later in this chapter.

The `@Remote` annotation belongs to the `javax.ejb` package, as we can see from the `import` statement. Annotations belong to a number of packages. We list annotations and their corresponding packages in Appendix A.

Note that we have decided to place our code in the `ejb30.session` package. The interface is implemented by the EJB container, and not the application, when the bean is deployed to the container.

Next we code the bean class itself, `TimeServiceBean.java`:

```
package ejb30.session;
import java.util.*;
import javax.ejb.Stateless;

@Stateless
public class TimeServiceBean implements TimeService {
    public String getTime() {
        Formatter fmt = new Formatter();
        Calendar cal = Calendar.getInstance();
        fmt.format("%tr", cal);
        return fmt.toString();
    }
}
```

This class contains the implementation of the `getTime()` business method. This looks like any Java class, all we have added is the `@Stateless` annotation. This indicates to the EJB container that the session bean is stateless.

Annotations

The option of annotations is one of the most important features which distinguishes EJB 3 from earlier versions. Metadata annotations were introduced in Java SE 5. For this reason Java SE 5 or higher must be used in conjunction with EJB 3.

In EJB 2.x, instead of annotations we use XML **deployment descriptors**. As the name suggests, deployment descriptors are read by the application server when the EJB is deployed. We will discuss the deployment process later in this chapter. The deployment descriptor, `ejb-jar.xml`, for the above session bean is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         metadata-complete="true" version="3.0"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                             http://java.sun.com/xml/ns/javaee/
                             ejb-jar_3_0.xsd">
<enterprise-beans>
<session>
<display-name>TimeServiceBean</display-name>
<ejb-name>TimeServiceBean</ejb-name>
```

```
<business-remote>ejb30.session.TimeService</business-remote>
<ejb-class>ejb30.session.TimeServiceBean</ejb-class>
<session-type>Stateless</session-type>
<transaction-type>Container</transaction-type>
<security-identity>
<use-caller-identity/>
</security-identity>
</session>
</enterprise-beans>
</ejb-jar>
```

Note the lines:

```
<business-remote>ejb30.session.TimeService</business-remote>
```

and

```
<session-type>Stateless</session-type>
```

indicate that we have a remote stateless session bean. All other lines correspond to default values for an annotated session bean.

Annotations are far simpler to use than deployment descriptors. Deployment descriptors tend to be verbose, not very readable by humans and error prone when manually edited. Any non-trivial deployment descriptor file really needs a tool both for writing and reading. However annotations have been criticized on the grounds that deployment aspects are tangential to the definition of a bean and should not intrude in the bean's code. The use of annotations means recompiling a bean when we only want to change a deployment aspect; modifying a deployment descriptor does not warrant recompilation. However annotations are optional in EJB 3. We can still use XML deployment descriptors if we wish. Furthermore we can mix both annotations and deployment descriptors in EJB 3. In such cases deployment descriptors always override annotations.

One approach might be to use annotations for those aspects of an EJB which rarely change between deployments. For example, the relationship of an entity which we will discuss in Chapter 4, is likely to be invariant. The same can be said for transaction attributes of a session bean, which we will discuss in Chapter 7. On the other hand database table names onto which entities are mapped (covered in Chapter 3) or security aspects (covered in Chapter 12), for example, are more likely to change between deployments. Consequently deployment descriptors are more appropriate here.

On large projects the teams or individuals responsible for deployment may not be the same as those responsible for development. It is unrealistic to expect the former to add or modify any annotations within code developed by others. In these circumstances deployment descriptors would be used.

However, because this is an introductory book and we want to keep its length reasonably short we will only make use of annotations in nearly all our examples.

Creating a Session Bean Client

Next we will create a client which is a standalone Java application. The client simply references a session bean, invokes the bean's `getTime()` method, then prints out the result:

```
package ejb30.client;
import javax.naming.*;
import ejb30.session.*;

public class Client {
    public static void main(String args[]) throws Exception {
        InitialContext ctx = new InitialContext();
        TimeService timeService =
        (TimeService) ctx.lookup("ejb30.session.TimeService");
        String time = timeService.getTime();
        System.out.println("Time is: " + time);
    }
}
```

Note that because the client is remote we need to use JNDI (Java Naming and Directory Interface) to lookup the session bean. All EJB containers implement JNDI. JNDI is a set of interfaces which has many implementations: RMI, LDAP, and CORBA amongst others. There are two steps for performing a JNDI lookup. The first step is to create a JNDI connection by creating a `javax.naming.InitialContext` object. One way to do this is to create a `java.util.Properties` object, set JNDI properties specific to the EJB container, then pass the properties object as an argument to `InitialContext`. This will take the form:

```
Properties env  = new Properties();
// Use env.put to set container specific JNDI properties
InitialContext ctx = new InitialContext(env);
```

An alternative method is to define the properties in a `jndi.properties` file and use the no-arg `InitialContext` constructor:

```
InitialContext ctx = new InitialContext();
```

The JNDI properties are then accessed at runtime. The GlassFish container provides a default `jndi.properties` file which is contained in `appserv-rt.jar`. This JAR file must be in the client's classpath, as we shall shortly see when we examine the Ant build file.

The second step is to use the `InitialContext` object to lookup the remote EJB. In our example this is done by the statement:

```
TimeService timeService =
             (TimeService) ctx.lookup("ejb30.session.TimeService");
```

Note the argument passed is the global JNDI name. This is generated by the container at deploy time. The form of the global JNDI name is also container-specific. In the case of GlassFish this is the fully qualified remote session bean interface name. In our example this is `ejb30.session.TimeService`. The lookup result is directly cast to the session bean interface type, namely `TimeService`.

We can now invoke the session bean business method:

```
String time = timeService.getTime();
```

The use of JNDI is one of the more awkward aspects of EJB technology. Fortunately in EJB 3 the only occasion its use is mandatory is when a client invokes EJB from outside the container. If we invoke an EJB from within a container, then we can use dependency injection annotation metadata instead of JNDI lookups. We will return to this shortly.

Running the Example

First we need to compile the source code. The Ant script is run from the command line, as follows:

cd C:\EJB3Chapter02\glassfish\lab1

C:\EJB3Chapter02\glassfish\lab1>ant compile

This will compile the interface, `TimeService.java`, the session bean, `TimeServiceBean.java`, and the client, `Client.java`.

The next step is to package the session bean into an EJB module. An EJB module consists of one or more EJBs contained in a JAR file. In our example we have just one EJB, `TimeService`, and we name the JAR file, `TimeService.jar`:

C:\EJB3Chapter02\glassfish\lab1>ant package-ejb

We can use the `jar -tvf` command to examine the contents of the JAR file:

```
C:\EJB3Chapter02\glassfish\lab1\build>jar  -tvf  TimeService.jar
   ...META-INF/
   ...META-INF/MANIFEST.MF
   ...ejb30/
   ...ejb30/session/
   ...ejb30/session/TimeService.class
   .. ejb30/session/TimeServiceBean.class
```

Note the JAR file contains the standard `META-INF` directory and the default manifest file, `MANIFEST.MF`.

The next step is to deploy the program. We can do this within the GlassFish administrator console, which has an option for deploying an EJB module. Alternatively we can use an Ant script as follows:

```
C:\EJB3Chapter02\glassfish\lab1>ant deploy
```

Deployment is essentially the process by which applications are transferred to the control of the application server. The developer copies the EJB module to designated sub-directories within the application server directory structure. However, behind the scenes a lot more takes place during deployment. Typically some sort of verification takes places to check that the contents of the EJB module are well formed and comply with the Java EE specification. If annotations are used, the application server may generate corresponding XML deployment descriptor files. Additional vendor-specific XML files and classes and interfaces are typically generated. Entities will be mapped to a database and optionally corresponding database tables may be created during deployment. We will discuss entities and databases in Chapter 3. During deployment security roles are mapped to user groups. We will discuss security in Chapter 12.

Note that some application servers do not have the option of deploying EJB modules. Such modules need to be placed in an EAR (Enterprise Application Archive) file before deployment. We shall see an example of an EAR file shortly.

Finally we can run the example with the run-client Ant command:

```
C:\EJB3Chapter02\glassfish\lab1>ant run-client
   Buildfile: build.xml
   compile:
   run-client:
     [java] Time is: 03:21:17 PM
   BUILD SUCCESSFUL
```

The Program Directory Structure

Before we describe the Ant build scripts in more detail we will look at the directory structure in which the source code is placed.
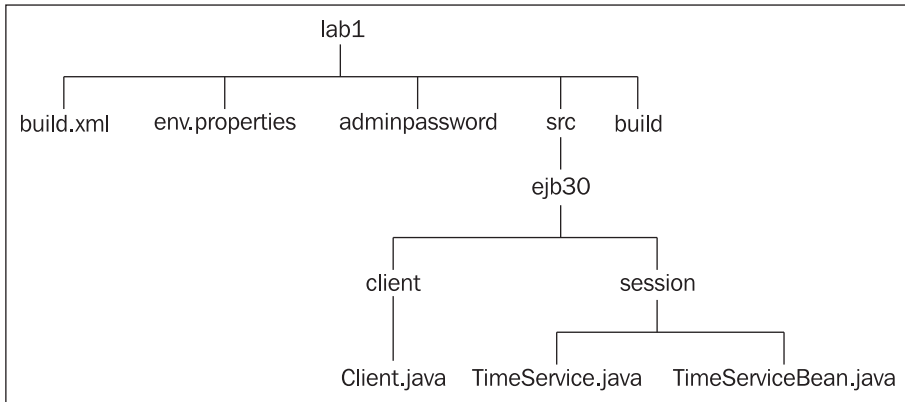
lab1: This is the root directory and contains the following files and subdirectories:

- **build.xml**: This is the Ant build script itself. We will describe it in more detail shortly.

- **env.properties:** This file contains runtime environment property settings and is read by the Ant build script. Currently the file contains just one value, `glassfish.home`, which is the root directory in which GlassFish is installed. When writing this book GlassFish was installed on Windows in the `C:/glassfish` directory, so the contents of `env.properties` is:

  `glassfish.home=C:/glassfish`

- **adminpassword:** This file contains the GlassFish administrator password required for the deployment task. This is identified by `AS_ADMIN_PASSWORD`. The initial password, which can be reset of course, is `adminadmin`. So the contents of `adminpassword` are:

  `AS_ADMIN_PASSWORD=adminadmin`

- **src:** This subdirectory contains the source code. Recall the client program, `Client.java`, is placed in the `ejb30.client` package. The package name is reflected in the subdirectory structure. Subdirectories have also been created to correspond to the `ejb30.session` package in which `TimeService` and `TimeServiceBean` have been placed.

- **build:** This subdirectory contains files created during the build process. For example, class files created during compilation and JAR files created during packaging will be placed here.

The Ant Build Script

We use an Ant build file for automating the compilation of client and session bean source code as well as packaging and deploying the session bean. We also use the build file for running the client. All these steps are defined as Ant targets. The Ant build file is listed below:

```
<project name="ejb30notebook" basedir=".">

<property name="build.dir" value="${basedir}/build" />
<property name="src.dir" value="${basedir}/src" />
<property file="env.properties" />

<path id="j2ee.classpath">
      <pathelement location="${build.dir}"/>
      <fileset dir="${glassfish.home}/lib">
          <include name="javaee.jar"/>
          <include name="appserv-rt.jar"/>
      </fileset>
</path>

<target name="clean">
   <delete dir="${build.dir}"/>
   <mkdir dir="${build.dir}" />
   <mkdir dir="${build.dir}/lib" />
</target>

<target name="all">
    <antcall target="clean"/>
    <antcall target="compile"/>
    <antcall target="package-ejb"/>
    <antcall target="deploy"/>
    <antcall target="run-client"/>
</target>
```

```
<target name="compile">
    <javac destdir="${build.dir}"
        srcdir="${src.dir}"
        classpathref="j2ee.classpath"/>
</target>


<target name="package-ejb" depends="compile">
    <jar jarfile="${build.dir}/TimeService.jar">
     <fileset dir="${build.dir}">
        <include name="ejb30/session/**" />
     </fileset>

   </jar>
</target>


<target name="deploy">
    <exec executable="${glassfish.home}/bin/asadmin"
        failonerror="true"
        vmlauncher="false">
    <arg line="deploy --user admin --passwordfile
        adminpassword    ${build.dir}/TimeService.jar"/>
    </exec>
</target>


<target name="undeploy">
    <exec executable="${glassfish.home}/bin/asadmin"
        failonerror="true"
        vmlauncher="false">
     <arg line="undeploy --user admin  --passwordfile
        adminpassword  TimeService"/>
    </exec>
  </target>


<target name="run-client" depends="compile">
    <java classname="ejb30.client.Client" fork="yes"
        classpathref="j2ee.classpath"/>
</target>

</project>
```

Within the `path` tag we have the classpath, identified by `j2ee.classpath`. This contains the `javaee.jar` file (containing libraries such as `javax.ejb`) required for compilation and `appserv-rt.jar` required for runtime execution of a client.

The `clean` target deletes and recreates the `build` directory. This is used to ensure that all generated files such as class and JAR files are recreated from scratch.

The `all` target performs all the tasks required to deploy and run the example.

The `deploy` target uses the GlassFish `asadmin` tool to deploy the `TimeService.jar` file. The line:

```
<arg line="deploy --user admin --passwordfile
    adminpassword    ${build.dir}/TimeService.jar"/>
```

specifies that deployment is performed by the `admin` user. The `admin` password is held in a file named `adminpassword`.

There is also an `undeploy` target, which removes `TimeService.jar` from the application server.

The Application Client Container

Although accessing an EJB from a client using JNDI is simpler than in EJB 2.x, it is still rather awkward. The good news is that we can dispense with JNDI altogether if the client runs from within an application client container (ACC). The EJB 3 specification does not mandate that an EJB-compliant application server provides an ACC but makes its inclusion optional. Consequently not all EJB-compliant application servers provide an ACC, however GlassFish does.

An ACC enables a client executing in its own JVM outside the EJB container to access a number of Java EE services such as JMS resources, security, and metadata annotations.

If the previous client example is run from an ACC then we can use dependency injection annotation metadata instead of JNDI lookups. Here is the new version of `Client`:

```
package ejb30.client;
import javax.naming.*;
import ejb30.session.*;
import javax.ejb.EJB;
public class Client {
    @EJB
    private static TimeService timeService;
    // injected field must be static
    public static void main(String args[]) throws Exception {
        String time = timeService.getTime();
        System.out.println("Time is: " + time);
    }
}
```

We use the `@EJB` annotation to instruct the container to lookup the `TimeService` bean and inject a bean reference into the `timeService` field. This is an example of **field injection**. Another kind of dependency injection is **setter injection**; we shall see an example of this later in this chapter. We can then invoke any of the `TimeService` methods such as `timeService.getTime()`.

Because the `Client` class runs in a static context, as a main method, the injected field must also be static.

Building the Application

We need to modify the build process if we are using an ACC. The build steps are application sever-specific so what follows in this section applies only to GlassFish. First we create a `manifest.mf` file with the contents:

```
Main-Class: ejb30.client.Client
```

We need to do this because the client will be bundled in a JAR file, as we shall see shortly, and we need to indicate which class in the JAR file is the application's entry point. The compilation and EJB packaging tasks require no change. However in order to use the ACC we must package the client in a JAR file. The following Ant script creates a `Client.jar` file.

```
<target name="package-client" depends="compile">
    <jar jarfile="${build.dir}/Client.jar"
         manifest="${config.dir}/manifest.mf">
         <fileset dir="${build.dir}">
                <include name="ejb30/client/Client.class" />
         </fileset>
    </jar>
</target>
```

Note that we have included the customized manifest file. The next step is to create an Enterprise Application Archive (EAR) file. An EAR file is a standard JAR file, with an `ear` extension, that contains other JAR files embedded within it. The JAR files can be EJB modules or any Java classes, such as an application client deployed in a JAR file. An EAR file can also contain a web module, identified by a `war` extension. We shall see an example of this later in the book.

In our example the EAR file, which we name `TimeService.ear`, will contain the EJB module `TimeService.jar` and the client JAR file, `Client.jar`. The Ant script to create an EAR file is:

```
<target name="package-ear" depends="package-ejb">
        <jar destfile="${build.dir}/TimeService.ear"
             basedir="${build.dir}"
             includes="TimeService.jar Client.jar"/>
</target>
```

Again we can use the `jar –tvf` command to examine the contents of the EAR file:

```
C:\EJB3Chapter02\glassfish\lab2\build>jar –tvf  TimeService.ear
    ...META-INF/
    ...META-INF/MANIFEST.MF
    ...Client.jar
    .. TimeService.jar
```

We deploy the EAR file rather than the EJB module. The Ant script shows this:

```
<target name="deploy">
    <exec executable="${glassfish.home}/bin/asadmin"
          failonerror="true"
          vmlauncher="false">
      <arg line="deploy --user admin –passwordfile
          adminpassword    ${build.dir}/TimeService.ear"/>
    </exec>
</target>
```

Alternatively we can deploy an EAR file from the GlassFish administrator console. There is an option for deploying an enterprise application contained in an EAR file.

Finally we need to modify the Ant run-client script:

```
<target name="run-client">
    <exec executable="${glassfish.home}/bin/appclient"
          failonerror="true"
          vmlauncher="false">
      <arg line="-client
              ${glassfish.home}/domains/domain1/generated/xml/
                j2ee-apps/TimeService/TimeServiceClient.jar
                -mainclass ejb30.client.Client"/>
```

Note that we start up the GlassFish ACC by executing the `appclient` program. At deployment GlassFish creates a directory under the `domains/domain1/generated/ xml/j2ee-apps` directory. The name of this generated directory is the same as the EAR file without the extension, `TimeService` in our case. In this directory GlassFish creates a JAR file with the same name as the directory but with `Client` appended, `TimeServiceClient.jar` in our case. The contents of this JAR file are those of the deployed EAR file together with some GlassFish generated deployment XML files. When we execute `appclient` we need to supply the full pathname of `TimeServiceClient.jar` as well as the fully qualified main application client class.

From this point on, throughout the book we assume that clients will always run in an ACC.

Stateless Session Bean's LifeCycle

It is import to stress that a session bean's lifecycle is controlled by the container and not the application. The following state diagram shows the stateless session bean's lifecycle:
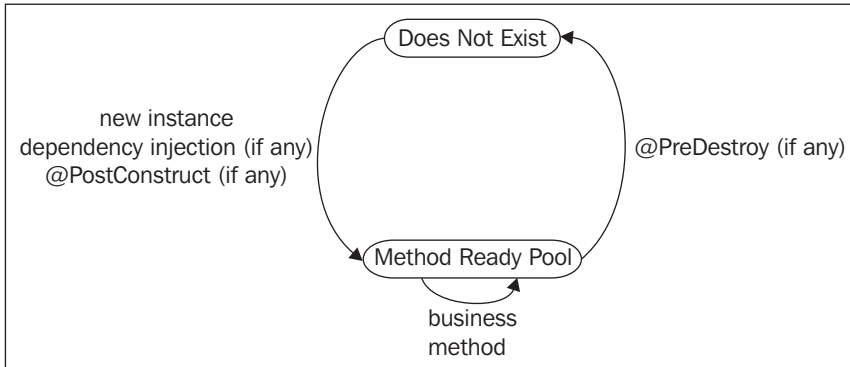
The initial state of a stateless session bean is the **does-not-exist** state. This would be the case before a container starts up, for example. The next state is the **method-ready pool**. When the container starts up, it typically creates a number of stateless session bean instances in the method-ready pool. However the container can decide at any time to create such instances. In order to create an instance in the method-ready pool, the container performs the following steps:

1. The bean is instantiated.

2. The container injects the bean's `SessionContext`, if applicable. In our example we have not made use of the `SessionContext`. The `SesionContext` is used by a bean to query the container about the bean's status or context.

3. The container performs any other dependency injection that is specified in the bean's metadata. Again in our example we have not specified any such metadata.

4. The container then invokes a `PostConstruct` callback method if one is present in the bean. Again we have not invoked such a method in our bean example. The `PostConstruct` method would be used for initializing any resources used by the bean. For example, the session bean may make use of a JMS queue for sending messages. The connection queue used by JMS could be initialized in the `PostConstruct` method. A `PostConstruct` method is called only once in the life of an instance, when it has transitioned from the does-not-exist state to the method-ready pool.

The container then calls a business method on the bean. Each business method call could originate from a different client. Conversely when a client invokes a business method on a stateless session bean any instance in the method-ready pool can be chosen by the container to execute the method.

After a business method has been invoked, the container may decide to destroy the bean instance (typically if the container decides there are too many instances in the method-ready pool) or reuse the instance on behalf of any client that invokes its business methods.

When the container does decide to destroy the instance it first invokes the PreDestroy callback method if one is present in the bean. In our example we have not invoked such a method. The PreDestroy method would be used for tidying up activities, such as closing connections that may have been opened in the PostConstruct method. A PreDestroy method is called only once in the life of an instance, when it is about to transition to the does-not-exist state.

These features of instance pooling and instance sharing mean that stateless session beans scale well to large number of clients.

The listing below shows a modified implementation of TimeServiceBean with a SessionContext and callback methods added. At this stage the methods merely print out messages. Later in this book we shall see more meaningful examples.

```
@Stateless
public class TimeServiceBean implements TimeService {
    @Resource private SessionContext ctx;
    @PostConstruct
    public void init() { System.out.println(
                "Post Constructor Method init() Invoked"); }

    public String getTime() {
        System.out.println(ctx.getInvokedBusinessInterface());
        Formatter fmt = new Formatter();
        Calendar cal = Calendar.getInstance();
        fmt.format("%tr", cal);
        return fmt.toString();
    }

    @PreDestroy
    public void tidyUp() {
        System.out.println(
                "Pre Destruction Method tidyUp() Invoked");
    }
}
```

Note that there can be at most only one `PostConstruct` and one `PreDestroy` method. The methods can take any name but must be `void` with no arguments. In the case of GlassFish, the messages printed out by these methods will appear in the container's log file, which can be viewed from the GlassFish administrator console.

Note the statement:

```
@Resource private SessionContext ctx;
```

This uses the `@Resource` annotation to inject a reference to the `javax.ejb.SessionContext` object into the `ctx` field. Any kind of dependency injection, apart from two exceptions, is signaled with the `@Resource` annotation. The two exceptions are `@EJB`, which is used for injecting EJB references and `@PersistenceContext`, which is used for injecting `EntityManager` instances. We will cover the `@PersistenceContext` annotation in Chapter 3.

In the `getTime()` method we have added the statement:

```
System.out.println(ctx.getInvokedBusinessInterface());
```

This uses one of `SessionContext` methods, namely `getInvokedBusinessInterface()`. This method returns the session bean interface through which the current method is invoked. In our example the result is `ejb30.session.TimeService`. This is a rather artificial use of `getInvokedBusinessInterface()` since our session bean implements just one interface, namely `TimeService`. However a session bean could implement two interfaces: a remote interface and a local interface. We will discuss local interfaces later in this chapter. One client could invoke `TimeServiceBean` through the remote interface and another through the local interface, `getInvokedBusinessInterface()`, which would inform us whether the current client is remote or local.

There are a more methods in the `SessionContext` interface regarding the session beans current transaction and security contexts. We shall return to this in the chapters on Transactions and Security.

Stateful Session Beans

In contrast to stateless session beans, **stateful** session beans maintain state for an individual client over one or more method requests. A stateful session bean is not shared among clients, and a client's reference to a bean only ends when the client ends the session or the session times out. The state is not written to a database but held in the containers cache and is lost when the container crashes.

The classic example of a stateful session bean is the online shopping cart. The user adds one or more items to a shopping cart, possibly over a long time period while the user visits other web sites or is interrupted by a phone call. Finally the user may decide to purchase the items in the cart, cancel his or her shopping cart session, or even abandon the session without explicitly terminating it.

The following example illustrates some aspects of a shopping cart. Our shopping cart will be limited to adding items to a cart and listing the current contents of a cart. First we define a remote interface as follows:

```
@Remote
public interface ShoppingCart {
    public void initialize();
    public void addItem(String item);
    public Collection<String> getItems();
    public void finished();
}
```

The `initialize()` method performs initialization; we shall see what this means shortly. The `addItem()` method adds an item to the cart. The `getItems()` method returns the current contents of the cart. The `finished()` method is invoked when the client has finished their session and typically removes the session bean, so allowing the container to reclaim resources.

Next we look at the bean implementation:

```
@Stateful
public class ShoppingCartBean implements ShoppingCart {
    private ArrayList<String> items;
    @PostConstruct
    public void initialize() {
        items = new ArrayList<String>();
    }
    public void addItem(String item) {
        items.add(item);
    }
    public Collection<String> getItems() {
        return items;
    }
    @Remove
    public void finished() {
        System.out.println(
                "Remove method finished() Invoked");
    }
}
```

First note that we have used the @Stateful annotation to indicate this is a stateful session bean. We have used a @PostConstruct method, initialize(), to create an ArrayList named items. Stateful session beans may contain instance variables, such as items, which are modified by the bean. The reason, of course, is that the state of such variables is preserved from one method invocation to another. As with stateless beans we can also have a @PreDestroy method, although we have not created one in this example. The addItem() and getItems() methods are straightforward and need no further comment.

We use the @Remove annotation to indicate that after the annotated method has executed, the client no longer needs the session. The container can then delete the bean instance. If a @PreDestroy method exists, it will be executed after the @Remove method. Typically the method would perform tidying up such as closing connections. In our example the finished() method merely prints a message.

It is not compulsory to have a @Remove annotated method, but it is a good practice. Without such a method the bean instance will continue to occupy the container's cache until a timeout period is reached. At this point the bean instance is deleted by the container. The timeout period is configured by the container's administrator. By having a @Remove annotated method we are freeing up container resources as soon as we have made use of them.

The listing below shows a possible client:

```
public class Client {
    @EJB
    private static ShoppingCart shoppingCart;
    public static void main(String args[]) throws Exception {
        shoppingCart.addItem("Bread");
        shoppingCart.addItem("Milk");
        shoppingCart.addItem("Tea");
        System.out.println("Contents of your cart are:");
        Collection<String> items = shoppingCart.getItems();
        for (String item : items) {
            System.out.println(item);
        }
        shoppingCart.finished();
    }
}
```

Stateful Session Bean's LifeCycle

The following state diagram shows the stateful session bean's lifecycle:
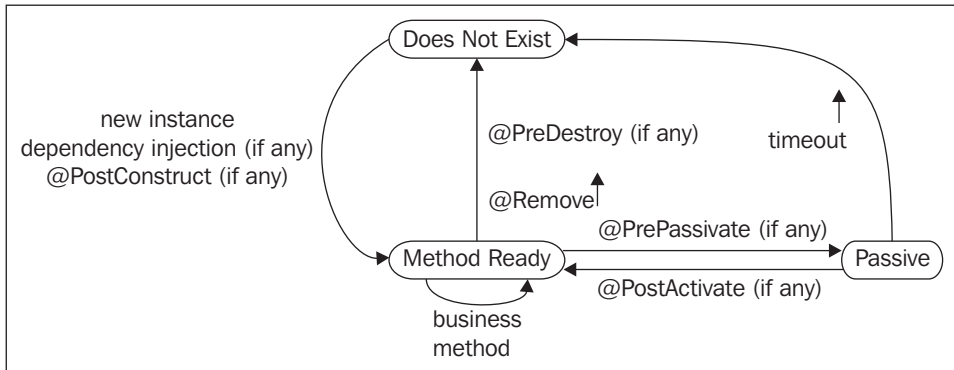
The lifecycle of a stateful session bean differs from that of a stateless session bean in a number of respects. There is no pool of identical bean instances, as each instance serves a single client. A stateful session bean transitions from a does-not-exist to a method-ready state as soon as a client performs a business interface lookup or a dependency injection of a bean. So this step is client, and not container initiated. Once a stateful session bean is in a method-ready state, the container performs the same steps as with a method-ready stateless session bean:

1. The bean is instantiated.

2. The container injects the bean's `SessionContext`, if applicable.

3. The container performs any other dependency injection that is specified in the bean's metadata.

4. The container then invokes a `PostConstruct` callback method if one is present in the bean.

The container then calls a business method on the bean.

Recall that a session can involve considerable idle time if a client decides to wait, or perform other activities, between method calls. The container's cache could become full of idle stateful session bean instances. To limit such instances, the container can swap these from the cache out to disk or other storage. This process is called **passivation** and involves serialization of the bean instance. When the client decides to invoke a method on a passivated bean, then the container deserializes the saved state and invokes the requested method. This process is known as **activation**. Note that the method may not be on the original bean instance, but this is safe as at activation we restore the state which was saved at passivation.

The passivation algorithm is container-dependent, and typically is dependent on factors such as the maximum number of beans which may be held in the containers cache, the maximum bean idle time, and the victim selection policy of the container. The GlassFish victim selection policy has the following options: Least Recently Used (LRU), Not Recently Used (NRU), and First In First Out (FIFO). All of these factors can be configured by the container administrator.

The bean can provide a `@PrePassivate` callback method, which is executed immediately before passivation and a `@PostActivate` callback method, which is executed immediately after activation. A session bean's state may contain open resources such as open sockets, database or JMS connections that cannot be handled by the container during passivation. In these cases the resource will be closed in the `@PrePassivate` method and reopened in the `@PostActivate` method.

A bean can move from a method-ready state to a does-not-exist state if a `@Remove` annotated method is invoked. A bean can also do this if a container configured timeout period has been reached. In either case a `@PreDestroy` annotated method is first invoked.

A passivated bean can also move to a does-not-exist state if a timeout period has been reached. Note in this case any `@PreDestroy` annotated method will **not** be invoked.

The listing below shows a modified implementation of `ShoppingCartBean` with the `@PrePassivate` and `@PostActivate` callback methods added. The methods simply print out corresponding messages.

```
@Stateful
public class ShoppingCartBean implements ShoppingCart {
    private ArrayList<String> items;

    @PostConstruct
    public void initialize() {
        items = new ArrayList<String>();
    }
    public void addItem(String item) {
        items.add(item);
    }
    public Collection<String> getItems() {
        return items;
    }
    @PrePassivate
    public void logPassivation() {
        System.out.println(
            "PrePassivate method logPassivation() Invoked");
```

```
    }
    @PostActivate
    public void logActivation() {
        System.out.println(
            "PostActivate method logActivation() Invoked");
    }
    @Remove
    public void finished() {
        System.out.println(
            "Remove method finished() Invoked");
    }
}
```

Note that there can be at most only one `PrePassivate` and one `PostActivate` method. The methods can take any name but must be `void` with no arguments.

Note that to simulate bean idle time, and hence conditions for passivation and activation, we have added a delay of 59 seconds between method calls from the Client:

```
public class Client {
    public static void main(String args[]) throws Exception {
        ...
        shoppingCart.addItem("Bread");
        shoppingCart.addItem("Milk");
        Thread.sleep(59000);
        shoppingCart.addItem("Tea");
    }
}
```

To test activation and passivation with GlassFish you will also need to log in as a container administrator and set the `Max Cache Size` to say `2`. The `Cache Resize` quantity, which is the number of beans to passivate when the cache is full, is set to `2` and the `Cache Idle Timeout` to say `15 seconds`. Then execute 3 simultaneous client sessions. This process may be different for other EJB containers.

Local Interfaces

Up to this point all our examples have used the session bean remote interface, as the clients have run in their own JVM outside the EJB container. Behind the scenes, a remote interface uses the RMI-IIOP protocol for network operations. This protocol stipulates that method arguments are passed by value and not by reference. Passing by value means that an object being passed from the client to the remote bean, or vice versa, is first serialized then passed over the network then deserialized. This all

has an impact in terms of performance. Even if our client is a session bean invoking another in the same container there is a performance overhead if we use a remote interface because of the serialization and deserialization taking place. For this reason EJB technology provides a local interface option for session beans. Method arguments are passed by reference and not by value so improving performance.

To illustrate all this we shall create a stateless session bean which will be invoked by and run in the same EJB container, as our original `TimeServiceBean`. This invoked bean will just return the string `Have a Nice Day`. First we define the beans interface, `NiceDayService`:

```
package ejb30.session;
import javax.ejb.Local;

@Local
public interface NiceDayService {
    public String getMessage();
}
```

Note that we have prefixed the interface definition with the `@Local` annotation. This indicates to the EJB container that this bean may only be invoked by a local client running in the same container. The interface consists of just one method definition, `getMessage()`. Next we look at the bean implementation, `NiceDayServiceBean`:

```
package ejb30.session;
import java.util.*;
import javax.ejb.Stateless;

@Stateless
public class NiceDayServiceBean implements NiceDayService {
    public String getMessage() {
        return " - Have a Nice Day";
    }
}
```

This is all straightforward. We note that the session bean is stateless and the `getMessage()` method returns the string `- Have a Nice Day`.

Now let's look as to how we might modify the `TimeServiceBean` to invoke `NiceDayService`. The `TimeServiceBean` will append the `NiceDayService` message to the current time. Here is the modified code for `TimeServiceBean`:

```
package ejb30.session;
import java.util.*;
import javax.ejb.Stateless;
import javax.ejb.EJB;

@Stateless
```

```
public class TimeServiceBean implements TimeService {
    private @EJB NiceDayService niceDay;
    public String getTime() {
        Formatter fmt = new Formatter();
        Calendar cal = Calendar.getInstance();
        fmt.format("%tr", cal);
        return fmt.toString() +
                                niceDay.getMessage();

    }
}
```

We use the `@EJB` annotation to instruct the container to lookup the `NiceDayService` bean and inject a bean reference into the `niceDay` field. Recall this is an example of field injection. We can then invoke the `niceDay.getMessage()` method.

An alternative to field injection is setter injection. In this case a method rather than a field is annotated. The following version of `TimeServiceBean` uses setter injection:

```
@Stateless
public class TimeServiceBean implements TimeService {
    private NiceDayService niceDay;
    public String getTime() {
        Formatter fmt = new Formatter();
        Calendar cal = Calendar.getInstance();
        fmt.format("%tr", cal);
        return fmt.toString() + niceDay.getMessage();
    }
    @EJB
    public void setNiceDay(NiceDayService niceDay) {
        this.niceDay = niceDay;
    }
}
```

By annotating the `setNiceDay()` method with the `@EJB` annotation, we are instructing the container to lookup the `NiceDayService` bean, then invoke the `setNiceDay()` method passing the `NiceDayService` bean reference as a parameter to `setNiceDay()`.

Summary

We have covered a lot of ground in this chapter. We have seen that session beans are an EJB technology for encapsulating business logic. Session beans can be either stateless or stateful. We had our first examples of using metadata annotations in our beans. We looked at packaging and deploying session beans using Ant scripts. We showed how a client running outside an EJB container would invoke a session bean using JNDI. We also showed how a client running in an Application Client Container can use dependency injection instead of JNDI to invoke a session bean.

We described the lifecycle for both stateless and stateful session beans and looked at examples of session bean lifecycle callback methods.

3

Entities

Entities are classes that need to be persisted, usually in a relational database. In this chapter we cover the following topics:

- EJB 3 entities
- Java persistence API
- Mapping an entity to a database table
- Metadata defaults
- Introduction to the entity manager
- Packaging and deploying entities
- Generating primary keys
- Overriding metadata defaults

Introduction

Entities are classes that need to be persisted; their state is stored outside the application, typically in a relational database. Unlike session beans, entities do not have business logic other than validation. As well as storing such entities, we want to query, update, and delete them.

The EJB 3 specification recognizes that many applications have the above persistence needs without requiring the services (security, transactions) of an application server EJB container. Consequently the persistence aspects of EJB 3 have been packaged as a separate specification—the Java Persistence API (JPA). JPA does not assume we have a container and can even be used in a Java SE (Standard Edition) application. As well as persistence, JPA deals with Object/Relational Mapping and Queries, these are covered in Chapters 4 and 5 respectively. Most of our examples assume that the persistence engine exists within an EJB 3 container such as GlassFish or JBoss. In Chapter 6 we shall show examples of persistence outside a container.

Successful standalone object-relational mapping products such as open source Hibernate and proprietary Oracle Toplink have implemented these persistence technologies for a number of years. Creators of Oracle Toplink and Hibernate have been influential in the development of the JPA specification. So, readers who are familiar with either Hibernate or Toplink will recognize similarities between the JPA and those products. Furthermore, under the covers, the GlassFish application server implements the JPA using Toplink and the JBoss application server uses Hibernate. These are pluggable defaults however, so it is possible to implement the JPA in GlassFish using Hibernate for example.

The JPA can be regarded as a higher level of abstraction sitting on top of JDBC. Under the covers the persistence engine converts JPA statements into lower level JDBC statements.

EJB 3 Entities

In JPA, any class or POJO (Plain Old Java Object) can be converted to an entity with very few modifications. The following listing shows an entity `Customer.java` with attributes `id`, which is unique for a Customer instance, and `firstName` and `lastName`.

```
package ejb30.entity;
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
public class Customer implements java.io.Serializable {
    private int id;
    private String firstName;
    private String lastName;
    public Customer() {}
    @Id
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getFirstname() { return firstName; }
    public void setFirstname(String firstName) {
        this.firstName = firstName;
    }
    public String getLastname() { return lastName; }
    public void setLastname(String lastName) {
        this.lastName = lastName;
    }
    public String toString() {
        return "[Customer Id =" + id + ",first name=" +
                firstName + ",last name=" + lastName + "]";
    }
}
```

The class follows the usual JavaBean rules. The instance variables are non-public and are accessed by clients through appropriately named getter and setter accessor methods. Only a couple of annotations have been added to distinguish this entity from a POJO. Annotations specify entity metadata. They are not an intrinsic part of an entity but describe how an entity is persisted or, as we shall see in Chapter 4, how an entity is related to other entities. The `@Entity` annotation indicates to the persistence engine that the annotated class, in this case `Customer`, is an entity. The annotation is placed immediately before the class definition and is an example of a **class level** annotation. We can also have **property-based** and **field-based** annotations, as we shall see.

The `@Id` annotation specifies the primary key of the entity. The `id` attribute is a primary key candidate. Note that we have placed the annotation immediately before the corresponding getter method, `getId()`. This is an example of a property-based annotation. A property-based annotation must be placed immediately before the corresponding getter method, and not the setter method. Where property-based annotations are used, the persistence engine uses the getter and setter methods to access and set the entity state.

An alternative to property-based annotations are field-based annotations. We show an example of these later in this chapter. Note that all annotations within an entity, other than class level annotations, must be all property-based or all field-based.

The final requirement for an entity is the presence of a no-arg constructor.

Our Customer entity also implements the `java.io.Serializable` interface. This is not essential, but good practice because the Customer entity has the potential of becoming a detached entity. Detached entities must implement the `Serializable` interface. We will discuss detached entities in Chapter 6.

At this point we remind the reader that, as throughout EJB 3, XML deployment descriptors are an alternative to entity metadata annotations.

Comparison with EJB 2.x Entity Beans

An EJB 3 entity is a POJO and not a component, so it is referred to as an entity and not an entity bean. In EJB 2.x the corresponding construct is an entity bean component with the same artifacts as session beans, namely an XML deployment descriptor file, a remote or local interface, a home or localhome interface, and the bean class itself. The remote or local interface contains getter and setter method definitions. The home or local interface contains definitions for the `create()` and `findByPrimaryKey()` methods and optionally other finder method definitions. As with session beans, the entity bean class contains callback methods such as `ejbCreate()`, `ejbLoad()`, `ejbStore()`, `ejbRemove()`, `ejbActivate()`, `ejbPassivate()`, and `setEntityContext()`.

The EJB 3 entity, being a POJO, can run outside a container. Its clients are always local to the JVM. The EJB 2.x entity bean is a distributed object that needs a container to run, but can have clients from outside its JVM. Consequently EJB 3 entities are more reusable and easier to test than EJB 2.x entity beans.

In EJB 2.x we need to decide whether the persistence aspects of an entity bean are handled by the container (Container Managed Persistence or CMP) or by the application (Bean Managed Persistence or BMP).

In the case of CMP, the entity bean is defined as an abstract class with abstract getter and setter method definitions. At deployment the container creates a concrete implementation of this abstract entity bean class.

In the case of BMP, the entity bean is defined as a class. The getter and setter methods need to be coded. In addition the `ejbCreate()`, `ejbLoad()`, `ejbStore()`, `ejbFindByPrimaryKey()`, and any other finder methods need to be coded using JDBC.

Mapping an Entity to a Database Table

We can map entities onto just about any relational database. GlassFish includes an embedded Derby relational database. If we want GlassFish to access another relational database, Oracle say, then we need to use the GlassFish admin console to set up an Oracle data source. We also need to refer to this Oracle data source in the `persistence.xml` file. We will describe the `persistence.xml` file later in this chapter. These steps are not required if we use the GlassFish default Derby data source. All the examples in this book will use the Derby database.

EJB 3 makes heavy use of defaulting for describing entity metadata. In this section we describe a few of these defaults.

First, by default, the persistence engine maps the entity name to a relational table name. So in our example the table name is CUSTOMER. If we want to map the Customer entity to another table we will need to use the `@Table` annotation as we shall see later in this chapter.

By default, property or fields names are mapped to a column name. So ID, FIRSTNAME, and LASTNAME are the column names corresponding to the `id`, `firstname`, and `lastname` entity attributes. If we want to change this default behavior we will need to use the `@Column` annotation as we shall see later in this chapter.

JDBC rules are used for mapping Java primitives to relational datatypes. So a `String` will be mapped to VARCHAR for a Derby database and VARCHAR2 for an Oracle database. An `int` will be mapped to INTEGER for a Derby database and NUMBER for an Oracle database.

The size of a column mapped from a `String` defaults to 255, for example `VARCHAR(255)` for Derby or `VARCHAR2(255)` for Oracle. If we want to change this column size then we need to use the `length` element of the `@Column` annotation as we shall see later in this chapter.

To summarize, if we are using the GlassFish container with the embedded Derby database, the Customer entity will map onto the following table:

CUSTOMER
ID INTEGER PRIMARY KEY
FIRSTNAME VARCHAR(255)
LASTNAME VARCHAR(255)

Most persistence engines, including the GlassFish default persistence engine, Toplink, have a schema generation option, although this is not required by the JPA specification. In the case of GlassFish, if a flag is set when the application is deployed to the container, then the container will create the mapped table in the database. Otherwise the table is assumed to exist in the database.

Introducing the EntityManager

We will use a remote stateless session bean for business operations on the Customer entity. We will call our session bean interface `BankService`. In later chapters, we will add more entities and business operations that are typically used by a banking application. At this stage our banking application deals with one entity—Customer, and business methods `addCustomer()` and `findCustomer()`, which respectively add a customer to the database and retrieve a customer entity given the customers identifier. The interface listing follows:

```
package ejb30.session;
import javax.ejb.Remote;
import ejb30.entity.Customer;

@Remote
public interface BankService {
    void addCustomer(int custId, String firstName,
                     String lastName);
    Customer findCustomer(int custId);
}
```

The code below shows the session bean implementation, `BankServiceBean`:

```
package ejb30.session;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import ejb30.entity.Customer;
import javax.persistence.PersistenceContext;
@Stateless
public class BankServiceBean implements BankService {

    @PersistenceContext(unitName="BankService")
    private EntityManager em;

    public Customer findCustomer(int custId) {
        return ((Customer)
                        em.find(Customer.class, custId));
    }
    public void addCustomer(int custId, String firstName,
                            String lastName) {
        Customer cust = new Customer();
        cust.setId(custId);
        cust.setFirstname(firstName);
        cust.setLastname(lastName);
        em.persist(cust);
    }
}
```

The `EntityManager` is a service provided by the persistence engine which provides methods for persisting, finding, querying, removing, and updating entities. The set of managed entity instances is known as a **persistence context**. It is important to stress that a persistence context is associated with a managed set of entity *instances* or *objects*. If we have, say, 10 clients accessing the bean, we will have 10 persistence contexts. Associated with an `EntityManager` is a **persistence unit** which specifies configuration information. The statement:

```
@PersistenceContext(unitName="BankService")
private EntityManager em;
```

defines an `EntityManager` instance `em` with an associated persistence context and a persistence unit named `BankService`. The actual persistence unit configuration is specified in a `persistence.xml` file which we shall see shortly.

Actually, behind the scenes, the `@PersistenceContext` annotation causes the container to:

- Use the `EntityManagerFactory` to create an instance of the `EntityManager`
- Bind the `EntityManager` instance to the `BankService` persistence unit
- Inject the `EntityManager` instance into the `em` field.

Now let's take a look at the `addCustomer()` method. First, the statement

```
Customer cust = new Customer();
```

creates, as expected, an instance of the Customer entity. However, at this stage the entity instance is not managed. The instance is not yet associated with a persistence context. Managed entity instances are also referred to as **attached** instances and unmanaged instances as **detached** instances.

The next few statements invoke the Customer setter methods in the usual way. The statement:

```
em.persist(cust);
```

invokes the `EntityManager.persist()` method. At this stage the entity instance is not necessarily written immediately to the database. Rather, the entity instance becomes managed or attached and associated with a persistence context. The `EntityManager` manages the synchronization of a persistence context with the database. There may be more setter methods in our method after the `persist()` statement. The `EntityManager` is unlikely to update, or flush, the database after each setter as database write operations are expensive in terms of performance. More likely the entity state would be kept in a cache and flushed to the database at the end of the current transaction. The latest point at which the `EntityManager` will flush to the database is when the transaction commits, although the `EntityManager` may chose to flush sooner.

By default, a transaction starts when a method is invoked, and is committed on leaving a method. We will discuss transactions further in Chapter 7.

Now let's look at the `findCustomer()` method. The statement:

```
return ((Customer) em.find(Customer.class, custId));
```

invokes the `EntityManager.find()` method. This method retrieves an entity by its primary key. The parameters of `find()` are the entity class and a primary key, which is an `Object` type, which uniquely identifies the entity. In our example we use the variable `custId` as the primary key (the **autoboxing** feature of Java SE 5 converts the supplied `int` type to an `Object` type).

The `EntityManager` service also provides a query language, **JPQL**, which we will cover in Chapter 5. Furthermore the `EntityManager` provides entity management methods, such as merging detached entities. At this stage the reader might think that entities should always be in an attached (managed) state, however we shall see examples in Chapter 6 where entities do exist in a detached state.

The following listing is an example of how a client might invoke the `BankService` methods to create and then find an entity.

```
package ejb30.client;
import javax.naming.Context;
import javax.naming.InitialContext;
import ejb30.session.*;
import ejb30.entity.Customer;
import javax.ejb.EJB;
public class BankClient {
    @EJB
    private static BankService bank;
    public static void main(String[] args) {
        try {
            int custId = 0;
            String firstName = null;
            String lastName = null;

            try {
                custId = Integer.parseInt(args[0]);
                firstName = args[1];
                lastName = args[2];
            } catch (Exception e) {
                System.err.println(
                 "Invalid arguments entered, try again");
                System.exit(0);
            }
            // add customer to database
            bank.addCustomer(custId, firstName, lastName);
            Customer cust = bank.findCustomer(custId);
            System.out.println(cust);
        } catch (Throwable ex) {
            ex.printStackTrace();
        }
    }
}
```

The code is fairly straightforward. The variables `custId`, `firstName`, and `lastName` are initialized to parameters supplied to the main method when it is invoked. The client uses the `BankService.addCustomer()` method to create and add a Customer entity to the database. The `BankService.findCustomer()` method is then invoked to retrieve the entity just created.

We now return to the persistence unit which we referred to in the `@PersistenceContext` annotation that we used in the `BankService` session bean. The actual configuration is specified in an XML file, `persistence.xml`, which is shown below:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
version="1.0">
<persistence-unit name="BankService"/>
</persistence>
```

Again we make heavy use of defaults here. A `persistence.xml` file is required whenever an application makes use of `EntityManager` services, even if we rely entirely on defaults. In our case we have just used the `name` element to define the `BankService` persistence unit. Note the name must match the `unitName` used earlier in the `@PersistenceContext` annotation. There are a number of persistence unit elements, in particular regarding transactions and data sources. We will return to these elements later in the book.

Packaging and Deploying Entities

As we have introduced a number of new source code artifacts that were not present in the previous chapter on session beans, the packaging process will be a little different. First we look at the program directory structure for our sample application.

The Program Directory Structure

Below is the program directory structure for the `BankClient` application which invokes the `BankServiceBean` in order to add the `Customer` entity to the database.
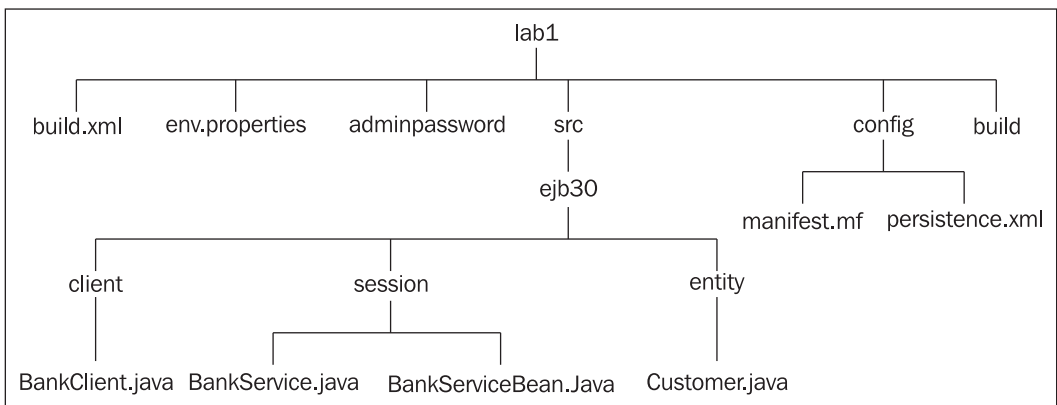
There are a couple of differences in the directory structure from the one described in the previous chapter. We have added an `entity` subdirectory; this contains the entity source code, `Customer.java`. We have also added a `config` subdirectory containing the `manifest.mf` and `persistence.xml` files.

Building the Application

Because we are persisting entities from an EJB container we need to place the `persistence.xml` file in the `META-INF` directory within the EJB module, `BankService.jar`. The `package-ejb` target in the Ant build file does this:

```
<target name="package-ejb" depends="compile">
    <jar jarfile="${build.dir}/BankService.jar">
        <fileset dir="${build.dir}">
            <include name="ejb30/session/**" />
          <include name="ejb30/entity/**" />
        </fileset>
        <metainf dir="${config.dir}">
            <include name="persistence.xml" />
         </metainf>
    </jar>
</target>
```

We can use the `jar -tvf` command to examine the contents of the JAR file:

```
C:\EJB3Chapter03\glassfish\lab1\build>jar  -tvf  BankService.jar
   ...META-INF/
   ...META-INF/MANIFEST.MF
   ...ejb30/
   ...ejb30/entity/
   ...ejb30/session/
   ...ejb30/entity/Customer.class
   ...ejb30/session/BankService.class
   ...ejb30/session/BankServiceBean.class
   ...META-INF/persistence.xml
```

This is only one of a number of ways to package the persistence unit. Entities can be persisted not only from an EJB container but also from a web container or from a Java SE application running outside the container. We shall see an example of Java SE persistence in Chapter 6. Consequently the persistence unit (entity classes together with the `persistence.xml` file) can be placed in `WEB-INF/classes` directory of a WAR file or in a Java SE application client jar file.

Alternatively we can package a persistence unit into a separate jar file. This jar file can then be added to the `WEB-INF/lib` directory of a WAR file, the root of an EAR file or in the library directory of an EAR file. Adding a persistence unit to a Java EE module limits its scope to that module. If we place the persistence unit in an EAR it is visible to all modules within the application.

In our example we want the database tables to be created at deploy time, and dropped when we undeploy an application from the container. We modify the `deploy` target in our Ant build file, adding a `createtables=true` clause:

```
<target name="deploy">
  <exec executable="${glassfish.home}/bin/asadmin"
        failonerror="true"
        vmlauncher="false">
    <arg line="deploy --user admin --passwordfile
        adminpassword --createtables=true
        ${build.dir}/BankService.ear"/>
  </exec>
</target>
```

Similarly we add a `droptables=true` clause for the `undeploy` target.

```
<target name="undeploy">
  <exec executable="${glassfish.home}/bin/asadmin"
        failonerror="true"
        vmlauncher="false">
    <arg line="undeploy --user admin  --passwordfile
        adminpassword  --droptables=true  BankService"/>
  </exec>
</target>
```

Note that before we can deploy `BankService` within GlassFish, we need to start up the embedded Derby database. We do this from the command-line with the `asadmin` utility:

```
C:\> asadmin start-database
```

Field-Based Annotations

Recall we used a property-based annotation for the primary key @Id in the Customer entity. This results in the persistence engine using getter and setter methods to access and set the entity state. In this section we will modify the Customer entity to use field-based, rather than property-based annotations. The following listing demonstrates this:

```
@Entity
public class Customer implements java.io.Serializable {
    @Id
    private int id;
    private String firstName;
    @Basic
    private String lastName;
    public Customer() {};
    public Customer(int id, String firstName,
                    String lastName){
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
    }
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getFirstname() { return firstName; }
    public void setFirstname(String firstName) {
        this.firstName = firstName;
    }
    public String toString() {
        return "[Customer Id =" + id + ",first name=" +
                firstName + ",last name=" + lastName + "]";
    }
}
```

Note that we have placed the primary key @Id annotation immediately before the field declaration. If we use a field-based annotation then all other annotations, other than class level annotations such as @Entity, must also be field-based annotations. In the case of field-based annotations the persistence engine will use reflection to access and set the entity state. Getter and setter methods may be present in the entity but they are ignored by the persistence engine. To demonstrate this point we have removed the getLastName() and setLastName() methods from the Customer entity.

We have seen that attributes other than the primary key are mapped to a database by default. We call these mappings **basic mappings**. A basic mapping is used where the attribute is a Java primitive type or any Serializable type such as a String. We can explicitly flag these mappings using the @Basic annotation. We have done this for the lastName field.

Generating Primary Keys

Up to now we have relied on the application to set an entity's primary key by supplying a primary key identifier as a parameter. However, we may want to relieve the application of this responsibility and use automatically generated keys. JPA provides the **Table**, **Sequence**, **Auto**, and **Identity** strategies for generating primary keys.

Table Strategy

With this strategy the persistence engine uses a relational database table from which the keys are generated. This strategy has the advantage of portability; we can use it with any relational database. Here is an example of how we would specify a table generation strategy for the id attribute of the Customer entity:

```
@TableGenerator(name="CUST_SEQ",
                table="SEQUENCE_TABLE",
                pkColumnName="SEQUENCE_NAME",
                valueColumnName="SEQUENCE_COUNT")
@Id
@GeneratedValue(strategy=GenerationType.TABLE,
                generator="CUST_SEQ")
public int getId() { return id; }
```

First we use the `@TableGenerator` annotation to specify the table used for key generation. This annotation can be placed on the primary key attribute, as we have done here, or on the entity class.

The `name` element in our example `CUST_SEQ`, identifies the generator. The `table` element is the name of the table that stores the generated values. In our case we have chosen `SEQUENCE_TABLE` as the table name. There is a default table element, but this is dependent on the persistence engine being used. In the case of GlassFish, this table name is `SEQUENCE`. The `pkColumnName` element is the name of the primary key column in the sequence table. We have chosen `SEQUENCE_NAME` as the column name. Again the default is persistence engine dependent. In the case of GlassFish this value is `SEQ_NAME`. The `valueColumnName` element is the name of the column that stores the last key value generated. The default is persistence engine dependent. In the case of GlassFish this value is `SEQ_COUNT`.

One element of `@TableGenerator` for which we assumed the default is `pkColumnValue`. This is the `String` value that is entered in the `pkColumnName` column. The default is persistence engine dependent. In the case of GlassFish, this value is set to the name element of `@TableGenerator`. In our example this is `CUST_SEQ`. So `SEQUENCE_TABLE` will initially have the following row present:

```
SQL> SELECT * FROM SEQUENCE_TABLE;

    SEQUENCE_NAME    SEQUENCE_COUNT
    -------------    --------------
    CUST_SEQ                      0
```

If we want to store several sequences in the same sequence table, for example a separate sequence for each entity, then we need to manually specify the `pkColumnValue` element. The following statement sets the `pkColumnValue` to `CUSTOMER_SEQ`:

```
@TableGenerator(name="CUST_SEQ",
                table="SEQUENCE_TABLE",
                pkColumnName="SEQUENCE_NAME",
                valueColumnName="SEQUENCE_COUNT",
                pkColumnValue="CUSTOMER_SEQ")
```

We should mention two other `@TableGenerator` elements: `initialValue` and `allocationSize`. `initialValue` is the initial value assigned to the primary key sequence. The default value is 0. The sequence is incremented by a value of 1. The `allocationSize` is the cache size into which the persistence engine reads from the sequence table. The default value is 50.

The `@GeneratedValue` annotation specifies the key generation strategy with the `strategy` element. In our case we have chosen the `TABLE` strategy. The default strategy is `AUTO` which we describe later in this chapter. The `generator` element provides the name of the primary key generator. In our case this is `CUST_SEQ` and must match the name element of the `@TableGenerator` annotation. The `@GeneratedValue` annotation is a field-based or property-based annotation, so must be present immediately before the primary key field or getter method.

Sequence Strategy

Some databases, such as Oracle, have a built-in mechanism called sequences for generating keys. To invoke such a sequence we need to use the `@SequenceGenerator` annotation. For example:

```
@SequenceGenerator(name="CUST_SEQ",
                   sequenceName="CUSTOMER_SEQUENCE")
```

As with the `@TableGenerator` annotation, the `name` element identifies the generator. The `sequenceName` element identifies the database sequence object. `initialValue` is the initial value assigned to the primary key sequence. The default differs from the `@TableGenerator` equivalent, and is equal to 1. The `allocationSize` is the cache size into which the persistence engine reads from the sequence. The default value is 50.

The `@SequenceGenerator` annotation can be placed on the primary key attribute or on the entity class.

As with Table generated sequences, we use the `@GeneratedValue` annotation to specify the generation strategy. For example,

```
@Id
@GeneratedValue(strategy=GenerationType.SEQUENCE,
                generator="CUST_SEQ")
```

This time we have specified the `SEQUENCE` strategy. The name of the primary key generator is `CUST_SEQ`, and this must match the name element of the `@SequenceGenerator` annotation. Remember the `@GeneratedValue` annotation is a field-based or property-based annotation, so it must be present immediately before the primary key field or getter method.

Identity Strategy

Some databases, such as Microsoft SQL Server, use an identity column for generating keys. To use this we specify the `IDENTITY` strategy in the `@GeneratedValue` annotation:

```
@GeneratedValue(strategy=GenerationType.IDENTITY)
```

Note that there is no `generator` element that we have for the Table and Sequence strategies.

Auto Strategy

The final strategy is the `AUTO` strategy. With this strategy the persistence engine selects the strategy. In the case of GlassFish the `TABLE` strategy is selected. We can specify an `AUTO` strategy either explicitly:

```
@GeneratedValue(strategy=GenerationType.AUTO)
```

or implicitly:

```
@GeneratedValue
```

as the default strategy is AUTO.

In the case of GlassFish the default sequence table has the name SEQUENCE, with columns SEQ_NAME and SEQ_COUNT:

SEQUENCE
SEQ_NAME VARCHAR(50) PRIMARY KEY
SEQ_COUNT DECIMAL

Overriding Metadata Defaults

In this section we return to the Customer entity and override some of the default mapping options.

First we want to use the CLIENT table with columns CUSTOMER_ID, FIRST_NAME and LAST_NAME to store Customer entities. The primary key is CUSTOMER_ID. The table definition is:

CLIENT
CUSTOMER_ID NUMBER(38) PRIMARY KEY
FIRST_NAME VARCHAR(30) NOT NULL
LAST_NAME VARCHAR(30) NOT NULL

The modified listing for Customer.java is shown below:

```
@Entity
@Table(name = "CLIENT")
public class Customer implements java.io.Serializable {
    private int id;
    private String firstName;
    private String lastName;
    public Customer() {};
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name = "CUSTOMER_ID")
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    @Column(name = "FIRST_NAME")
    public String getFirstname() { return firstName; }
    public void setFirstname(String firstName) {
        this.firstName = firstName;
    }
```

```
    @Column(name = "LAST_NAME")
    public String getLastname() { return lastName; }
    public void setLastname(String lastName) {
        this.lastName = lastName;
    }
    public String toString() {
        return "[Customer Id =" + id + ",first name=" +
                firstName + ",last name=" + lastName + "]";
    }
}
```

Note we have used the `@Table` annotation to specify the table name as `CLIENT`. The default table name is the entity name, `CUSTOMER` in our example. The `catalog` element specifies the database catalog where the table is located. Many database systems, Derby included, do not support the concept of a catalog. So in our example we can leave this element out. The `schema` element specifies the database schema where the table is located. The default is persistence provider specific, in the case of the GlassFish APP is the default schema for the embedded Derby database. Again, in our example, we rely on the default.

We have explicitly specified an AUTO primary key generation strategy:

```
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
```

Recall that GlassFish selects the TABLE strategy in this case.

Recall that by default column names are set to attribute names. To override this we use the `@Column` annotation. For example,

```
    @Column(name = "CUSTOMER_ID")
    public int getId() { return id; }
```

will map the `id` attribute to the `CUSTOMER_ID` column. In a similar fashion we map the `firstName` and `lastName` attributes to the `FIRST_NAME` and `LAST_NAME` columns respectively.

Summary

Entities are classes that need to be persisted, usually in a relational database. The persistence aspects of EJB 3 have been packaged as a separate specification, the Java Persistence API (JPA), so that applications that do not need EJB container services can still persist their entities to a relational database. Persistence services are handled by a persistence engine. In this chapter we make use of the Toplink persistence engine that comes bundled with the GlassFish application server.

Any Java class, or POJO, can be converted to an entity using metadata annotations. We described by means of an example the default rules for mapping an entity to a relational database table.

We introduced the `EntityManager` service, which provides methods for persisting, finding, querying, removing and updating entities. We saw examples of the `EntityManager.persist()` and `EntityManager.find()` methods. We introduced the concept of a persistence context, which is the set of managed entity instances.

We looked at Ant scripts for packaging and deploying an application which uses entities.

We examined strategies for generating primary keys. Finally we looked at examples of overriding default rules for mapping entities to relational tables.

4

Object/Relational Mapping

Entities do not exist in isolation and are usually associated with other entities. In this chapter we examine how to map these associations onto a relational database. We will cover the following topics:

- One-to-one, one-to-many, and many-to-many associations.
- Default object/relational mapping values.
- Overriding the default object/relational mapping values.
- The `@Embedded`, `@Embeddable`, `@Enumerated`, and `@MapKey` annotations.
- Composite primary keys
- Entity inheritance mapping strategies

O/R Mapping Default Behavior

In the previous chapter we saw how to persist individual entities to a relational database. In this chapter, we take a further look at object-relational mapping with EJB 3. In any real world application, entities do not exist in isolation but have relationships with other entities. In this chapter we shall see how these relationships are mapped onto a relational database.

Object-relational mapping is specified using annotations, but XML mapping files can be used as an alternative. Annotations make extensive use of defaulting. This means that not only do we not have to specify any annotation options if we are satisfied with the defaults, but in many cases we do not have to provide an annotation itself. In this case a default annotation is used.

This defaulting capability is particularly helpful in cases where we start from a Java object model and derive our database schema from this model. However, if we are mapping our Java model to an existing database schema we will need to explicitly specify any mapping options. This is also necessary if our persistence provider does not provide an automatic database schema generation facility. Even if we have automatic schema generation and we are using a Java object model to determine the database schema, we would need to override defaults as part of fine-tuning our application before deploying it in production. For these reasons we will cover many of the object-relational mapping options later in this chapter rather than just describing the defaults.

To begin with, however, we will make full use of defaults to demonstrate the non-intrusiveness of EJB 3.

A Banking Example Application

We will use a simple banking example throughout this chapter. The following diagram shows an initial object model, which consists of Customer, Account, Address, and Referee entities. We will modify this model repeatedly throughout this chapter to illustrate EJB 3 Object-Relational mapping capabilities.
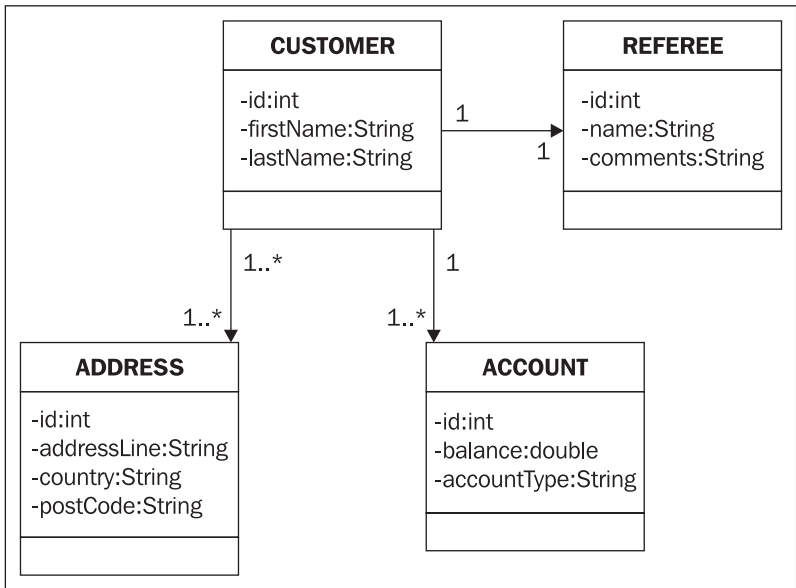
Note that, in our model, a customer can have many accounts (checking or savings, for example) and can have multiple addresses. Also more than one customer can share the same address. This is encapsulated by a many-to-many relationship between customer and address. We have also assumed a one-to-one relationship between customer and referee. A customer has to provide one referee before they can open an account. Note that the model is unidirectional. A Customer object can reference an Account object for example, but an Account object cannot reference a Customer object.

Customer Entity

The following is the code for the Customer class, `Customer.java`:

```java
package ejb30.entity;
import java.io.Serializable;
import java.util.Collection;
import javax.persistence.*;

@Entity
public class Customer implements Serializable {
    private int id;
    private String firstName;
    private String lastName;
    private Referee referee;
    private Collection<Address> addresses;
    private Collection<Account> accounts;

    public Customer() {}

    @Id
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    public String getFirstName() { return firstName; }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() { return lastName; }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    @OneToOne
    public Referee getReferee() { return referee; }
    public void setReferee(Referee referee) {
        this.referee = referee;
    }
```

```
    @OneToMany
    public Collection<Account> getAccounts() {
        return accounts;
    }
    public void setAccounts(Collection<Account> accounts) {
        this.accounts = accounts;
    }
    @ManyToMany
    public Collection<Address> getAddresses() {
        return addresses;
    }
    public void setAddresses(Collection<Address> addresses) {
        this.addresses = addresses;
    }
    public String toString() {
        return "[Customer Id =" + id + ",first name=" +
                firstName + ",last name=" + lastName +
                ", referee=" + referee + ",addresses=" +
                addresses  + ",accounts=" + accounts + "]";
    }
}
```

The `@Entity` annotation, as we have seen, indicates that this entity is to be mapped onto a database by the `EntityManager`. We have not used a `@Table` annotation. The default table name that is generated is the entity name in upper case, namely CUSTOMER. The `@Id` annotation indicates that the following field is the primary key. We have not used the `@Column` annotation with the `id` field, so the default is the field name in upper case, namely ID. Recall from Chapter 3 that Java primitive types are mapped onto relational database column types according to JDBC to SQL mapping rules. These resulting SQL types will depend on the database. In the case of the Derby database embedded with GlassFish, fields of type `int` are mapped to a database type of INTEGER. A primary key cannot have NULL values so the `id` field is mapped to:

```
ID INTEGER NOT NULL, PRIMARY KEY (ID)
```

The `firstName` property is a `String` type for which the default Derby mapping is VARCHAR(255).

The `@OneToOne` annotation indicates there is a one-to-one mapping between the Customer and Referee entities. In this case the CUSTOMER table will contain a foreign key to the REFEREE table. The default name for the foreign key column is a concatenation of the following 3 items:

- the name of the relationship field of the Customer entity in uppercase, namely REFEREE.

- an underscore, "_".

- the name of the primary key column of the Referee entity in uppercase, namely ID.

In this example the resulting foreign key column name is REFEREE_ID. The type of REFEREE_ID is the same as the primary key column of the referenced REFEREE table, namely INTEGER. If we want to override these defaults we need to use the @JoinColumn annotation, which we will cover later in this chapter. To summarize, the default CUSTOMER table will be:

CUSTOMER
ID INTEGER PRIMARY KEY
FIRSTNAME VARCHAR(255)
LASTNAME VARCHAR(255)
REFEREE_ID INTEGER FOREIGN KEY

The @OneToMany annotation indicates that there is a one-to-many relationship between the Customer entity, the **owner entity**, and the Account entity, the **inverse** or **owned entity**. The default behavior is to map onto a join table named CUSTOMER_ACCOUNT (owner table name + underscore + inverse table name). The join table consists of two foreign key columns, Customer_ID and accounts_ID. The first foreign key is the concatenation of the following:

- the name of the owning entity, in this case Customer.

- an underscore, "_".

- the name of the primary key column in the *owning* CUSTOMER table, namely ID.

The type of this foreign key column is the same as the type of the CUSTOMER table primary key column, namely INTEGER. The second foreign key column is the concatenation of the following:

- the relationship field of the owning Customer entity, namely accounts.

- an underscore, "_".

- the name of the primary key column of the *inverse* ACCOUNT table, namely ID.

The type of this foreign key column is the same as the type of ACCOUNT table primary key column, namely INTEGER. To summarize the resulting default join table will be:

CUSTOMER_ACCOUNT
Customer_ID INTEGER PRIMARY KEY FOREIGN KEY
accounts_ID INTEGER PRIMARY KEY FOREIGN KEY

Later in this chapter we will show how to override these defaults with the @JoinTable annotation.

The @ManyToMany annotation indicates there is a many-to-many relationship between the Customer and the Address entity. Recall this is a unidirectional relationship where Customer references Address, but Address does not reference Customer. In such cases Customer is regarded as the owning entity and Address the owned or inverse entity. The default behavior is to map onto a join table named CUSTOMER_ADDRESS (owner table name + underscore + inverse table name). The join table consists of two foreign key columns, Customer_ID and addresses_ID. The procedure for naming these columns is the same as the unidirectional one-to-many case described above. The resulting default join table will be:

CUSTOMER_ADDRESS
Customer_ID INTEGER PRIMARY KEY FOREIGN KEY
addresses_ID INTEGER PRIMARY KEY FOREIGN KEY

Finally the Customer class contains an overridden toString() method which is used when printing out the values of the Customer object from a client application.

Because our model is unidirectional and the only references between entities are those from Customer, no relationship annotations are required for the remaining classes.

Account Entity

The code for the Account class, Account.java is listed below:

```
package ejb30.entity;
import java.io.Serializable;
import javax.persistence.*;

@Entity
public class Account implements Serializable {
    private int id;
    private double balance;
    private String accountType;

    public Account() {}

    @Id
```

```
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public double getBalance() { return balance; }
    public void setBalance(double balance) {
        this.balance = balance;
    }
    public String getAccountType() { return accountType; }
    public void setAccountType(String accountType) {
        this.accountType = accountType;
    }
    public String toString() {
        return "[account id =" + id + ",balance=" +
                balance + ",account type=" + accountType + "]";
    }
}
```

The only item to note is the default mapping of the `balance` field. The database type to which a field is mapped depends on the underlying database. In the case of the GlassFish embedded Derby database, a field of type `double` is mapped to a `FLOAT` database type. The default table that the Account entity is mapped to is:

ACCOUNT
ID INTEGER PRIMARY KEY
ACCOUNTTYPE VARCHAR(255)
BALANCE FLOAT

Address Entity

The code for Address class, `Address.java` is listed below:

```
package ejb30.entity;
import java.io.Serializable;
import javax.persistence.*;
@Entity
public class Address implements Serializable {
    private int id;
    private String addressLine;
    private String country;
    private String postCode;
    public Address() {}
    @Id
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
```

```
    public String getAddressLine() { return addressLine; }
    public void setAddressLine(String addressLine) {
        this.addressLine = addressLine;
    }
    public String getCountry() { return country; }
    public void setCountry(String country) {
        this.country = country;
    }
    public String getPostCode() { return postCode; }
    public void setPostCode(String postCode) {
        this.postCode = postCode;
    }
    public String toString() {
        return "[address id=" + id + ",address=" +
                addressLine + ",country=" +country +
                ",post code=" + postCode +"]";
    }
}
```

The default mapping rules that we have already described apply here, and the resulting default mapped table is:

ADDRESS
ID INTEGER PRIMARY KEY
POSTCODE VARCHAR(255)
ADRESSLINE VARCHAR(255)
COUNTRY VARCHAR(255)

Referee Entity

The code for the Referee class, `Referee.java` is listed below:

```
package ejb30.entity;
import java.io.Serializable;
import javax.persistence.*;

@Entity
public class Referee implements Serializable {
    private int id;
    private String name;
    private String comments;

    public Referee() {}

    @Id
```

```
        public int getId() { return id; }
        public void setId(int id) { this.id = id; }
        public String getName() { return name; }
        public void setName(String name) { this.name = name; }
        public String getComments() { return comments; }
        public void setComments(String comments) {
            this.comments = comments;
        }
        public String toString() {
            return "[referee id=" + id + ",name=" + name +
                    ",comments=" + comments + "]";
        }
    }
```

No new mapping rules apply here, and the default mapped table is:

REFEREE
ID INTEGER PRIMARY KEY
NAME VARCHAR(255)
COMMENTS VARCHAR(255)

Testing the Application

To test our application we create a session bean interface, `BankService`, with methods `createCustomers()` and `findCustomer()` as follows:

```
@Remote
public interface BankService {
    void createCustomers();
    Customer findCustomer(int custId);
}
```

The `createCustomers()` method populates and then persists the Customer, Address, Account, and Referee entities. The `findCustomer()` method, which is invoked after `createCustomers()`, retrieves a Customer entity for a given customer identifier. Now let's take a look at some of the code for the interface implementation, the stateless session bean `BankServiceBean`:

```
@Stateless
public class BankServiceBean implements BankService {
    private EntityManager em;
    ...
    public void createCustomers() {
```

```
            Referee r1 = new Referee();
            r1.setId(1);
            r1.setName("SIR JOHN DEED");
            r1.setComments("JUDGE");
            em.persist(r1);
            Customer c1 = new Customer();
            c1.setId(1);
            c1.setFirstName("SIMON");
            c1.setLastName("KING");
            c1.setReferee(r1);
            Account a1 = new Account();
            a1.setId(1);
            a1.setBalance(430.5);
            a1.setAccountType("C");
            ArrayList<Account> accounts1 =
                new ArrayList<Account>();
            accounts1.add(a1);
            c1.setAccounts(accounts1);
            em.persist(a1);
            Address add1 = new Address();
            add1.setId(1);
            add1.setAddressLine("49, KINGS ROAD MANCHESTER");
            add1.setCountry("UK");
            add1.setPostCode("MN1 2AB");
            ArrayList<Address> addresses1 =
                new ArrayList<Address>();
            addresses1.add(add1);
            c1.setAddresses(addresses1);
            em.persist(add1);
            em.persist(c1);
            ...
```

In the above code we use the `Referee` and `Customer` methods to populate their respective entities. We can use the `EntityManager.persist()` method to persist Referee as soon as its fields are populated. After populating the Customer entity we do not persist it as we have not yet populated the referenced Account and Address entities. Next, we create a new Account instance `a1` and populate its fields. We then create an `ArrayList` of type `Account` and add `a1` to this list as follows:

```
ArrayList<Account> accounts1 =
    new ArrayList<Account>();
accounts1.add(a1);
```

We can now invoke the Customer `setAccounts()` method:

```
c1.setAccounts(accounts1);
```

In this case customer `c1` has only one account, `a1`. If a customer has more than one account then each account would be added to the above `ArrayList`.

We handle addresses in a similar manner to accounts. Then we can persist our customer object.

The `findCustomer()` implementation simply uses the entity manager `find()` method that we have seen in the previous chapter:

```
public Customer findCustomer(int custId) {
    return ((Customer) em.find(Customer.class, custId));
}
```

The following code fragment shows how we might invoke the above methods from a client, `BankClient`:

```
custId = Integer.parseInt(args[0]);
bank.createCustomers();
Customer cust = bank.findCustomer(custId);
System.out.println(cust);
```

We need to modify the `build.xml` file to include passing of an argument to `BankClient`. For example:

```
<target name="run-client">
    <exec executable="${glassfish.home}/bin/appclient"
          failonerror="true"
          vmlauncher="false">
      <arg line="-client
          ${glassfish.home}/domains/domain1/generated/xml/
          j2ee-apps/BankService/BankServiceClient.jar
          -mainclass ejb30.client.BankClient 4"/>
    </exec>
</target>
```

If we run the client we get the following output:

```
run-client:
    [java] [Customer Id =4,first name=EDWARD,last name=COOK,
           referee=[referee id=4,name=RICHARD BRANSON,
                    comments=HE SHOULD BANK WITH US],
           addresses={IndirectList: not instantiated},
           accounts={IndirectList: not instantiated}]
```

Note that, null values are printed out for addresses and accounts. This is because the default is to lazily load associated entities in a one-to-many and many-to-many relationship. By this we mean that when we load a Customer object from the database we do not automatically load associated Account and Address objects. We will discuss this in more detail later in this chapter.

O/R Mapping Overriding Defaults

In this section we modify our object model and make it bidirectional. We will also take the opportunity to override some mapping defaults. Shown below is a UML diagram for the bidirectional model.
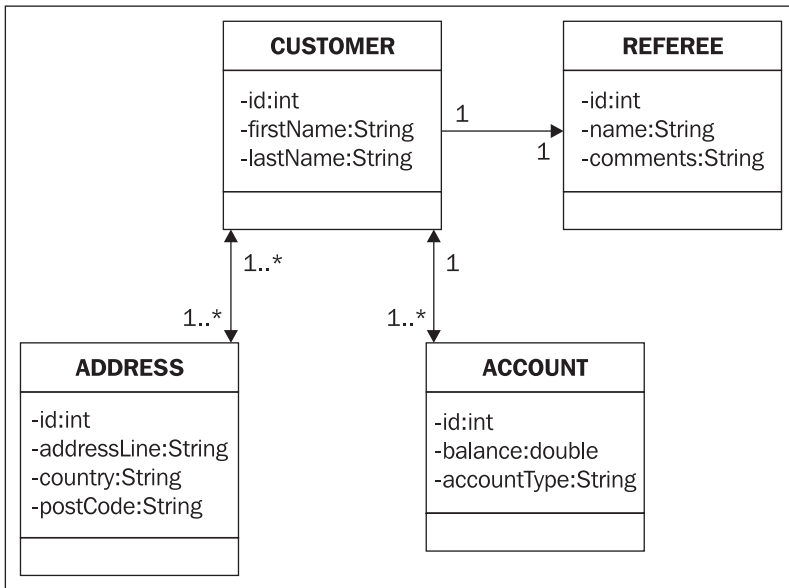
A Customer entity can now be referenced by Account and Address entities. However, we still leave the Customer and Referee relationship unidirectional; we have no need in our application to reference a Customer entity from a Referee entity.

Customer Entity

First we modify the Customer entity; the listing is shown below with modifications highlighted. We will discuss these modifications in detail.

```
@Entity
public class Customer implements Serializable {
    private int id;
    private String firstName;
    private String lastName;
    private Referee referee;
    private Collection<Address> addresses;
    private Collection<Account> accounts;
    public Customer() {}
    @Id
    @Column(name="CUSTOMER_ID")
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    @Column(name="FIRST_NAME", length=30)
    public String getFirstName() { return firstName; }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    @Column(name="LAST_NAME", length=30)
    public String getLastName() { return lastName; }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    @OneToOne
    public Referee getReferee() { return referee; }
    public void setReferee(Referee referee) {
        this.referee = referee;
    }
    @OneToMany(mappedBy="customer", fetch=EAGER)
    public Collection<Account> getAccounts() {
        return accounts;
    }
    public void setAccounts(Collection<Account> accounts) {
        this.accounts = accounts;
    }
    @ManyToMany(fetch=EAGER)
    @JoinTable(
        name="CUSTOMER_ADDRESS",
        joinColumns=@JoinColumn(
                    name="CUST_ID",
                    referencedColumnName="CUSTOMER_ID"),
        inverseJoinColumns=@JoinColumn(
                    name="ADD_ID",
                    referencedColumnName="ADDRESS_ID")
```

```
        )
        public Collection<Address> getAddresses() {
            return addresses;
        }
        public void setAddresses(Collection<Address> addresses) {
            this.addresses = addresses;
        }
        public String toString() {
            return "[Customer Id =" + id + ",first name=" +
                    firstName + ",last name=" + lastName + ",
                    referee=" + referee + ",addresses=" +
                    addresses + ",accounts=" + accounts + "]";
        }
    }
```

To override the default column name, use the `name` element of the `@Column` annotation. The following code fragment specifies the column name for the `id` field to be `CUSTOMER_ID`:

```
@Column(name="CUSTOMER_ID")
public int getId() { return id; }
```

We will name primary key columns for all our entities according to the format ENTITY + underscore + ID. Recall the Derby database default column length for `String` fields is 255. To override this use the `length` element of the `@Column` annotation. The following renames the column name for the `firstName` field and gives a length of 30:

```
@Column(name="FIRST_NAME", length=30)
public String getFirstName() { return firstName; }
```

Next we examine the one-to-many relationship between Customer and Account. The relationship is now bidirectional, the many side of a bidirectional one-to-many relationship is always the owning side. So Account is the owning side. In the inverse side of a bidirectional, one-to-many relationship we need to specify the owning field using the `mappedBy` element. The `customer` field in the owner Account entity references the inverse Customer entity relationship. The resulting `@OneToMany` annotation will be:

```
@OneToMany(mappedBy="customer")
public Collection<Account> getAccounts() {
    return accounts;
}
```

Recall that when we printed out the Customer entity in the previous section, the associated Account values were null. This is because the default is to **lazily load** associated entities in a one-to-many relationship. By this we mean that when we load a Customer object from the database we do not automatically load associated Account objects. This has obvious performance benefits at the loading stage. If the application then requires an associated Account object then this is retrieved when the application makes an explicit reference to that object. This is reasonable if, in our application, a Customer makes only occasional reference to an Account object. If, in our application, the Customer object frequently references an Account, then it makes sense to override the default and initialize associated Account objects at load time. This is called **eager loading**; we use the `fetch` element in the `@OneToMany` annotation for this:

```
@OneToMany(mappedBy="customer", fetch=EAGER)
public Collection<Account> getAccounts() {
    return accounts;
}
```

In some circumstances the default is for eager loading to take place, for example with one-to-one mappings or with basic mappings. We can override this default. For example, if we had a large column which is infrequently referenced:

```
@Basic(fetch=LAZY)
```

However, a lazy loading override is taken as a hint to the persistence provider. The persistence provider may eagerly load the field in any case if it calculates negligible performance overhead. Typically an entire database row will occupy contiguous blocks on a disk, so there is no gain in retrieving just some of the columns of the database row. The reverse is not true. The persistence provider will always perform an eager fetch if this is explicitly specified.

We will now take a look at the `@ManyToMany` annotation options. Recall that for many-to-many relationships it is arbitrary which is the owning side; we have chosen Customer to be the owning side and Address the inverse side. The default is to map to a join table. To override the default, or just to make mappings explicit, we use the `@JoinTable` annotation as follows:

```
@ManyToMany(fetch=EAGER)
    @JoinTable(
        name="CUSTOMER_ADDRESS",
        joinColumns=@JoinColumn(
                    name="CUST_ID",
                    referencedColumnName="CUSTOMER_ID"),
        inverseJoinColumns=@JoinColumn(
                    name="ADD_ID",
```

```
                    referencedColumnName="ADDRESS_ID")
          )
public Collection<Address> getAddresses() {
    return addresses;
}
```

The `name` element specifies the join table name; in our example `CUSTOMER_ADDRESS` happens to be the same as the default. The `joinColumns` element specifies the join table foreign key column or columns which references the owning table. As our foreign key consists of a single column, we use an embedded `@JoinColumn` annotation. In the expression,

```
@JoinColumn(
  name="CUST_ID",
  referencedColumnName="CUSTOMER_ID")
```

the `name` element is the name of the foreign key column in the join table. In our case we choose to name this column `CUST_ID`, rather than accept the default `CUSTOMER_ID`. We would need to do this if we were mapping onto an existing join table with the foreign key column named `CUST_ID`. The `referencedColumnName` is the name of the column referenced by this foreign key. In our example this is the primary key column of the owning, `CUSTOMER`, table which we earlier specified as `CUSTOMER_ID`.

In some cases we may have a composite foreign key, consisting of two or more columns, although it is rare to have more than two columns. In such situations we would use the `@JoinColumns` annotation. Composite foreign keys usually occur when we are mapping our object model onto a legacy database. We shall see an example of this later in this chapter.

We return to the final element in our `@JoinTable` annotation example, namely `inverseJoinColumns`. This specifies the join table foreign key column or columns which reference the inverse table. Again we use an embedded `@JoinColumn` annotation:

```
@JoinColumn(
  name="ADD_ID",
  referencedColumnName="ADDRESS_ID")
```

Here we have overridden the default `ADDRESS_ID`, and named our foreign key column as `ADD_ID`. This references the primary key column, `ADDRESS_ID`, of the inverse `ADDRESS` table. Again `inverseJoinColumns` may refer to composite foreign columns, in which case these will be specified using the `@JoinColumns` annotation.

Note that we have added a `fetch=EAGER` clause to the `@ManyToMany` annotation. For many-to-many relationships lazy loading is the default behavior.

Account Entity

Now let's modify the Account entity so that it can handle the bidirectional relationship with Customer. The listing below has the modifications highlighted:

```java
@Entity
public class Account implements Serializable {
    private int id;
    private double balance;
    private String accountType;
    private Customer customer;
    public Account() {}

    @Id
    @Column(name = "ACCOUNT_ID")
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    public double getBalance() { return balance; }
    public void setBalance(double balance) {
        this.balance = balance;
    }

    @Column(name="ACCOUNT_TYPE", length=2)
    public String getAccountType() { return accountType; }
    public void setAccountType(String accountType) {
        this.accountType = accountType;
    }

    @ManyToOne
    @JoinColumn(name="CUSTOMER_ID",
                referencedColumnName="CUSTOMER_ID")
    public Customer getCustomer() { return customer; }
    public void setCustomer(Customer customer) {
        this.customer = customer;
    }

    public String toString() {
        return "[account id =" + id + ",balance=" +
                balance + ",account type=" + accountType + "]";
    }
}
```

The relationship between the Account and Customer entities is many-to-one so we need to add a @ManyToOne annotation. We also need to define a field customer of type Customer with associated getters and setters. The following code fragment shows this:

```
@ManyToOne
@JoinColumn(name="CUSTOMER_ID",
            referencedColumnName="CUSTOMER_ID")
public Customer getCustomer() { return customer; }
public void setCustomer(Customer customer) {
    this.customer = customer;
}
```

Note that, we have also used the @JoinColumn annotation to explicitly name the foreign key column in the ACCOUNT table and the referenced primary key column in the CUSTOMER table.

Address Entity

Now we modify the Address entity so that it can handle the bidirectional relationship with customer. A code fragment with the modifications is shown below:

```
@Entity
public class Address implements Serializable {
    private int id;
    private String addressLine;
    private String country;
    private String postCode;
    private Collection<Customer> customers;
    //
    @ManyToMany(mappedBy="addresses", fetch=EAGER)
    public Collection<Customer> getCustomers() {
        return customers; }
    public void setCustomers(Collection<Customer> customers) {
        this.customers = customers;
    }
    // other getters and setters
}
```

Note the @ManyToMany annotation. Recall that we have chosen Customer to be the owning side and Address to be the inverse side of the many-to-many relationship. In the inverse side we need to specify the owning field with a mappedBy element. The owning field is the field in the Customer entity, namely addresses, which references the Address entity. We can see this in the following code fragment from the Customer entity:

```
@Entity
public class Customer implements Serializable {
    private int id;
    private String firstName;
    private String lastName;
    private Referee referee;
    private Collection<Address> addresses;
    private Collection<Account> accounts;

    public Customer() {}

    @Id
    @Column(name="CUSTOMER_ID")
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
      ...
    @ManyToMany(fetch=EAGER)
    @JoinTable(
        name="CUSTOMER_ADDRESS",
        joinColumns=@JoinColumn(
                    name="CUST_ID",
                    referencedColumnName="CUSTOMER_ID"),
        inverseJoinColumns=@JoinColumn(
                    name="ADD_ID",
                    referencedColumnName="ADDRESS_ID")
    )
    public Collection<Address> getAddresses() {
        return addresses;
    }
    public void setAddresses(Collection<Address> addresses) {
        this.addresses = addresses;
    }
    ...
}
```

Reverting back to the Address entity, note that we have added a `fetch=EAGER` clause to override the default lazy loading behavior.

We do not need to modify Referee as the relationship remains unidirectional with no reference to Customer from Referee.

BankServiceBean

To test our new bidirectional model we need to modify `BankServiceBean`. First we need to modify the `createCustomer()` method so that bidirectional references from Account to Customer and from Address to Customer are included. A code fragment with the modifications is shown below:

```
@Stateless
public class BankServiceBean implements BankService {
    @PersistenceContext(unitName="BankService")
    private EntityManager em;
    public void createCustomers() {
        Referee r1 = new Referee();
        r1.setId(1);
        r1.setName("SIR JOHN DEED");
        r1.setComments("JUDGE");
        em.persist(r1);
        Customer c1 = new Customer();
        c1.setId(1);
        c1.setFirstName("SIMON");
        c1.setLastName("KING");
        c1.setReferee(r1);
        Account a1 = new Account();
        a1.setId(1);
        a1.setBalance(430.5);
        a1.setAccountType("C");
        a1.setCustomer(c1);
        ArrayList<Account> accounts1 =
              new ArrayList<Account>();
        accounts1.add(a1);
        c1.setAccounts(accounts1);
        em.persist(a1);
        Address add1 = new Address();
        add1.setId(1);
        add1.setAddressLine("49, KINGS ROAD MANCHESTER");
        add1.setCountry("UK");
        add1.setPostCode("MN1 2AB");
        ArrayList<Address> addresses1 =
              new ArrayList<Address>();
        addresses1.add(add1);
        c1.setAddresses(addresses1);
        em.persist(add1);
        em.persist(c1);
```

```
        Referee r2 = new Referee();
        r2.setId(2);
        r2.setName("DR WILLIAM SMITH");
        r2.setComments("MEDICAL PRACTIONER");
        em.persist(r2);
        Customer c2 = new Customer();
        c2.setId(2);
        c2.setFirstName("JANE");
        c2.setLastName("KING");
        c2.setReferee(r2);
        Account a2 = new Account();
        a2.setId(2);
        a2.setBalance(99);
        a2.setAccountType("C");
        a2.setCustomer(c2);
        ArrayList<Account> accounts2 =
            new ArrayList<Account>();
        accounts2.add(a2);
        c2.setAccounts(accounts2);
        em.persist(a2);
        ArrayList<Customer> customers1 =
            new ArrayList<Customer>();
        customers1.add(c1);
        customers1.add(c2);
        add1.setCustomers(customers1);
        em.persist(add1);
        c2.setAddresses(addresses1); // same address as
                                     // John King
        em.persist(c2);
        ...

    }

    public Customer findCustomer(int custId) {
        return ((Customer) em.find(Customer.class, custId));
    }
    public Customer findCustomerForAccount(int accountId) {
        Account account = (Account) em.find(Account.class,
                                            accountId);
        return account.getCustomer();
    }
    public Collection<Customer> findCustomersForAddress(
```

```
                                  int addressId) {
        Address address = (Address) em.find(Address.class,
                                          addressId);
        Collection<Customer> customers =
                                  address.getCustomers();
        return customers;
    }
  }
```

Note for Account object `a1` we add the statement:

```
    a1.setCustomer(c1);
```

This provides the link from Account to Customer; we already have a link from Customer to Account. Similar statements would be added for remaining Account and associated Customer objects.

When creating a link from Address to Customer for our first address object, `add1`, we note that two customers, `c1` and `c2`, share this address. We deal with this by creating a list, `customers1`, containing the two customers and then populating the address object `add1` with `customers1`:

```
    ArrayList<Customer> customers1 =
        new ArrayList<Customer>();
    customers1.add(c1);
    customers1.add(c2);
    add1.setCustomers(customers1);
```

Finally note that we have added two methods to `BankServiceBean`. The `findCustomerForAccount()` method retrieves a Customer object for a supplied `account id`. The `findCustomersForAddress()` method retrieves a Collection of Customer objects for a supplied `address id`.

O/R Mapping Additional Annotations

We will modify our application again and introduce some more object-relational mapping annotations. Specifically we will cover the `@Embedded`, `@Embeddable`, `@Enumerated`, and `@MapKey` annotations.

First we decide to embed the Referee entity within the Customer entity. An embedded Referee entity shares the identity of the owning Customer entity. A Referee entity cannot exist on its own without a corresponding Customer entity. Such a Referee entity cannot be independently persisted, it is persisted by default whenever the owning Customer entity is persisted. However, we still map a Referee object onto a relational database in the usual manner.

We also decide to add an **enumerated type**, gender, as a Customer attribute.

To indicate an entity is embedded EJB 3 provides two annotations: @Embedded and @Embeddable. @Embedded is used within the owning entity and @Embeddable within the embedded entity. Below is the modified version of the Customer entity:

```
@Entity
public class Customer implements Serializable {
    private int id;
    private String firstName;
    private String lastName;
    private Referee referee;
    private Collection<Address> addresses;
    private Map<String, Account> accounts =
        new HashMap<String, Account>();

    public Customer() {}

    @Id
    @Column(name="CUSTOMER_ID")
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    @Column(name="FIRST_NAME", length=30)
    public String getFirstName() { return firstName; }
    public void setFirstName(String firstName) {
        this.firstName = firstName; }

    @Column(name="LAST_NAME", length=30)
    public String getLastName() { return lastName; }
    public void setLastName(String lastName) {
        this.lastName = lastName; }

    @OneToOne
    @Embedded
    public Referee getReferee() { return referee; }
    public void setReferee(Referee referee) {

        this.referee = referee; }

    @OneToMany(mappedBy="customer", fetch=EAGER)
    @MapKey()
    public Map<String, Account> getAccounts() {
        return accounts; }
    public void setAccounts(Map<String, Account> accounts) {
        this.accounts = accounts; }

    @ManyToMany(fetch=EAGER)
    @JoinTable(
        name="CUSTOMER_ADDRESS",
        joinColumns=@JoinColumn(
```

```
                              name="CUST_ID",
                              referencedColumnName="CUSTOMER_ID"),
        inverseJoinColumns=@JoinColumn(
                              name="ADD_ID",
                              referencedColumnName="ADDRESS_ID")
    )
    public Collection<Address> getAddresses() {
        return addresses; }
    public void setAddresses(Collection<Address> addresses) {
        this.addresses = addresses; }
    private Gender gender;
    @Column(name="GENDER", length=20)
    @Enumerated(STRING)
    public Gender getGender() { return gender; }
    public void setGender(Gender gender) {
        this.gender = gender;
    }
    public String toString() {
        return "[Customer Id =" + id + ",first name=" +
                firstName + ",last name=" + lastName + ",
                referee=" + referee + ",addresses=" +
                addresses + ",accounts=" + accounts +
                ",gender=" + gender + "]";
    }
}
```

As you can see in the section above, we have added the `@Embedded` annotation on the `getRefeee()` method to indicate that the Referee entity is embedded.

Note that we have added a new field, `gender`, to the Customer entity. The `gender` field will be persisted as an enumerated type with enumeration constants of MALE and FEMALE. This is implemented in EJB 3 using the `@Enumerated` annotation. This can be a field-based or property-based annotation. In our example we prefix the `getGender()` method with `@Enumerated`:

```
private Gender gender;
@Column(name="GENDER", length=20)
@Enumerated(STRING)
public Gender getGender() { return gender; }
public void setGender(Gender gender) {
    this.gender = gender;
}
```

Note that we have specified STRING in the @Enumerated annotation. This specifies that the field will be mapped as a string. If we specify ORDINAL, which is the default, then the field will be mapped as an integer.

We now need to create the enumeration class itself, Gender.java:

```
package ejb30.entity;
public enum Gender { MALE, FEMALE }
```

Because we specified a value of STRING in the @Enumerated annotation, one of the string values MALE or FEMALE will be stored in the database. If we had specified ORDINAL, then 0 would be stored in the database if the enumeration value is MALE and 1 would be stored if the enumeration value is FEMALE.

We will make one more modification to the Customer entity. We want to store Account objects associated with a Customer as a **Map**. We define an accounts object to be a Map in the usual manner:

```
private Map<String, Account> accounts =
    new HashMap<String, Account>();
```

This defines a Map, accounts, where the key is a String type and the value is an Account object.

We now need to use the @MapKey annotation and modify the associated getter and setter as follows:

```
@OneToMany(mappedBy="customer", fetch=EAGER)
@MapKey()
public Map<String, Account> getAccounts() {return accounts;}
public void setAccounts(Map<String, Account> accounts) {
    this.accounts = accounts;
}
```

The @MapKey defaults to storing the primary key of the associated entity as the map key. In this case the id property of Account is stored. If we want to specify a property other than the primary key we need to use the name attribute of @MapKey. For example, if in our banking example we also had a rollNumber property which is unique for an Account object, we would specify:

```
@MapKey(name=rollNumber)
```

Note that we also modified the return type of getAccounts() and the argument type of setAccounts().

Referee Class

We now turn to the Referee class:

```
//@Entity
@Embeddable
public class Referee implements Serializable {
    private int id;
    private String name;
    private String comments;
    // getters and setters
}
```

We have added the `@Embeddable` annotation to indicate that all Referee fields are persisted only when the Customer entity is persisted. Because Referee cannot be persisted on its own it is no longer an entity so we do not use the `@Entity` annotation. There are no other changes to the Referee class. We cannot use any mapping annotations apart from `@Basic`, `@Column`, `@Lob`, `@Temporal`, and `@Enumerated` within an embeddable class.

BankServiceBean

We need to make a few changes to `BankServiceBean` as shown in the following code fragment:

```
@Stateless
public class BankServiceBean implements BankService {
    @PersistenceContext(unitName="BankService")
    private EntityManager em;

    public void createCustomers() {
        Referee r1 = new Referee();
        r1.setId(1);
        r1.setName("SIR JOHN DEED");
        r1.setComments("JUDGE");
   //   em.persist(r1);

        Customer c1 = new Customer();
        c1.setId(1);
        c1.setFirstName("SIMON");
        c1.setLastName("KING");
        c1.setReferee(r1);
        c1.setGender(Gender.MALE);

        Account a1 = new Account();
        a1.setId(1);
```

```
        a1.setBalance(430.5);
        a1.setAccountType("C");
        a1.setCustomer(c1);
        Map<String, Account> accounts1 =
            new HashMap<String, Account>();
        accounts1.put("1", a1);
        c1.setAccounts(accounts1);
        em.persist(a1);
        ...
    }

    public Map<String, Account> findAccountForCustomer(
                                        int custId) {
        Customer customer = (Customer) em.find(Customer.class,
                                        custId);
        Map<String, Account> accounts =
                            customer.getAccounts();
        return accounts;
    }
    // other finder methods
}
```

First, as Referee is now an embedded entity, statements which persist instances of
the Referee entity need to be deleted:

```
//    em.persist(r1);
```

Next, as each Customer object now has an associated Gender enumerated type, we
need to invoke the setGender() method passing either a MALE or FEMALE argument:

```
c1.setGender(Gender.MALE);
```

Finally, we have added the findAccountForCustomer() method which returns a
Map of accounts for a given customer.

Composite Primary Keys

It is good practice to have a single column as a primary key. It is also good practice
for the primary key value to be obtained from a sequence and not contain any data.
A primary key containing data is very hard to update. A primary key obtained from
a sequence is known as a **surrogate key**. The main advantage of a surrogate key is
that we never need to update the key. However, there may be occasions where we
need to map an entity to a legacy relational database which has a composite primary
key for the corresponding table.

Suppose our Customer entity has a composite primary key of FIRST_NAME, LAST_NAME in the corresponding CUSTOMER table. First we must define a primary key class, CustomerPK.java, as follows:

```java
public class CustomerPK implements Serializable {
    private String firstName = "";
    private String lastName = "";
    public CustomerPK() {}
    public CustomerPK(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    // getter and setter methods.
    // equals and hashcode method.
}
```

A primary key class must be public, serializable, and have a public no-arg constructor. The class must also define equals and hashcode methods.

Now let's look at the Customer entity:

```java
@Entity
@IdClass(CustomerPK.class)
public class Customer implements Serializable {
    ...
    @Id
    @Column(name="FIRST_NAME", length=30)
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    @Id
    @Column(name="LAST_NAME", length=30)
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    ...
}
```

Note that we use the class level `@IdClass` annotation to specify the primary key class associated with the Customer entity. We use the property-based `@Id` annotation for each of the getter methods corresponding to the fields, `firstName` and `lastName`, making up the primary key.

Now look at the Account entity:

```
@Entity
public class Account implements Serializable {
    ...

    @ManyToOne
    @JoinColumns({
        @JoinColumn(name="CUSTOMER_FNAME",
                    referencedColumnName="FIRST_NAME"),
        @JoinColumn(name="CUSTOMER_LNAME",
                    referencedColumnName="LAST_NAME")
    })
    public Customer getCustomer() {
        return customer;
    }
    ...
}
```

When defining the many-to-one relationship between Account and Customer, we need to modify the join column definitions. We now have more than one join column; actually two columns which comprise the composite key. Consequently we use the `@JoinColumns` annotation wrapped around the individual `@JoinColumn` annotations.

O/R Inheritance Mapping

Inheritance is a distinctive aspect of object oriented technology. Inheritance could not be utilized in earlier versions of EJB as an EJB 2.x entity bean cannot extend another EJB 2.x entity bean. With EJB 3, entities are more object oriented in that one entity can extend another. However, this leaves us with the matter of mapping entities in an inheritance hierarchy onto a relational database.

There are a number of strategies we can use to map an inheritance hierarchy onto relational database tables. As an example, suppose the Account class has **CheckingAccount** and **SavingsAccount** subclasses:
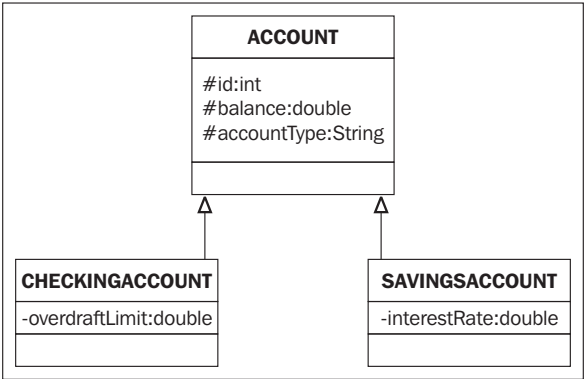
CheckingAccount and SavingsAccount extend the base Account class. As the names suggest, they are entities that represent checking accounts and savings accounts respectively.

SINGLE_TABLE Strategy

The first inheritance mapping option is the **Single Table** strategy. All the classes in a hierarchy are mapped to a single table. The table must have a **discriminator column** whose value for a given row identifies the associated subclass. The ACCOUNT table below results from implementing the Single Table strategy for the above Account inheritance hierarchy. A couple of rows have been added for illustration.

ACCOUNT Table

ACCOUNT_ ID	BALANCE	ACCOUNT_ TYPE	INTERESTRATE	OVERDRAFTLIMIT
1	430.5	C		100.0
5	5200	S	4.5	

ACCOUNT_TYPE is the discriminator column. A value of **C** indicates a checking account, a value of **S** indicates a saving account.

This strategy is efficient for both read and write operations because there is no need to perform joins. It is a rather wasteful use of database storage especially if there are many subclasses each containing a large number of attributes. Columns corresponding to subclass attributes, such as INTERESTRATE and OVERDRAFTLIMIT in our example, must be nullable.

We now turn to the code required to implement this strategy. The Account entity is shown below:

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="ACCOUNT_TYPE")
public abstract class Account implements Serializable {
    protected int id;
    protected double balance;
    protected String accountType;
    // getters and setters
    @Column(name="ACCOUNT_TYPE", length=2)
    public String getAccountType() {
        return accountType;
    }
    ...
}
```

Note the Account entity is an **abstract class**. This is because the Account entity will never be instantiated, only CheckingAccount and SavingsAccount entities will be instantiated. Consequently there is no constructor for the Account entity. The `@Inheritance` annotation indicates that a Single Table mapping strategy is used. The `@DiscriminatorColumn` annotation specifies that the ACCOUNT_TYPE column acts as the discriminator column. Note that we need to use the `@Column` annotation to map the `accountType` field with the ACCOUNT_TYPE column.

Now take a look at the CheckingAccount entity:

```
@Entity
@DiscriminatorValue("C")
public class CheckingAccount extends Account {
    private double overdraftLimit;
    public CheckingAccount() {
    }
    // getter and setter
}
```

Note the use of the `@DiscriminatorValue` annotation to indicate that a checking account has a value of C in the discriminator column. As CheckingAccount is a concrete class, it contains a constructor.

The code for SavingsAccount has a similar structure to CheckingAccount:

```
@Entity
@DiscriminatorValue("S")
public class SavingsAccount extends Account {
    private double interestRate;
    public SavingsAccount() {}
    // getter and setter
}
```

This time the `@DiscriminatorValue` annotation indicates that a savings account has a value of `S` in the discriminator column.

An important feature of the Single Table strategy is that polymorphic queries are supported. By a polymorphic query we mean that the `FROM` clause of the query includes not only instances of the *concrete* entity class to which it refers, but all subclasses of that class as well. If the query refers to an *abstract* entity class then it is polymorphic if it includes just the subclasses of that class. So the query,

```
SELECT a FROM Account a
```

will retrieve both CheckingAccounts and SavingsAccounts. We will cover queries in some detail in Chapter 5.

JOINED Strategy

In this strategy the root of the class hierarchy is represented by a single table. Each subclass is represented by a separate table which contains columns corresponding to fields which are specific to the subclass. The subclass table also contains a column that represents the primary key. The primary key column also acts as a foreign key to the root primary key. A Joined strategy for the Account inheritance hierarchy results in the following tables:

ACCOUNT Table

ACCOUNT_ID	BALANCE	ACCOUNT_TYPE
1	430.5	C
5	5200	S

CHECKINGACCOUNT Table

ACCOUNT_ID	OVERDRAFTLIMIT
1	100.0

SAVINGSACCOUNT Table

ACCOUNT_ID	INTERESTRATE
5	4.5

Note in both the CHECKINGACCOUNT and SAVINGSACCOUNT tables, ACCOUNT_ID is a foreign key referencing ACCOUNT(ACCOUNT_ID).

This strategy is efficient with respect to data storage, however as queries require joins, they are inefficient especially when inheritance hierarchies are wide or deep. Like the Single Table strategy, the Joined strategy also supports polymorphic queries.

We will take a look at the code to implement a Joined strategy for the Account inheritance hierarchy. First we take the Account entity:

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="ACCOUNT_TYPE")
public abstract class Account implements Serializable {
    protected int id;
    protected double balance;
    protected String accountType;
    // getters and setters
}
```

There is only one change from the Single Table strategy case, we specify a Joined strategy in the @Inheritance annotation. No changes are required for the CheckingAccount and SavingsAccount entities.

Table per Concrete Class Strategy

The final strategy is for each concrete subclass to have its own table. Each class has all its properties, including inherited properties, mapped to columns of the corresponding table. For the Account inheritance hierarchy this will result in the following tables:

CHECKINGACCOUNT Table

ACCOUNT_ID	BALANCE	OVERDRAFTLIMIT
1	430.5	100.0

SAVINGSACCOUNT Table

ACCOUNT_ID	BALANCE	INTERESTRATE
5	5200	4.5

Note there is no `ACCOUNT` table. This strategy is efficient in terms of data storage and has the added advantage that a discriminator column is not required. However, this strategy does not support polymorphism very well. It is also inefficient when queries span more than one table as an SQL `UNION` is typically required.

This strategy is optional. The EJB 3 specification does not require persistence providers to support this strategy. As the GlassFish application server, with the embedded Toplink persistence engine, does not support this strategy we will not go further into coding details.

Summary

Entities do not exist in isolation but are usually associated with other entities. This chapter examined how we map these associations onto a relational database. We first looked at an example with unidirectional one-to-one, one-to-many, and many-to-many associations. We described the `@OneToOne`, `@OneToMany` and `@ManyToMany` annotations for mapping associations to a relational database. Initially we relied on default object/relational mapping values.

We then described a bidirectional example and showed how we could override default object/relational mapping values.

We described eager and lazy loading mechanisms. We gave examples of the `@Embedded`, `@Embeddable`, `@Enumerated` and `@MapKey` annotations. We saw how to handle composite primary keys.

Finally we looked at a number of object/relational inheritance mapping strategies.

5

The Java Persistence
Query Language

EJB 3 provides the **Java Persistence Query Language (JPQL)**. In this chapter we will cover:

- Projections
- GROUP BY and HAVING clauses
- Joins
- Subqueries
- Dynamic Queries
- Functions
- Queries with parameters
- Handling Date and Time
- Bulk update and delete operations
- Native SQL

Introduction

A rather limited query language, **EJB QL**, was provided in EJB 2.0 and 2.1. JPQL enhances EJB QL by providing the following features: JOINs, GROUP BY, HAVING, projection, subqueries, dynamic queries, bulk update and delete. We shall see examples of all these features in this chapter.

JPQL is similar to the SQL language for querying relational databases. However, JPQL queries are expressed in Java objects rather than tables and columns. As SQL dialects differ between relational database vendors, JPQL has an added advantage of portability across databases. The entity manager uses the JDBC configuration specified in the `persistence.xml` file to convert the JPQL query to native SQL.

There may be occasions where potential queries cannot be expressed in JPQL, for example if we want to execute stored procedures. To account for this, JPQL provides the ability to invoke native SQL queries. However, the portability advantage of JPQL is lost in this case.

Simple Queries

All examples in this chapter use the Bank Account scenario described in Chapter 4, with Customer, Address, Referee, and Account persisted entities.

To get all instances of the Customer entity type one can use the query:

```
SELECT c FROM Customer c
```

The `c` in the above query is referred to in JPQL as an **identification variable**. The type returned by a JPQL query depends on the expressions making up the query. In our example the return type is a list of Customer entities. This differs from JDBC where the return type is always wrapped in a `ResultSet`.

The above query can be created dynamically at runtime by supplying the query as a string argument using the `EntityManager.createQuery()` method. This returns a `javax.persistence.Query` instance:

```
Query query = em.createQuery("SELECT c FROM Customer c");
```

The alternative to dynamically specifying queries is to use **named queries**, we shall see examples of these later in this chapter. As our query returns a list, we would use the `Query.getResultList()` method to execute the query:

```
List<Customer> customers = query.getResultList();
```

If the query returned a single result, we would use the `Query.getSingleResult()` method. We will see an example later in this chapter.

We can create a method `simpleQuery()`, say, with the above two methods chained into one. The `simpleQuery()` method would typically be placed in the `BankServiceBean` session bean as follows:

```
public List<Customer> simpleQuery() {
    return (List<Customer>)em.createQuery(
                        "SELECT c FROM Customer c").getResultList();

}
```

Projection

If we just want to retrieve the first and last name of Customer entities, we could use **projection** in JPQL as follows:

```
SELECT c.firstName, c.lastName FROM Customer c
```

The variables `c.firstName` and `c.lastName` are examples of a JPQL **path expression**. A path expression is an identification variable followed by the **navigation operator** (`.`) and a **state-field** or **association-field**. Persistent fields of the Customer entity such as `firstName` and `lastName` are examples of state-fields. We shall see examples of path expressions with association-fields later in this chapter.

In the above query a list of Object arrays, rather than Customer entities, is returned. The following code shows how we might print out the results of our query:

```
List<Object[]> names = query.getResultList();
for (Object[] name : names) {
    System.out.println("First Name: " + name[0] +
                       "Last Name: " + name[1]);
}
```

To remove duplicate values the `DISTINCT` operator is used:

```
SELECT DISTINCT c.lastName FROM Customer c
```

In this query a list of String objects is returned.

Conditional Expressions

The `WHERE` clause is used to filter results. For example,

```
SELECT c FROM Customer c WHERE c.lastName = 'PAGE'
```

retrieves only those Customer entities with `lastName` equal to `'PAGE'`.

A large number of constructs are provided for constructing conditional expressions that make up a `WHERE` clause.

More than one condition can be used in a `WHERE` clause by means of the `AND` operator:

```
SELECT c FROM Customer c WHERE c.lastName = 'KING'
                         AND c.firstName = 'SIMON'
```

As well the `=` (equal) operator JPQL provides the following comparison operators:

`>`, `>=`, `<`, `<=`, `<>` **(not equal)**.

The `IS NULL` operator tests whether or not a single-valued path expression is a `NULL` value:

```
SELECT ad FROM Address ad WHERE ad.postCode IS NULL
```

There is a negated version of the operator, `NOT NULL` as in:

```
SELECT ad FROM Address ad WHERE ad.postCode IS NOT NULL
```

The `BETWEEN` operator is used for checking whether a value is within a given range. `BETWEEN` is inclusive, so in the query:

```
SELECT ac FROM Account ac WHERE ac.balance BETWEEN 0 AND 100
```

accounts with balances equal to 0 and 100 will be selected. The above query is equivalent to

```
SELECT ac FROM Account ac WHERE ac.balance >= 0
                          AND ac.balance <= 100
```

`NOT BETWEEN` is the negated version of the operator.

The `IN` operator tests whether a single-valued path expression is a member of a collection:

```
SELECT ac FROM Account ac WHERE ac.id IN (1,2)
```

This query is equivalent to

```
SELECT ac FROM Account ac WHERE (ac.id = 1) OR (ac.id = 2)
```

`NOT IN` is the negated version of the operator.

The `LIKE` operator enables selection based on pattern matching. The format is

```
string_expression LIKE pattern_string
```

The `string_expression` must have a string value. The `pattern_string` consists of standard characters together with optional wildcard characters. The wildcard characters are underscore (_) for single characters and percent (%) for a sequence of characters. The sequence can be empty. For example, the query

```
SELECT c FROM Customer c WHERE c.firstName LIKE '_A%E%'
```

will select Customers with `firstName` having an A as a second character followed by at least one E anywhere in the string. So 'JANE', 'JAMES', 'MAE' will all match. The pattern matching is case sensitive.

If the pattern string itself contains an underscore or percent sign, we need to prefix that character with an escape character which we specify in the ESCAPE clause:

```
SELECT ad FROM Address ad WHERE ad.postCode
                          LIKE '%$_%'  ESCAPE '$'
```

This query will retrieve all post codes containing an underscore. In this case we have defined a dollar sign ($) as our escape character.

The ORDER BY clause will return the results of a query sorted in ascending or descending sequence.

The sort order is specified using the ASC or DESC keyword. The default is ASC. For example:

```
SELECT ac FROM Account ac ORDER BY ac.balance
```

or

```
SELECT ac FROM Account ac ORDER BY ac.balance DESC
```

It is possible to have multiple ORDER BY items. For example, accounts can be sorted by ascending account type and descending balance:

```
SELECT ac FROM Account ac ORDER BY ac.accountType ASC,
                                   ac.balance DESC
```

Aggregate Functions

We can summarize or aggregate data using an aggregate function. For example,

```
SELECT AVG(a.balance) from Account a
```

will return the average balance over all accounts. The argument to AVG must be numeric. Note that the return type of the AVG function is Double. As a single value is returned we execute the query using the Query.getSingleResult() method.

The remaining aggregate functions provided by JPQL are listed below.

MAX returns the maximum value of the group given an argument which must correspond to an orderable state-field type. Valid orderable state-field types are numeric types, string types, character types, or date types. The MAX return type is equal to the type of the state-field argument.

MIN returns the minimum value of the group given an argument which must correspond to an orderable state-field type. The MIN return type is equal to the type of the state-field argument.

SUM returns the sum of values of the group given a numeric argument. The return type of SUM is either `Long`, `Double`, `BigInteger`, or `BigDecimal` depending on the type of the argument.

COUNT returns the number of values in the group. The argument to COUNT can be an identification variable, state-field, or association-field. The COUNT return type is `Long`. If there are no values to which COUNT can be applied, the result is 0.

The argument to an aggregate function may be preceded by the keyword DISTINCT to specify that duplicate values are to be eliminated before the aggregate function is applied. In the query

```
SELECT DISTINCT COUNT (c.lastName) from Customer c
```

if, for example, three customers share the same last name they will be counted as one.

When applying any aggregate function, any field with a null value will be eliminated regardless of whether the keyword DISTINCT is specified.

GROUP BY

The previous example returns the average balance over all accounts. Typically when using an aggregate function we want to group the results according to some criteria. For example, we may want to calculate average account balances according to account type. For this we would use the GROUP BY clause, as in:

```
SELECT a.accountType, AVG(a.balance) from Account a
GROUP BY a.accountType
```

Any item that appears in the SELECT clause other than the aggregate function must also appear in the GROUP BY clause.

HAVING

We can restrict results retuned by a GROUP BY expression by adding a HAVING clause. Only those groups that satisfy the HAVING condition will be returned. For example, the query

```
SELECT count(a), a.accountType, AVG(a.balance) from Account a GROUP BY
a.accountType HAVING count(a.accountType) > 1
```

will return only those account types that have at least two values.

Queries with Relationships

A `SELECT` clause can refer to persistent relationship fields. For example, in the query

```
SELECT c.referee FROM Customer c
```

recall that `referee` is a persistent variable of type `Referee` which has a one-to-one relationship with the Referee entity. An example of an JPQL **association-field** is `c.referee`. Note that the return type in this query is `Referee`.

A path expression can be composed from other path expressions if the original path expression evaluates to a single-valued association-field. The query

```
SELECT c.lastName, c.referee.name FROM Customer c
```

is legal as the path expression `c.referee` is a single-valued association-field. The query

```
SELECT c.accounts FROM Customer c
```

is illegal as the association-field `c.accounts` evaluates to a collection.

Joins

The queries in the previous section performed an implicit **join**. A join occurs whenever we need to navigate across two or more entities. We have seen that we cannot construct a composite path expression from a collection association-field. This prevents us from navigating across a one-to-many or many-to-many entity relationship. We can do this explicitly using one of the `JOIN` operators.

Inner Joins

Take a look at the following query:

```
SELECT c.lastName, a.addressLine FROM Customer c
                            INNER JOIN c.addresses a
```

For each customer this will retrieve their last name and all associated address lines (recall a customer can have more than one address in our model). The identification variable `a` is defined by the path expression `c.addresses`, and this identification variable is used in the expression `a.addressLine`. In the case of an inner join, if a customer does not have an address then the customer's last name will not be retrieved by the above query. Note that the use of the keyword `INNER` is optional in JPQL.

The query from the previous section

```
SELECT c.referee FROM Customer c
```

can be written using an explicit join as follows:

```
SELECT r FROM Customer c JOIN c.referee r
```

An inner join can also be explicitly specified by the use of a Cartesian product in the FROM clause together with a join condition in the WHERE clause. Typically we would use this where there is no explicit relationship between the entities in our model. In the implicit inner join

```
SELECT c FROM Customer c, Referee r WHERE c.lastName = r.name
```

there is no explicitly modeled relationship between customer and referee names. This query will list customers who happen to have the same name as a referee.

Outer Joins

An outer join is used where matching values in a join condition need not be present. The keywords LEFT OUTER JOIN are used in JPQL. For example, the query

```
SELECT c.lastName, a.addressLine FROM Customer c
                          LEFT OUTER JOIN c.addresses a
```

will include those customers that do not have an address. In this case a.addressLine will be set to NULL. Note that the use of the keyword OUTER is optional, so LEFT JOIN and LEFT OUTER JOIN are synonymous..

Fetch Joins

A fetch join will eagerly load related data even if we have specified a lazy loading strategy when defining the relationship. Recall the query

```
SELECT c FROM Customer c
```

will not fetch related address data. This is because there is a many-many relationship between Customer and Address, and the default behavior in many-many relationships is to lazily load related data. Of course we can specify a fetch=EAGER clause when defining the Customer Address relationship. The fetch join query

```
SELECT c FROM Customer c JOIN FETCH c.addresses
```

will fetch related address data even if we have specified a fetch=LAZY clause when defining the Customer Address relationship.

A `LEFT JOIN FETCH` will additionally perform an outer join. So the query

```
SELECT c FROM Customer c LEFT JOIN FETCH c.addresses
```

will include customers who do not have an address.

Collection Comparison Expressions

The `IS EMPTY` clause tests whether a collection-valued expression has no elements. For example, the query

```
SELECT c FROM Customer c WHERE c.addresses IS EMPTY
```

returns only those customers with no addresses.

The `MEMBER OF` clause tests whether a value of an entity path expression is a member of a collection valued path expression. The query

```
SELECT DISTINCT c FROM Customer c, Address a
                     WHERE a MEMBER OF c.addresses
```

returns only those customers who have addresses.

Constructor Expressions

By using a constructor in the `SELECT` clause we can instantiate a Java object with the results of our query. This Java object does not need to be an entity or be mapped onto a database. The class name corresponding to this object must be fully qualified. For example, the query

```
SELECT NEW ejb30.entity.CustomerRef(
           c.firstName, c.lastName, c.referee.name)
FROM Customer c
```

creates a `CustomerRef` object which contains a customer's first and last name together with the associated referee name. The following is a partial listing for the `CustomerRef` class:

```
package ejb30.entity;
import java.io.Serializable;
public class CustomerRef implements Serializable {
    private String firstName;
    private String lastName;
    private String refereeName;
    public CustomerRef() {}
    public CustomerRef(String firstName, String lastName,
```

```
                        String refereeName) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.refereeName = refereeName;
    }
// getter and setter methods
}
```

Note the presence of a constructor exactly matching that in the `SELECT` statement.

SubQueries

Subqueries can be used in the `WHERE` or `HAVING` clause. A subquery is a `SELECT` statement within parentheses which in turn is part of a conditional expression. For example, the query

```
SELECT a FROM Account a WHERE a.balance >
        (SELECT MIN(a2.balance) FROM Account a2) )
```

will retrieve those accounts which have a balance greater than the minimum balance. Note we use a different indicator variable, `a2`, in the subquery to distinguish it from the indicator variable, `a`, in the main query.

An `EXISTS` expression is true if the result of the subquery consists of one or more values. For example, the query

```
SELECT c FROM Customer c WHERE EXISTS
        (SELECT a2 FROM Account a2 WHERE a2.customer = c
          AND a2.accountType='S')
```

retrieves customers who have at least one savings account.

An `ALL` expression is true if the conditional expression is true for all values in the subquery result. For example, the query

```
SELECT c FROM Customer c WHERE c.monthlyIncome >
        ALL (SELECT a.balance FROM c.accounts a)
```

retrieves customers whose monthly income is greater than all their associated accounts.

An `ANY` expression is true if the conditional expression is true for some value in the subquery result. For example, the query

```
SELECT c FROM Customer c WHERE c.monthlyIncome >
          ANY (SELECT a.balance FROM c.accounts a)
```

retrieves customers whose monthly income is greater than at least one of their associated accounts.

Functions

JPQL provides a number of functions which may be used in the WHERE or HAVING clause of a query.

CONCAT

CONCAT returns a string that is a concatenation of its arguments. For example,

```
SELECT a FROM Address a WHERE CONCAT(a.postCode, a.country)
                              LIKE 'M%UK'
```

SUBSTRING

SUBSTRING returns a portion of a string which is supplied as an argument to the function. The first argument is the input string, the second argument is an integer indicating the starting position of the substring, and the third argument is the length of the substring. The first position of the string is denoted by 1. For example,

```
SELECT a FROM Address a WHERE SUBSTRING(a.postCode, 1, 3)
                              = 'RC3'
```

TRIM

TRIM trims a specified character from a string. The syntax of the TRIM function is

```
TRIM ([TRIM_SPECIFICATION] [TRIM_CHARACTER] FROM INPUT_STRING)
```

The trim specification takes one of the values LEADING, TRAILING, or BOTH. BOTH is the default. The default trim character is a space (or blank). For example, the query

```
SELECT a FROM Address a WHERE TRIM(FROM a.country) = 'UK'
```

will trim any leading or trailing spaces from the country field.

LOWER and UPPER

The LOWER and UPPER functions convert a string to lower and upper case respectively. For example,

```
SELECT a FROM Address a WHERE LOWER(a.postCode) = '2ho 1ru'
                              AND UPPER(a.country) = 'UK'
```

LENGTH

LENGTH returns the length of a string as an integer. For example,

```
SELECT a FROM Address a WHERE LENGTH(a.addressLine) = 22
```

LOCATE

LOCATE returns the first position of a given search string within a target string starting at a specified position. The syntax of the LOCATE function is:

```
LOCATE(search_string, target_string [, start_postion] )
```

The default for start position is 1, the beginning of the target string. The returned position is an integer; if the search string is not found the function returns 0. In the query

```
SELECT a FROM Address a WHERE SUBSTRING(a.postCode,
                       LOCATE(' ', a.postCode), 4) = ' 9XY'
```

we locate the first space in the postCode field and use this as the start position in the SUBSTRING function to test whether the next four characters are equal to ' 9XY'

ABS

ABS returns the absolute value of a supplied argument. The argument type can be integer, float, or double. The function return type is the same as the argument type. For example,

```
SELECT a FROM Account a WHERE ABS(a.balance) > 50
```

SQRT

SQRT returns the square root of the supplied numeric argument as type double. For example,

```
SELECT a FROM Account a WHERE SQRT(ABS(a.balance)) < 12
```

MOD

MOD returns the modulus of two integer arguments as type integer. For example,

```
SELECT a FROM Account a WHERE MOD(a.id, 2) = 0
```

returns all accounts with even identifiers.

SIZE

`SIZE` returns the number of elements in a supplied collection. If the collection is empty, zero is returned. For example, the query

```
SELECT c FROM Customer c WHERE SIZE(c.accounts) = 3
```

returns all customers who have exactly `3` accounts.

Queries with Parameters

Positional or named parameters can be used in `WHERE` or `HAVING` clauses of a query. This enables the same query to be re-executed a number of times with different parameters.

Positional Parameters

Parameters are designated by a question mark prefix followed by an integer, numbered starting from 1.

For example, the following query uses two positional parameters:

```
SELECT c FROM Customer c WHERE c.firstName LIKE ?1 AND
                               c.lastName LIKE ?2
```

The `Query.setParameter()` method is used to bind actual values to the positional parameters. The first argument of `setParameter()` is an integer denoting the parameter position, the second argument is the actual value. The following listing shows how we might wrap the above query into a method which creates and executes the query:

```
public List<Customer> parameterQuery(String firstParam,
                                     String secondParam) {
    return (List<Customer>)
           em.createQuery(
                        "SELECT c FROM Customer c WHERE " +
                        "c.firstName LIKE ?1  " +
                        "AND c.lastName LIKE ?2")
                        .setParameter(1, firstParam)
                        .setParameter(2, secondParam)
                        .getResultList();
}
```

Named Parameters

A named parameter is prefixed by the colon ":" symbol, as follows:

```
SELECT c FROM Customer c WHERE c.firstName LIKE :firstString
                         AND c.lastName LIKE :secondString
```

Again the `Query.setParameter()` method is used to bind actual values to the named parameters. This time the first argument of `setParameter()` is the named parameter and the second argument is the actual value. The following method creates and executes the query:

```
public List<Customer> parameterQuery(String firstParam,
                                     String secondParam) {
    return (List<Customer>)
            em.createQuery(
                "SELECT c FROM Customer c WHERE " +
                "c.firstName LIKE :firstString " +
                "AND c.lastName LIKE :secondString")
            .setParameter("firstString", firstParam)
            .setParameter("secondString", secondParam)
            .getResultList();
}
```

Named Queries

To this point, all queries have been defined dynamically by supplying a string containing the query to the `EntityManager.createQuery()` method. A named query is an alternative with improved run time performance of the query. A named query is predefined using the `@NamedQuery` annotation in the class definition of any entity. The `@NamedQuery` annotation takes two arguments: the name of the query and the query text itself.

For example, we can insert the following named query in the Customer class definition:

```
@NamedQuery(name="findSelectedCustomers",
            query = "SELECT c FROM Customer c " +
                    "WHERE c.firstName LIKE :firstString " +
                    " AND c.lastName LIKE :secondString")
```

This query could be placed in any class definition, but as it relates to customers it makes sense to place it in the Customers class definition. The scope of a named query is the current persistence context, so query names are unique with respect to the current persistence context. For example, you cannot have a `findCustomersAndAccounts` named query defined in both the Customer and Account class definitions.

To execute the named query we use the `EntityManager.createNamedQuery()` method, as follows:

```
public List<Customer> parameterQuery(String firstParam,
                                     String secondParam) {
    return (List<Customer>)
            em.createNamedQuery("findSelectedCustomers")
            .setParameter("firstString", firstParam)
            .setParameter("secondString", secondParam)
            .getResultList();
}
```

If we want to place more than one named query in a class definition, as well as annotating each query with `@NamedQuery` as above, the entire set would be wrapped in a `@NamedQueries` annotation. For example, we might have the `findCheckingAccounts` and `findSavingsAccounts` named queries placed in the Accounts class definition as follows:

```
@NamedQueries({
    @NamedQuery(name="findCheckingAccounts",
                query="SELECT a FROM Account a WHERE " +
                    "a.balance > :firstString AND " +
                    "a.accountType = 'C' "),
    @NamedQuery(name="findSavingsAccounts",
                query="SELECT a FROM Account a WHERE " +
                    "a.balance > :firstString AND " +
                    "a.accountType = 'S' ")
})
```

The queries would be executed using the `EntityManager.createNamedQuery()` method as described above.

Handling Date and Time

In this section we look at how date and time is handled. If we want to persist a field of type `java.util.Date` we need to use the `@Temporal` annotation. This is described in the next subsection. The subsection "Queries with Date Parameters" shows how we handle a query with a `java.util.Date` or `java.util.Calendar` parameter. Finally the subsection "Datetime Functions" describes the date and time functions provided by JPQL.

@Temporal annotation

Suppose we add a new field, `dateOpened`, of type `java.util.Date` in the Account entity. The `dateOpened` variable indicates the date on which the account was opened. We must use the `@Temporal(TemporalType.DATE)` annotation, which can be either field-based or property-based, to indicate that the field will be mapped onto the `java.sql.Date` database type. This is shown in the following code fragment:

```
@Entity
public class Account implements Serializable {
    private int id;
    private double balance;
    private String accountType;
    private Customer customer;
    private Date dateOpened;
        ...

    @Temporal(TemporalType.DATE)
    public Date getDateOpened() {
        return dateOpened;
    }
    public void setDateOpened(Date dateOpened) {
        this.dateOpened = dateOpened;
    }
    ...
```

Other possible values of the `TemporalType` enumerated type are:

- `TemporalType.TIME` for mapping onto `java.sql.Time`
- `TemporalType.TIMESTAMP` for mapping onto `java.sql.Timestamp`

Note that, we must also use the `@Temporal` annotation for persisting fields of type `java.util.Calendar`.

Queries with Date Parameters

If we have a query with a `java.util.Date` or `java.util.Calendar` parameter we need to use one of the overloaded methods of `Query.setParameter()` with `TemporalType` as one of the arguments. For example, the following query retrieves accounts opened on a supplied date:

```
public List<Account> parameterQuery(Date firstParam) {
    return (List<Account>)
        em.createQuery(
            "SELECT a FROM Account a WHERE " +
            "a.dateOpened = ?1")
            .setParameter(1, firstParam, TemporalType.DATE)
            .getResultList();
    }
```

The overloaded date methods of `Query.setParameter()` for positional parameters are:

*   `setParameter(int position, Date value, TemporalType temporalType)`

*   `setParameter(int position, Calendar value, TemporalType temporalType)`

And for named parameters:

*   `setParameter(String name, Date value, TemporalType temporalType)`

*   `setParameter(String name, Calendar value, TemporalType temporalType)`

Datetime Functions

JPQL provides the following datetime functions:

*   CURRENT_DATE returns the value of the current date

*   CURRENT_TIME returns the value of the current time

*   CURRENT_TIMESTAMP returns the values of the current timestamp

Note these values are retrieved from the database server. These times may not be exactly the same as the current JVM time if the JVM in which the query is running is on a different server from the database server. This is only an issue for critical real time applications.

As an example, the following query uses the `CURRENT_DATE` function:

```
SELECT a FROM Account a WHERE a.dateOpened = CURRENT_DATE
```

Bulk Update and Delete

Bulk update and delete operations allow you to delete or update a number of entity instances of a *single* entity class including its subclasses. For example, the statement

```
DELETE FROM Account a WHERE a.customer IS NULL
```

will remove from the database all account entities that do not have an associated customer. A delete operation does not cascade to related entities, such as Customer in our example. For bulk update and delete operations we use the `Query.executeUpdate()` method to execute the query. So we can wrap the above query in a `deleteOrphanedAccounts()` method, say, as follows:

```
public void deleteOrphanedAccounts() {
    em.createQuery("DELETE FROM Account a WHERE " +
                   "a.customer IS NULL").executeUpdate();
}
```

The bulk update syntax is similar to that in SQL. For example, the query

```
UPDATE Account a SET a.balance = a.balance * 1.1
WHERE a.accountType = 'C'
```

will increase all checking account balances by 10 percent.

Bulk update and delete operations apply directly on the database and do not modify the state of the currently managed entity. Since the state of the persistence context is not synchronized with the database, care should be taken when using bulk update and deletes. Such operations should be performed in a separate transaction or at the beginning of a transaction before the affected entities have been accessed. We discuss transactions in Chapter 7.

Native SQL

Native SQL allows one to utilize proprietary database SQL features such as accessing stored procedures. Furthermore, native SQL enables you to manually optimize queries using hints and specific indexes. We would expect persistence engines to continually improve their JPQL query optimization capabilities and so reduce the need to use native SQL. Unlike JPQL, native SQL is potentially database specific, so loss of portability is the cost of using it. An alternative to native SQL for database SQL is JDBC; however native SQL has the advantage of returning the results of queries as entities.

The results of a native SQL query can be one or more entity types, non-entity or scalar values, or a combination of entities and scalar values. Native SQL queries are created using the `EntityManager.createNativeQuery()` method. This method is overloaded and the version used depends on whether a single entity type, multiple entity types or scalar values are returned, so we will demonstrate this through a number of examples.

The following query returns a single, Account, entity type:

```
createNativeQuery(
   "SELECT id, balance, accountType, customer_id FROM Account",
      Account.class);
```

The first parameter of `createNativeQuery()` is the native SQL itself and the second parameter is the class of the resulting entity instance or instances. Note that all columns must be present in the query including any foreign key columns. `customer_id` is a foreign key column in the above example. All column names, or column aliases if they are used in the native SQL, must match with the entities field name.

The following query returns an Account and a Customer entity type:

```
createNativeQuery(
    "SELECT a.id AS ACC_ID, a.balance, a.accountType," +
    "a.customer_id, c.id, c.firstName, c.lastName " +
    "FROM Account a, Customer c " +
    "WHERE a.customer_id = c.id", "CustomerAccountResults");
```

Note in the native SQL we have used `ACC_ID` as a column alias to distinguish the Account `id` column from the Customer `id` column. This time the second parameter of `createNativeQuery()`, `CustomerAccountResults`, is the name of a result set mapping. We must use a result set mapping whenever more than one entity is retrieved from the query. The result set mapping is specified using the `@SqlResultSetMapping` annotation which can be placed on any entity class. The result set mapping name is unique within the persistence unit. The code for the `CustomerAccountResults` result set mapping is:

```
@SqlResultSetMapping(
    name="CustomerAccountResults",
    entities={@EntityResult(
            entityClass=ejb30.entity.Account.class,
            fields={@FieldResult(name="id",
                                 column="ACC_ID"),
                @FieldResult(name="customer",
                                 column="CUSTOMER_ID") } ),
         @EntityResult(
          entityClass=ejb30.entity.Customer.class)
        }
)
```

Each `@EntityResult` annotation specifies an entity returned by the query. The `entityClass` parameter specifies the fully qualified entity class.

The `@FieldResult` annotation needs to be used whenever there is a mismatch between the column name or alias and the entity field name. The `name` parameter specifies the entities field name. The `column` parameter specifies the column or alias name. In our example we map the `ACC_ID` column alias to the Account `id` field; the `CUSTOMER_ID` column is mapped to the Account customer field.

A native SQL query may return a scalar result rather than an entity. For example, the following query returns all Customer last names as a `List` of `String` values, rather than a `List` of entity instances:

```
createNativeQuery("SELECT lastName FROM Customer");
```

In this case there is just one argument to `createNativeQuery()`, the native SQL string itself.

It is possible for a native SQL query to return a combination of entities and scalar values. For example, in the following query we retrieve an Account entity together with a scalar value representing the last name of the corresponding Customer:

```
createNativeQuery(
    "SELECT a.id, a.balance, a.accountType, a.customer_id, " +
    "c.lastName FROM Account a, Customer c "  +
    "WHERE a.customer_id = c.id", "AccountScalarResults");
```

The corresponding result set mapping, `AccountScalarResults`, is:

```
@SqlResultSetMapping(
    name="AccountScalarResults",
    entities={@EntityResult(
                entityClass=ejb30.entity.Account.class,
                fields=
                  {@FieldResult(name="customer",
                                column="CUSTOMER_ID")} )
    },
    columns={@ColumnResult(name="LASTNAME")}
)
```

The `@ColumnResult` annotation is used to specify a column as a scalar value. Recall the `@SqlResultSetMapping` annotation can be placed on any entity class. However if a given entity class has more than one `@SqlResultSetMapping` then these must be enclosed within a `@SqlResultSetMappings` annotation. For example, suppose that both the `AccountScalarResults` and the `CustomerAccountResults` result set mappings are placed on the Customer entity. The result will be:

```
@SqlResultSetMappings({
    @SqlResultSetMapping(
        name="CustomerAccountResults",
        entities={@EntityResult(
                    entityClass=ejb30.entity.Account.class,
                    fields={@FieldResult(
                            name="id",
                            column="ACC_ID"),
                        @FieldResult(
                            name="customer",
                            column="CUSTOMER_ID") } ),
                @EntityResult(
                    entityClass=ejb30.entity.Customer.class)
                }
    ),
    @SqlResultSetMapping(
        name="AccountScalarResults",
        entities={@EntityResult(
                    entityClass=ejb30.entity.Account.class,
                    fields={@FieldResult(
                            name="customer",
                            column="CUSTOMER_ID")} )
                },
        columns={@ColumnResult(name="LASTNAME")}
    )
})
```

So far, all our examples of native SQL have been dynamic queries, we can also use named native SQL queries. A named native query is defined on any entity class using the `@NamedNativeQuery` annotation. As an example, take our earlier query which retrieves a single Account entity type. The corresponding named native query is specified with:

```
@NamedNativeQuery(
    name="findAccount",
    query="SELECT a.id, a.balance, a.accountType, " +
            "a.customer_id FROM Account a",
    resultClass=Account.class)
```

We execute the above `findAccount` named native query with the `EntityManager.createNamedQuery()` method:

```
createNamedQuery("findAccount");
```

A named native query may need to refer to a results set mapping. For example, take the earlier query which retrieves an Account entity type together with a scalar value representing the last name of the corresponding Customer. The corresponding named native query is:

```
@NamedNativeQuery(
    name="findAccountLastname",
    query="SELECT a.id, a.balance, a.accountType, " +
          "a.customer_id, c.lastName FROM Account a, " +
          "Customer c WHERE a.customer_id = c.id",
    resultSetMapping="AccountScalarResults")
```

If a given entity class has more than one `@NamedNativeQuery` then these must be enclosed within a `@NamedNativeQueries` annotation. For example, suppose that both the `findAccount` and the `findAccountLastname` named native queries are placed in the Customer entity. The result will be:

```
@NamedNativeQueries({
    @NamedNativeQuery(
        name="findAccountLastname",
        query="SELECT a.id, a.balance, a.accountType, " +
              "a.customer_id, c.lastName FROM Account a, " +
              "Customer c WHERE a.customer_id = c.id",
        resultSetMapping="AccountScalarResults"),
    @NamedNativeQuery(
        name="findAccount",
        query="SELECT a.id, a.balance, a.accountType, " +
              "a.customer_id FROM Account a",
        resultClass=Account.class)
})
```

Summary

In this chapter we described the Java Persistence Query Language or JPQL.

JPQL is similar to the SQL language for querying relational databases. However, JPQL queries are expressed in Java objects rather than tables and columns. We looked at examples of simple queries and queries with conditional expressions using WHERE clauses. We described aggregate functions including the GROUP BY and HAVING clause.

We saw examples of joins and subqueries. JPQL provides a number of functions which may be used in the WHERE or HAVING clause of a query. Queries can also have positional or named parameters and we saw examples of these. We covered bulk update and delete operations.

Finally JPQL provides a Native SQL feature which enables one to use proprietary database SQL features such as accessing stored procedures.

6

Entity Manager

In this chapter we will take a more detailed look at the entity manager. In particular we will cover:

- Application-managed entity manager
- Container-managed entity manager
- Entity manager methods
- Cascade operations
- Extended persistence context
- Entity lifecycle callback methods
- Entity listener classes

We have already seen examples of the entity manager in previous chapters. All these examples made use of container-managed entity managers, where an entity manager's lifecycle is managed by the EJB 3 container. To allow for situations where we do not require the services offered by an EJB 3 container but would like to use the persistence model, the Java Persistence API provides **application-managed entity managers**. We will first take a look at application-managed entity managers, before returning to entity manager topics which apply irrespective of whether an entity manager is container-managed or application-managed.

Application-managed Entity Manager

To allow for situations where we do not require the services offered by an EJB 3 container but would like to use the persistence model, the Java Persistence API provides application-managed entity managers. Typically a Java application, such as a Swing program, would make use of an application-managed entity manager. These standalone Java applications are referred to as Java SE (Standard Edition) applications. Application-managed entity managers can also be used in servlets.

With application-managed entity managers, the application manages the entity manager's lifecycle. Container-managed entity managers normally use **JTA** (Java Transaction API) transactions and while it is possible to use JTA transactions with application-managed entity managers, normally **resource-local transactions** are used. An API, the `EntityTransaction` interface, is provided for resource-local transactions. This interface provides methods for beginning, committing and rolling back transactions. JTA transactions are discussed in more detail in Chapter 7.

Whenever an application uses a container-managed entity manager it injects an `EntityManager` instance using the `@PersistenceContext` annotation as follows:

```
@PersistenceContext(unitName="BankService")
private EntityManager em;
```

We have seen several examples of this. Behind the scenes the container creates an `EntityManagerFactory`; this factory is used to create `EntityManager` instances whenever the `@PersistenceContext` annotation is used. Finally the container, and not the application, closes both the `EntityManager` and `EntityManagerFactory`.

In the case of application-managed entity managers we need to explicitly create an `EntityManagerFactory`. We also need to explicitly close both the `EntityManager` and `EntityManagerFactory`. Let's look at an example.

The listing below shows a Java application, `BankClient`, which adds a customer to the database and then retrieves the same customer. The program uses the same Customer entity we have seen in previous chapters.

```
public class BankClient {
    public static void main(String[] args) {
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory(
            "BankService");
        EntityManager em = emf.createEntityManager();
        int custId = 0;
        String firstName = null;
        String lastName = null;
        try {
            custId = Integer.parseInt(args[0]);
            firstName = args[1];
            lastName = args[2];
        } catch (Exception e) {
            System.err.println(
                "Invalid arguments entered, try again");
            System.exit(0);
        }
```

```
        addCustomer(em, custId, firstName, lastName);
        Customer cust = em.find(Customer.class, custId);
        System.out.println(cust);
        em.close();
        emf.close();
    }

    private static void addCustomer(EntityManager em,
            int custId, String firstName, String lastName) {
        EntityTransaction trans = em.getTransaction();
        trans.begin();
        Customer cust = new Customer();
        cust.setId(custId);
        cust.setFirstname(firstName);
        cust.setLastname(lastName);
        em.persist(cust);
        trans.commit();
    }
}
```

Note that the application first creates a reference to the `EntityManagerFactory` associated with our persistence unit, `BankService`. We then use the `EntityManagerFactory.createEntityManager()` method to create an instance of the `EntityManager`:

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory(
    "BankService");
EntityManager em = emf.createEntityManager();
```

Since adding a new customer to the database is a transactional operation, we need to explicitly use the `EntityTransaction` interface. The `EntityTransaction.getTransaction()` method returns the interface:

```
EntityTransaction trans = em.getTransaction();
```

We start a transaction with the `EntityTransaction.begin()` method and after persisting our new customer entity with the `EntityManager.persist()` method, we commit the transaction with the `EntityTransaction.commit()` method. The `EntityTransaction` interface also has `rollback()` and `isActive()` methods. These are respectively for rolling back a transaction and for determining whether a transaction is in progress.

We need to modify the `persistence.xml` file to indicate that we are using resource-local transactions in a Java SE environment.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns=http://java.sun.com/xml/ns/persistence
              version="1.0">
<persistence-unit name="BankService"
              transaction-type="RESOURCE_LOCAL">
<class>ejb30.entity.Customer</class>
<properties>
   <property name="toplink.jdbc.driver"
     value="org.apache.derby.jdbc.ClientDriver"/>
   <property name="toplink.jdbc.url"
     value="jdbc:derby://localhost:1527/sun-appserv-samples"/>
   <property name="toplink.jdbc.user" value="app"/>
   <property name="toplink.jdbc.password" value="app"/>
</properties>
</persistence-unit>
</persistence>
```

Note that we have specified the transaction-type as `RESOURCE_LOCAL`. The `<class>` element provides the name of a class which the persistence provider adds to its set of managed classes. We are not required to do this in a container-managed environment as the container obtains the managed classes from the JAR file containing the `persistence.xml` file.

The `<properties>` element specifies vendor specific properties. The EJB 3 reference implementation uses Toplink as the persistence provider, so the above `<properties>` elements are all Toplink specific.

- `toplink.jdbc.driver` is the name of the JDBC driver. If we are using the Derby database the driver is `org.apache.derby.jdbc.ClientDriver`.

- `toplink.jdbc.url` is the JDBC connection URL to the database. For the Derby database embedded within GlassFish, this value is `jdbc:derby://localhost:1527/sun-appserv-samples`. `sun-appserv-samples` is the default database name when using the GlassFish embedded Derby database.

- `toplink.jdbc.user` and `toplink.jdbc.password` are the database user and password respectively. For the GlassFish embedded Derby database, the user and password both default to `"app"`.

Entity Manager Merge

In this and the remaining sections we discuss topics which are applicable to both container-managed and application-managed entity managers. However, our examples will assume a container-managed entity manager is being used.

As we have seen, an entity becomes managed when we use the `EntityManager.persist()` method. At this point the entity is associated with a persistence context. While an entity is managed the state of the entity is automatically synchronized with the database by the entity manager. By default a persistence context is scoped to a transaction. When the transaction ends, the persistence context ends and the entity is no longer managed. An unmanaged entity is referred to as a detached entity and is no longer associated with a persistence context. This means that further changes to the entity are no longer reflected in the database. If an entity is passed by value, through the remote interface to another application tier for example, then it also becomes detached. A typical scenario in 3-tier applications is to create a remote instance of an entity in the web-tier, possibly modify the entity, and then update the database with the new entity values.

To cater for the above scenario, the `EntityManager.merge()` method is provided. This synchronizes the state of a detached entity with the database.

As an example, suppose we have a stateless session bean, `BankServiceBean`, as listed below:

```
@Stateless
public class BankServiceBean implements BankService {
    @PersistenceContext(unitName="BankService")
    private EntityManager em;
    public Customer findCustomer(int custId) {
        return ((Customer) em.find(Customer.class, custId));
    }
    public void addCustomer(int custId, String firstName,
                            String lastName) {
        Customer cust = new Customer();
        cust.setId(custId);
        cust.setFirstName(firstName);
        cust.setLastName(lastName);
        em.persist(cust);
    }
    public void updateCustomer(Customer cust) {
        Customer mergedCust = em.merge(cust);
    }
}
```

A new method is `updateCustomer()`. This takes a detached Customer entity, `cust`, as an argument. The `EntityManager.merge()` method takes the detached entity as an argument and creates a managed entity, `mergedCust`. From this point on any setter methods on `mergedCust` will be synchronized with the database.

The code fragment below shows how a remote client first adds a customer to the database then creates a remote Customer object by invoking the `BankService.findCustomer()` method. After modifying this object the `BankService.updateCustomer()` method is invoked. `updateCustomer()` performs the merge.

```
@EJB
private static BankService bank;
bank.addCustomer(custId, firstName, lastName);
Customer cust = bank.findCustomer(custId);
cust.setFirstName("Michael");
bank.updateCustomer(cust);
```

Note with EJB 3 how straightforward it is to create a remote object from an EJB. A Customer object is merely returned by the session bean `findCustomer()` method. The only constraint is that the Customer object is serialized. In earlier EJB technology you would have to use the **Value Object** (or **Data Transfer Object**) pattern to do this.

Entity Manager Methods

In this section we look at some more `EntityManager` methods.

remove()

The `remove()` method flags an entity for removal from the database. Note that the entity must be a managed entity. The removal need not take place immediately. Removal occurs whenever the `EntityManager` decides to synchronize, or **flush**, the persistence context with the database. Typically this occurs when a transaction is committed. The code below shows a `BankServiceBean` method, `removeCustomer()`, which would be invoked in order to remove a Customer entity.

```
public void removeCustomer(Customer cust) {
    Customer mergedCust = em.merge(cust);
    em.remove(mergedCust);
}
```

Note that the method first merges an entity to ensure it is managed before removing it.

contains()

The `contains()` method is used to check whether an entity is managed in the current persistence context. It returns a `boolean` which is set to true if the entity is managed, otherwise it set to false. For example:

```
if (em.contains(cust)) {
    System.out.println("cust is managed");
}
```

flush()

Flushing, or synchronizing a persistence context with the database, is usually under the control of the entity manager. Typically flushing occurs at transaction commit time, although the entity manager may flush at other times, such as before query execution, if it sees fit. The `flush()` method enables the application to force synchronization of the persistence context with the database. The syntax is simply:

```
em.flush()
```

setFlushMode()

By default, whenever a query is executed within a transaction the entity manager ensures that any updates to the database are included in the query results. Typically this is done by flushing before query execution. This behavior is known as a **flush mode** of `AUTO`. A flush mode of `COMMIT` will cause flushing to occur only at transaction commit. A `COMMIT` flush mode will improve query performance as there are fewer calls to the database. This mode would be used if we know the query is not affected by potential database updates.

The flush mode can be set on the entity manager using the `setFlushMode()` method. This mode will then apply to all queries. The `setFlushMode()` method expects an argument of either `FlushModeType.COMMIT` or `FlushModeType.AUTO`. Recall `FlushModeType.AUTO` is the default behavior. For example

```
em.setFlushMode(FlushModeType.COMMIT);
```

We can also set the flush mode on a query object on a per query basis. For example:

```
Query query = em.createQuery(
            "SELECT c FROM Customer c WHERE c.id = ?1")
    .setParameter(1, custId) ;
query.setFlushMode(FlushModeType.AUTO);
```

The query flush mode will override any entity manager flush mode.

refresh()

The `refresh()` method is used where we want to ensure that any change to the database is synchronized with a managed entity. For example:

```
em.refresh(mergedCust);
```

Note that `mergedCust` must be a managed entity. The `refresh()` method is more likely to be used in a long running transaction.

clear()

The `clear()` method clears the persistence context. This causes all managed entities to become detached. The syntax is simply:

```
em.clear()
```

Note that any updates to entities will not be flushed to the database by the `clear()` method.

Cascade Operations

Cascading can occur whenever we have relationships between entities. For example, there is a one-to-one relationship between the Customer and Referee entities in our application. By allowing a `persist` operation to cascade, whenever we persist a Customer entity, the associated Referee entity is automatically persisted. There is no need to persist the Referee entity in a separate operation. By allowing a `remove` operation to cascade, whenever we remove a Customer entity, the associated Referee is automatically removed.

Cacading can occur for one-to-one, one-to-many, many-to-one, and many-to-many relationships. Cascading is possible for `persist`, `remove`, `merge`, and `refresh` operations. There is no default cascading behavior for any operation and for any relationship.

persist

To cascade a `persist` operation, set the `cascade` element of the relationship annotation to `CascadeType.PERSIST`. For example, we could cascade a `persist` from the Customer to the Referee entity by modifying the `@OneToOne` annotation within the Customer entity as follows:

```
@OneToOne(cascade=CascadeType.PERSIST)
    public Referee getReferee() { return referee; }
```

If we wanted to cascade a `persist` from a Referee entity to a Customer entity we would need to set the cascade option from within the Referee entity.

remove

To cascade a `remove` operation set the `cascade` element of the relationship annotation to `CascadeType.REMOVE`. The cascade `remove` only applies to one-to-one and one-to-many relationships. For example, we could cascade a `remove` from the Customer to associated Account entities as follows:

```
@OneToMany(fetch=EAGER, cascade=CascadeType.REMOVE)
public Collection<Account> getAccounts() { return accounts; }
```

As a result, whenever a Customer entity is removed, associated Account entities are automatically removed.

A cascade `remove` should be used with caution. In the above example it may make sense for Account entities to exist on their own, or be part of a relationship other than the Customer-Account one.

merge

To cascade a `merge` operation set the cascade element of the relationship annotation to `CascadeType.MERGE`. For example, we could cascade a `merge` from the Customer to associated Account entities as follows:

```
@OneToMany(fetch=EAGER, cascade=CascadeType.MERGE)
public Collection<Account> getAccounts() { return accounts; }
```

In this example, assume we detach a Customer entity, update the detached Customer and its associated Account entities and then merge the Customer entity. Without a cascade, any updating of an Account entity will be lost after the merge, unless the Account entity is also merged. With a cascade `merge` we only need to merge the Customer entity to have the associated Account entities merged as well.

refresh

To cascade a `refresh` operation, set the cascade element of the relationship annotation to `CascadeType.REFRESH`. For example, we could cascade a `refresh` from the Customer to associated Account entities as follows:

```
@OneToMany(fetch=EAGER, cascade=CascadeType.REFRESH)
public Collection<Account> getAccounts() { return accounts; }
```

This has the effect that when we `refresh` a managed Customer entity, associated Account entities are automatically refreshed.

all

We can specify more than one cascade type to a relationship annotation by simply listing the cascade types as follows:

```
@OneToMany(fetch=EAGER,
            cascade={CascadeType.PERSIST, CascadeType.MERGE,
                    CascadeType.REMOVE} )
public Collection<Account> getAccounts() { return accounts; }
```

If we want to specify all four cascade types (`persist`, `merge`, `remove` and `refresh`), we can use the `ALL` value as follows:

```
@OneToMany(fetch=EAGER, cascade=CasCadeType.ALL )
public Collection<Account> getAccounts() { return accounts; }
```

Extended Persistence Context

So far, all of our examples have used **transaction scoped** entity managers. Whenever a new transaction started, a new persistence context was created. This was true for both container-managed and application-managed entity managers. In the case of application-managed entity managers, as we have seen, the application explicitly starts and ends the transaction. In the case of container-managed entity managers, by default a transaction starts when a remote client invokes a session bean method. The transaction ends when that session bean method ends. This is the default transaction behavior and as we shall see in Chapter 7, it is possible to configure transaction start and end points to some extent.

Transaction scoped entity managers have a rather restrictive effect when it comes to applications using stateful session beans.

Suppose we have a remote client, `BankClient`, which invokes three methods in turn on a stateful session bean, `BankService`. The first method adds a new Customer entity, the second and third methods update the customers' first and last name respectively. The code fragment from `BankClient` will look like:

```
@EJB
private static BankService bank;
bank.addCustomer(custId, firstName, lastName);
bank.updateFirstName("Michael");
bank.updateLastName("Angelo");
```

Note that because `BankServiceBean` is a stateful session bean, the state of the Customer entity created by the `addCustomer()` method is saved. Consequently the `updateFirstName()` and `updateLastName()` methods can access this Customer entity. A code fragment from `BankServiceBean()` is shown below:

```
@Stateful
public class BankServiceBean implements BankService {
    @PersistenceContext(unitName="BankService")
    private EntityManager em;
    private Customer cust;
    public void addCustomer(int custId, String firstName,
                            String lastName) {
        cust = new Customer();
        cust.setId(custId);
        cust.setFirstName(firstName);
        cust.setLastName(lastName);
        em.persist(cust);
    }
    public void updateFirstName(String firstName) {
        cust.setFirstName(firstName);
        Customer mergedCust = em.merge(cust);
    }
    public void updateLastName(String lastName) {
        cust.setLastName(lastName);
        Customer mergedCust = em.merge(cust);
    }
```

Because each method invocation is a separate transaction, a new persistence context is created at the start of each method call and is closed as soon as the method ends. Consequently the Customer entity `cust` created by the `addCustomer()` method becomes detached as soon as `addCustomer()` finishes. This means the `updateFirstName()` and `updateLastName()` methods must perform a `merge` operation in order to create a managed entity:

```
Customer mergedCust = em.merge(cust);
```

To enable a persistence context to span more than one transaction, EJB 3 provides a container-managed **Extended Persistence context**. This only applies to stateful session beans. The persistence context starts as soon as we perform a JNDI look-up or dependency injection on the stateful session bean. The persistence context lasts until the stateful session bean is destroyed, usually by a `@Remove` method of the bean. As a result, entities created by a session bean method remain managed throughout the life of the session bean and there is no need to perform any `merge` operations.

To indicate that a persistence context is extended we add the `type=PersistenceContextType.EXTENDED` element to the `@PersistenceContext` annotation:

```
@PersistenceContext(unitName="BankService",
                    type=PersistenceContextType.EXTENDED)
```

We can now modify the `updateFirstName()` and `updateLastName()` methods, removing the `merge` step:

```
public void updateFirstName(String firstName) {
    cust.setFirstName(firstName);
}
public void updateLastName(String lastName) {
    cust.setLastName(lastName);
}
```

Entity LifeCycle Callback Methods

We saw session bean callback methods in Chapter 2. Callback methods are also provided for events in an entity's lifecycle. The callback events are:

- `PostLoad`
- `PrePersist`
- `PostPersist`
- `PreRemove`
- `PostRemove`
- `PreUpdate`
- `PostUpdate`

An entity can define a method corresponding to one or more of these events by prefixing the method with a corresponding annotation. So a `PostLoad` method would be prefixed by `@PostLoad` and so on.

The `PostLoad` callback method is invoked by the entity manager after an entity has been loaded by a query, or after a `refresh` operation. For example, in the Customer entity we could add a `PostLoad` method as follows:

```
@PostLoad
public void postLoad(){
    System.out.println("customer.postLoad method invoked");
}
```

Note we do not have to name our method `postLoad()`. Any name is acceptable provided the method returns a `void`, has no arguments and does not throw a checked exception. This applies to any of the entity lifecycle callback methods.

The `PrePersist` event occurs immediately after a `persist()` method is invoked, but before any database insert. A `PrePersist` event also occurs if a `merge` operation causes the creation of a new, managed entity instance. If an entity is persisted as a result of a cascade then the `PrePersist` event also occurs.

The `PostPersist` event occurs immediately after a database insert. This usually occurs at transaction commit time, but can occur after an explicit flush or if the entity manager decides to flush. In the latter two cases a rollback could still occur after a `PostPersist`.

The `PreRemove` event occurs immediately after a remove method is invoked but before any database delete. If an entity is removed as a result of a cascade then the `PreRemove` event also occurs.

The `PostRemove` event occurs immediately after a database delete. This usually occurs at transaction commit time, but can occur after an explicit flush or if the entity manager decides to flush. In the latter two cases a rollback could still occur after a `PostRemove`.

A `PreUpdate` event occurs immediately before a database update. A database update can occur when an entity is updated or at transaction commit time, but can also occur after an explicit flush or if the entity manager decides to flush.

A `PostUpdate` event occurs immediately after a database update.

An entity can have callback methods for any number of lifecycle events, but for any one lifecycle event there can be at most only one callback method. However the same callback method can be invoked for two or more lifecycle events.

Entity Listeners

An alternative to placing entity lifecycle callback methods in the entity itself is to place the methods in one or more separate classes. These classes are known as **entity listeners**. The code below is an example of an entity listener:

```
public class CustomerListener{
    public CustomerListener() {}
    @PostLoad
    public void postLoad(Customer cust){
        System.out.println(
            "custListener.postLoad method invoked " +
```

```
                cust.getId() );
        }
        @PrePersist
        public void prePersist(Customer cust){
            System.out.println(
                "custListener.prePersist method invoked " +
                cust.getId() );
        }
        @PostPersist
        public void postPersist(Customer cust){
            System.out.println(
                "custListener.postPersist method invoked " +
                cust.getId() );
        }
        @PreUpdate
        public void preUpdate(Customer cust){
            System.out.println(
                "custListener.preUpdate method invoked " +
                cust.getId() );
        }
        @PostUpdate
        public void postUpdate(Customer cust){
            System.out.println(
                "custListener.postUpdate method invoked " +
                cust.getId() );
        }
        @PreRemove
        public void preRemove(Customer cust){
            System.out.println(
                "custListener.preRemove method invoked " +
                cust.getId() );
        }
        @PostRemove
        public void postRemove(Customer cust){
            System.out.println(
                "custListener.postRemove method invoked" +
                cust.getId() );
        }
    }
```

Note that the lifecycle methods within an entity listener take an object as an argument. In our example, a Customer object is the argument; this indicates that the method will only be invoked by lifecycle events occurring on a Customer entity.

We can construct a more general entity listener which applies to all entities. For example:

```
public class BankListener{
    public BankListener() {}
    @PrePersist
    public void prePersist(Object o){
        System.out.println(
            "bankListener.prePersist method invoked " +
             o.getClass().getName() );
    }
}
```

In this case because the argument is of type `Object`, the method will be invoked by a `PrePersist` event on any entity.

To indicate which entity listeners are applicable for a given entity, we need to add the `@EntityListeners` annotation in the entity itself. For example, the code below is a fragment from the Customer entity:

```
@Entity
@EntityListeners({CustomerListener.class, BankListener.class})
public class Customer implements Serializable {
    @PrePersist
    public void prePersist(){
        System.out.println(
            "customer.prePersist method invoked");
    }
    ...
}
```

In our example both `CustomerListener` and `BankListener` are applicable. Note that the entity itself can still have a lifecycle method present. Lifecycle methods are executed in the order in which the entity listeners are listed in the `@EntityListeners` annotation followed by any lifecycle methods in the entity itself. In our example the `@PrePersist` method in `CustomerListener` will be executed first, followed by the `@PrePersist` in `BankListener` and finally the `@PrePersist` method in Customer will be executed last.

A lifecycle callback method is applicable to inherited classes in an inheritance hierarchy. Suppose we have an Account entity with a `@PrePersist` method and a subclass `SavingsAccount`. Then the `@PrePersist` method will be invoked for a persist operations on `SavingsAccount` as well as `Account` itself. We could also define lifecycle callback methods in the inherited classes themselves. Lifecycle callback methods are executed in inheritance hierarchy order, base class before subclass. For example, a `@PrePersist` callback method in `Account` will be executed before any `@PrePersist` method in `SavingsAccount`.

Entity listener classes also apply to inherited classes in an inheritance hierarchy. Suppose we have an Account entity with an associated entity listener class, `AccountListener.class`, as follows:

```
public class AccountListener {
    @PrePersist
    protected void prePersistAccount(Object account) {...}
}
```

Suppose `SavingsAccount` is a subclass of Account with an associated entity listener class, `SavingsAccountListener.class`, as follows:

```
public class SavingsAccountListener {
    @PrePersist
    protected void prePersistSavingsAccount(
                        Object savingsAccount) {...}
}
```

A persist operation on `SavingsAccount` will cause the `prePersistAccount()` and `prePersistSavingsAccount()` methods to be executed in order.

Summary

In this chapter we examined `EntityManager` services in more detail. Application-managed entity managers are provided by the JPA for Java applications which do do not require the services offered by an EJB 3 container but still require persistence. We provided an example of using an application-managed entity manager.

We discussed the concept of managed and detached entities and described a number of `EntityManager` methods such as `merge()`. We described cascade operations.

We covered the extended persistence context which enables a persistence context to span more than one transaction. An extended persistence context can only be used by stateful session beans.

We looked at entity lifecycle callback methods. These methods can be placed in the entity itself; an alternative is to place such methods in separate classes known as entity listeners.

7
Transactions

In this chapter we will cover the following topics:

- ACID properties of a transaction
- Container-managed transactions
- Container-managed transaction demarcation
- Session synchronization interface
- Doomed transactions
- Transaction isolation levels
- Lost update problem
- Optimistic and pessimistic locking
- How EJB 3 implements optimistic locking
- Bean-managed transactions and the user transaction interface

Introduction

Up to now our examples have, for the most part, used transactions by default. Most of our examples have used container-managed transactions. As the name suggests, the transaction lifecycle is controlled by the EJB container. Container-managed transactions use the Java Transaction API, or JTA. Container-managed transactions have a default start and end point, however it is possible to configure these as we shall see later in this chapter. Java SE applications, which run outside an EJB container and so do not have JTA available, must use resource-local transactions. Resource-local transactions use the `EntityManager` interface. The application uses `EntityManager` methods to explicitly start and end transactions. We saw an example of this in Chapter 6.

A transaction is a sequence of one or more steps that add, modify, or delete persistent data. Typically the data is persisted to a database. All steps must succeed in which case the transaction succeeds or is committed. If any one step fails then the transaction as a whole fails. In the case of failure the transaction is rolled back so that the state of the data reverts to the state at the start of the transaction. The classic example of a transaction is a transfer of funds from Bank Account A to Bank Account B. The transaction consists of two steps:

1. Subtract $100 from Account A
2. Add $100 to Account B

For the transaction to succeed, both these steps have to succeed. Were step 1 to succeed and step 2 to fail for any reason (for example Account B does not exist or is currently unavailable), the transaction would undo or rollback step 1.

Transactions are said to have **ACID** properties:

- **Atomicity** — Either all steps of a transaction succeed or none succeed.
- **Consistency** — A transaction either commits in a valid database state or is rolled back to its original valid state.
- **Isolation** — Transaction changes are not visible to other transactions until the transaction commits.
- **Durability** — Committed data is permanent and survives any system crashes.

The atomicity, consistency, and durability properties are desirable in their entirety. However, as we shall see later in this chapter, there are degrees of isolation. The maximum level of isolation may not be the most desirable as there may be a high performance penalty to pay.

Container-Managed Transaction Demarcation

When using Container-managed transactions we need to decide where a transaction starts and where it ends. Container-managed demarcation policies are defined by **transaction attributes**. There are six transaction attributes: SUPPORTS, NOT_SUPPORTED, REQUIRED, REQUIRES_NEW, MANDATORY, NEVER**.**

These can be set on the session bean's class or method. Transaction attributes can also be set for message driven beans as we shall see in Chapter 8.

A Java application method which is not in a transaction may invoke a session bean method. This session bean method execution may span a transaction, so the method start and end will demarcate the transaction. In another scenario a session bean method which is in a transaction may invoke another session bean method. Do we start a new transaction for this second method, or is the second method run within the first method's transaction? The transaction attributes cater for these scenarios.

SUPPORTS

If the caller is in a transaction the annotated method will execute in that transaction. If the caller is not in a transaction, no transaction is created and the method executes outside a transaction context.

NOT_SUPPORTED

If the caller is in a transaction, then that transaction is suspended. No transaction is created and the annotated method executes outside a transaction context. If the caller is not in a transaction, no transaction is created and the annotated method executes outside a transaction context.

REQUIRED

The annotated method is required to execute in a transaction but not necessarily a new transaction. If a transaction is already active then the method will execute in that transaction. If no transaction is active, a new transaction is started. This transaction attribute is the default.

REQUIRES_NEW

In this case annotated methods must start in a new transaction. If the calling method is in a transaction, then that transaction is suspended. The new transaction is either committed or rolled back; the original transaction is then resumed.

MANDATORY

The annotated method must start in the caller's transaction. If the caller is not in a transaction, an exception is thrown.

NEVER

The annotated method must never be invoked in a transaction context. If the caller is in a transaction, an exception is thrown. If the caller is not in a transaction, no transaction is created and the annotated method executes outside a transaction context.

Examples of Transaction Attributes

We will show examples of all the transaction attributes starting with the default, `REQUIRED`.

REQUIRED Example

A `REQUIRED` annotated method is required to execute in a transaction but not necessarily a new transaction. If a transaction is already active then the method will execute in that transaction. If no transaction is active, a new transaction is started.

Suppose we have a stateless session bean, `BankServiceBean`, and we want to create an audit record every time we add a customer to the database by means of the `addCustomer()` method. Suppose this method invokes a method, `addAuditMessage()`, on another stateless session bean `AuditServiceBean`. This method creates and then persists an Audit entity. The Audit entity attributes are simply an identifier and a message string.

The normal sequence of events is:

1. start `addCustomer()` method
2. persist Customer entity
3. start `addAuditMessage()` method
4. persist Audit entity
5. end `addAuditMessage()` method
6. end `addCustomer()` method

A transaction starts at step 1 and ends at step 6. Suppose that a runtime error occurs in the course of step 6. The transaction will be rolled back: consequently as well as no customer there will be no corresponding audit entry in the database. This is because the `addAuditMessage()` method runs within the existing transaction started in step 1. The code for `AuditServiceBean` is shown below:

```
@Stateless
@TransactionAttribute(
        TransactionAttributeType.REQUIRED)
```

```
public class AuditServiceBean implements AuditService {
    @PersistenceContext(unitName="BankService")
    private EntityManager em;
    public void addAuditMessage (int auditId,
                                   String message) {
        Audit audit = new Audit();
        audit.setId(auditId);
        audit.setMessage(message);
        em.persist(audit);
    }
}
```

Note that we have used the `@TransactionAttribute` annotation to specify the transaction attribute. As REQUIRED is the default transaction attribute we did not have to use the annotation, but we have done so to explicitly show the transaction attribute for this bean. We have used the annotation on the class. This means that all `AuditServiceBean`'s methods will run with the specified transaction attribute. It is possible to apply the `@TransactionAttribute` annotation on a method. We shall see an example of this shortly.

The `BankServiceBean.addCustomer()` method will look like:

```
@Stateless
public class BankServiceBean implements BankService {
    private @EJB AuditService audit;

    @PersistenceContext(unitName="BankService")
    private EntityManager em;
    public void addCustomer(int custId, String firstName,
                            String lastName) {
        Customer cust = new Customer();
        cust.setId(custId);
        cust.setFirstName(firstName);
        cust.setLastName(lastName);
        em.persist(cust);
        audit.addAuditMessage(1, "customer add attempt");
        if (custId == 10) {
            throw new RuntimeException(
                "add customer - simulated system failure");
        }
    }
}
```

REQUIRES_NEW Example

A REQUIRES_NEW annotated method must start in a new transaction. If the calling method is in a transaction then that transaction is suspended. The new transaction is either committed or rolled back; the original transaction is then resumed.

Recall in the previous section we had a stateless session bean, BankServiceBean, and we created an audit record every time we added a customer to the database by means of the addCustomer() method. This method invokes the addAuditMessage() method on another stateless session bean, AuditServiceBean.

The normal sequence of events was:

1. start addCustomer() method
2. persist Customer entity
3. start addAuditMessage() method
4. persist Audit entity
5. end addAuditMessage()
6. end addCustomer() method

A transaction starts at step 1 and ends at step 6. Suppose that a runtime error occurs in the course of step 6. If a REQUIRED transaction attribute applies to the addAuditMessage() method, the transaction will be rolled back. Consequently as well as no customer, there will be no corresponding audit entry in the database. Suppose in these circumstances we still want to create an audit entry recording that an attempt was made to add a customer. We can achieve this by setting the transaction attribute of the addAuditMessage() to REQUIRES_NEW. In this way a new transaction is started at the beginning of step 3 and committed at step 5. A subsequent runtime error will not rollback this transaction. The code for AuditServiceBean is shown below:

```
@Stateless
public class AuditServiceBean implements AuditService {
    @PersistenceContext(unitName="BankService")
    private EntityManager em;
    @TransactionAttribute(
        TransactionAttributeType.REQUIRES_NEW)
    public void addAuditMessage (int auditId,
                                 String message) {
        Audit audit = new Audit();
        audit.setId(auditId);
        audit.setMessage(message);
        em.persist(audit);
    }
}
```

We have used the `@TransactionAttribute` annotation on the method. Only the `addAuditMessage()` method runs as a `REQUIRES_NEW` transaction. Other methods run with the default on the class, `REQUIRED`, transaction attribute. An on the method annotation will override an on the class annotation.

NOT_SUPPORTED Example

A `NOT_SUPPORTED` annotated method cannot be invoked in a transactional context. If the caller of a `NOT_SUPPORTED` annotated method is in a transaction then that transaction is suspended. No transaction is created and the annotated method executes outside a transaction context. If the caller is not in a transaction, no transaction is created and the annotated method executes outside a transaction context.

Suppose the `AuditServiceBean` rather than writing to a database writes an audit message to a simple sequential file. In this scenario `AuditServiceBean` cannot support a transaction. If after writing a message to the file we have a failure then we cannot undo the write. If we use the default `REQUIRED` setting for the `AuditServiceBean` then the invoking `BankServiceBean` will rollback the adding of a customer to the database. If we don't want this behavior but want a new customer to be added to the database, even if there is a failure in auditing, we can use the `NOT_SUPPORTED` attribute to the `AuditServiceBean` as follows:

```
@Stateless
public class AuditServiceBean implements AuditService {

    @TransactionAttribute(
        TransactionAttributeType.NOT_SUPPORTED)
    public void addAuditMessage (int auditId,
                                 String message) {
        System.out.println(
            "audit id:" + auditId + " message:" + message);
        if (auditId == 1) {
            throw new RuntimeException(
                "add customer - simulated system failure");
        }
    }
}
```

SUPPORTS Example

A `SUPPORTS` annotated method inherits the transaction environment from the caller. So if the caller is in a transaction, the annotated method will execute in that transaction. If the caller is not in a transaction, the method executes outside a transaction context.

Suppose the `BankServiceBean` has an `addCustomer()` method which, after persisting a customer, invokes the `countQuery()` method. `countQuery()` returns the number of customers in the database. The code is shown below:

```
@Stateless
public class BankServiceBean implements BankService {
    @PersistenceContext(unitName="BankService")
    private EntityManager em;
            ...
    public Long addCustomer(int custId, String firstName,
                            String lastName) {
        Customer cust = new Customer();
        cust.setId(custId);
        cust.setFirstName(firstName);
        cust.setLastName(lastName);
        em.persist(cust);
        Long count = countQuery(custId);
        return count;
    }
    @TransactionAttribute(TransactionAttributeType.SUPPORTS)
    public Long countQuery() {
        return (Long) em.createQuery(
          "SELECT COUNT(c) from Customer c").getSingleResult();
    }
}
```

We have given the `countQuery()` method a transaction attribute of SUPPORTS. This means that `countQuery()` runs in the same transaction as `addCustomer()`. If a runtime failure occurs in `countQuery()`, the persist operation in `addCustomer()` will be rolled back.

We could also invoke `countQuery()` from a non-transactional Java application client, `BankClient` say, as follows:

```
public class BankClient {
    @EJB
    private static BankService bank;
    public static void main(String[] args) {
        ...
        try {
            Long count = bank.countQuery();
            System.out.println("no customers = " + count);
        } catch (Exception e) {
            System.out.println(
                    "Exception querying customers");
        }
        ...
}
```

Because `BankClient` is non-transactional any runtime failure in `countQuery()` will not cause any rollback in `BankClient`. Typically, `SUPPORTS` is used in conjunction with read-only methods.

MANDATORY Example

A `MANDATORY` annotated method must start in the caller's transaction. If the caller is not in a transaction, an exception is thrown.

We will revert to the example in the REQUIRED Example section where `BankServiceBean` creates an audit record every time we add a customer to the database. This is done by invoking the `AuditServiceBean.addAuditMessage()` method.

This time we will give the `addAuditMessage()` method a transaction attribute of `MANDATORY`, as follows:

```
@Stateless
public class AuditServiceBean implements AuditService {
    @PersistenceContext(unitName="BankService")
    private EntityManager em;

    @TransactionAttribute(TransactionAttributeType.MANDATORY)
    public void addAuditMessage (int auditId,
                                        String message) {
        Audit audit = new Audit();
        audit.setId(auditId);
        audit.setMessage(message);
        em.persist(audit);
    }
}
```

We can invoke `addAuditMessage()` from the `BankServiceBean.addCustomer()` method as follows:

```
@Stateless
public class BankServiceBean implements BankService {

    private @EJB AuditService audit;

    @PersistenceContext(unitName="BankService")
    private EntityManager em;

    public void addCustomer(int custId, String firstName,
                              String lastName) {
        Customer cust = new Customer();
        cust.setId(custId);
```

```
        cust.setFirstName(firstName);
        cust.setLastName(lastName);
        em.persist(cust);
        audit.addAuditMessage(1, "customer add attempt");
    }
    ...
}
```

Recall the default transaction attribute for addCustomer() is REQUIRED, so addCustomer() will be in a transaction when it is invoked. The addAuditMessage() method will run in the same transaction as addCustomer(). In this respect addAuditMessage() behaves the same as if it were annotated with REQUIRED attribute.

However, if we attempt to invoke addAuditMessage() from a non-transactional caller such as BankClient, an EJBTransactionRequiredException will be thrown.

MANDATORY is not a frequently used transaction attribute. Typically we would use it when we want to ensure that the annotated method can only be invoked as part of an enveloping transaction.

NEVER Example

A NEVER annotated method must *never* be invoked in a transaction context. If the caller is in a transaction, an exception is thrown. If the caller is not in a transaction, no transaction is created and the annotated method executes outside a transaction context.

We will revert to the example in the NOT_SUPPORTED Example section. AuditServiceBean, rather than writing to a database, writes an audit message to a simple sequential file. Now we will annotate the addAuditMessage() method with the NEVER transaction attribute as follows:

```
@Stateless
public class AuditServiceBean implements AuditService {
    @TransactionAttribute(TransactionAttributeType.NEVER)
    public void addAuditMessage (int auditId,
                                 String message) {
        System.out.println(
            "audit id:" + auditId + " message:" + message);
    }
}
```

Assume that, as previously, we invoke addAuditMessage() from the BankServiceBean.addCustomer() method as follows:

```
@Stateless
public class BankServiceBean implements BankService {
    private @EJB AuditService audit;
    @PersistenceContext(unitName="BankService")
```

```
    private EntityManager em;

       ...
    public void addCustomer(int custId, String firstName,
                            String lastName) {
        Customer cust = new Customer();
        cust.setId(custId);
        cust.setFirstName(firstName);
        cust.setLastName(lastName);
        em.persist(cust);
        try {
            audit.addAuditMessage(1, "customer add attempt");
        } catch (Exception e) {
            System.out.println("Exception auditing customer");
        }
    }
       ...
}
```

Because `addCustomer()` runs in an transactional context, as a consequence of the default `REQUIRED` transaction attribute, invoking `addAuditMessage()` will cause an `EJBException` to be thrown.

However, we can invoke `addAuditMessage()` from a non-transactional caller such as `BankClient`.

`NEVER` is the least used transaction attribute. Typically we would use it when a method writes to a simple text file.

Controlling Container Managed Transactions

When using container managed transactions, the application usually just needs to demarcate any transactions. In general we do not need to explicitly commit or rollback a transaction: the container takes care of this. However sometimes we need to explicitly commit and rollback transactions even when using container managed transactions. We may want to use a cache with stateful session beans and commit the data when the cache is full rather than at the end of each transaction. To do this we need to implement the `SessionSynchronization` interface. This is described in the next subsection. In the course of processing a business method we may need to abort a transaction. This is done using the `SessionContext.setRollbackOnly()` method. This ensures the container will not commit the current transaction. We describe this in the "Doomed Transactions" subsection.

SessionSynchronization Interface

There are occasions when we want the application to have some control over commits and rollbacks in container managed stateful session beans. For example, in a shopping cart application we may decide to cache a running total of purchased items in one transaction and commit them to the database after all items have been purchased. Let's assume that after writing the cache to the database but before committing we decide to rollback the transaction. The database will be rolled backed successfully but the stateful session bean cache, which is non transactional, will still contain the purchased items. We need a mechanism to synchronize the database with the session.

We resolve this by implementing the `SessionSynchronization` interface in the stateful session bean. This interface has the following methods: `afterBegin()`, `beforeCompletion()`, and `afterCompletion()`. The `afterBegin()` method is invoked by the container before it invokes the first business method in a transaction. The `beforeCompletion()` method is invoked by the container after the business method has finished but before the transaction commits. At this point we can update the database with the session cache. This method can also be used to rollback a transaction by invoking the `SessionContext.setRollbackOnly()` method. We will see an example of this in the next section. The `afterCompletion()` method is invoked by the container after the transaction has completed. The method has a `boolean` parameter which is set to true if the transaction was committed and false if the transaction was rolled back.

Below is a code fragment for a stateful session bean, `ShoppingCartBean`, which implements the `SessionSynchronization` interface.

```
@Stateful
    public class ShoppingCartBean implements ShoppingCart,
                                 SessionSynchronization {
    @PersistenceContext(unitName="ShoppingService")
    private EntityManager em;
    private Shopping sc;
    private List<String> items;
    @PostConstruct
    public void initialize() {
        items = new ArrayList<String>();
    }
    public void addItem(String item) {
        items.add(item);
    }
    public void afterBegin() {
        System.out.println("afterbegin method invoked");
    }
    public void beforeCompletion() {
```

```
        System.out.println("beforeCompletion method invoked");
        int itemId = 0;
        for (String item : items) {
            itemId++;
            sc = new Shopping();
            sc.setId(itemId);
            sc.setItem(item);
            em.persist(sc);
        }
    }
    public void afterCompletion(boolean committed) {
        System.out.println("afterCompletion method invoked");
        if (committed == false) {
            System.out.println("committed = false");
            items = null;
        }
    }
}
```

The method `addItem()` adds an individual shopping cart item to the cache, items, which is a `List` of type `String`.

The `beforeCompletion()` method updates the database with the contents of the items cache. Note that each individual item is represented by the Shopping entity. The `afterCompletion()` method is invoked after the transaction has been completed. If the transaction was committed, we don't need to take any action. If however the transaction was rolled back, identified by the `boolean committed` being set to false, we need to manually rollback the items cache by setting it to null.

`ShoppingServiceBean` is an example of how we might invoke the above `ShoppingCartBean`. `ShoppingServiceBean` provides a service where a number of items are added to a shopping cart as one transaction. This is done by the method `doShopping()` which invokes the `ShoppingCart` beans `addItem()` method for adding individual items to the cart. The code is shown below:

```
@Stateful
public class ShoppingServiceBean implements ShoppingService {
    private @EJB ShoppingCart shoppingCart;
    public void doShopping() throws ShoppingCartException {
        shoppingCart.addItem("Bread");
        shoppingCart.addItem("Milk");
        shoppingCart.addItem("Tea");
            ...
        throw new ShoppingCartException();
    }
    ...
```

After adding the items to the cart we can simulate a failure in the `doShopping()` method by throwing a `ShoppingCartException`. An application exception, by default, does not cause a rollback unless we add the annotation `@ApplicationExcep tion(rollback=true)` in the `ShoppingCartException` code:

```
@ApplicationException(rollback=true)
public class ShoppingCartException extends Exception {
    public ShoppingCartException() {}
}
```

Doomed Transactions

We have seen two ways in which a container-managed transaction can be rolled back. One way is when the EJB container throws a `RemoteException` or `RunTimeException` for system exceptions such as database or network connection errors. When a system exception occurs, the container will automatically rollback the transaction. The client application is not expected to handle a system exception. The application will merely catch the system exception and re-throw an `EJBException`.

The second way a container-managed transaction can be rolled back is to throw an application exception with the annotation `@ApplicationException(rollback=true)`. An application exception is an exception that occurs while executing business logic, typically some kind of validation. The client application is expected to handle an application exception. We may optionally decide to rollback an application exception, using the `rollback=true` qualifier, depending on the application's business logic.

The third way a container-managed transaction can be rolled back is to use the `SessionContext` interface `setRollbackOnly()` method. The transaction is not rolled back immediately but on completion of the method. We call this **dooming** a transaction.

We may want to check if a transaction is doomed or not to avoid subsequent resource intensive processing if a transaction is already doomed, for example. The `SessionContext` interface provides the `getRollbackOnly()` method for this purpose.

We can modify the `ShoppingCartBean` to doom a transaction in the `addItem()` method as shown below:

```
@Stateful
public class ShoppingCartBean implements ShoppingCart,
                                SessionSynchronization {
    @Resource SessionContext  ctx;
    @PersistenceContext(unitName="ShoppingService")
```

```
        private EntityManager em;
        private Shopping sc;
        private List<String> items;

        ...
        public void addItem(String item) {
            if (ctx.getRollbackOnly() == false ) { // if true do
                                                   // nothing
                if (item.equals("Milk")) {
                    ctx.setRollbackOnly();
                } else {
                    items.add(item);
                }
            }
        }
```

In our example we have no milk in stock so any transaction requesting milk will be doomed. The line `ctx.setRollbackOnly()` does this. The `getRollbackOnly()` method returns true if the transaction has been doomed, so we avoid adding subsequent items by the line

```
        if (ctx.getRollbackOnly() == false ) {
```

Concurrency and Database Locking

In this section we cover concurrency and database locking topics. Isolation levels deal with concurrency issues that arise when two or more transactions simultaneously operate on the same data. We discuss isolation levels in the next subsection. The "Lost Update Problem" subsection describes a problem that can occur when transactions concurrently update the same data. The lost update problem is resolved by one of two strategies: pessimistic locking and optimistic locking. Optimistic locking is implemented in EJB 3 using version fields: we discuss this in the "Versioning" subsection.

Isolation Levels

We mentioned earlier that isolation was the third of the ACID properties of transactions. Isolation levels deal with concurrency issues that arise when two or more transactions are simultaneously operating on the same data. There are three problems that can occur with concurrent transactions: the **dirty read**, **unrepeatable read,** and **phantom read** problems. Each isolation level successively deals with each of these problems. The strictest isolation level, serializable, deals with all three problems. However, this comes at a high performance cost and a less strict isolation level may be applied in practice.

The **dirty read** problem can be described by the following scenario:

An account entity initially has a balance of 100.

- Transaction 1 updates the balance to 200.
- Transaction 2 reads new balance of 200.
- Transaction 1 rolls back restoring the balance to the original value of 100.
- Transaction 2 now has an incorrect, or dirty, value of 200.

The **unrepeatable read** problem can be described by the following scenario:

- Transaction 1 reads account id and balance for all accounts with balance less than 150.
- Transaction 2 updates an account entity changing its balance from 100 to 200.
- Transaction 1 reads the data again including detailed information such as account name
- An account which appeared in the initial summary read is not present in the second detailed read.

This problem only occurs where there are multiple reads in same transaction. In many cases the reads will be in separate transactions so this problem is fairly rare.

The **phantom** read problem can be described by the following scenario:

- Transaction 1 reads account ID and balance for all accounts with balance less than 150.
- Transaction 2 adds a new account entity with a balance of 100.
- Transaction 1 reads the data again including detailed information such as account name
- A new account now appears in the second detailed read which was not present in the initial summary read.

A phantom read is similar to the unrepeatable read, except that data is inserted rather than updated by the second transaction.

There are four transaction isolation levels: **Read Uncommitted**, **Read Committed**, **Repeatable Read,** and **Serializable**.

The **Read Uncommitted** level will allow transactions to read uncommitted data written by a concurrent transaction and so allow dirty reads. Unrepeatable reads and phantom reads may also occur with this level.

The **Read Committed** level will prevent transactions from reading uncommitted data written by a concurrent transaction. Dirty reads are thus prevented, however unrepeatable reads and phantom reads may still occur.

The **Repeatable Read** level ensures that whenever committed data is read from the database, subsequent reads of the same data in the same transaction will always have the same values as the first read. So the unrepeatable read problem is prevented. Phantom reads can still occur.

The **Serializable** level is the strictest isolation level and prevents dirty reads, unrepeatable reads and phantom reads.

The table below summarizes the effect of isolation levels.

Isolation Level	Dirty Read	Unrepeatable Read	Phantom Read
Read Uncommitted	YES	YES	YES
Read Committed	NO	YES	YES
Repeatable Read	NO	NO	YES
Serializable	NO	NO	NO

These isolation levels have been part of the SQL92 standard for relational database management systems (RDBMS). Most RDBMS do not provide the Read Uncommitted level as this level is not sufficient for nearly all applications. The serializable level, while guaranteeing the highest level of data integrity, is too costly in performance terms, so most RDBMS provide the serializable level as an option but not as a default. The default level is usually Read Committed or Repeatable Read. The Oracle RDBMS, for example, offers Read Committed as a default level and Serializable as an option. (Oracle also provides a non-standard Read Only isolation level).

At the EJB 3 level there are no APIs for setting transaction isolation levels; the specification assumes an EJB 3 compliant container has an isolation level of Read Committed. It is possible to programmatically enforce a Repeatable Read isolation level by explicit read and write locks. This also ensures portability. We shall see examples of read and write locks later in this chapter.

Lost Update Problem

This is a classic problem with transactions that concurrently update the same data. As an example, suppose a Customer entity initially has attributes `firstName='LEONARDO'` and `lastName='DAVINCI'`.

We have two transactions updating this entity in the following sequence:

1. Transaction 1 reads the Customer entity
2. Transaction 2 reads the same Customer entity
3. Transaction 1 updates its copy of `firstName` to `MICHAEL`
4. Transaction 2 updates its copy of `lastName` to `ANGELO`
5. Transaction 2 commits data - value on database is `LEONARDO ANGELO`
6. Transaction 1 commits data - value on database is `MICHAEL DAVINCI`

Transaction 2's update of the `lastName` to `ANGELO` has been lost. This is because the value of `lastName` in transaction 1's copy of the data was `DAVINCI` and transaction 1 committed after transaction 2.

There are two strategies to deal with this problem: **pessimistic locking** and **optimistic locking**. In the case of **pessimistic locking,** a transaction locks the source data for the duration of the transaction. During this period no other transaction can update the data. In our scenario, transaction 1 will lock the Customer entity and only when the transaction has finished can transaction 2 perform its update. Pessimistic locking carries an obvious performance cost and is not really a viable option where a large number of transactions can update the same data.

The alternative strategy is **optimistic locking**. With this strategy no locks are held for the duration of the transaction in the optimistic assumption that no other transaction will concurrently attempt to update the same data. If however a second or subsequent transaction does attempt an update it will be rolled back.

EJB 3 provides an optimistic locking mechanism by means of **versioning** which we will now look at in some detail.

Versioning

Optimistic locking is implemented in EJB 3 by means of version fields. The version field contains the version number, which is incremented whenever a transaction updates an entity. Before committing, a transaction also checks that the version number on the database is the same as the initial version number. If the version numbers are different this means another transaction has updated the data since it was read and an `OptimisticLockException` is thrown and the transaction is rolled back.

Version fields are not automatically generated by the container; rather the application adds a version field to the class and annotates it with `@Version`. So versioning is optional, but highly desirable for any entity which has any likelihood of concurrent updates.

The code for the Customer entity has been modified to include a version field:

```
@Entity
public class Customer implements java.io.Serializable {
    private int id;
    private String firstName;
    private String lastName;
    private int version;
    ...
    @Version
    protected int getVersion() { return version; }
    protected void setVersion(int version) {
        this.version = version;
    }
    ...
    }
```

Note that the version field can have any name as long as it is annotated with `@Version`. Unsurprisingly we have named our version field, `version`. We have used property-based annotation but field-based annotation is also supported. We have chosen an `int` type for `version` however `Integer`, `short`, `Short`, `long`, `Long`, and `Timestamp` types are also supported.

The `updateCustomer()` method in the `BankServiceBean` is modified to catch an `OptimisticLockException`:

```
public void updateCustomer(Customer cust) {
    try {
        Customer mergedCust = em.merge(cust);
    } catch (OptimisticLockException e) {
        throw new EJBException(e);
    }
  }
```

In the above code fragment we attempt to update the Customer entity on the database by the statement:

```
Customer mergedCust = em.merge(cust);
```

Before updating the Customer, the persistence provider will check if another transaction has concurrently updated the Customer entity using the version field as described above. If another transaction has concurrently updated the Customer entity, the persistence provider will rollback the current transaction and throw an `OptimisticLockException`. The above code catches the `OptimisticLockException` and re-throws an `EJBException`, thus aborting the `BankServiceBean` session bean. However rather than re-throwing an `EJBException`, on catching an `OptimisticLockException` the application could attempt to repeat the transaction.

Read and Write Locking

Repeatable read isolation can be achieved using a **read lock** in EJB 3. This is done by means of the `EntityManager.lock()` method. This method takes the entity being read locked as the first parameter and `LockModeType.READ` as the second parameter. The following code fragment read locks the Customer entity `cust`:

```
private EntityManager em;
Customer cust = em.find(Customer.class, custId);
em.lock(cust, LockModeType.READ);
```

The transaction which has obtained the lock can now perform two repeatable reads. If a second transaction updates the data being read between the two repeatable reads then one of the transactions will throw an exception and roll back.

A **write lock** is similar to a read lock except the entities version field is also incremented. A lock mode of `LockModeType.WRITE` is a parameter to the `EntityManager.lock()` method:

```
em.lock(cust, LockModeType.WRITE);
```

Note that to use read or write locks the corresponding entity must have a version field. As we saw in the section on isolation levels, the unrepeatable read problem occurs infrequently, so you will rarely to need to use read or write locks.

UserTransaction Interface

The default behavior is for EJBs to use **container managed transaction (CMT) demarcation**. However it is possible for an EJB to use **bean managed transaction (BMT) demarcation.** With BMT the bean programmatically specifies transaction start and commit points. The Java Transaction API (JTA) is provided for this purpose. Specifically the JTA is implemented by the `javax.transaction.UserTransaction` interface. Although BMT gives the application control over the transaction demarcation, this does mean more complex code and risk of programming errors. For this reason use of BMT in the EJB-tier is not recommended.

Suppose we want to demarcate transactions from the web-tier rather than from the EJB-tier. As the transaction is initiated outside an EJB container we cannot use CMT. In this case we would use JTA to programmatically demarcate any transactions.

As an example, suppose we have a servlet which adds a customer to the database. The user is prompted by an html form for the customer's first and last name. The html form looks like:

The corresponding code is:

```html
<html>
<body>
<hr>
<form method="post" action="addCustomer">
<p>Customer Id: <input type="text" name="customerId"
                        size="3"> </p>
<p>First Name: <input type="text" name="firstName"
                        size="20"> </p>
<p>Last Name: <input type="text" name="lastName"
                        size="20"> </p>
<br>
<p>
<input type="submit" value="addCustomer"/>
<input type="reset" value="cancel"/>
</form>
<br> <a href="index.html">Back to Menu</a>
</body>
</html>
```

When the user submits the form, `AddCustomerServlet` is invoked. This adds the customer to the database and generates an html page to indicate that this has been done. The code for `AddCustomerServlet` follows:

```
public class AddCustomerServlet extends HttpServlet {
    @PersistenceUnit
    private EntityManagerFactory emf;
    @Resource
    private UserTransaction utx;
    public void doPost(HttpServletRequest req ,
                       HttpServletResponse resp)
               throws ServletException, IOException {
        try {
            utx.begin();
        } catch (Exception e) {
            throw new ServletException(e);
        }
        EntityManager em = emf.createEntityManager();
        try {
            resp.setContentType("text/html");
            PrintWriter out = resp.getWriter();
            String stringId = req.getParameter("customerId");
            int id = Integer.parseInt(stringId);
            String firstName = req.getParameter("firstName");
            String lastName = req.getParameter("lastName");
            Customer cust = new Customer();
            cust.setId(id);
            cust.setFirstName(firstName);
            cust.setLastName(lastName);
            em.persist(cust);
            out.println("Added new customer. " +
             "<br><a href=\"index.html\"> Back to menu </a>");
            out.println("</body> </html> ");
            utx.commit();
        } catch (Exception e) {
            try {
                utx.rollback();
            } catch (Exception ee) {}
            throw new ServletException(e);
        } finally {
            em.close();
        }
    }
}
```

First note that we use the `@PersistenceUnit` annotation to inject an `EntityManagerFactory` object into the variable `emf`. We do not need to specify the unit name if only one persistence unit exists. We then use the `EntityManagerFactory.createEntityManager()` method to create an `EntityManager` instance:

```
EntityManager em = emf.createEntityManager();
```

We do **not** use the statement

```
@PersistenceContext(unitName="BankService")
private EntityManager em;
```

which we have used in all our session beans. The reason for this is that an `EntityManager` object is not thread safe. This does not affect session beans as they are single threaded. However a servlet is multi-threaded, so multiple concurrent threads would use the same `EntityManager` object. However an `EntityManagerFactory` object is thread safe, so each thread will create its own `EntityManager` instance.

We use the `@Resource` annotation to inject a `javax.transaction.UserTransaction` object into the variable `utx`. We use the `UserTransaction.begin()` method to begin our transaction:

```
utx.begin();
```

The transaction is committed or rolled back using the `UserTransaction.commit()` and `UserTransaction.rollback()` methods respectively.

We should briefly mention the remaining `UserTransaction` methods.

- `setRollbackOnly()`: This dooms a transaction similar to the `SessionContext.setRollbackOnly()` method we described earlier in the context of session beans.

- `setTransactionTimeout(int)`: This sets the maximum amount of time, in seconds, that a transaction can run before it is aborted.

- `getStatus()`: This returns a constant indicating the status of the transaction. For example, STATUS_ACTIVE indicates the transaction is active, STATUS_MARKED_ROLLBACK indicates the transaction is doomed possibly as a result of a `setRollbackOnly()` call.

Summary

Applications which run under an EJB container have their transactions managed by the container using the Java Transaction API (JTA). Applications which run outside an EJB container can make use of resource-local transactions provided by the `EntityManager` interface. In Chapter 6 we saw an example of resource local transactions. In this chapter we looked at container managed transactions (CMT) using the JTA.

We described the ACID properties a transaction must possess before moving onto CMT transaction attributes. These were `SUPPORTS`, `NOT_SUPPORTED`, `REQUIRED`, `REQUIRES_NEW`, `MANDATORY`, `NEVER`.

We covered the `SessionSynchronization` interface: this provides stateful session beans using CMT control over commits and rollbacks.

We described doomed transactions before moving onto concurrency and locking topics. We discussed isolation levels, the third ACID property. We described three problems that occur with concurrent transactions: the dirty read, unrepeatable read and phantom read problems. The four isolation levels, Read Uncommitted, Read Committed, Repeatable Read, and Serializable, deal with these problems.

We described the lost update problem and saw two ways of dealing with it: pessimistic locking and optimistic locking. In EJB 3 optimistic locking is implemented using versioning.

Finally we looked at bean managed transactions (BMT). The `UserTransaction` interface is provided to implement BMT.

8
Messaging

In this chapter we discuss the Java Message Service API before moving on to Message-driven beans. In particular we will cover:

- Java Message Service (JMS) API
- Point-to-point and publish/subscribe messaging models
- Queue producer and consumer examples
- Topic producer and consumer examples
- Message-driven beans (MDBs)
- MDB activation configuration properties
- MDB lifeCycle
- Sending message confirmation to a client
- MDBs and transactions

Introduction

So far our examples have all been synchronous. For example, a client invokes a session bean which performs some operation while the client waits. When the session bean completes its operation (or operations), control is passed back to the client and the client continues with the remaining operations. For some applications an asynchronous mode of operation is more appropriate, particularly in business to business applications. For example, a client may send a message requesting that an order be processed. The order processing may take some time. Once the message has been sent, the client continues with its operations. The client does not need to wait until the order has been processed. Of course the client will need to be notified in some way that the message has been received and in due course that the order has been processed. Even if the order processing is near instantaneous we may still prefer the asynchronous messaging model because we don't want to couple the client and server systems.

The client is referred to as the **message producer**. The recipient of the message is referred to as a **message consumer**. In certain messaging systems, as we shall see, there may be more than one consumer for a given message.

Message-driven beans are asynchronous message consumers. Typically a message-driven bean, or **MDB**, is based on the **Java Message Service (JMS)** API. Since EJB 2.1 MDBs may implement a messaging system other than JMS. However, we will only cover JMS-based MDBs in this book. All EJB-compliant application servers must support both MDBs and the JMS API.

The JMS API is not only implemented by EJB-compliant application servers, but by many standalone message-oriented-middleware (MOM) products such as IBM MQ. These products provide services such as load balancing and guaranteed message delivery.

Before looking at MDB examples we first take a brief look at the JMS API.

Java Message Service (JMS) API

The JMS API is a set of interfaces and classes that support two basic messaging modes: **point-to-point** and **publish/subscribe**.

In the **point-to-point** mode, the message producer sends a message to a **queue**. The message is read just once by a consumer. After the message is read it is deleted from the queue. There may be more than one potential consumer, however once a message has been read by one consumer it cannot be read by any of the remaining consumers. A consumer does not need to be registered with the queue at the time the message is sent. If no other consumer has read the message then the consumer can read the message as soon as it (the consumer) accesses the queue.

In the **publish/subscribe** mode, the message producer sends a message to a **topic**. One or more consumers register with, or subscribe to, a topic. The message remains in the topic until all consumers have read the message. If a consumer has not registered with a topic at the time the message is sent then it will be unable to subsequently read the message.

A queue or topic consumer can be either synchronous or asynchronous. If a consumer is synchronous and there is no message in the topic or queue, the consumer will hang until the message arrives. An asynchronous consumer is registered as a listener to the queue or topic; when a message arrives the listener is notified and the asynchronous consumer then reads the message.

JMS does not guarantee message delivery in any order. So if a producer sends a part A message followed by a part B message, you cannot assume the consumer will read part A before part B.

There are a number of steps a queue or topic producer must undertake and a number of steps depending on whether the consumer is synchronous or asynchronous. These are best described by examples.

Queue Producer and Consumer Examples

We will start with a Java application, `QueueProducer.java`, acting as a JMS queue producer. Recall how we use a queue in point-to-point messaging. Point-to-point is used when you want a particular recipient, or consumer, to process a given message just once. An example would be a message to process an order—we would want only one recipient to process this order and that recipient should process the order only once. Another example is an instruction to open a specific bank account. `QueueProducer.java` just sends a message `Open bank account 1234` to the queue. The code is shown below:

```
package ejb30.client;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.jms.*;

public class QueueProducer {
    public static void main(String[] args) {
        ConnectionFactory cf = null;
        Connection conn = null;
        Queue queue = null;
        try {
            InitialContext ctx = new InitialContext();
            cf = (ConnectionFactory)
                    ctx.lookup("BankServiceConnectionFactory");
            queue = (Queue) ctx.lookup("BankServiceJMSQueue");
        } catch (NamingException e) {
            e.printStackTrace();
        }
        try {
            conn = cf.createConnection();
            Session sess = conn.createSession(false,
                            Session.AUTO_ACKNOWLEDGE);
```

```
          MessageProducer mp = sess.createProducer(queue);
          TextMessage msg = sess.createTextMessage(
                         "Open bank account 1234");
          mp.send(msg);
          System.out.println("JMS Message sent");
      } catch (JMSException ex) {
          System.out.println(
              "JMS Exception:" + ex.getMessage() );
      } finally {
          try {
              conn.close();
          } catch (JMSException ex) {}
      }
    }
  }
```

The first step is to lookup a **connection factory**. In our example BankServiceConnectionFactory is the connection factory, and because this is a standalone Java application we need to use JNDI to look it up. It is possible for a session bean to act as a queue producer in which case we can use dependency injection to obtain a connection factory. We shall see an example of this later in this chapter. We will also describe how we create a connection factory later in the section "Running the Queue Producer and Synchronous Queue Consumer Examples".

Next we use JNDI to lookup the queue. In our example BankServiceJMSQueue is the queue. Again we shall shortly see how this is created.

Next we create a Connection object using the connection factory createConnection() method:

```
    conn = cf.createConnection();
```

The next step is to create a Session object using the Connection.createSession() method:

```
    Session sess = conn.createSession(false,
                       Session.AUTO_ACKNOWLEDGE);
```

The first parameter, false, indicates that we do not want to use transactions. Session.AUTO_ACKNOWLEDGE indicates that the consumer should send an acknowledgment to the JMS server immediately on receipt. However as the current program is a producer and not a consumer, this parameter will be ignored.

We now create a `MessageProducer` object using the `Session.createProducer()` method:

```
MessageProducer mp = sess.createProducer(queue);
```

We create a `TextMessage` object which holds a message in the form of a `String` object:

```
TextMessage msg = sess.createTextMessage(
"Open bank account 1234");
```

`TextMessage` is one of the message types that JMS supports. Alternatively, we could use one of the following JMS Message types:

- `MapMessage` contains a Java `Map`
- `ObjectMessage` contains a serializable Java object
- `BytesMessage` contains an array of bytes
- `StreamMessage` contains a stream of primitive Java types

We would use one of these message types when we want to send data, and not just a message, for the recipient to process.

Finally we use the `MessageProducer.send()` method to send the message to the queue:

```
mp.send(msg);
```

We have chosen to send a single message to the queue. We can of course send multiple messages by repeatedly invoking the `send()` method.

Finally, we release resources by closing the `Connection` object after we have finished.

Synchronous Queue Consumer Example

In this example a Java application, `SynchQueueConsumer`, simply synchronously reads a message from the same queue used in the previous example. Because a queue is being used, `SynchQueueConsumer` can read this message only once. Once the message has been read, it is removed from the queue and no other consumer can read the message. Because `SynchQueueConsumer` is a synchronous consumer, if there is no message in the queue `SynchQueueConsumer` will hang until the message arrives. A synchronous consumer operates in a *pull* manner; it proactively polls the queue for a message.

```
package ejb30.client;
import javax.naming.Context;
import javax.naming.InitialContext;
```

```
import javax.naming.NamingException;
import javax.jms.*;
public class SynchQueueConsumer{
    public static void main(String[] args) {
        ConnectionFactory cf = null;
        Connection conn = null;
        Queue queue = null;
        try {
            InitialContext ctx = new InitialContext();
            cf = (ConnectionFactory)
                    ctx.lookup("BankServiceConnectionFactory");
            queue = (Queue) ctx.lookup("BankServiceJMSQueue");
        } catch (NamingException e) {
            e.printStackTrace();
        }
        try {
            conn = cf.createConnection();
            Session sess = conn.createSession(false,
                               Session.AUTO_ACKNOWLEDGE);
            MessageConsumer mcon = sess.createConsumer(queue);
            conn.start();
            Message message = mcon.receive(0);
            TextMessage msg = (TextMessage) message;
            System.out.println(
                    "JMS Message:" + msg.getText() );

        } catch (JMSException ex) {
            System.out.println(
                    "JMS Exception:" + ex.getMessage() );
        } finally {
            try {
                conn.close();
            } catch (JMSException ex) {}
        }
    }
}
```

As with our queue producer program, we use JNDI to look up the
BankServiceConnectionFactory connection factory and the BankServiceJMSQueue
queue. Again, as before, we create a Connection and a Session object.

We next create a `MessageConsumer` object using the `Session.createConsumer()` method:

```
MessageConsumer mcon = sess.createConsumer(queue);
```

We start the connection, and then use the `MessageConsumer.receive()` method to receive any message from the queue:

```
conn.start();
Message message = mcon.receive(0);
```

Recall that a synchronous consumer will hang if there is no message in the queue. The `receive()` method argument indicates the timeout in milliseconds. If the value is 0 the program will hang indefinitely.

Next we cast the message to a `TextMessage` type and print out the result:

```
TextMessage msg = (TextMessage) message;
System.out.println( "JMS Message:" + msg.getText() );
```

We have read a single message from the queue and assumed it is a `TextMessage` type. In a production version of the program we would read all messages from the queue and have some mechanism to indicate the last message, possibly a null message or `End of Message` message. We would filter out any messages which are not of type `TextMessage` or handle these types separately.

Finally, as before, we close the `Connection` object.

Running the Queue Producer and Synchronous Queue Consumer Examples

Before we can run the examples, we need to create the connection factory and queue. This process is application server-dependent. In the case of GlassFish there are two ways we can do this. First we can logon to the Administration Console by entering `http://localhost:4848/` in the browser. Then use the `Resources` option to manually set up the connection factory and queue.

Alternatively we can create an Ant script which uses the GlassFish `asadmin` utility to perform the setup from the command line:

```
<target name="create-connection-factory">
  <exec executable="${glassfish.home}/bin/asadmin"
          failonerror="true"
          vmlauncher="false">
  <arg line="create-jms-resource --user admin
     --passwordfile adminpassword
```

```
          --restype javax.jms.ConnectionFactory --enabled=true
          BankServiceConnectionFactory"/>
     </exec>
  </target>
  <target name="create-queue">
    <exec executable="${glassfish.home}/bin/asadmin"
          failonerror="true"
          vmlauncher="false">
    <arg line="create-jms-resource --user admin
      --passwordfile adminpassword
      --restype javax.jms.Queue --enabled=true
      --property Name=BankServiceJMSQueue BankServiceJMSQueue"/>
    </exec>
  </target>
```

We would also have corresponding targets for deleting the connection factory and queue.

Note in the statement,

```
<arg line="create-jms-resource --user admin
   --passwordfile adminpassword
   --restype javax.jms.Queue --enabled=true
   --property Name=BankServiceJMSQueue BankServiceJMSQueue"/>
```

the clause `Name=BankServiceJMSQueue` refers to the *value* of the queue, whereas the last item in the line is the JNDI name of the queue. Normally we don't need to distinguish between the two items and both can be set to be the same. If the queue had a JNDI name of `BankServiceJMSQueue` and a value of `BankServiceJMSQueueValue`, then the statement would read:

```
<arg line="create-jms-resource --user admin
   --passwordfile adminpassword
   --restype javax.jms.Queue --enabled=true
   --property Name=BankServiceJMSQueueValue
     BankServiceJMSQueue"/>
```

Now run the `QueueProducer` program. After it completes, run `SynchQueueConsumer`. The program will read the message. Recall that in point-to-point mode a consumer does not need to be registered with the queue at the time the message is sent. If we run `SynchQueueConsumer` again, the program will hang until `QueueProducer` is run again. This is normal behavior for a synchronous queue consumer.

An Asynchronous Queue Consumer Example

In this example a Java application, `AsynchQueueConsumer`, simply asynchronously reads a message from the same queue used in the previous examples. This program differs from the synchronous consumer example in that it registers with the JMS server a listener object that implements the `MessageListener` interface. The `MessageListener` interface has an `onMessage()` method which the listener object must implement. When a message arrives on the queue, the JMS server will invoke the `onMessage()` method. So, in contrast to the synchronous case where a message is *pulled* from the queue, in the asynchronous case the message is *pushed* from the queue. The code for AsynchQueueConsumer is listed below:

```
public class AsynchQueueConsumer{
    public static void main(String[] args) {
        ConnectionFactory cf = null;
        Connection conn = null;
        Queue queue = null;
        MessageListener listener = null;
        try {
            listener = new TextMessageListener();
            InitialContext ctx = new InitialContext();
            cf = (ConnectionFactory)
                    ctx.lookup("BankServiceConnectionFactory");
            queue = (Queue) ctx.lookup("BankServiceJMSQueue");
        } catch (NamingException e) {
            e.printStackTrace();
        }
        try {
            conn = cf.createConnection();
            Session sess = conn.createSession(false,
                            Session.AUTO_ACKNOWLEDGE);
            MessageConsumer mcon = sess.createConsumer(queue);
            mcon.setMessageListener(listener);
            conn.start();
        } catch (JMSException ex) {
            System.out.println(
                    "JMS Exception:" + ex.getMessage() );
        } finally {
            try {
                conn.close();
            } catch (JMSException ex) {}
        }
    }
}
```

We have highlighted where the program differs from a synchronous consumer.

As with our queue producer program, we use JNDI to look up the
`BankServiceConnectionFactory` connection factory and the `BankServiceJMSQueue`
queue. Again, as before, we create `Connection`, `Session` and `MessageConsumer`
objects. Our listener object is named `TextMessageListener`. We shall see
the code for `TextMessageListener` shortly. We create an instance of the
`TextMessageListener` class and register it as a listener using the `MessageConsumer`.
`setMessageListener()` method :

```
listener = new TextMessageListener();
mcon.setMessageListener(listener);
```

Next we start the connection. From this point messages are delivered to the listener.
Finally we close the connection.

We need to create the `TextMessageListener` class which must implement the
`MessageListener` interface. This interface has just one method, `onMessage()`,
which `TextMessageListener` must implement. When a message arrives on
the queue, the JMS server will invoke the `onMessage()` method. The code for
`TextMessageListener` is shown below:

```
public class TextMessageListener implements MessageListener {
    public void onMessage(Message message) {
        try {
            TextMessage msg = (TextMessage) message;
            System.out.println(
                    "JMS Message:" + msg.getText() );
        } catch (JMSException ex) {
            ex.printStackTrace();
        }
    }
}
```

The `onMessage()` method has an argument of type `Message`, namely the incoming
message. We cast the message to a `TextMessage` type before printing it out. Again in
a production version, we would be more defensive and either cater for, or filter out,
other JMS Message types.

Running the Asynchronous Queue Consumer Example

First run the `QueueProducer` program. After it completes run `AsynchQueueConsumer`. The program will read the message. Run `AsynchQueueConsumer` again. Unlike the synchronous version, the program will not hang. It will terminate with no message being printed out.

Topic Producer and Consumer Examples

We will start with a Java application, `TopicProducer.java`, acting as a JMS topic producer. Recall we use a topic in publish/subscribe messaging. Publish/subscribe is used for one-to-many delivery of messages. Typically this mode is used for broadcasting information to a large number of consumers who may not even be known to the topic producer. An example in the banking industry would be the notification of an imminent interest rate cut. `TopicProducer.java` just sends a message `"Savings account interest rate cut by 0.5 %"`

The code is very similar to `QueueProducer`.

```java
public class TopicProducer {
    public static void main(String[] args) {
        ConnectionFactory cf = null;
        Connection conn = null;
        Topic topic = null;
        try {
            InitialContext ctx = new InitialContext();
            cf = (ConnectionFactory)
                    ctx.lookup("BankServiceConnectionFactory");
            topic = (Topic) ctx.lookup("BankServiceJMSTopic");
        } catch (NamingException e) {
            e.printStackTrace();
        }
        try {
            conn = cf.createConnection();
            Session sess = conn.createSession(false,
                            Session.AUTO_ACKNOWLEDGE);
            MessageProducer mp = sess.createProducer(topic);
            TextMessage msg = sess.createTextMessage(
                "Savings account interest rate cut by 0.5 %");
            mp.send(msg);
            System.out.println("JMS Message sent");
        } catch (JMSException ex) {
```

```
            System.out.println(
                    "JMS Exception:" + ex.getMessage() );
        } finally {
            try {
                conn.close();
            } catch (JMSException ex) {}
        }
    }
}
```

We lookup the same connection factory, `BankServiceConnectionFactory`, that we used in the `QueueProducer` example. However, we could create a separate connection factory dedicated to topics if we so wished. The advantage of having separate connection factories is that the application server administrator can individually configure each connnection factory. Instead of a queue we look up a topic, in our example the topic is `BankServiceJMSTopic`:

```
topic = (Topic) ctx.lookup("BankServiceJMSTopic");
```

We create `Connection` and `Session` objects as before. We create a `MessageProducer` object using the `Session` object. However, this time we supply the topic as an argument:

```
MessageProducer mp = sess.createProducer(topic);
```

As before, we use the `send()` method of the `MessageProducer` object to send the message to the topic. Finally we close the connection.

Synchronous Topic Consumer Example

In this example a Java application, `SynchTopicConsumer`, simply synchronously reads a message from the same topic used in the previous example.

```
public class SynchTopicConsumer{
    public static void main(String[] args) {
        ConnectionFactory cf = null;
        Connection conn = null;
        Topic topic = null;
        try {
            InitialContext ctx = new InitialContext();
            cf = (ConnectionFactory)
                    ctx.lookup("BankServiceConnectionFactory");
            topic = (Topic) ctx.lookup("BankServiceJMSTopic");
        } catch (NamingException e) {
            e.printStackTrace();
```

```
            }
            try {
                conn = cf.createConnection();
                Session sess = conn.createSession(false,
                            Session.AUTO_ACKNOWLEDGE);
                MessageConsumer mcon = sess.createConsumer(topic);

                conn.start();
                Message message = mcon.receive(0);
                TextMessage msg = (TextMessage) message;
                System.out.println(
                        "JMS Message:" + msg.getText() );

            } catch (JMSException ex) {
                System.out.println(
                        "JMS Exception:" + ex.getMessage() );
            } finally {
                try {
                    conn.close();
                } catch (JMSException ex) {}
            }
        }
    }
```

The code for this program is almost the same as for `SynchQueueConsumer` described earlier. One difference is that we perform a JNDI lookup of `BankServiceJMSTopic` rather than a queue:

```
    topic = (Topic) ctx.lookup("BankServiceJMSTopic");
```

The second difference is that we use the topic as an argument when creating the `MessageConsumer` object:

```
    MessageConsumer mcon = sess.createConsumer(topic);
```

Running the Topic Producer and Synchronous Topic Consumer Examples

Before we can run the examples we need to create the topic. As with connection factories and queues, this process is application server-dependent. In the case of GlassFish we can do this manually by means of the Administration Console, or by creating an Ant script which uses the GlassFish `asadmin` utility to perform the setup from the command line:

```
<target name="create-topic">
    <exec executable="${glassfish.home}/bin/asadmin"
        failonerror="true"
        vmlauncher="false">
```

```
        <arg line="create-jms-resource --user admin
          --passwordfile adminpassword --restype javax.jms.Topic
          --enabled=true --property Name=BankServiceJMSTopic
          BankServiceJMSTopic"/>
      </exec>
  </target>
```

Now run the `TopicProducer` program. After it completes, run
`SynchTopicConsumer`. This program will hang unlike the synchronous queue
consumer example shown earlier. This is because `SynchTopicConsumer` was not
registered with the topic at the time that `TopicProducer` sent the message. If we run
`TopicProducer` again then the hanging `SynchTopicConsumer` will read the message
and terminate.

An Asynchronous Topic Consumer Example

In this example a Java application, `AsynchTopicConsumer`, simply asynchronously
reads a message from the same topic used in the previous example. The code for this
program is very similar to the asynchronous queue consumer program described
earlier. We also need to create an instance of a `MessageListener` class and register it
as a listener. The code is shown below:

```
public class AsynchTopicConsumer{
    public static void main(String[] args) {
        ConnectionFactory cf = null;
        Connection conn = null;
        Topic topic = null;
        MessageListener listener = null;
        try {
            listener = new TextMessageListener();
            InitialContext ctx = new InitialContext();
            cf = (ConnectionFactory)
                  ctx.lookup("BankServiceConnectionFactory");
            topic = (Topic) ctx.lookup("BankServiceJMSTopic");
        } catch (NamingException e) {
            e.printStackTrace();
        }
        try {
            conn = cf.createConnection();
            Session sess = conn.createSession(false,
                            Session.AUTO_ACKNOWLEDGE);
            MessageConsumer mcon = sess.createConsumer(topic);
            mcon.setMessageListener(listener);
            conn.start();
```

```
            try
            {
                Thread.sleep(30000);
            } catch (Exception e) {}
        } catch (JMSException ex) {
            System.out.println(
                    "JMS Exception:" + ex.getMessage() );
        } finally {
            try {
                conn.close();
            } catch (JMSException ex) {}
        }
    }
}
```

We use JNDI to look up the `BankServiceConnectionFactory` connection factory and the `BankServiceJMSTopic` topic. As before, we create `Connection`, `Session`, and `MessageConsumer` objects. We also create an instance of the `TextMessageListener` class and register it as a listener. However the program differs from the queue example in that after starting the connection we let the program sleep for 30 seconds.

The reason for this is, you may recall, that a consumer must be registered with the topic at the time the message is sent. In this way, providing we run the topic producer within 30 seconds of the consumer starting, we will be able to read any message in the topic. Alternatively we could have created an infinite loop which would require some sort of end of message from the producer to break from the loop.

Running the Asynchronous Topic Consumer Example

First run the `AsynchTopicConsumer` program. In another window immediately afterwards, run a second copy of `AsynchTopicConsumer`. Finally, in a third window run the `TopicProducer` program. All three programs should be started within 30 seconds of each other. Because this is a topic and a topic can have any number of subscribers, both copies of `AsynchTopicConsumer` will receive the message sent by `TopicProducer`.

Motivation for Message-Driven Beans

In enterprise scale environments we would like our messaging systems to take advantage of the services offered by an EJB container. These include security, transactions, concurrency, and the scalability of the EJB model. A session bean cannot be a message consumer since its business methods must be synchronously invoked by a client. However, a session bean can act as a message producer as we shall see. To meet these requirements, EJB provides message-driven beans (MDBs). An added bonus of MDBs is that much of the boilerplate code of creating and starting connections and creating sessions and listeners is dispensed with.

MDBs are EJB components which act as asynchronous message consumers. Since EJB 2.1, MDBs can support non-JMS messaging systems as well as JMS messaging systems, but we will describe only JMS-based MDBs in this book.

MDBs do not have remote or local business interfaces. JMS-based MDBs must implement the `MessageListener` interface. As we have seen, this interface has just one method, `onMessage()`. This method, as well as receiving the message, also performs any business logic not unlike session beans. MDBs can use point-to-point (queue) or publish/subscribe (topic) modes. MDBs are stateless. Like session beans, MDBs can invoke other session beans and can interact with entities by means of the `EntityManager`.

A Simple Message-Driven Bean Example

For this example we revert to the banking scenario. Now we will get a session bean to send a message to a JMS queue requesting that a new customer be added to the database. The message will be a Customer object with `id`, `firstName`, and `lastName` attributes. On receipt of the message, the MDB simply adds the customer to the database.

A Session Bean Queue Producer

We have modified the `BankServiceBean.addCustomer()` method so that it acts as a queue producer. The `addCustomer()` method is invoked synchronously by a client in the usual manner. The code for `addCustomer()` is shown below:

```
@Stateless
public class BankServiceBean implements BankService {
    @Resource(mappedName="BankServiceConnectionFactory")
    private ConnectionFactory cf;
    @Resource(mappedName="BankServiceJMSQueue")
    private Queue queue;
    ...
```

```
    public void addCustomer(int custId, String firstName,
                            String lastName) {
    try{
        Customer cust = new Customer();
        cust.setId(custId);
        cust.setFirstName(firstName);
        cust.setLastName(lastName);
        Connection conn = cf.createConnection();
        Session sess = conn.createSession(false,
                            Session.AUTO_ACKNOWLEDGE);
        MessageProducer mp = sess.createProducer(queue);
        ObjectMessage objmsg = sess.createObjectMessage();
        objmsg.setObject(cust);
        mp.send(objmsg);
        conn.close();
    } catch (JMSException e) {
        e.printStackTrace();
    }
}
...
```

As we have a session bean running within an EJB container, we can dispense with JNDI lookups and use the `@Resource` annotation to inject the connection factory and queue into corresponding variables. We assume that a connection factory with JNDI name `BankServiceConnectionFactory` has already been created. We also assume that a queue with JNDI name `BankServiceJMSQueue` has already been created. We saw in the section "Running the Queue Producer and Synchronous Queue Consumer Examples" how to create the `BankServiceConnectionFactory` and `BankServiceJMSQueue` in the case of the GlassFish application server. Here we choose to use the same connection factory and JMS queue, but we could create a connection factory and JMS queue specifically for MDBs.

We create a customer object and set its attributes.

We create `Connection`, `Session`, and `MessageProducer` objects in the same way as in the Java application client queue producer described earlier. However, instead of creating a `TextMessage` object, we create an `ObjectMessage` object using the `Session.createObjectMessage()` method:

```
    ObjectMessage objmsg = sess.createObjectMessage();
```

We use the `ObjectMessage.setObject()` method to assign the customer object to the message:

```
    objmsg.setObject(cust);
```

Finally we use the `MessageProducer.send()` method to send the message to the queue.

A Message-Driven Bean Queue Consumer

We will create an MDB named `ProcessCustomerBean` which reads a customer object from the message queue and persists the customer to the database. The code is shown below:

```java
package ejb30.MDB;
import javax.persistence.EntityManager;
import ejb30.entity.Customer;
import javax.persistence.PersistenceContext;
import javax.ejb.EJBException;
import javax.ejb.MessageDriven;
import javax.ejb.ActivationConfigProperty;
import javax.jms.MessageListener;
import javax.jms.Message;
import javax.jms.ObjectMessage;

@MessageDriven(activationConfig = {
@ActivationConfigProperty(propertyName = "destinationType",
                          propertyValue = "javax.jms.Queue")
}, mappedName="BankServiceJMSQueue" )
public class ProcessCustomerBean implements MessageListener {
    @PersistenceContext(unitName="BankService")
    private EntityManager em;
    public void onMessage(Message message) {
        try {
            if (message instanceof ObjectMessage) {
                ObjectMessage objmsg = (ObjectMessage)
                                        message;
                Customer cust = (Customer) objmsg.getObject();
                Customer mergedCust = em.merge(cust);
                System.out.println("MDB: Customer persisted");
            } else {
                System.out.println("Wrong type of message");
            }
        } catch(Exception e) {
            throw new EJBException(e);
        }
    }
}
```

Recall there is no remote or local business interface for MDBs. We use the `@MessageDriven` annotation to indicate this is an MDB. We also use the `@ActivationConfigProperty` to specify a configuration property, namely `destinationType`. Configuration properties can be application server dependent, however there are a number, such as `destinationType`, which are common to all EJB 3 compliant application servers. We will see a few more of these configuration properties later in this chapter. We use the `mappedName` element to specify the JNDI name of the queue being read from.

We only describe the use of annotations with MDBs in this book. However, as with the rest of EJB 3, it is possible to use XML deployment descriptors with MDBs either as an alternative to, or in conjunction with annotations.

The MDB must implement the `MessageListener` interface. As with asynchronous Java application message consumers, we must implement an `onMessage()` method. This method has no return value. Within the method, we ignore any messages which are not of `ObjectMessage` type. We cast the message to `ObjectMessage` type, then obtain the customer object using the `ObjectMessage.getObject()` method. Finally we persist the customer object using the `EntityManager.merge()` method.

MDB Activation Configuration Properties

We have already seen the `destinationType` activation configuration property. In this section we will describe a few more configuration properties which are common to all EJB 3 compliant application servers.

acknowledgeMode

We have already come across message acknowledgment in the context of JMS Java applications. In the context of MDBs, without message acknowledgment from the EJB container to the JMS server, the server may re-send messages to the container. There are two values for `acknowlegeMode`

- `Auto-acknowledge`—the EJB container sends an acknowledgement as soon as the message is received.

- `Dups-ok-acknowledge`—the EJB container sends an acknowledgment any time after it has received the message. Consequently duplicate messages may be sent by the JMS server. The MDB must be programmed to handle duplicates.

An example of the syntax is:

```
@ActivationConfigProperty(propertyName = "acknowledgeMode",
                          propertyValue = "Auto-acknowledge")
```

The `acknowledgeMode` property is only needed if the MDB is using bean-managed transactions (BMT).

There is no need for this property if the MDB is using Container-Managed Transactions (CMT). With CMT if the transaction succeeds, the EJB container will send a message acknowledgment to the JMS server in any case. If the transaction fails, the message is not acknowledged and the message is put back in the queue and re-sent.

We have more to say about MDBs and transactions later in this chapter.

subscriptionDurability

You may recall in the publish/subscribe mode, one or more consumers register with a topic. The message remains in the topic until all consumers have read the message. Normally if a consumer has not connected to and registered with a topic at the time the message is sent then it will be unable to subsequently read the message. This default case is described as **non-durable**. In the case of MDBs, topic connection and registration usually occurs at deployment time, so if a message has already been sent to a topic, the MDB will not be able to read it. If, once we have deployed the MDB, the EJB container crashes after the message has been sent, the connection between the EJB container and JMS server will be lost. Consequently the MDB will not be able to read the message after the EJB container is restarted. However, with a **durable** subscription, the JMS server will hold the message until the EJB container restarts in the event of a crash.

System performance is improved with a nondurable subscription but at the cost of reliability.

An example of the syntax is:

```
@ActivationConfigProperty(propertyName =
                          "subscriptionDurability",
                          propertyValue = "NonDurable")
```

or

```
@ActivationConfigProperty(propertyName =
                          "subscriptionDurability",
                          propertyValue = "Durable")
```

messageSelector

A message selector allows the MDB to filter out certain messages. For example, we would like to modify the MDB described in the previous section to only process messages which relate to priority customers.

We do this as follows:

```
@ActivationConfigProperty(propertyName = "messageSelector",
                          propertyValue = "Priority = 'HIGH'")
```

The producer would need to set the priority using the `ObjectMessage.setStringProperty()` method as follows:

```
objmsg.setStringProperty("Priority", "HIGH" );
```

The `propertyValue` clause can be an expression similar to one in an SQL `WHERE` clause.

MessageDrivenContext

An MDB may decide to implement the `MessageDrivenContext` interface. This interface is similar to the `SessionContext` interface provided for session beans, and provides information about an MDBs environment. We use the `@Resource` annotation to inject the context object into an MDB:

```
@Resource private MessageDrivenContext ctx;
```

There are a number of methods such as `setRollbackOnly()` and `getRollbackOnly()` which are identical to the corresponding `SessionContext` methods. These are used in the context of transactions, which we discuss later in this chapter.

Remaining `MessageDrivenContext` methods deal with Timer or Security aspects, which we cover in later chapters.

MDB LifeCycle

The MDB lifecycle is similar to the stateless session bean lifecycle. The following state diagram shows the MDB's lifecycle:
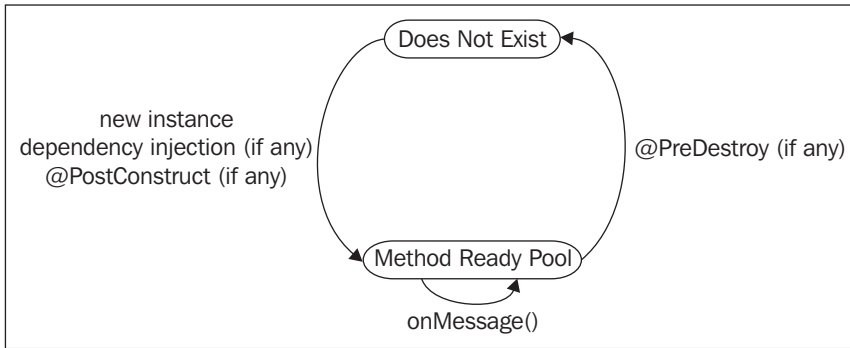
There are just two MDB lifecycle states: does-not-exist and method-ready pool. The initial state of an MDB is the does-not-exist state. This would be the case before a container starts up. The next state is the method-ready pool. When the container starts up it typically creates a number of MDB instances in the method-ready pool. However, the container can decide at any time to create such instances. In creating an instance in the method-ready pool, the container performs the following steps:

1. The MDB is instantiated.

2. The container injects the bean's `MessageDrivenContext`, if applicable.

3. The container performs any other dependency injection that is specified in the bean's metadata.

4. The container invokes a `PostConstruct` callback method if one is present in the bean. The `PostConstruct` method would be used for initializing any resources used by the bean. A `PostConstruct` method is called only once in the life of an instance, when it has transitioned from does-not-exist state to the method-ready pool. Note there can be at most only one `PostConstruct` method. The method can take any name but must be `void` with no arguments.

When a message arrives, the bean's `onMessage()` method is invoked. The container may have created multiple instances of MDBs which handle the same JMS destination, so any bean instance may be chosen by the container to execute the `onMessage()` method.

At some stage the container may decide to destroy the bean instance, usually when the container shuts down or when the container decides there are too many instances in the method ready pool. In this case the bean transitions from the method-ready-pool state to the does-not-exist state.

The container will first invoke a `PreDestroy` callback method if one is present in the bean. The `PreDestroy` method is used for tidying up activities. A `PreDestroy` method is called only once in the life of an instance, when it is about to transition to the does-not-exist state. As with the `PostConstruct` method, there can be at most only one `PreDestroy` method. The method can take any name but must be `void` with no arguments.

MDB Example Revisited

We will go back to our earlier example and add in a few more features that we covered in the last few sections. To recap, we have a session bean, `BankServiceBean`, which sends a message to a JMS queue requesting that a new customer be added to the database. The message itself is a customer object. On receipt of the message, the MDB adds the customer to the database. This time we will use a topic instead of a queue.

In this example we will also assume that customers can be either high or low priority. To illustrate the message filtering capabilities of MDBs, the MDB we describe in this section will only process high priority customers. We can assume that a separate MDB will process low priority customers.

The only modification we make to `BankServiceBean` is to indicate that we are using a JMS topic and that we have a high priority customer. We do this using the `ObjectMessage.setStringProperty()` method. The MDB will subsequently filter out any low priority customers. The code below highlights the modifications to `BankServiceBean`:

```
@Stateless
public class BankServiceBean implements BankService {
    @Resource(mappedName="BankServiceConnectionFactory")
    private ConnectionFactory cf;
    @Resource(mappedName="BankServiceJMSTopic")
    private Topic topic;
    @PersistenceContext(unitName="BankService")
    private EntityManager em;
    public void addCustomer(int custId, String firstName,
                            String lastName) {
        try{
            Customer cust = new Customer();
```

```
            cust.setId(custId);
            cust.setFirstName(firstName);
            cust.setLastName(lastName);
            Connection conn = cf.createConnection();
            Session sess = conn.createSession(false,
                            Session.AUTO_ACKNOWLEDGE);
            MessageProducer mp = sess.createProducer(topic);
            ObjectMessage objmsg = sess.createObjectMessage();
            objmsg.setObject(cust);
            objmsg.setStringProperty("Priority", "HIGH" );
            mp.send(objmsg);
            conn.close();
        } catch (JMSException e) {
            e.printStackTrace();
        }
    }
    ...
```

The revised code for the MDB `ProcessCustomerBean` is shown below:

```
@MessageDriven(activationConfig = {
@ActivationConfigProperty(propertyName = "destinationType",
                          propertyValue = "javax.jms.Topic"),
@ActivationConfigProperty(propertyName = "acknowledgeMode",
                          propertyValue = "Auto-acknowledge"),
@ActivationConfigProperty(propertyName =
                                "subscriptionDurability",
                          propertyValue = "NonDurable"),
@ActivationConfigProperty(propertyName = "messageSelector",
                          propertyValue = "Priority = 'HIGH'")
}, mappedName="BankServiceJMSTopic" )
public class ProcessCustomerBean implements MessageListener {
    @PersistenceContext(unitName="BankService")
    private EntityManager em;
    @PostConstruct
    public void init() {
        System.out.println(
                "Post Constructor Method init() Invoked");
    }
    public void onMessage(Message message) {
        try {
            if (message instanceof ObjectMessage) {
                ObjectMessage objmsg = (ObjectMessage)
                                        message;
                Customer cust = (Customer) objmsg.getObject();
```

```
                Customer mergedCust = em.merge(cust);
                System.out.println("MDB: Customer persisted");
            } else {
                System.out.println("Wrong type of message");
            }
        } catch(Exception e) {
            throw new EJBException(e);
        }
    }
    @PreDestroy
    public void tidyUp() {
        System.out.println(
            "Pre Destruction Method tidyUp() Invoked");
    }
}
```

We indicate that we are using a topic and not a queue. We have added three more activation configuration properties: `acknowlegeMode`, `subscriptionDurability`, and `messageSelector`. We have set the `acknowledgeMode` to `Auto-acknowledge`; however as we are using CMT by default this will have no effect. We have added the property just as an example of the syntax.

The `subscriptionDurability` property has been set to `NonDurable`, and the `messageSelector` property indicates we should only read messages with priority set to `HIGH`.

We have added a `PostConstruct` method `init()` and a `PreDestroy` method `tidyUp()`.

Sending Message Confirmation to a Client

We have already seen how an EJB container acknowledges message receipt with the JMS server. In this section we look at how a message producer client can be informed that its message has been received by an MDB.

This is done by the client setting the `JMSReplyTo` attribute with the message. `JMSReplyTo` contains the destination, queue or topic, to which the MDB should send its confirmation message. The MDB uses the `ObjectMessage.getJMSReplyTo()` method to obtain the confirmation destination. The MDB then acts as a message producer to send a message to the confirmation destination.

We will look at the example we have used throughout this chapter. Instead of using a session bean to send a JMS message to an MDB, we will use a Java application program. Recall that a session bean cannot act as a JMS message consumer. The Java application program, `TopicProducer`, will act both as a topic producer and a synchronous queue consumer. The code for `TopicProducer` is shown below:

```java
public class TopicProducer {
    @Resource(mappedName="BankServiceConnectionFactory")
    private static ConnectionFactory cf;
    @Resource(mappedName="BankServiceJMSTopic")
    private static Topic topic;
    @Resource(mappedName="BankServiceJMSQueue")
    private static Queue queue;
    public static void main(String[] args) {
        Connection conn = null;
        Connection conn2 = null;
        try {
            Customer cust = new Customer();
            cust.setId(10);
            cust.setFirstName("LEONARDO");
            cust.setLastName("DAVINCI");
            conn = cf.createConnection();
            Session sess = conn.createSession(false,
                            Session.AUTO_ACKNOWLEDGE);
            MessageProducer mp = sess.createProducer(topic);
            ObjectMessage objmsg = sess.createObjectMessage();
            objmsg.setObject(cust);
            objmsg.setStringProperty("Priority", "HIGH" );
            objmsg.setJMSReplyTo(queue);
            mp.send(objmsg);
            System.out.println("JMS Message sent");
            // now program acts as synchronous consumer
            // to get confirmation
            conn2 = cf.createConnection();
            Session sess2 = conn2.createSession(false,
                            Session.AUTO_ACKNOWLEDGE);
            MessageConsumer mcon = sess.createConsumer(queue);
            conn.start();
            Message message = mcon.receive(0);
            TextMessage msg = (TextMessage) message;
            System.out.println(
                    "JMS Message:" + msg.getText() );
        } catch (JMSException ex) {
            System.out.println(
                    "JMS Exception:" + ex.getMessage() );
```

```
            } finally {
                try {
                    conn.close();
                } catch (JMSException ex) {}
            }
        }
    }
```

This time we choose to deploy `TopicProducer` in the application client container. This means we can use the `@Resource` annotation to inject `BankServiceJMSTopic` and `BankServiceJMSQueue` into corresponding variables rather than using JNDI lookups. `BankServiceJMSTopic` is used for sending the message and `BankServiceJMSQueue` for receiving confirmation. The program also creates a second `Connection` object `conn2` which is subsequently used to obtain the message confirmation. The program creates a Customer object and sends a message in the usual way. The only addition is the line:

```
    objmsg.setJMSReplyTo(queue);
```

This specifies the destination, `BankServiceJMSQueue` in our case, to which the MDB should reply. After sending the message, the program now acts as a synchronous queue consumer and reads the confirmation message. The code is the standard synchronous consumer code that we have already seen. Note the program assumes there will be just one confirmation message of type `TextMessage`. Of course, in the real world, we would not want to assume this. Note the program hangs until a message arrives in the confirmation queue. The program will hang indefinitely if no confirmation is sent.

We will now take a look at the modifications required for the `ProcessCustomerBean` MDB to send a confirmation.

```
    @MessageDriven(mappedName="BankServiceJMSTopic" )
    public class ProcessCustomerBean implements MessageListener {
        @Resource(mappedName="BankServiceConnectionFactory")
        private ConnectionFactory cf;
         ....
        public void onMessage(Message message) {
            try {
                if (message instanceof ObjectMessage) {
                    ObjectMessage objmsg = (ObjectMessage)
                                                message;
                    cust = (Customer) objmsg.getObject();
                    Customer mergedCust = em.merge(cust);
                    System.out.println("MDB: Customer persisted");
                    sendConfirmation(objmsg, cf);
                } else {
                    System.out.println("Wrong type of message");
```

```
            }
        } catch(Exception e) {
            throw new EJBException(e);
        }
    }
    private void sendConfirmation(ObjectMessage objmsg,
                                     ConnectionFactory cf) {
        Connection conn = null;
        try {
            Queue queue = (Queue) objmsg.getJMSReplyTo();
            conn = cf.createConnection();
            Session sess = conn.createSession(false,
                            Session.AUTO_ACKNOWLEDGE);
            MessageProducer mp = sess.createProducer(queue);
            TextMessage msg = sess.createTextMessage(
                            "Customer added to databae");
            mp.send(msg);
        } catch (JMSException ex) {
            System.out.println(
                "JMS Exception:" + ex.getMessage() );
        } finally {
            try {
                conn.close();
            } catch (JMSException ex) {}
        }
    }
}
```

First note that we inject an instance of the `BankServiceConnectionFactory`
into the `cf` variable. We need this later to send the confirmation message. There
is no change to the `onMessage()` method other than a call to a new method,
`sendConfirmation()`, after the customer is persisted.

The `sendConfirmation()` method obtains the destination by using the
`ObjectMessage.getJMSReplyTo()` method:

```
Queue queue = (Queue) objmsg.getJMSReplyTo();
```

In our example the destination is a queue, but you can have a topic as a destination.
We then use the usual queue producer code to send the confirmation as a
`TextMessageType`.

MDBs and Transactions

Because JMS decouples the message producer from the consumer, it is not possible for a transaction to span the sending and receipt of a message. For example, you cannot have a transaction starting when a client sends a message to a recipient and committing after the client has received confirmation. In many cases a client does not know who the recipient is. Even if such transactions were possible, they would be very long running. Good transaction design encourages keeping transactions short.

Consequently the only transaction attributes available when using Container Managed Transactions (CMT) with MDBs are `REQUIRED` and `NOT_SUPPORTED`. The remaining transaction attributes deal with client initiated transactions, so are not applicable to MDBs. An MDB with the `REQUIRED` attribute will always execute in a new transaction. As with session beans, we would use the `NOT_SUPPORTED` attribute if the MDB writes to a legacy database or simple file system which does not support transactions.

As an alternative to CMT, MDBs can use Bean Managed Transactions (BMT). We need to be careful with exception handling when using CMT with MDBs. You may recall a system exception is defined as a `RuntimeException`, `RemoteException`, or `EJBException`. If a system exception is thrown, the container will roll back the transaction and discard the MDB instance. The message will not be acknowledged and the JMS server may resend the message. A new MDB instance will read the message. It is quite possible a second attempt to read the message will still cause the same system exception to be thrown. Consequently we have the possibility of an infinite loop of message rereads occurring. In practice it is possible to configure the JMS server to attempt a fixed number of message redeliveries before giving up.

A similar scenario occurs when the transaction is rolled back, either with the `MessageDrivenContext.setRollbackOnly()` method or if the MDB is annotated with `@ApplicationException(rollback=true)` and an application exception is thrown. The container will not discard the instance, but the message will not be acknowledged and the possibly infinite message retry scenario will occur.

For these reasons the MDB `onMessage()` method should not throw system exceptions or rollback transactions.

Summary

We first described the Java Message Service (JMS) API. We described two messaging modes that JMS supports: point-to-point and publish/subscribe. In point-to-point mode a client, or message producer, sends a message to a queue. In publish/ subscribe mode a message producer sends a message to a topic. A message recipient is referred to a message consumer. A queue or topic consumer can be either synchronous or asynchronous.

We looked at a number of examples of Java applications running outside an EJB container using JMS. We looked at queue and topic producers and both synchronous and asynchronous queue and topic consumers.

Message Driven Beans (MDBs) are provided where our messaging systems need the services offered by an EJB container. MDBs are EJB components which act as asynchronous message consumers. Although MDBs may implement non JMS messaging systems, all EJB compliant servers must support JMS-based MDBs.

MDBs can use point-to-point (queue) or publish/subscribe (topic) modes.

JMS-based MDBs must implement the `onMessage()` method of the `MessageListener` interface. This method receives the message and performs any business logic.

We looked at MDB activation configuration properties and described the MDB lifecycle. We gave an example of an MDB sending message confirmation to a client.

Finally we covered MDBs and transactions.

9
EJB Timer Service

EJB provides a basic scheduling capability, the timer service. In this chapter we will cover:

- The EJB timer service
- A single event example
- An interval event example
- The Timer interface
- Timers and transactions

Introduction

Most enterprises use operating system schedulers, such as the Unix Cron, for scheduling business processes to run at a fixed point in time or at intervals. Typically these are batch or workflow processes. A scheduling capability is provided for EJBs; the difference being that EJBs initiate the scheduler and that the business process is encapsulated as an EJB method. Of course, with EJB timers, you also have platform independence.

Because the EJB timer service is a container-managed service, it can be used in conjunction with stateless session beans or MDBs. Currently timers cannot be used with stateful session beans, although the EJB specification group has indicated that this may be rectified in a future release. As EJB 2.x entity beans are container-managed, they can use timers. However as EJB 3 entities are not container-managed, they cannot use timers.

An EJB creates a timer by invoking the `TimerService.createTimer()` method. The timer specifies a scheduled event either as a fixed point in time or as an interval. The same EJB will have a `@Timeout` annotated callback method which is invoked by the container when the scheduled event occurs. This method will contain the business process logic. Instead of a `@Timeout` annotated method, the EJB can implement an `ejbTimeout()` method.

Timer Service Examples

In the context of banking applications, we might use timers for housekeeping activities such as deleting temporary files, reporting on customers with no accounts, accounts with no corresponding customers, or unusual account activity. However, in order to provide the reader with examples that he or she can run themselves we will provide examples, which while not very realistic, do demonstrate the use of the timer service.

A Single Event Example

Suppose in our banking scenario we want to give all savings accounts a $10 bonus after a period of time. We first modify the stateless session `BankServiceBean.createAccounts()` method to create a timer as soon as we have created the accounts.

```
@Stateless
public class BankServiceBean implements BankService {
    @Resource
    private TimerService ts;

    public void createAccounts() {

        a1 = new Account();
        a1.setId(1);
        a1.setBalance(99);
        a1.setAccountType("C");
        em.persist(a1);
        a2 = new Account();
        a2.setId(2);
        a2.setBalance(520);
        a2.setAccountType("S");
        em.persist(a2);
        a3 = new Account();
        a3.setId(3);
        a3.setBalance(900);
        a3.setAccountType("S");
        em.persist(a3);

        Timer timer = ts.createTimer(6000, null);
    }
        ...
```

Note that we inject a `javax.ejb.TimerService` object into the variable `ts`. Alternatively, the `TimerService` object can be obtained from the `SessionContext`. We will show an example of this later in this chapter.

The first argument of the `TimerService.createTimer()` method specifies the duration in milliseconds. So the timer is scheduled to expire in 6 seconds from the moment the method is invoked. The duration is specified in milliseconds because this is the granularity used in the underlying Java SE platform. The timer service, however, cannot guarantee whether the timer will execute at exactly the specified time and so is not suitable for real time applications. The timer service is designed for processes that execute at coarse-grained durations and intervals. We have picked a figure of 6 seconds, so that the reader running the examples can quickly see the timer service at work.

The second argument to the `createTimer()` method is a serializable `info` object. If we do not have an `info` object, as is the case in our example, we set the argument to null. We will show an example of the use of the `info` object later in this chapter.

The `createTimer()` method returns a `javax.ejb.Timer` object, which is a notification to the container of a timed event to be scheduled.

The `createTimer()` method is overloaded. The first form of the method is:

```
createTimer(long duration, Serializable info)
```

This form was used in the `BankServiceBean` example above. The second form of the `createTimer()` method is:

```
createTimer(Date expiration, Serializable info)
```

In this case a fixed date, rather than a relative duration, is provided as the first argument.

Within the `BankServiceBean` we now add a timeout callback method. This method will be invoked by the container when a timeout event occurs. In our example this occurs 6 seconds after we have invoked `createTimer()`. We can give the callback method any name, in our case `awardBonus()`, but it must be prefixed by the `@Timeout` annotation.

```
@Timeout    // add 10 dollar bonus to all savings accounts
public void awardBonus(Timer timer) {
    System.out.println("Timer invoked");
    List<Account> accounts = em.createQuery(
       "SELECT ac FROM Account ac WHERE ac.accountType = 'S'")
       .getResultList();
    for (Account account : accounts) {
```

```
            double balance = account.getBalance();
            balance = balance + 10.0;
            account.setBalance(balance);
            em.persist(account);
        }
    }
```

A bean can have at most one timeout method. The timeout method must be `void` and have a `Timer` object as a parameter. In this example we do not make any use of the `Timer` object, but just add `10` dollars to all savings accounts. Rather than a name of `awardBonus()` prefixed with the `@Timeout` annotation, the timeout callback method could be implemented as an `ejbTimeout()` method. We will show an example of this shortly.

Typically timers are used in stateless session beans for batch processing or for processes that apply to a number of entities.

An Interval Event Example

Suppose in our banking example we want to add 1% interest to all savings accounts at specified intervals. The code below shows the modified `BankServiceBean`:

```
    @Stateless
    public class BankServiceBean implements BankService,
                                                TimedObject {

        @Resource
        private SessionContext ctx;
        ....
        public void createAccounts() {

            a1 = new Account();
            a1.setId(1);
            a1.setBalance(99);
            a1.setAccountType("C");
            em.persist(a1);
            a2 = new Account();
            a2.setId(2);
            a2.setBalance(520);
            a2.setAccountType("S");
            em.persist(a2);
            a3 = new Account();
            a3.setId(3);
            a3.setBalance(900);
            a3.setAccountType("S");
```

```
        em.persist(a3);
        TimerService ts = ctx.getTimerService();
        Timer timer = ts.createTimer(3000, 3000, null);
    }
    ...
```

Note this time rather than inject a `TimerService` object directly into our code, we obtain it using the `SessionContext.getTimerService()` method. In EJB 2.x this was the only way of obtaining a `TimerService` object. The call to `createTimer()` this time takes the form:

```
createTimer(long initialDuration, long intervalDuration,
            Serializable info)
```

So the timer will initially expire 3 seconds from the moment the method is invoked and subsequently at 3 second intervals. Note that no final expiration time is given; the timer will be active until it is cancelled. We will discuss cancellation later in this chapter. Again we do not supply an `info` object, so the third argument is null.

An alternative form of the `createTimer()` method for interval events is:

```
createTimer(Date initialExpiration, long intervalDuration,
            Serializable info)
```

In this case a fixed date rather than a relative duration is provided as an argument. Subsequent intervals are specified as relative durations in milliseconds. There is no `createTimer()` method where subsequent intervals can be specified as a `Date` type or a collection of `Date` types. We cannot specify an expiration time of every Monday and Wednesday at noon, for example.

We now add a timeout callback method to `BankServiceBean`. Instead of using a `@Timeout` annotated method as in the previous section we will implement the `ejbTimeout()` method. This is the only method in the `javax.ejb.TimedObject` interface which `BankServiceBean` must implement:

```
@Stateless
public class BankServiceBean implements BankService,
                                    TimedObject {
```

The `ejbTimeout()` method returns `void` and has a `Timer` object as a parameter:

```
// add 1% interest to all savings accounts
public void ejbTimeout(Timer timer) {
    System.out.println("Timer invoked");
    List<Account> accounts = em.createQuery(
      "SELECT ac FROM Account ac WHERE ac.accountType = 'S' ")
```

```
        .getResultList();
    for (Account account : accounts) {
        double balance = account.getBalance();
        balance = balance + (balance * 0.01);
        account.setBalance(balance);
        em.persist(account);
    }
}
```

Again we do not make any use of the `Timer` object, but just add 1% to all savings accounts. Implementing an `ejbTimeout()` method was the only option available in EJB 2.x.

Message-driven bean timers

We should at this point mention message-driven bean (MDB) timers. MDB timers are very similar to stateless session bean timers. The only difference is that MDB timers are created in the `onMessage()` method in response to an incoming message rather than a client initiated method call.

A Timer Interface Example

Recall that a `Timer` is an object that represents a timed event that has been scheduled by the EJB timer service. A `Timer` object implements the `javax.ejb.Timer` interface. This interface contains a number of methods which provide information about the timed event. Further information about the timed event can be provided by the application supplying an `info` object to the `createTimer()` method. This section shows examples of `Timer` methods and using `info` objects. We will also show how to cancel a timer.

We take the previous example where we add 1% interest to all savings accounts at specified intervals. However, this time we will inject a `TimerService` object rather than obtaining it through the `SessionContext`:

```
@Stateless
public class BankServiceBean implements BankService {
    @Resource
    private TimerService ts;

    public void createAccounts() {
        Formatter fmt = new Formatter();
        Calendar cal = Calendar.getInstance();
        fmt.format("%tk:%tM:%tS", cal, cal, cal);
```

```
                    // current time formatted as
                    // 24hours:minutes:seconds eg "16:10:02"
        String info =
              "Timer to add interest, started at " + fmt;
        Timer timer = ts.createTimer(3000, 3000, info);
    }
```

In the `createAccounts()` method, we obtain the current date and time and use the J2SE v5.0 Formatter to format it to a **24hours:minutes:seconds** pattern. We then create a string, `info`, which concatenates a brief description of the timer with the formatted date and time. This `info` string is then supplied as the final argument of the `createTimer()` method. Note that this argument can be any serializable object; a `String` type is serializable of course.

We do not make any changes to the timeout callback method. However, we add a new method `cancelTimers()` which unsurprisingly cancels all timers associated with the `BankServiceBean`:

```
public void cancelTimers() {
    for (Object obj : ts.getTimers()) {
        Timer timer = (Timer)obj;
        System.out.println("Next Timeout:" +
                            timer.getNextTimeout() );
        System.out.println("Time Remaining:" +
                            timer.getTimeRemaining() );
        System.out.println("Timer Info:" + timer.getInfo());
        timer.cancel();
    }
}
```

In addition to the overloaded `createTimer()` methods, the `TimerService` interface provides the `getTimers()` method for obtaining all active timers associated with the current bean. This returns a collection of `Timer` objects. We iterate through this collection of `Timer` objects. In our example we have only one `Timer` object. We then use some of the `Timer` interface methods to obtain more information about each `Timer` in turn.

- The `getNextTimeout()` method returns a `Date` object which represents the date on which the timer will expire next.

- The `getTimeRemaining()` method returns a `long` value which represents the time in milliseconds before the timer next expires.

- The `getInfo()` method returns the serializable `info` object which is associated with a timer at its creation. The `info` object is a means of identifying a timer, especially if we have more than one timer associated with the current bean.

The following shows typical output from the `cancelTimers()` method:

```
Next Timeout Wed Feb 28 17:22:56 CST 2007
Time Remaining 2326
Timer Info  Timer to add interest, started at 17:22:23
```

Finally we cancel the timer using the `Timer` method `cancel()`.

There is one more `Timer` method, `getHandle()`, which we have not shown in our example. This returns a `TimerHandler` object, which is a reference to the `Timer` object and which the application can persist for later use. Provided a single event timer has not expired or an interval event timer has not been cancelled, we can retrieve the reference using the `TimerHandler.getTimer()` method.

Timers and Transactions

Because timers are persistent objects, a timer that has not yet expired will survive a container crash or application server shutdown. The handling of timers that expire during server shutdowns is application server specific. For example GlassFish will not attempt at startup to retrospectively invoke the bean's timeout callback method for timers that expired during shutdown.

The `createTimer()` method executes in the scope of the executing method. If the transaction rolls back, then the container will undo the timer creation.

If a method cancels a timer within a transaction, and the transaction is rolled back, then the container will roll back the timer cancellation.

A timeout callback method may have a transaction attribute of `REQUIRES_NEW` or `REQUIRED`. If `REQUIRED` is specified as a transaction attribute, the container will start a new transaction in any case. So a `REQUIRED` attribute has exactly the same behavior as a `REQUIRES_NEW` attribute. This is done so that a method can use the default, `REQUIRED`, transaction attribute and still get the correct `REQUIRES_NEW` behavior. If a timeout callback method rolls back a transaction, then the container will reinvoke the callback method at least once.

Summary

The EJB timer service provides a basic scheduling capability, initiated by stateless session beans or MDBs. An EJB creates a timer by invoking the `TimerService.createTimer()` method. The created timer is actually a `javax.ejb.Timer` object, which is a notification to the container of a timed event to be scheduled. The scheduled event can be either a fixed point in time or an interval.

The same EJB will have a `@Timeout` annotated callback method which is invoked by the container when the scheduled event occurs. This method contains any business process logic. The callback method has the `Timer` object as a parameter. There are a number of methods in the `javax.ejb.Timer` interface which provide information about the timed event.

We also saw how to cancel timers. We made use of `TimerService.getTimers()` method which provides all active timers associated with the current bean.

Finally we looked at timers and transactions.

10
Interceptors

The idea behind **interceptors** is to separate common, or cross-cutting, code from business methods. Examples of such common code are logging, auditing, performance monitoring, or security checks. Such aspects are shared among business methods and are often tangential to any business logic. Instead of cluttering business methods with cross-cutting code, any such code is encapsulated in separate methods or classes called interceptors. Whenever a business method is invoked by an EJB container, its associated interceptors are automatically invoked first.

In this chapter we will cover:

- Interceptor methods
- Interceptor classes
- Default interceptors
- Interceptor communication

Interceptors are associated with **Aspect Oriented Programming (AOP)** and is a new feature introduced in version 3.0 of the EJB specification.

EJB 3 provides the ability to apply custom made interceptors to the business methods of session and message-driven beans. These interceptors take the form of methods annotated with the `@AroundInvoke` annotation. The interceptor can be defined in the bean class itself or in a separate class. There can be only one interceptor method per class.

We will start with a session bean example.

Interceptor Methods

We will modify the `BankServiceBean` stateless session bean and add an interceptor which logs the time taken to execute each business method within the bean. We name the interceptor method `methodStats()`. The code is shown below:

```
package ejb30.session;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import ejb30.entity.Customer;
import javax.persistence.PersistenceContext;
import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;

@Stateless
public class BankServiceBean implements BankService {
    ...
    @AroundInvoke
    public Object methodStats(InvocationContext invctx)
                                        throws Exception {
        String name = invctx.getMethod().getName();
        long startTime = System.nanoTime();
        System.out.println("Starting method: " + name);
        try {
            return invctx.proceed();
        } finally {
            long elapsedTime =
                System.nanoTime() - startTime;
            System.out.println("Finished method: " + name +
                " Method took " + elapsedTime +
                " nanoseconds to execute");
        }
    }
}
```

The `@AroundInvoke` annotation identifies the associated method as an interceptor. As with all of EJB 3 we can use deployment descriptors instead of, or in conjunction with, annotations to specify interceptors. An interceptor method can have any name but must have a return type of `Object`, throw `Exception` and have a `javax.interceptor.InvocationContext` object as a parameter. The `InvocationContext` interface has a number of methods which provide information regarding the business method being invoked.

The `InvocationContext.getMethod()` method returns, as a `Method` type, the business method invoking the interceptor. The `Method.getName()` method can then be invoked to obtain the invoking method name:

```
String name = invctx.getMethod().getName();
```

The `InvocationContext.proceed()` method has the signature:

```
public Object proceed() throws Exception
```

The `proceed()` method will invoke the next interceptor if there is more than one interceptor associated with the current business method. In our example we have only one interceptor; we will see later in this chapter how two or more interceptors can be associated with a business method. If there is only one interceptor, or if the current interceptor is the last in a chain, `proceed()` will invoke the business method.

As the name suggests `@AroundInvoke` indicates that the intercept wraps itself around the business method. The code prior to the `proceed()` method is executed before the business method is invoked. The code in the `finally` clause is executed after the business method is invoked.

Only one `@AroundInvoke` annotated method may be present in a bean.

The interceptor method does not have to invoke the next interceptor in the chain or the business method itself. For example, an interceptor method may perform some security check. If this check fails then the `proceed()` method will not be invoked.

Interceptor Classes

Interceptors can be defined in a separate class. The class will contain an `@AroundInvoke` annotated method as described in the previous section. The code below shows an interceptor class, `Auditor`, containing the `audit()` method. The `audit()` method updates the Audit entity with the class and method name of the intercepted business method, together with the current date and time. Of course a real world auditing application will store a lot more information, such as the identity of the user or process invoking the business method, as well as an indication as to whether the business method has executed successfully. The code for the `Auditor` interceptor class is shown below:

```
public class Auditor {
    @PersistenceContext(unitName="BankService")
    private EntityManager em;
    @AroundInvoke
    public Object audit(InvocationContext invctx)
                                  throws Exception {
```

```
    String className = invctx.getClass().getName();
    String methodName = invctx.getMethod().getName();
    Formatter fmt = new Formatter();
    Calendar cal = Calendar.getInstance();
    // standard date time format - day month date
    // 24hours:minutes:seconds tzone year
    // eg "Fri Mar 09 17:12:03 CST 2007"
    fmt.format("%tc", cal);
    String message = className + "." + methodName +
                    " invoked at " + fmt;
    Audit audit = new Audit();
    audit.setMessage(message);
    em.persist(audit);
    return invctx.proceed();
  }
}
```

The code is fairly straightforward. We have added a call to the `InvocationContext.getClass()` method to obtain the class invoking the interceptor. Note this time there is no interceptor code to execute after the `proceed()` call.

Only one `@AroundInvoke` annotated method can exist within a class. When a bean instance is created, interceptor instances are created for each associated interceptor class. When a bean instance is removed the associated interceptor instances are destroyed. Interceptors share the naming context of the associated bean so, for example, we can inject the `EJBContext` or `PersistenceContext` into the interceptor class. In the `Auditor` interceptor we injected the `BankService PersistenceContext` into the class.

Interceptor method invocations occur within the same transaction and security context as the associated business method. We will cover EJB security in Chapter 12.

The code below is a modified fragment of `BankServiceBean` and shows how we indicate that the `Auditor` interceptor class is applicable for this bean:

```
@Stateless
@Interceptors(Auditor.class)
public class BankServiceBean implements BankService {
    // ... some business methods
    @ExcludeClassInterceptors
    public List<Audit> findAllAudits() {
        List<Audit> audits = em.createQuery(
            "SELECT aud FROM Audit aud").getResultList();
        return audits;
    }
```

```
    @AroundInvoke
    public Object methodStats(InvocationContext invctx)
                                        throws Exception {
        String name = invctx.getMethod().getName();
        long startTime = System.nanoTime();
        System.out.println("Starting method: " + name);
        try {
            return invctx.proceed();
        } finally {
            long elapsedTime =
                System.nanoTime() - startTime;
            System.out.println("Finished method: " + name +
                " Method took " + elapsedTime +
                " nanoseconds to execute");
        }
    }
}
```

Note the annotation:

```
    @Interceptors(Auditor.class)
```

This indicates the class which will be used as an interceptor for every business method in `BankServiceBean`. If we want to associate more than one interceptor class with the current bean, we simply list them delineated by braces:

```
    @Interceptors({Auditor.class, AnotherInterceptor.class})
```

The interceptors are processed in left to right order.

By prefixing a class declaration with the `@Interceptors` annotation we indicate that the interceptor or interceptors apply to *all* business methods in the class. This is known as a class-level interceptor. Alternatively we can prefix individual methods with the `@Interceptors` annotation. In that case the interceptor(s) only apply to the annotated methods. This is known as a method-level interceptor.

If a class declaration has been prefixed with the `@Interceptors` annotation, we can exclude individual bean business methods from having corresponding interceptors invoked. This is done with the `@ExcludeClassInterceptors` annotation. We have excluded the `findAllAudits()` method in this way.

Note that `BankServiceBean` still includes the `methodStats()` interceptor method. We could of course put this method in a separate interceptor class, but have chosen not to do so. In this case any interceptor class methods will be invoked before any interceptor methods in the bean itself. So the `Auditor.audit()` interceptor will be invoked before `methodStats()`.

More precisely the correct intercept invocation order is as follows:

1. First default interceptors, if any, are invoked. We will see shortly how to create default interceptors.

2. Next class-level interceptors, if any, are invoked in the order specified within the `@Interceptors` annotation.

3. Next method-level interceptors, if any, are invoked in the order specified within the `@Interceptors` annotation.

4. Finally any bean interceptor method is invoked.

It is possible to override this order as we shall see in the next section.

Default Interceptors

A default interceptor is invoked whenever any business method is invoked on any bean within the deployment. Default interceptors cannot be defined using annotations but must be specified using deployment descriptors within an `ejb-jar.xml` file. Suppose we want the `Auditor` interceptor class to be the default interceptor for all EJBs. The `ejb-jar.xml` file to do this is shown below:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd">
<interceptors>
<interceptor>
<interceptor-class>ejb30.session.Auditor</interceptor-class>
<around-invoke>
<method-name>audit</method-name>
</around-invoke>
</interceptor>
</interceptors>
<assembly-descriptor>
<interceptor-binding>
<ejb-name>*</ejb-name>
<interceptor-class>ejb30.session.Auditor</interceptor-class>
</interceptor-binding>
</assembly-descriptor>
</ejb-jar>
```

Note that before we define a default interceptor, we must specify the interceptor class and method itself. The interceptor class is specified using the `<interceptor-class>` deployment descriptor. This is embedded within the `<interceptor>` deployment descriptor. Recall that there may be more than one interceptor class. In that case each interceptor class would be embedded within their own `<interceptor> </interceptor>` pair. The code:

```
<around-invoke>
<method-name>audit</method-name>
</around-invoke>
```

specifies that `audit()` is the interceptor method.

The default interceptor is specified within the `<interceptor-binding>` deployment descriptor. This in turn must be embedded within `<assembly-descriptor>`. The `<assembly-descriptor>` contains application-assembly information such as security roles, transaction attributes as well as default interceptors. We need to specify the default interceptor class with the `<interceptor-class>` descriptor. We may have more than one default interceptor class. In this case each default interceptor class would have their own `<interceptor-class>` descriptor. Multiple default interceptors will be invoked in the order that they are specified within the `ejb-jar.xml` file.

The wildcard in

```
<ejb-name>*</ejb-name>
```

indicates that the default interceptor is applicable to all EJBs in the current deployment. If we wanted the default interceptor to only apply to the `BankServiceBean` session bean we would specify:

```
<ejb-name>BankServiceBean</ejb-name>
```

The `ejb-jar.xml` file must be packaged in the `META-INF` directory of the EJB JAR file alongside the `persistence.xml` file.

Note that because we have specified `Auditor` as an interceptor class using deployment descriptors in the `ejb-jar.xml` file, we do not need to prefix the `audit()` method in the `Auditor` class with the `@AroundInvoke` annotation:

```
public class Auditor {
    @PersistenceContext(unitName="BankService")
    private EntityManager em;
    public Object audit(InvocationContext invctx)
                                throws Exception {
        ...
    }
}
```

The `BankServiceBean` no longer needs to prefix the class declaration with the `@Interceptors(Auditor.class)` annotation, as `Auditor` is the default interceptor class.

We can exclude individual business methods from having default interceptors invoked by means of the `@ExcludeDefaultInterceptors` annotation. For example we could exclude the `findAllAudits()` method in this way:

```
@Stateless
public class BankServiceBean implements BankService {
    // ... some business methods

    @ExcludeDefaultInterceptors
    public List<Audit> findAllAudits() {
        List<Audit> audits = em.createQuery(
            "SELECT aud FROM Audit aud").getResultList();
        return audits;
    }
}
```

Recall the interceptor invocation order is default, class-level and then method-level interceptors followed by the bean interceptor method.

It is possible to override this order using the `interceptor-order` deployment descriptor element in the `ejb-jar.xml` file. Suppose we have a default interceptor, `DefaultInterceptor`, and a class-level interceptor, `ClassLevelInterceptor`, on `BankServiceBean`. Then we can change the invocation order by adding the following in the `ejb-jar.xml` file:

```
<interceptor-binding>
<ejb-name>ejb30.session.BankServiceBean</ejb-name>
<interceptor-order>
   <interceptor-class>
       ejb30.session.ClassLevelInterceptor</interceptor-class>
   <interceptor-class>
       ejb30.session.DefaultInterceptor</interceptor-class>
</interceptor-order>
</interceptor-binding>
```

Interceptor Communication

If a business method has two or more interceptors, then it is possible for interceptors to pass data between each other within the same business method invocation. This is done using the `InvocationContext.getContextData()` method. This method returns a `Map` object containing the data passed between the interceptors. This is best illustrated with an example.

As the `Auditor` class interceptor is fired first for any business method invocation, we will get this interceptor to send a message to any subsequent interceptors in the chain. The message merely indicates that the `Auditor` was the first interceptor in the chain. The code below shows the modified version of `Auditor`:

```
public class Auditor {
    @PersistenceContext(unitName="BankService")
    private EntityManager em;
    public Object audit(InvocationContext invctx)
                                  throws Exception {
        String className = invctx.getClass().getName();
        String methodName = invctx.getMethod().getName();
        Formatter fmt = new Formatter();
        Calendar cal = Calendar.getInstance();
        // standard date time format - day month date
        // 24hours:minutes:seconds tzone year
        // eg "Fri Mar 09 17:12:03 CST 2007"
        fmt.format("%tc", cal);
        String message = className + "." + methodName +
                    " invoked at " + fmt;
        Audit audit = new Audit();
        audit.setMessage(message);
        em.persist(audit);
        String msg = "First interceptor for " + methodName +
                    " was Auditor.audit";
        invctx.getContextData().put("interceptMessage", msg);
        return invctx.proceed();
    }
}
```

As `InvocationContext.getContextData()` returns a `Map` object, we use the `Map.put()` method to populate the `Map`. In our case we use `interceptMessage` as the key and a `String` message as the `Map` value. Of course we could add more objects to the `Map`.

We now modify the `BankServiceBean.methodStats()` interceptor to read the `Map` returned by `getContextData()`. As `methodStats()` is an interceptor method within a bean, it is last to be invoked after any default and class interceptors. The code is shown below:

```
@Stateless
public class BankServiceBean implements BankService {
    // ... other business methods

    @ExcludeDefaultInterceptors
```

```
        public List<Audit> findAllAudits() {
            List<Audit> audits = em.createQuery(
                "SELECT aud FROM Audit aud").getResultList();
            return audits;
        }
        @AroundInvoke
        public Object methodStats(InvocationContext invctx)
                                        throws Exception {
            String name = invctx.getMethod().getName();
            long startTime = System.nanoTime();
            String msg = (String)
                invctx.getContextData().get("interceptMessage");
            if (msg == null) {
                msg = "No interceptor message for " + name;
            }
            System.out.println(msg);
            System.out.println("Starting method: " + name);
            try {
                return invctx.proceed();
            } finally {
                long elapsedTime =
                    System.nanoTime() - startTime;
                System.out.println("Finished method: " + name +
                    " Method took " + elapsedTime +
                    " nanoseconds to execute");
            }
        }
    }
```

Using `interceptMessage` as the `key` we obtain the corresponding `value` using the `Map.get()` method in the statement:

```
String msg = (String)
    invctx.getContextData().get("interceptMessage");
```

The `methodStats()` interceptor cannot assume that any other interceptor has set the context data `Map`, or indeed that any other interceptor has even been fired. For example, because we excluded any default interceptors for the `findAllAudits()` business method, `methodStats()` will be the first interceptor to be fired for any invocation of `findAllAudits()`. In these cases `getContextData()` will return a null `Map`, and we print out a message to indicate this.

Summary

Interceptors are a new EJB 3 feature which separate common, or cross-cutting, code from business methods. Interceptors can be defined for session and message-driven beans. We can add an interceptor method to a bean by annotating that method with the `@AroundInvoke` annotation. The interceptor method wraps itself around the associated business method.

Interceptors can be defined in a separate class. A bean can have one or more interceptor classes associated with it denoted by the `@Interceptors` annotation.

Default interceptors are specified by XML deployment descriptors rather than annotations and apply when deployed. Default interceptors apply for all beans in the deployment.

If a business method has two or more interceptors, then it is possible for interceptors to pass messages between each other within the same business method invocation. This is done using the `InvocationContext.getContextData()` method.

11

Implementing EJB 3 Web Services

Web service technology is a huge and evolving topic with numerous specifications. In this chapter we just scratch the surface and focus on how to expose a Java application and a session bean as a web service, and how to invoke a web service from a Java client. In this chapter we will cover the following topics:

- Overview of web service concepts
- Exposing a Java application as a web service
- The Web Service Definition Language (WSDL)
- The GlassFish wsgen tool
- Exposing a session bean as a web service

Overview of Web Service Concepts

The idea behind web services is to expose an application as a service for clients over the web independently of the application and clients' execution environments. The application may be fine grained, such as a mathematical calculation, or coarse grained, such as processing an order. Interoperability is an important goal of web services and all leading technology providers, including Microsoft, have subscribed to web services standards. This means a C# client running under .NET can invoke a web service implemented as a Java EJB in a Java EE environment and vice-versa. The interoperability goal of web services means that such services are more loosely coupled than EJBs.

There are two main organizations which develop web service standards: The World Wide Web Consortium (W3C) and The Organization for the Advancement of Structured Information Standards (OASIS). The key web service standards are SOAP and WSDL, maintained by W3C, and UDDI maintained by OASIS. We will take a brief look at these standards.

The SOAP Protocol

SOAP originally stood for 'Simple Object Access Protocol' but this was dropped by the W3C with version 1.2 of the standard as it was considered to be misleading (SOAP does not access objects in particular). SOAP no longer is an acronym. SOAP is a protocol for exchanging XML-wrapped messages over computer networks normally using HTTP or HTTPS. A web service client sends a request in XML over the internet using the HTTP or HTTPS protocol to a web service provider. The web service provider will parse this XML request, execute the service, and send a response in XML back to the client. Both the XML request and response use the SOAP protocol. A SOAP message consists of an envelope element; within the envelope are one or more optional header elements and one or more body elements. The header element consists of infrastructure items such as logging or security. The body element contains the message being exchanged. Below is a SOAP request for a web service that multiplies two numbers. The request supplies the two numbers, 3 and 4, as arguments:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:ns1="http://endpoint/">
<soapenv:Body>
<ns1:multiply>
<arg0>3</arg0>
<arg1>4</arg1>
</ns1:multiply>
</soapenv:Body>
</soapenv:Envelope>
```

Below is a SOAP response from the previous request. The response contains the result, 12, from the multiplication operation.

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:ns1="http://endpoint/">
<soapenv:Body>
```

```
<ns1:multiplyResponse>
<return>12</return>
</ns1:multiplyResponse>
</soapenv:Body>
</soapenv:Envelope>
```

There is no need to learn any SOAP syntax. All SOAP requests and responses are generated by software artifacts that are described later in this chapter.

The WSDL Standard

The web service interface is specified by the **WSDL**. WSDL is also an XML document which provides all the information required to locate and access a web service. The web service provider has the responsibility of creating the WSDL document. We will see an example of a WSDL document later in this chapter.

The UDDI Standard

Another concept in web services technology is the **web service registry**. Having created a web service, the provider may wish to publicize the service by publishing the WSDL document in a registry. The registry could belong to the organization providing the web service or be provided by a third party. A client wishing to use a web service will use the registry's directory service to locate a web service then retrieve the associated WSDL document. The client then uses the WSDL document to create a SOAP web service request. The registry operations of publishing, searching and retrieving are covered by the **Universal Description, Discovery, and Integration** (**UDDI**) standard.

However, in practice most web services are not for universal consumption. Often web services are deployed in business to business (B2B) environments, legacy integration, or Service Oriented Architecture (SOA) implementations. In all these cases, the use of a UDDI registry is optional.

SOA and Web Services

SOA implementations are a growing use of web services. SOA characterizes a business process as a number of discrete coarse grained services. These services are not necessarily web services. For example, they could be services based on asynchronous messaging technology such as JMS. However, in many cases an SOA service will be a web service.

Creating a Java Application Web Service

Any Java EE 5 compliant application server implements the **Java API for XML-Based Web Services** (**JAX-WS**) 2.0 specification (JSR 224). JAX-WS defines the mapping between Java and WSDL. The actual mapping between Java and XML (recall WSDL is an XML document) is specified by the **JAXB 2.0** (Java Architecture for XML Binding) API which any Java EE 5 application server implements. A JAX-WS runtime system converts Java API calls and responses from and to SOAP messages. Furthermore any Java EE 5 compliant server will usually provide tools for both generating WSDL documents from a Java web service implementation class and for generating a Java web service implementation class from a WSDL document. In web services terminology such an implementation class is known as an **endpoint implementation class**. A Java EE 5 application server may also automatically generate a WSDL document when the Java endpoint implementation class is deployed to the server.

A Java application is deployed as a web service in a web container. A stateless session bean, on the other hand, is deployed as a web service in an EJB container.

We will start with an example that creates a web service that multiplies two integers. We can do this in two ways. One approach is to first create a WSDL document then generate a Java endpoint implementation from the WSDL. This is known as the top-down approach. The second approach is to first create a Java endpoint implementation class then generate the corresponding WSDL document. This is known as the bottom-up approach.

The top-down approach is likely to be used when creating web services from scratch. The WSDL document is a contract or specification that the web service implements. We are more likely to achieve interoperability in a heterogeneous environment if we use the top-down approach. However, this approach requires mastering the WSDL protocol before developing the simplest web service and so entails a steep learning curve.

The bottom-up approach is likely to be used when we have existing applications that we want to expose as web services. The risk with a bottom-up approach is that we may use language constructs which are not portable and so compromise interoperability. However, the bottom-up approach is easier to learn; we are not confronted with having to assimilate all the details of WSDL before we can start. From a learning point of view, it is easier to start with a bottom-up approach and after sufficient familiarity with WSDL then switch to a top-down approach. In this chapter we take the bottom-up approach.

Below is a listing of an endpoint implementation class, `Arithmetic.java`, that multiplies two integers. There are only a few modifications we need to make to a standard Java class in order to convert it to an endpoint implementation class. This is a result of JAX-WS 2.0 taking the EJB 3 approach of using of annotations and defaults.

```java
package endpoint;
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public class Arithmetic {
    public Arithmetic() {}

    @WebMethod
    public int multiply(int a, int b) {
        int c = a * b ;
        return c;
    }
}
```

The `@WebService` annotation marks the class as a web service. There are a number of annotation properties some of which we will cover later, but for now we will stick with the defaults.

The `@WebMethod` annotation identifies the method to be exposed as a web service operation.

If there are no `@WebMethod` annotations in the endpoint implementation class then all public methods will be exposed as web service operations.

There are a few restrictions on the endpoint implementation class:

- The class must be an outer public class
- The class must not be final
- The class must not be abstract
- The class must have a default public constructor
- The class must not define a `finalize()` method

Furthermore, any method that is exposed as a web service operation must be public. The method's parameters and return types must be supported by JAXB. These supported Java types include all Java primitives as well as a number of standard Java classes such as `java.lang.String` and `java.util.Date`.

The method may include a throws java.rmi.RemoteException clause but is not required to do so. In EJB 2.x a method exposed as a web service operation has to include a throws java.rmi.RemoteException clause. This is because in EJB 2.x a RemoteException is *checked*. In EJB 3 a RemoteException is *unchecked*, so including a throws RemoteException clause is optional.

Creating an Endpoint Implementation Interface

Optionally, we may provide an endpoint implementation interface. Suppose we have a Mathematics interface which contains just one method, multiply. The Mathematics interface is implemented by MathematicsImpl. The Mathematics interface code is shown below:

```
package endpoint;
import javax.jws.WebService;

@WebService
public interface Mathematics {
    public int multiply(int a, int b);
}
```

As with an endpoint implementation class, the @WebService annotation marks the interface as a web service. All methods declared within the interface are exposed as web service operations.

If we do provide an interface then the implementation class should specify the interface with the endpointInterface property of the @WebService annotation. The class, MathematicsImpl.java, which implements the Mathematics interface, is shown below:

```
@WebService(endpointInterface = "endpoint.Mathematics")
public class MathematicsImpl implements Mathematics {
    public MathematicsImpl() {}
    public int multiply(int a, int b) {
        int c = a * b ;
        return c;
    }
    public int add(int a, int b) {
        int c = a + b ;
        return c;
    }
}
```

We do not need to prefix any methods with the `@WebMethod` annotation. Note that we have included the `add()` method. However, as this method is not declared in the interface it will *not* be exposed as a web service operation.

The WSDL Document

We will revert to our original `Arithmetic` endpoint implementation class. This will have the corresponding WSDL file shown below. We do not need to create such a file manually. A WSDL file will either be created by the application server at deploy time or we can use an application server supplied tool to generate a WSDL file. In this chapter, we will not attempt to cover all the WSDL syntax. We will describe just a few WSDL elements to help the reader get started with implementing Java web services.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://endpoint/"
             name="ArithmeticService"
             xmlns:tns="http://endpoint/"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <xsd:schema>
      <xsd:import namespace=http://endpoint/
                  schemaLocation="ArithmeticService_schema1.xsd"/>
    </xsd:schema>
  </types>
  <message name="multiply">
    <part name="parameters" element="tns:multiply"/>
  </message>
  <message name="multiplyResponse">
    <part name="parameters" element="tns:multiplyResponse"/>
  </message>
  <portType name="Arithmetic">
    <operation name="multiply">
      <input message="tns:multiply"/>
      <output message="tns:multiplyResponse"/>
    </operation>
  </portType>
  <binding name="ArithmeticPortBinding" type="tns:Arithmetic">
    <soap:binding
            transport="http://schemas.xmlsoap.org/soap/http"
            style="document"/>
    <operation name="multiply">
```

```
        <soap:operation soapAction=""/>
        <input>
          <soap:body use="literal"/>
        </input>
        <output>
          <soap:body use="literal"/>
        </output>
      </operation>
    </binding>
    <service name="ArithmeticService">
      <port name="ArithmeticPort"
            binding="tns:ArithmeticPortBinding">
        <soap:address location="http://localhost:8080/
                          arithmetic-webservice/ArithmmeticService"/>
      </port>
    </service>
  </definitions>
```

In WSDL terminology an abstract web service description is defined as a **port type**. The port type describes the available web service operations. A concrete web service instance is known as a **port** and is defined by the concatenation of port type, protocol binding and endpoint address. A WSDL **service** is a collection of ports.

The <portType> Element

The `<portType>` element describes the available web service operations:

```
<portType name="Arithmetic">
```

The name of a port type defaults to the service endpoint interface, or the service endpoint implementation class, if no interface has been provided. A port type is the only WSDL element which corresponds to a service endpoint interface if an interface has been provided. All other WSDL elements that correspond to a service endpoint implementation class do so even if an interface has been provided.

The available web service operations are indicated by the `<operation>` element:

```
<operation name="multiply">
```

The name of an operation defaults to the Java method name.

The <binding> Element

This defines the message format and protocol details for a web service. In the above WSDL document we have specified SOAP over HTTP as the binding protocol. This is the standard web services protocol currently available, but in principle we could specify other protocols if and when they become available.

The <service> Element

The WSDL service is indicated by the `<service>` element:

```
<service name="ArithmeticService">
  <port name="ArithmeticPort"
        binding="tns:ArithmeticPortBinding">
    <soap:address location="http://localhost:8080/
                      arithmetic-webservice/ ArithmmeticService"/>
  </port>
</service>
```

The name of a service defaults to the name of the endpoint implementation class with "Service" appended. Within the service element are listed the ports comprising the service. In our example WSDL document has only one port making up the service. The port is indicated by the `<port>` element and specifies the protocol binding and endpoint URL address.

The URL address is created at deploy time and takes one of two forms. If the web service is deployed in a web container, the URL takes the form:

```
http://<hostname>:<port>/context-root/<WSDL service name>
```

When the GlassFish application server is started with the `asdamin start-domain` command, a message provides the available hostname and port for web applications. The context root defaults to the name of the WAR file although we can specify a different context root at deploy time. We will show how the WAR file is created in the "Deploying A Java Application as Web Service" section, later in this chapter.

If the web service is deployed as a session bean in the EJB container we can dispense with context root in the URL address:

```
http://<hostname>:<port>/<WSDL service name>
```

The <message> and <types> Elements

The WSDL `<message>` element describes the parameters and return types of a web service message. The data types of these parameters and return types are specified in the WSDL `<types>` element:

```
<types>
  <xsd:schema>
    <xsd:import namespace="http://endpoint/"
               schemaLocation="ArithmeticService_schema1.xsd"/>
  </xsd:schema>
</types>
```

The data types can be described in the WSDL file itself or, as in our example, in a separate XML Schema Definition (XSD) file. The name of this XSD file is given by the `schemaLocation` attribute. The default XML target namespace for both the WSDL and XSD documents is generated as follows:

1. reverse the order of the Java package components
2. prefix the result with `http://`
3. add a / suffix to the result

So a Java package of `com.example.endpoint` will map to a namespace of `http://endpoint.example.com/`.

In our example the Java package is simply `endpoint`, so the corresponding default namespace is `http://endpoint/`.

The GlassFish WSGEN Tool

We mentioned earlier that application servers provide tools for generating WSDL files from a Java endpoint implementation class. We will describe the GlassFish `wsgen` tool. If the reader is using another Java EE application server then this section may be skipped.

The first use of `wsgen` is to generate a WSDL file from a Java implementation class. This use is optional in GlassFish. A WSDL file will be automatically created by GlassFish at deploy time. `wsgen` is run after the Java endpoint implementation class has been compiled. The following Ant script will generate a WSDL file:

```
<target name="run-wsgen">
    <mkdir dir="build/wsgen"/>
    <exec executable="${glassfish.home}/bin/wsgen.bat"
          failonerror="true" >
      <arg line="-cp build -r build/wsgen –wsdl
           endpoint.Arithmetic"/>
    </exec>
</target>
```

- -cp (abbreviation for **-classpath**): This option specifies where to find the input class files. In our example this is in the build directory.

- -r: This option is used only in conjunction with the -wsdl option. It specifies where to place the generated WSDL and associated files.

- -wsdl: This option specifies that wsgen should generate a WSDL file.

The final argument is the fully qualified endpoint implementation class, namely endpoint.Arithmetic. The result of the above script is to create an ArithmeticService.wsdl file in the build/wsgen directory. The name of the WSDL file is the same as the service name, which you will recall defaults to the name of the endpoint implementation class with "Service" appended. As well as the WSDL file, the associated XSD file, ArithmeticService_Schema1.xsd, is also created.

The second purpose of wsgen is to generate a number of Java classes required for **marshalling** and **unmarshalling** of method invocations, responses and exceptions during the execution of a web service. The actual marshalling and unmarshalling is performed by the container using the JAXB 2.0 API. Recall any Java EE 5 compliant application server implements JAXB 2.0. The process of transforming a collection of Java objects into XML format is known as marshalling. The inverse process of transforming an XML document into a collection of Java objects is known as unmarshalling.

Again this use of wsgen is optional. These Java artifacts are automatically created by GlassFish at deploy time. The Ant script to generate the artifacts is:

```
<target name="run-wsgen">
        <mkdir dir="build/wsgen"/>
        <exec executable="${glassfish.home}/bin/wsgen.bat"
            failonerror="true" >
          <arg line="-cp build -keep -d build
              endpoint.Arithmetic"/>
        </exec>
</target>
```

-keep specifies to keep the generated artifacts.

-d specifies where to place the generated artifacts. In our example this is the build directory. In fact, under the build directory wsgen will create the subdirectory endpoint/jaxws and place the artifacts in this subdirectory. The generated artifacts are actually source and compiled JavaBeans:

- Multiply.class
- Multiply.java
- MultiplyResponse.class
- MultiplyResponse.java

The `Multiply` JavaBean invokes the web service, and the `MultiplyResponse` JavaBean handles the response. If the endpoint implementation class contained exceptions then an additional JavaBean would be generated for each exception.

Deploying a Java Application as a Web Service

Recall that a Java application web service is deployed as a class in a web container. So after compiling the Java application we need to package it in a Web Archive (WAR) file. Below is a listing of the contents of a WAR file, for the `Arithmetic.java` example:

```
>jar -tvf arithmetic-webservice.war
     0 Wed Apr 18  14:18:08 BST 2007 META-INF/
   106 Wed Apr 18  14:18:06 BST 2007 META-INF/MANIFEST.MF
     0 Wed Apr 18  14:16:18 BST 2007 WEB-INF/
     0 Wed Apr 18  14:16:18 BST 2007 WEB-INF/classes/
     0 Wed Apr 18  14:18:08 BST 2007 WEB-INF/classes/endpoint/
   286 Wed Apr 18  14:18:08 BST 2007
  WEB-INF/classes/endpoint/Arithmetic.class
  >
```

We have named the WAR file `arithmetic-webservice.war`.

The next step is to deploy the WAR file. This process is application server-dependent, below is an Ant target for deploying the WAR file to the GlassFish application server using the `asadmin deploy` command:

```
<target name="deploy">
    <exec executable="${glassfish.home}/bin/asadmin"
          failonerror="true"
          vmlauncher="false">
      <arg line="deploy --user admin
          --passwordfile adminpassword
          ${build.dir}/arithmetic-webservice.war"/>
    </exec>
</target>
```

The GlassFish Admin Console Test Harness

In this section we describe the GlassFish web service test harness. If the reader is using another Java EE application server then this section may be skipped. First launch the application server admin console from the browser by entering `http://<hostname>:<port>`. For example:

```
http://localhost:4848
```

Note that when the application server is started with the `asdamin start-domain` command a message provides the hostname and port for the admin console. Under the **Web Services** heading in the **Common Tasks** pane on the left of the screen, click on **Arithmetic**. This brings up the web service details in the right-hand pane:
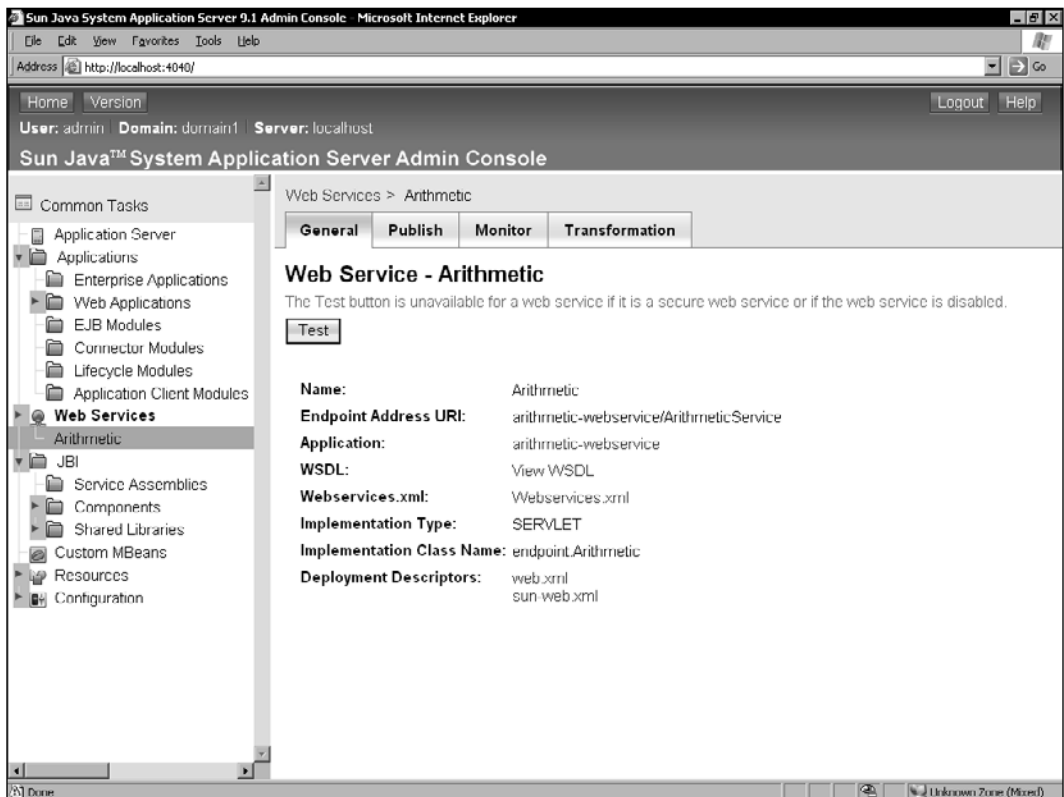
Next click on the **Test** button and the following screen appears:

Enter the two integers to be multiplied then click on the **multiply** button. The result will be returned together with the SOAP request and response.

Creating a Java Web Service Client

Having created a WSDL file we can publish it to a UDDI registry as described in the introduction. Alternatively, assuming we have a trusted client, we can provide the client with the WSDL document. The client need not be a Java client, but does need to run in an environment which implements Web Service standards. The client's environment has to have some mechanism for creating a SOAP request from the provided WSDL file.

In this section we describe how a Java client invokes a web service given a WSDL file. This is done by generating **stubs** or **proxies** from the WSDL document. The proxy is the local client representation of the remote web service. The client then invokes the web service through the proxy.

GlassFish provides a tool, `wsimport`, for generating the proxies. Below is an Ant script for generating the proxies from the `ArithmeticService` WSDL file created earlier:

```
<target name="run-wsimport">
    <exec executable="${glassfish.home}/bin/wsimport"
          failonerror="true"
          vmlauncher="false">
      <arg line="-keep -d build/client
```

```
                http://localhost:8080/arithmetic-webservice/
                  ArithmeticService?WSDL"/>
    </exec>
  </target>
```

`-keep` indicates that generated files should be kept.

`-d` specifies the directory in which to place the generated files, in our case `build/client`.

The final argument is the location of the WSDL file. This is the URL address of the WSDL file suffixed with `?WSDL` or `?wsdl`. We saw earlier in the section on WSDL how the URL address is created.

Having generated the proxies, we can now create a Java client, `WebServiceClient.java` that invokes the `Arithmetic` web service.

```
package client;
import javax.xml.ws.WebServiceRef;
import endpoint.ArithmeticService;
import endpoint.Arithmetic;
public class WebServiceClient {
    @WebServiceRef(wsdlLocation=
        "http://localhost:8080/arithmetic-webservice/
          ArithmeticService?WSDL")
    static ArithmeticService service;
    public static void main(String[] args) {
        try {
            int a = 3;
            int b = 4;
            Arithmetic port = service.getArithmeticPort();
            int result = port.multiply(3, 4);
            System.out.println("Result of multiplying "
                + a + " by " + b + " using webservice is: "
                + result);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Note that we have imported `endpoint.ArithmeticService` and `endpoint.Arithmetic`. These are proxy classes that were earlier generated by the `wsimport` tool. The package name, `endpoint`, is derived from the XML target namespace specified in the WSDL file.

The `@WebServiceRef` annotation is used to declare a reference to a web service. The web service is the generated `ArithmeticService` proxy. The `wsdlLocation` property indicates the location of the corresponding WSDL document. We inject `ArithmeticService` into the `service` variable.

We next obtain a reference to a port using the `ArithmeticService` proxy `getArithmeticPort()` method. The port is another proxy, `Arithmetic`, which implements the service endpoint interface. Finally we invoke the port's `multiply` operation.

The only steps that remain are to compile and run the client.

Overriding JAX-WS Annotation Defaults

In this section we revisit the `Arithmetic` endpoint implementation class and override some of the defaults. Below is the new listing of `Arithmetic` with the modifications highlighted:

```
package endpoint;
import javax.jws.WebService;
import javax.jws.WebMethod;
@WebService(
            name="Maths",
            serviceName="MathsWebService",
            targetNamespace="http://wsimport")
public class Arithmetic {
    public Arithmetic() {}
    @WebMethod(
                operationName="product")
    public int multiply(int a, int b) {
        int c = a * b ;
        return c;
    }
}
```

First we look at some of the properties of the `@WebService` annotation. The `name` property sets the name associated with the WSDL port type. This defaults to the Java class or interface name, `Arithmetic` in our case.

The `serviceName` property sets the WSDL service name as indicated by the `<service>` element. The default is the Java class name with "Service" appended, `ArithmeticService` in our case.

The `targetNamespace` property sets the XML namespace for WSDL and XSD documents. We described earlier in the WSDL section how the default is derived—in our case the default is `http://endpoint/`.

Next we consider the `@WebMethod` annotation. The `operationName` property sets the name of the WSDL operation. This defaults to the name of the Java method, `multiply` in our case.

We have listed the corresponding WSDL file, with modifications highlighted:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://wsimport" name="MathsWebService"
             xmlns:tns=http://wsimport
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://wsimport"
                  schemaLocation="MathsWebService_schema1.xsd"/>
    </xsd:schema>
  </types>
  <message name="product">
    <part name="parameters" element="tns:product"/>
  </message>
  <message name="productResponse">
    <part name="parameters" element="tns:productResponse"/>
  </message>
  <portType name="Maths">
    <operation name="product">
      <input message="tns:product"/>
      <output message="tns:productResponse"/>
    </operation>
  </portType>
  <binding name="MathsPortBinding" type="tns:Maths">
    <soap:binding transport= "http://schemas.xmlsoap.org/soap/http"
                  style="document"/>
    <operation name="product">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
```

```
      </operation>
    </binding>
    <service name="MathsWebService">
      <port name="MathsPort" binding="tns:MathsPortBinding">
        <soap:address location="http://localhost:8080/
                                arithmetic-webservice/ MathsWebService"/>
      </port>
    </service>
  </definitions>
```

We compile and package the web service in a WAR file named
`arithmetic-service.war` as before. We then run the GlassFish `wsimport`
tool as before. The web service Java client, `WebServiceClient.java`, is listed
below with modifications highlighted:

```
package client;
import javax.xml.ws.WebServiceRef;
import wsimport.MathsWebService;
import wsimport.Maths;
public class WebServiceClient {
    @WebServiceRef(wsdlLocation="http://localhost:8080/
                        arithmetic-webservice/ MathsWebService?WSDL")
    static MathsWebService service;
    public static void main(String[] args) {
        try {
            int a = 3;
            int b = 4;
            Maths port = service.getMathsPort();
            int result = port.product(3, 4);
            System.out.println("Result of multiplying "
                + a + " by " + b + " using webservice is: "
                + result);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Note that the generated proxy classes are now `MathsWebService` and `Maths`. The
names are derived from the new WSDL service name and port type respectively.
They reside in the `wsimport` package, the package name being derived from the
"`http://wsimport`" target namespace.

We obtain a port using the `getMathsPort()` method, rather than the `getArithmeticPort()` method. Finally we invoke the port's `product`, rather than `multiply` operation.

Deploying an EJB Session Bean as a Web Service

We have seen how we can expose a Java application as a web service by deploying it as a WAR file in a web container. JAX-WS also allows a stateless session bean to be exposed as a web service by deploying it as a JAR file in an EJB container. The JAX-WS annotations that we use are the same for a stateless session bean as those we used for a Java application.

Below is a listing for the `BankEndpointBean` stateless session bean which we have modified to be a web service. The EJB has one business method, `addCustomer()`, which adds a new customer to the database. We use the same Customer entity from earlier chapters.

```
package endpoint;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import entity.Customer;
import javax.persistence.PersistenceContext;
import java.util.*;
@Stateless
@WebService
public class BankEndpointBean {
    @PersistenceContext(unitName="BankService")
    private EntityManager em;
    private Customer cust;
    public BankEndpointBean() {}
    @WebMethod
    public String addCustomer(int custId, String firstName,
                              String lastName) {
        cust = new Customer();
        cust.setId(custId);
        cust.setFirstName(firstName);
        cust.setLastName(lastName);
        em.persist(cust);
        return "Customer added";
    }
}
```

As with Java applications, the `@WebService` annotation marks the stateless session bean as a web service. In this example we rely on all the default properties of `@WebService`. The `@WebMethod` annotation identifies the method to be exposed as a web service operation. If there are no `@WebMethod` annotations in the stateless session bean then all public methods will be exposed as web service operations.

Note that the stateless session bean does not need to implement a business interface. However, if we also have clients that will invoke the bean as an EJB rather than web service clients then the bean must implement the business interface.

We can optionally create a service endpoint interface. This is done in exactly the same way as for Java applications as described in the earlier "Creating an Endpoint Implementation Interface" section.

Packaging an EJB Web Service

A stateless session bean web service is deployed in an EJB container. So after compiling the session bean endpoint class we need to package the web service in a JAR file. Below is a listing of the JAR file, which we have named `BankWebService.jar`, for the `BankEndpointBean.java` example:

```
> jar -tvf BankWebService.jar
     0 Mon May 14 09:55:44 BST 2007 META-INF/
   106 Mon May 14 09:55:42 BST META-INF/MANIFEST.MF
     0 Mon May 14 09:50:24 BST endpoint/
   832 Mon May 14 09:55:44 BST endpoint/BankEndpointBean.class
     0 Mon May 14 09:55:12 BST entity/
   997 Mon May 14 09:55:44 BST entity/Customer.class
   193 Wed Nov 01 09:05:10 GMT 2006 META-INF/persistence.xml
```

Note that we have included the entity, `Customer.class`, and the `persistence.xml` file as is usual with EJB 3. Finally we deploy the JAR file. At deployment the WSDL and associated artifacts are automatically created, or as before we can use a tool such as `wsgen` to create these artifacts.

Creating an EJB Web Service Client

As with Java applications exposed as web services, we need to generate proxies from the WSDL document. For GlassFish we use the `wsimport` tool as described earlier. A Java client then invokes the web service through the proxies. Below is a listing for a Java client, `BankClient`, which invokes our session bean web service to add a customer to the database:

```
package client;
import javax.xml.ws.WebServiceRef;
import endpoint.BankEndpointBeanService;
import endpoint.BankEndpointBean;

public class BankClient {
    @WebServiceRef(wsdlLocation="http://localhost:8080/
                   BankEndpointBeanService/BankEndpointBean?WSDL")
    static BankEndpointBeanService service;

    public static void main(String[] args) {
        try {
            int custId = 0;
            String firstName = null;
            String lastName = null;
            try {
                custId = Integer.parseInt(args[0]);
                firstName = args[1];
                lastName =  args[2];
            } catch (Exception e) {
                System.err.println(
                    "Invalid arguments entered, try again");
                System.exit(0);
            }

            BankEndpointBean port =
                service.getBankEndpointBeanPort();
            String result = port.addCustomer(
                              custId, firstName, lastName);
            System.out.println(result);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

The client is similar to the WebServiceClient described earlier. The generated proxy classes are endpoint.BankEndpointBeanService and endpoint. BankEndpointBean. The class and package name are default values; we described their derivation earlier. The wsdlLocation property of the @WebServiceRef annotation specifies the location of the associated WSDL document:

```
"http://localhost:8080/BankEndpointBeanService/
    BankEndpointBean?WSDL"
```

Recall from the earlier section on WSDL that the WSDL location for a session bean deployed as a web service takes the form:

```
http://<hostname>:<port>/<WSDL service name>
```

There is no context root component in the address.

We obtain a port using the `getBankEndpointBeanPort` method as expected. Finally we invoke the port's `addCustomer` operation, again as expected.

Summary

We have covered a lot in this chapter. We had an overview of web service concepts and were introduced to the SOAP, UDDI and WSDL standards.

Any Java EE 5 compliant application server implements the JAX-WS 2.0 specification. This defines the mapping between Java and WSDL. A Java web service implementation class, which can either be a Java application or stateless session bean, is known as an endpoint implementation class

There are two ways of creating a Java web service. The top-down approach is to first create a WSDL document and then generate a Java endpoint implementation from the WSDL. The second approach is to first create a Java endpoint implementation class and then generate the corresponding WSDL document. This is known as the bottom-up approach.

Using the bottom-up approach we just need to add a few metadata annotations to a Java application in order to expose it as a web service. We looked at a simple Java application example and examined the corresponding WSDL document. We looked at the GlassFish `wsgen` tool, which generates a WSDL document from an endpoint implementation.

We deployed a Java application web service as a class in a web container. We saw how to create a Java web service client.

Finally, we looked at an example of exposing a stateless session bean as a web service by deploying it in an EJB container.

12

EJB 3 Security

Security is a wide ranging topic which operates on many levels and covers many technologies. The technologies involved include networks, operating systems, database systems, and application servers as well as manual procedures.

Application level security in the context of Java EE applications is provided by web and EJB containers. Since this is a book about EJB 3, most of this chapter will be concerned with EJB container security. However, as EJBs are often invoked from the web-tier we will take a brief look at web container security later in this chapter.

In this chapter we will cover the following topics:

- Java EE container security
- Authentication
- GlassFish authentication
- Authenticating an EJB client
- Declarative and programmatic EJB authorization
- Web-tier authentication and authorization

Java EE Container Security

There are two aspects covered by Java EE container security: authentication and authorization. **Authentication** is the process of verifying that users are who they claim to be. Typically this is performed by the user providing credentials such as a password. **Authorization**, or access control, is the process of restricting operations to specific users or categories of users. The EJB specification provides two kinds of authorization: declarative and programmatic, as we shall see later in this chapter.

The Java EE security model introduces a few concepts common to both authentication and authorization. A **principal** is an entity that we wish to authenticate. The format of a principal is application-specific but an example is a username. A **role** is a logical grouping of principals. For example, we can have administrator, manager, and employee roles. The scope over which a common security policy applies is known as a **security domain**, or **realm**.

Authentication

For authentication, every Java EE compliant application server provides **the Java Authentication and Authorization Service (JAAS)** API. JAAS supports any underlying security system. So we have a common API regardless of whether authentication is username/password verification against a database, iris or fingerprint recognition for example. The JAAS API is fairly low level and most application servers provide authentication mechanisms at a higher level of abstraction. These authentication mechanisms are application-server specific however. We will not cover JAAS any further here, but look at authentication as provided by the GlassFish application server.

GlassFish Authentication

There are three actors we need to define on the GlassFish application server for authentication purposes: **users, groups**, and **realms**. A user is an entity that we wish to authenticate. A user is synonymous with a principal. A group is a logical grouping of users and is not the same as a role. A group's scope is global to the application server. A role is a logical grouping of users whose scope is limited to a specific application. Of course for some applications we may decide that roles are identical to groups. For other applications we need some mechanism for mapping the roles onto groups. We shall see how this is done later in this chapter. A realm, as we have seen, is the scope over which a common security policy applies.

GlassFish provides three kinds of realms: **file**, **certificate**, and **admin-realm**. The file realm stores user, group, and realm credentials in a file named `keyfile`. This file is stored within the application server file system. A file realm is used by web clients using http or EJB application clients. The certificate realm stores a digital certificate and is used for authenticating web clients using https. The admin-realm is similar to the file realm and is used for storing administrator credentials. GlassFish comes pre-configured with a default file realm named `file`.

We can add, edit, and delete users, groups, and realms using the GlassFish administrator console. We can also use the `create-file-user` option of the `asadmin` command line utility. To add a user named `scott` to a group named `bankemployee`, in the `file` realm, we would use the command:

```
<target name="create-file-user">
  <exec executable="${glassfish.home}/bin/asadmin"
        failonerror="true"
        vmlauncher="false">
  <arg line="create-file-user --user admin
     --passwordfile userpassword --groups bankemployee scott"/>
  </exec>
</target>
```

- `--user` specifies the GlassFish administrator username, `admin` in our example.

- `--passwordfile` specifies the name of the file containing password entries. In our example this file is `userpassword`. Users, other than GlassFish administrators, are identified by `AS_ADMIN_USERPASSWORD`. In our example the content of the `userpassword` file is:

      AS_ADMIN_USERPASSWORD=xyz

  This indicates that the user's password is `xyz`.

- `--groups` specifies the groups associated with this user (there may be more than one group). In our example there is just one group, named `bankemployee`. Multiple groups are colon delineated. For example if the user belongs to both the `bankemployee` and `bankcustomer` groups, we would specify:

      --groups bankemployee:bankcustomer

- The final entry is the operand which specifies the name of the user to be created. In our example this is `scott`.

There is a corresponding `asadmin delete-file-user` option to remove a user from the file realm.

Mapping Roles to Groups

The Java EE specification specifies that there must be a mechanism for mapping local application specific roles to global roles on the application server. Local roles are used by an EJB for authorization purposes as we shall see later in this chapter. The actual mapping mechanism is application server specific. As we have seen in the case of GlassFish, the global application server roles are called groups. In GlassFish, local roles are referred to simply as roles. Suppose we want to map an `employee` role to the `bankemployee` group. We would need to create a GlassFish specific deployment descriptor, `sun-ejb-jar.xml`, with the following `<security-role-mapping>` element:

```
<security-role-mapping>
    <role-name>employee</role-name>
    <group-name>bankemployee</group-name>
</security-role-mapping>
```

We also need to access the configuration-security screen in the administrator console. We then *disable* the `Default Principal To Role Mapping` flag. If the flag is enabled then the default is to map a group onto a role with the same name. So the `bankemployee` group will be mapped to the `bankemployee` role.
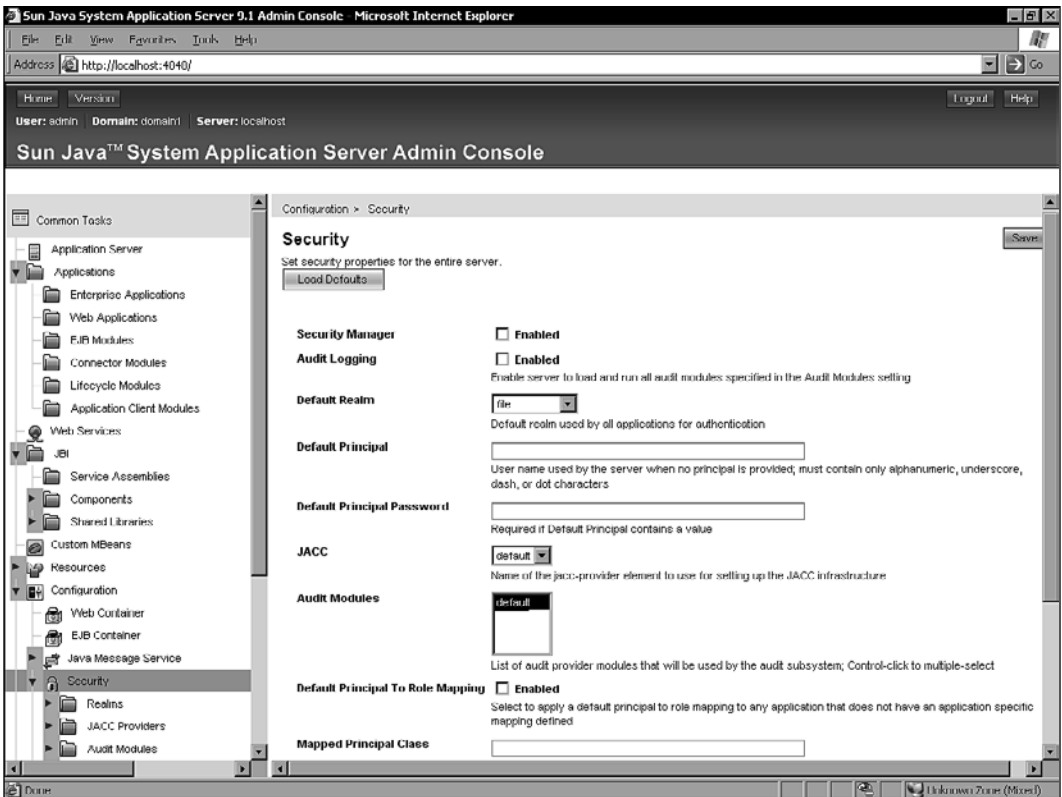
We can leave the default values for the other properties on the configuration-security screen. Many of these features are for advanced use where third party security products can be plugged in or security properties customized. Consequently we will give only a brief description of these properties here.

- **Security Manager:** This refers to the JVM security manager which performs code-based security checks. If the security manager is disabled GlassFish will have better performance. However, even if the security manager is disabled, GlassFish still enforces standard Java EE authentication/authorization.

- **Audit Logging:** If this is enabled, GlassFish will provide an audit trail of all authentication and authorization decisions through **audit modules**. Audit modules provide information on incoming requests, outgoing responses and whether authorization was granted or denied. Audit logging applies for web-tier and ejb-tier authentication and authorization. A default audit module is provided but custom audit modules can also be created.

- **Default Realm:** This is the default realm used for authentication. Applications use this realm unless they specify a different realm in their deployment descriptor. The default value is `file`. Other possible values are `admin-realm` and `certificate`. We discussed GlassFish realms in the previous section.

- **Default Principal:** This is the user name used by GlassFish at run time if no principal is provided. Normally this is not required so the property can be left blank.

- **Default Principal Password:** This is the password of the default principal.

- **JACC:** This is the class name of a JACC (Java Authorization Contract for Containers) provider. This enables the GlassFish administrator to set up third-party plug in modules conforming to the JACC standard to perform authorization.

- **Audit Modules:** If we have created custom modules to perform audit logging, we would select from this list.

- **Mapped Principal Class:** This is only applicable when Default Principal to Role Mapping is enabled. The mapped principal class is used to customize the `java.security.Principal` implementation class used in the default principal to role mapping. If no value is entered, the `com.sun.enterprise.deployment.Group` implementation of `java.security.Principal` is used.

Authenticating an EJB Application Client

Suppose we want to invoke an EJB, `BankServiceBean`, from an application client. We also want the application client container to authenticate the client. There are a number of steps we first need to take which are application server specific. We will assume that all roles will have the same name as the corresponding application server groups. In the case of GlassFish we need to use the administrator console and enable `Default Principal To Role Mapping`.

Next we need to define a group named `bankemployee` with one or more associated users. We have seen how to do this for GlassFish in the previous section.

An EJB application client needs to use **IOR (Interoperable Object Reference)** authentication. The IOR protocol was originally created for **CORBA** (Common Object Request Broker Architecture) but all Java EE compliant containers support IOR. An EJB deployed on one Java EE compliant vendor may be invoked by a client deployed on another Java EE compliant vendor. Security interoperability between these vendors is achieved using the IOR protocol. In our case the client and target EJB both happen to be deployed on the same vendor, but we still use IOR for propagating security details from the application client container to the EJB container.

IORs are configured in vendor specific XML files rather than the standard `ejb-jar.xml` file. In the case of GlassFish, this is done within the `<ior-security-config>` element within the `sun-ejb-jar.xml` deployment descriptor file. We also need to specify the invoked EJB, `BankServiceBean`, in the deployment descriptor. An example of the `sun-ejb-jar.xml` deployment descriptor is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD
  Application Server 9.0 EJB 3.0//EN"
  "http://www.sun.com/software/appserver/dtds/sun-ejb-jar_3_0-0.dtd">
<sun-ejb-jar>
  <enterprise-beans>
    <ejb>
      <ejb-name>BankServiceBean</ejb-name>
      <ior-security-config>
        <as-context>
          <auth-method>USERNAME_PASSWORD</auth-method>
          <realm>default</realm>
          <required>true</required>
        </as-context>
      </ior-security-config>
    </ejb>
  </enterprise-beans>
</sun-ejb-jar>
```

- The `as` in `<as-context>` stands for the IOR **authentication service**. This specifies authentication mechanism details.

- The `<auth-method>` element specifies the authentication method. This is set to USERNAME_PASSWORD which is the only value for an application client.

- The `<realm>` element specifies the realm in which the client is authenticated.

- The `<required>` element specifies whether the above authentication method is required to be used for client authentication.

When creating the corresponding EJB JAR file, the `sun-ejb-jar.xml` file should be included in the `META-INF` directory, as follows:

```
<target name="package-ejb" depends="compile">
        <jar jarfile="${build.dir}/BankService.jar">
            <fileset dir="${build.dir}">
                <include name="ejb30/session/**" />
                        <include name="ejb30/entity/**" />
                    </fileset>
                    <metainf dir="${config.dir}">
                <include name="persistence.xml" />
                        <include name="sun-ejb-jar.xml" />
            </metainf>
        </jar>
    </target>
</target>
```

As soon as we run the application client, GlassFish will prompt with a username and password form, as follows:
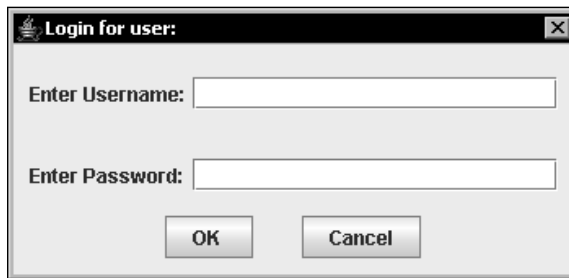
If we reply with the username `scott` and password `xyz` the program will run. If we run the application with an invalid username or password we will get the following error message:

```
javax.ejb.EJBException: nested exception is: java.rmi.AccessException:
CORBA NO_PERMISSION 9998 .....
```

EJB Authorization

Authorization, or access control, is the process of restricting operations to specific roles. In contrast with authentication, EJB authorization is completely application server independent. The EJB specification provides two kinds of authorization: declarative and programmatic. With declarative authorization all security checks are performed by the container. An EJB's security requirements are declared using annotations or deployment descriptors. With programmatic authorization

security checks are hard-coded in the EJBs code using API calls. However, even with programmatic authorization the container is still responsible for authentication and for assigning roles to principals.

Declarative Authorization

As an example, consider the `BankServiceBean` stateless session bean with methods `findCustomer()`, `addCustomer()` and `updateCustomer()`:

```
package ejb30.session;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import ejb30.entity.Customer;
import javax.persistence.PersistenceContext;
import javax.annotation.security.RolesAllowed;
import javax.annotation.security.PermitAll;
import java.util.*;

@Stateless
@RolesAllowed("bankemployee")
public class BankServiceBean implements BankService {

    @PersistenceContext(unitName="BankService")
    private EntityManager em;
    private Customer cust;

    @PermitAll
    public Customer findCustomer(int custId) {
        return ((Customer) em.find(Customer.class, custId));
    }

    public void addCustomer(int custId, String firstName,
                                        String lastName) {
        cust = new Customer();
        cust.setId(custId);
        cust.setFirstName(firstName);
        cust.setLastName(lastName);
        em.persist(cust);
    }

    public void updateCustomer(Customer cust) {
        Customer mergedCust = em.merge(cust);
    }
}
```

We have prefixed the bean class with the annotation:

```
@RolesAllowed("bankemployee")
```

This specifies the roles allowed to access any of the bean's method. So only users belonging to the `bankemployee` role may access the `addCustomer()` and `updateCustomer()` methods. More than one role can be specified by means of a brace delineated list, as follows:

```
@RolesAllowed({"bankemployee", "bankcustomer"})
```

We can also prefix a method with `@RolesAllowed`, in which case the method annotation will override the class annotation. The `@PermitAll` annotation allows unrestricted access to a method, overriding any class level `@RolesAllowed` annotation.

As with EJB 3 in general, we can use deployment descriptors as alternatives to the `@RolesAllowed` and `@PermitAll` annotations.

Denying Authorization

Suppose we want to deny all users access to the `BankServiceBean`. `updateCustomer()` method. We can do this using the `@DenyAll` annotation:

```
@DenyAll
public void updateCustomer(Customer cust) {
    Customer mergedCust = em.merge(cust);
}
```

Of course if you have access to source code you could simply delete the method in question rather than using `@DenyAll`. However suppose you do not have access to the source code and have received the EJB from a third party. If you in turn do not want your clients accessing a given method then you would need to use the `<exclude-list>` element in the `ejb-jar.xml` deployment descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd">
<enterprise-beans>
<session>
<ejb-name>BankServiceBean</ejb-name>
</session>
</enterprise-beans>
<assembly-descriptor>
<exclude-list>
```

```
<method>
    <ejb-name>BankServiceBean</ejb-name>
    <method-name>updateCustomer</method-name>
</method>
</exclude-list>
</assembly-descriptor>
</ejb-jar>
```

EJB Security Propagation

Suppose a client with an associated role invokes, for example, EJB A. If EJB A then invokes, for example, EJB B then by default the client's role is propagated to EJB B. However, you can specify with the `@RunAs` annotation that all methods of an EJB execute under a specific role.

For example, suppose the `addCustomer()` method in the `BankServiceBean` EJB invokes the `addAuditMessage()` method of the `AuditServiceBean` EJB:

```
@Stateless
@RolesAllowed("bankemployee")
public class BankServiceBean implements BankService {

    private @EJB AuditService audit;
    ....
    public void addCustomer(int custId, String firstName,
                                         String lastName) {
        cust = new Customer();
        cust.setId(custId);
        cust.setFirstName(firstName);
        cust.setLastName(lastName);
        em.persist(cust);
        audit.addAuditMessage(1, "customer add attempt");
    }
        ...
}
```

Note that only a client with an associated role of `bankemployee` can invoke `addCustomer()`. If we prefix the `AuditServiceBean` class declaration with `@RunAs("bankauditor")` then the container will run any method in `AuditServiceBean` as the `bankauditor` role, regardless of the role which invokes the method. Note that the `@RunAs` annotation is applied only at the class level, `@RunAs` cannot be applied at the method level.

```
@Stateless
@RunAs("bankauditor")
public class AuditServiceBean implements AuditService {
    @PersistenceContext(unitName="BankService")
    private EntityManager em;
```

```
@TransactionAttribute(
                    TransactionAttributeType.REQUIRES_NEW)
public void addAuditMessage (int auditId,
                                String message) {
    Audit audit = new Audit();
    audit.setId(auditId);
    audit.setMessage(message);
    em.persist(audit);
}
}
```

Programmatic Authorization

With **programmatic authorization** the bean rather than the container controls authorization. The `javax.ejb.SessionContext` object provides two methods which support programmatic authorization: `getCallerPrincipal()` and `isCallerInRole()`. The `getCallerPrincipal()` method returns a `java.security.Principal` object. This object represents the caller, or principal, invoking the EJB. We can then use the `Principal.getName()` method to obtain the name of the principal. We have done this in the `addAccount()` method of the `BankServiceBean` as follows:

```
Principal cp = ctx.getCallerPrincipal();
System.out.println("getname:" + cp.getName());
```

The `isCallerInRole()` method checks whether the principal belongs to a given role. For example, the code fragment below checks if the principal belongs to the `bankcustomer` role. If the principal does not belong to the `bankcustomer` role, we only persist the account if the balance is less than `99`.

```
if (ctx.isCallerInRole("bankcustomer")) {
    em.persist(ac);
} else if (balance < 99) {
    em.persist(ac);
}
```

When using the `isCallerInRole()` method, we need to declare all the security role names used in the EJB code using the class level `@DeclareRoles` annotation:

```
@DeclareRoles({"bankemployee", "bankcustomer"})
```

The code below shows the `BankServiceBean` EJB with all the programmatic authorization code described in this section:

```
package ejb30.session;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import ejb30.entity.Account;
import javax.persistence.PersistenceContext;
```

```
import javax.annotation.security.RolesAllowed;
import java.security.Principal;
import javax.annotation.Resource;
import javax.ejb.SessionContext;
import javax.annotation.security.DeclareRoles;
import java.util.*;

@Stateless
@DeclareRoles({"bankemployee", "bankcustomer"})
public class BankServiceBean implements BankService {

    @PersistenceContext(unitName="BankService")
    private EntityManager em;
    private Account ac;
    @Resource SessionContext ctx;

    @RolesAllowed({"bankemployee", "bankcustomer"})
    public void addAccount(int accountId, double balance,
                           String accountType) {
        ac = new Account();
        ac.setId(accountId);
        ac.setBalance(balance);
        ac.setAccountType(accountType);
        Principal cp = ctx.getCallerPrincipal();
        System.out.println("getname:" + cp.getName());
        if (ctx.isCallerInRole("bankcustomer")) {
            em.persist(ac);
        } else if (balance < 99) {
            em.persist(ac);
        }
    }
    .....
}
```

Where we have a choice declarative authorization is preferable to programmatic authorization. Declarative authorization avoids having to mix business code with security management code. We can change a bean's security policy by simply changing an annotation or deployment descriptor instead of modifying the logic of a business method. However, some security rules, such as the example above of only persisting an account within a balance limit, can only be handled by programmatic authorization. Declarative security is based only on the principal and the method being invoked, whereas programmatic security can take state into consideration.

Java EE Web Container Security

Because an EJB is typically invoked from the web-tier by a servlet, JSP page or JSF component, we will briefly mention Java EE web container security. The web-tier and EJB tier share the same security model. So the web-tier security model is based on the same concepts of principals, roles and realms.

Web-Tier Authorization

Web-tier authorization is similar to EJB authorization but with a few differences. Both programmatic and declarative authorization is supported for the web-tier. For programmatic authorization, the EJB methods of `getCallerPrincipal()` and `isCallerInRole()` have their equivalents of `getUserPrincipal()` and `isUserInRole()` in the `HttpServletRequest` interface. These methods would then be used in a servlet or JSP page. To use these methods we would need to add the `@DeclareRoles` annotation or the `<security-role>` deployment descriptor element as with EJBs. The `@RunAs` annotation can also be used at servlet class level to specify that a servlet runs as a specified role.

There is no annotation equivalent for web-tier declarative authorization; we have to use the `web.xml` deployment descriptor instead. In the code fragment below, the `bankemployee` and `bankcustomer` roles have access to all URLs in the application (specified by the URL pattern of `/*`). Of course we can extend this to allow given roles access to specific URLs.

```
<security-constraint>
<web-resource-collection>
<web-resource-name>bank service example</web-resource-name>
<url-pattern>/*<url-pattern>
</web-resource-collection>
<auth-constraint>
<role-name>bankemployee</role-name>
<role-name>bankcustomer</role-name>
</auth-constraint>
</security-constraint>
```

Recall that the mechanism for mapping local application specific roles to global roles (or roles to groups) on the application server is application server specific. In the case of GlassFish we would use the `<security-role-mapping>` element in the `sun-web.xml` deployment descriptor as we did in the `sun-ejb.xml` deployment descriptor for an EJB application client.

Transport Layer Security

Before we move on to web-tier authentication, we will briefly describe transport layer security, as we refer to this when we configure web-tier authentication. The transport layer is characterized by the **Secure Sockets Layer** (**SSL**) protocol and its successor the **Transport Layer Security** (**TLS**) protocol. Messages are encrypted using public-key encryption, and users are authenticated using digital certificates. The application server presents a certificate to the client's web browser. The purpose of the certificate is to verify that the site is what it claims to be. This is known as **server-side authentication**. The server may request the client to present their digital certificate. This is known as **client-side authentication**. **Mutual authentication** occurs when both server-side and client-side authentication are enabled. The SSL or TLS layers are typically used underneath http. In this case the https URL scheme is used to indicate a secure http connection.

The SSL and TLS protocols correspond to the transport layer of the **Open System Interconnection (OSI)** communication model.

Web-Tier Authentication

We have seen that an EJB application client uses IOR authentication and only one authentication mechanism, `USERNAME_PASSWORD`, is available. Web applications have a number of authentication mechanisms: **basic**, **form** and **client certificate**. Before we can authenticate a web application user we must have set up valid usernames, passwords and roles on the web or application server. This process is vendor specific. We have seen how to do this for the GlassFish application server.

Basic authentication requires a username and password to be entered on a login form supplied by the web container. The username and password are transmitted as part of an http request and authenticated by the web container. This mechanism is not secure. If the transmission is intercepted, the username and password can easily be decoded. Basic authentication is specified in the `<auth-method>` element within the `<login-config>` element in the `web.xml` deployment descriptor as follows:

```
<login-config>
<auth-method>BASIC</auth-method>
</login-config>
```

Form authentication is similar to basic authentication except that the login and error screens can be customized. Form authentication takes places over http and like basic authentication, is not secure. Form authentication is specified in the `web.xml` deployment descriptor as follows:

```
<login-config>
<auth-method>FORM</auth-method>
</login-config>
```

Client certificate authentication requires the web server to authenticate a client using the client's public key certificate. The authentication uses https (http over SSL) so is more secure than either basic or form authentication. Client certificate authentication is specified in the `web.xml` deployment descriptor as follows:

```
<login-config>
<auth-method>CLIENT</auth-method>
</login-config>
```

Example of Web-Tier Authentication and Authorization

We will take the example from the section "UserTransaction Interface" in Chapter 7. In that example we have a servlet, `AddCustomerServlet`, which adds a customer to the database. We will modify the example so that only users belonging to the `bankemployee` role can access the servlet using basic authentication. Basic authentication, you will recall, requires a username and password to be entered on a login form supplied by the web container before any web component such as a servlet can be accessed.

The only modification we need to make is in the `web.xml` file. We have listed the entire `web.xml` file below with the security modifications highlighted:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
       http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <servlet>
      <servlet-name>AddCustomerServlet</servlet-name>
      <servlet-class>ejb30.AddCustomerServlet</servlet-class>
  </servlet>
  <servlet-mapping>
      <servlet-name>AddCustomerServlet</servlet-name>
      <url-pattern>/addCustomer</url-pattern>
  </servlet-mapping>
  <security-constraint>
      <web-resource-collection>
          <web-resource-name>bank service</web-resource-name>
          <url-pattern>/*</url-pattern>
      </web-resource-collection>
      <auth-constraint>
          <role-name>bankemployee</role-name>
```

```
            </auth-constraint>
        </security-constraint>
        <login-config>
            <auth-method>BASIC</auth-method>
            <realm-name>file</realm-name>
        </login-config>
    </web-app>
```

To run the servlet enter `http://localhost:8080/BankService/index.html` in a browser. The address `http://localhost:8080` is the URL for web applications as shown by GlassFish on startup. If GlassFish provides a port other than 8080 then modify the URL accordingly. As soon as the URL has been entered, the web container will present a login form requesting the username and password. If a user belonging to the `bankemployee` role has been correctly entered we proceed to the servlet, otherwise the web container will generate an error page.

Summary

We have seen that there are two aspects to Java EE container security: authentication and authorization. Authentication is the process of verifying that users are who they claim to be. Authorization is the process of restricting operations to specific users or categories of users. The Java EE security model deals with principals, roles and realms. A principal is an entity that we wish to authenticate. A role is a logical grouping of principals. A realm is the scope over which a common security policy applies.

Defining principals, roles, and realms is application server specific; we saw how to do this for GlassFish.

We saw how to configure the GlassFish application client container to authenticate an EJB application client.

There are two kinds of EJB authorization mechanisms: declarative and programmatic. With declarative authorization, security is managed by the container. The EJB specifies the desired access control using annotations. With programmatic authorization, the EJB controls authorization.

Finally we had a brief look at web-tier authorization and authentication.

A

Annotations and Their Corresponding Packages

The following table shows EJB 3 annotations together with their corresponding packages. The list is not exhaustive, only annotations described in this book are shown here.

Annotation	Package
@Remote, @Stateless, @EJB, @Stateful, @Remove, @Local, @TransactionAttribute, @ApplicationException, @MessageDriven, @ActivationConfigProperty, @Timeout	javax.ejb
@Resource, @PostConstruct, @PreDestroy	javax.annotation
@PersistenceContext, @Entity, @Id, @Table,	javax.persistence
@Basic, @TableGenerator, @GeneratedValue, @SequenceGenerator, @Column, @OneToOne, @OneToMany, @ManyToMany, @JoinColumn,	
@JoinTable, @JoinColumns, @Embedded, @Embeddable, @Enumerated, @MapKey, @Lob,	
@Temporal, @IdClass, @Inheritance, @DiscriminatorColumn, @DiscriminatorValue,	
@NamedQuery, @NamedQueries, @SqlResultSetMapping, @EntityResult,	
@FieldResult, @ColumnResult, @SqlResultSetMappings, @NamedNativeQuery,	
@NamedNativeQueries, @PostLoad, @PrePersist,	
@PostPersist, @PreRemove, @PostRemove, @PreUpdate, @PostUpdate, @EntityListeners	
@Version	

Annotation	Package
@AroundInvoke, @Interceptors, @ExcludeClassInterceptors, @ExcludeDefaultInterceptors	javax.interceptor
@WebService, @WebMethod	javax.jws
@WebServiceRef	javax.xml.ws
@RolesAllowed, @PermitAll, @DenyAll, @RunAs, @DeclareRoles	javax.annotation.security

Index

A

ACC
  about 31
  application, building 32, 33
ACID properties, transaction
  atomicity 136
  consistency 136
  durability 136
  isolation 136
activation 40
activation configuration property, MDB 177
aggregate functions, JPQL
  COUNT 102
  GROUP BY clause 102
  HAVING clause 102
  Max 101
  Min 101
  Sum 102
annotations
  @ActivationConfigProperty 177
  @ApplicationException(rollback=true)
      annotation 148
  @AroundInvoke annotation 200, 201
  @Column annotation 76
  @ColumnResult annotation 117
  @DeclareRoles annotation 243
  @DenyAll annotation 241
  @DiscriminatorColumn annotation 95
  @DiscriminatorValue annotation 93, 94
  @EJB annotation 31
  @Embeddable annotation 88
  @Embedded annotation 86
  @Entity annotation 47
  @EntityListeners annotation 133
  @EntityResult annotation 116

@Enumerated annotation 86
@ExcludeClassInterceptors annotation 203
@ExcludeDefaultInterceptors annotation
      206
@FieldResult annotation 116
@GeneratedValue annotation 57-59
@Id annotation 47
@IdClass annotation 90
@Inheritance annotation 93
@Interceptors(Auditor.class) annotation
      206
@Interceptors annotation 203
@JoinColumn annotation 67
@JoinColumns annotation 91
@JoinTable annotation 77
@Lob annotation 88
@Local annotation 42
@ManyToMany annotation 66, 68
@MapKey annotation 87
@MessageDriven annotation 177
@NamedNativeQueries annotation 118
@NamedNativeQuery annotation 117
@NamedQueries 111
@NamedQuery 110
@OneToMany annotation 66, 67
@OneToOne annotation 65, 66
@PermitAll annotation 240
@PersistenceContext 36
@PersistenceContext annotation 51-53
@PostActivate callback method 40
@PostConstruct 35
@PostLoad 130, 131
@PostPersist 132
@PostRemove 132
@PreDestroy method 38
@PrePassivate callback method 40

O

O/R inheritance mapping
  account class, example 92
  joined strategy 94, 95
  single table strategy 92, 93
  strategies 91
  table per concrete class strategy 95, 96
O/R mapping
  about 63
  banking example 64
O/R mapping annotations
  @Embeddable annotation 88
  @Embedded annotation 85
  @Enumerated annotation 86
  @MapKey annotation 87
O/R mapping annotations, banking example
  BankServiceBean 88, 89
  composite primary keys 89-91
  referee class 88
O/R mapping defaults
  banking example 64
O/R mapping defaults, banking example
  account entity 68, 69
  address entity 69, 70
  application, testing 71-74
  customer entity 65-68
  referee entity 70, 71
O/R mapping defaults overriding, banking example
  account entity, modifying 79, 80
  address entity, modifying 80, 81
  BankServiceBean, modifying 82-84
  customer entity, modifying 75-78
  UML diagram, for bidirectional model 74
object/relational mapping.
    See **O/R mapping**
Open System Interconnection (OSI) 246
operators
  = (equal) operator 99
  BETWEEN operator 100
  DISTINCT operator 99
  IS NULL operator 100
  LIKE operator 100
  NOT BETWEEN operator 100
  NOT IN operator 100

optimistic locking, post update problem
  about 152
  implementing, by version fields 152, 153

P

packages 249, 250
packaging, entity 54
parameters
  named parameters 110
  named queries 110
  positional parameters 109
passivation 40
persistence context 50
persistence engine 8
persistence provider 8
persistence context, entity manager 123
pessimistic locking, post update problem 152
point-to-point mode, JMS API 160
primary keys
  auto strategy 59, 60
  generating, strategies 57
  identity strategy 59
  sequence strategy 58, 59
  table strategy 57, 58
principal 234
programmatic authorization 243, 244
projection, JPQL 99
publish/subscribe mode, JMS API 160

Q

queries
  with date parameters 113
  with parameters 109
  with relationships 103
queries, JPQL 98
queries with parameters
  named parameters 110
  positional parameters 109
queue producer and consumer, example
  about 161-163
  asynchronous queue consumer 167, 168
  running 165, 166
  synchronous queue consumer 163-165

R

read lock **154**
role **234**

S

security **233**
security manager **237**
sequence strategy, primary keys **58, 59**
server-side authentication **246**
session bean
  life cycle, callback methods 40
session beans
  about 9
  stateless session beans 22
SessionSynchronization interface
  implementing 146-148
Simple Object Access Protocol.
     See **SOAP protocol**
SOAP protocol 212
stateful session beans
  @PostActivate callback method 40
  @PrePassivate callback method 40
  about 36
  activation 39
  annotations, creating 23
  deployment descriptors 23, 24
  example 23, 37, 38
  lifecycle 39
  passivation 40
  session bean client, creating 25, 26
  source code, compiling 26
stateful session beans lifecycle versus
     stateless session beans lifecycle 39
stateless session beans
  about 22
  adminpassword, directory structure 28
  build, directory structure 29
  build.xml, directory structure 28
  env.properties, directory structure 28
  example 22
  lifecycle 34-36
  program directory structure 28
  src, directory structure 28
subqueries, JPQL
  ALL expression 106

  ANY expression 106
  EXISTS expression 106
surrogate key 89
synchronous queue consumer, example
  about 163-165
  running 165, 166
synchronous topic consumer, example
  about 170, 171
  running 171

T

table strategy, primary keys 57, 58
timer interface
  cancelTimers() method 195
  createTimer() method 191
  example 194
  getHandle() method 196
  getInfo() method 195
  getNextTimeout() method 195
  getTimeRemaining() method 195
  methods 195
  TimerHandler.getTimer() method 196
  TimerService.createTimer() method 189
  TimerService.getTimers() method 193
timers and transactions 196
timer service. *See* **EJB timer service**
timer service, example
  createTimer() method 191
  internal event example 192, 193
  single event example 190, 191, 192
  TimerService.createTimer() method 189
  TimerService.getTimers() method 193
TLS. *See* **Transport Layer Security**
top-down approach, Java web service 214
topic producer and consumer, example
  about 169, 170
  asynchronous topic consumer 172, 173
  running 171
  synchronous topic consumer 170, 171
transaction
  about 136
  ACID properties 136
  container-managed transaction demarca-
     tion 136
 dooming 148
  example 136

Business Process Management with JBoss jBPM

ISBN: 978-1-847192-36-3   Paperback: 300 pages

Develop business process models for implementation in a business process management system

1. Map your business processes in an efficient, standards-friendly way

2. Use the jBPM toolset to work with business process maps, create a customizable user interface for users to interact with the process, collect process execution data, and integrate with existing systems.

3. Use the SeeWhy business intelligence toolset as a Business Activity Monitoring solution, to analyze process execution data, provide real-time alerts regarding the operation of the process, and for ongoing process improvement
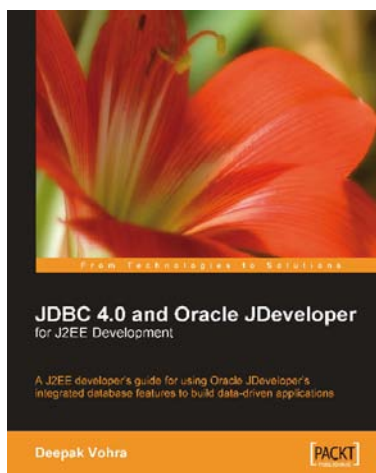
## JDBC 4.0 and Oracle JDeveloper for J2EE Development

ISBN: 978-1-847194-30-5   Paperback: 431 pages

A J2EE developer's guide to using Oracle JDeveloper's integrated database features to build data-driven applications

1. Develop your Java applications using JDBC and Oracle JDeveloper

2. Explore the new features of JDBC 4.0

3. Use JDBC and the data tools in Oracle JDeveloper

4. Configure JDBC with various application servers

Please check **www.PacktPub.com** for information on our titles
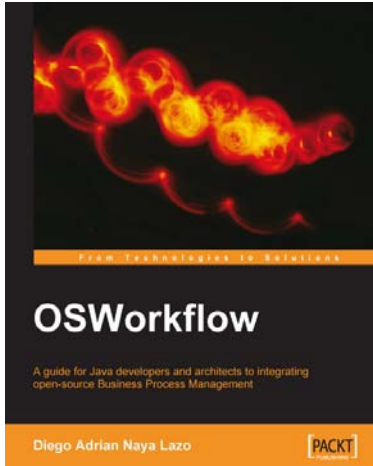
OSWorkflow

ISBN: 978-1-847191-52-6   Paperback: 200 pages

Get your workflow up and running with this step-by-step guide authored by an active developer of the OSWorkflow project with real-world examples

1. Basics of OSWorkflowl

2. Integrating business rules with Drools

3. Task scheduling with Quartz

Java EE 5 Development using GlassFish Application Server

ISBN: 78-1-847192-60-8   Paperback: 400 pages

The complete guide to installing and configuring the GlassFish Application Server and developing Java EE 5 applications to be deployed to this server

1. Concise guide covering all major aspects of Java EE 5 development

2. Uses the enterprise open-source GlassFish application server

3. Explains GlassFish installation and configuration

4. Covers all major Java EE 5 APIs

Please check **www.PacktPub.com** for information on our titles