SAMPLE CHAPTER

EJB3 IN ACTION

Debu Panda
Reza Rahman
Derek Lane

MANNING

EJB 3 in Action

by Debu Panda

Reza Rahman

Derek Lane

Sample Chapter 11

brief contents

PART 1 OVERVIEW OF THE EJB LANDSCAPE 1

1 ▪ What's what in EJB 3 3

2 ▪ A first taste of EJB 33

PART 2 UILDING BUSINESS LOGIC WITH EJB 3 69

3 ▪ Building business logic with session beans 71

4 ▪ Messaging and developing MDBs 110

5 ▪ Learning advanced EJB concepts 140

6 ▪ Transactions and security 176

PART 3 DIVING INTO THE JAVA PERSISTENCE API (JPA) 215

7 ▪ Implementing domain models 217

8 ▪ Object-relational mapping 250

9 ▪ Manipulating entities with EntityManager 293

10 ▪ Using the query API and JPQL to retrieve entities 340

PART 4 PUTTING EJB 3 INTO ACTION 385

11 ▪ Packaging EJB 3 applications 387

12 ▪ Effectively integrating EJB 3 across your application tiers 425

13 ▪ Taming wild EJBs: performance and scalability 455

PART 5 MIGRATION AND INTEROPERABILITY ISSUES 497

14 ▪ Migrating to EJB 3 499

15 ▪ Exposing EJBs as web services 537

16 ▪ EJB 3 and Spring 568

11 Packaging EJB 3 applications

This chapter covers

- Class loading concepts
- Packaging EJB 3 components
- Packaging EJB 3 entities
- O/R mapping with XML
- Deployment issues and best practices

In the previous chapters you learned how to build a business-logic tier with session and message-driven beans, and you used entities to support the persistence tier. The real success of Java EE applications lies in assembly and deployment, as this is the key to delivering on Java's promise of write once, run anywhere (WORA). If you fail to fully grasp this step, your application may not realize this level of portability.

A typical application has a handful of Java classes, and maintenance can be a nightmare if you are shipping your applications from one environment to another. To simplify maintenance you can create a Java archive (JAR) file. Typically, a JAR

Java platform roles: it's all about juggling hats

The Java EE platform defines different roles and responsibilities relating to development, assembly, and deployment of Java EE applications. In this book we are mainly interested in the Developer, Assembler, and Deployer roles, but we introduce you to all the roles so that you can be familiar with them. The roles defined by the specifications are

- Enterprise Bean Provider
- Application Assembler
- Deployer
- EJB Server Provider
- EJB Container Provider
- Persistence Provider
- System Administrator

The database administrator is not one of the defined Java EE roles. The database administrator may not even understand a line of Java code. However, the importance of this role cannot be overlooked, especially in large corporations where relational databases are outside the control of the application developers. Developers, Assemblers, and Deployers may need to work with the DBAs in order to successfully build and release Java EE applications.

It's all about the division of labor. Many believe that the difficulties of earlier EJB practices were a result of the division of the EJB roles. In reality, the previous EJB specifications were not the real culprit—the source of all the confusion is the Java EE specification. While the Java EE and EJB specifications define seven roles, the problem is that many project teams do not even have seven people—how can a two- or three-person team wear that many hats?

file is a file in zip format that contains classes. However, enterprise Java applications are packaged as specialized versions of JAR files—EAR, WAR, and EJB-JAR modules—before they can be deployed to a Java EE–compliant application server.

In this chapter we begin with a discussion of application packaging and deployment. The chapter also provides critical information on class loading, so that you can appreciate why the archives are packaged as they are. This is intended to provide you a better understanding of the packaging requirements for EJBs that include entities. We explain the need for deployment descriptors, and look at how to use them. Finally, we look at a persistence unit and how to perform object-relational (O/R) mapping using XML.

11.1 Packaging your applications

A typical enterprise Java application may contain several Java classes of different types, such as EJBs, servlets, JavaServer Faces (JSF) managed beans, and entity classes, as well as static files such as JSPs and HTML files. As we discussed in chapter 1, EJBs run in the EJB container whereas web applications such as servlets and JSF managed beans run in the web container. To run your application you have to make it available to the Java EE application server. This is known as *deployment*. Since EJB is a core part of the Java EE specification, you have to follow the Java EE standard for deployment.

To understand EJB packaging, you must consider how it fits into the bigger picture of Java EE packaging and know what constitutes a complete enterprise Java application. Up to this point we have focused on using EJB components such as session beans and MDBs to build business logic and JPA entities to implement your persistence code. However, your application will not be complete without a presentation tier that accesses the business logic you built with EJBs. For example, the EJBs we built for ActionBazaar do not make sense unless we have a client application accessing them. Most likely, you've used standard technologies such as JSP or JSF to build the web tier of your applications. These web applications, together with EJBs, constitute an enterprise application that you can deploy to an application server.

To deploy and run an application, you have to package the complete application together—the web module and EJBs—and deploy to an application server. Usually you will group similar pieces of the application together in modules. Java EE defines a standard way of packaging these modules in JAR files, and specifies the formats for these JARs. One of the advantages of having these formats defined as part of the specification is that they are portable across application servers.

Table 11.1 lists the archives or modules supported by Java EE 5 and their contents. Note that each archive type is used for packaging a specific type of module, such as EJB or web. For instance, a WAR is used to package a web-tier application module, and the EAR file is intended to be the über archive containing all the other archives so that in the end, you're only deploying one file. The application server will scan the contents of the EAR and deploy it. We discuss how an EAR is loaded by the server in section 11.1.2.

Table 11.1 Enterprise Java applications need to be assembled into specific types of JAR files before they can be deployed to an application server. These are the available module types as specified by Java EE.

Type	Description	Descriptor	Contents
CAR	Client application archives	`application-client.xml`	Thick Java client for EJBs.
EAR	Enterprise application archive	`application.xml`	Other Java EE modules such as EJB-JARs.
EJB-JAR	EJB Java archive	`ejb-jar.xml`	Session beans, message-driven beans, and optionally entities. Needs a `persistence.xml` if entities are packaged.
RAR	Resource adapter archives	`ra.xml`	Resource adapters.
WAR	Web application archives	`web.xml`	Web application artifacts such as servlets, JSPs, JSF, static files, etc. Entities can also be packaged in this module. Needs a `persistence.xml` if entities are packaged.

To create these files, you can use the `jar` utility that comes with JDK. The final step is to assemble all the JAR files into one EAR file for deployment. In 11.3.1 we show you a build script that creates a JAR file. Each of these JAR types contains an optional deployment descriptor that describes the archive. As we have been discussing throughout this book, you can use metadata annotations instead of a deployment descriptor.

In this chapter, we focus primarily on the EAR file and the EJB-JAR file, which contains the session and message-driven beans, as well as entities.

It's worth mentioning that entities can be packaged in most archive types. For example, the ability to package entities in WARs allows you to use the EJB 3 JPA in simple web applications or with lightweight frameworks such as Spring. Note that entities are not supported in RAR modules. This statement, however, begs the question of why Java EE does not have a different archive type to package EJB 3 entities, just as JBoss has the Hibernate Archive (HAR) to package persistence objects with Hibernate's O/R framework.

You may know the answer to this question if you have followed the evolution of the EJB 3 specification. For those who haven't, we now regale you with Tales from the Expert Group (cue spooky music)…

During the evolution of the EJB 3 Public Draft, the PAR (Persistence Archive) was introduced, which mysteriously vanished in the Proposed Final Draft. A huge, emotional battle was fought in the EJB and Java EE expert groups over whether to introduce a module type for a persistence module at the Java EE level, and suggestions were sought from the community at large, as well as from various developer forums. Many developers think a separate persistence module is a bad idea because entities are supported both outside and inside the container. Considering that persistence is inherently a part of any enterprise application, it makes sense to support packaging entities with most module types, instead of introducing a new module type specialized for packaging entities.

Now that you know what modules are supported and a little about how they were arrived at, shall we take a quick peek under the hood of an EAR module?

11.1.1 Dissecting the EAR file

To understand how deployment works, let's take a closer look at the EAR file, the top-level archive file that contains other Java EE archives when it is deployed to the application server. For instance, the ActionBazaar application contains an EJB module, a web module, a JAR containing helper classes, and an application client module. The file structure of the EAR file that ActionBazaar uses looks like this:

```
META-INF/application.xml
actionBazaar-ejb.jar
actionBazaar.war
actionBazaar-client.jar
lib/actionBazaar-commons.jar
```

`application.xml` is the deployment descriptor that describes the standard Java EE modules packaged in each EAR file. The contents of `application.xml` look something like listing 11.1.

Listing 11.1 Deployment descriptor for the ActionBazaar EAR module

```
<application>
  <module>
    <ejb>actionBazaar-ejb.jar</ejb>                    <-- EJB module
  </module>
  <module>
    <web>
      <web-uri>actionBazaar.war</web-uri>              <-- Web module
      <context-root>ab</context-root>
    </web>
  </module>
  <module>
    <java>actionBazaar-client.jar</java>               <-- Application client module
  </module>
</application>
```

If you review the EAR file descriptor in listing 11.1, you'll see that it explicitly identifies each of the artifacts as a specific type of module. When you deploy this EAR to an application server, the application server uses the information in the deployment descriptor to deploy each of the module types.

Java EE 5 made the deployment descriptor optional, even in the EAR. This is a departure from previous versions of Java EE, where it was mandatory. The Java EE 5.0–compliant application servers deploy by performing automatic detection based on a standard naming convention or reading the content of archives; see http://java.sun.com/blueprints/code/namingconventions.html.

Next, let's take a look at how application servers deploy an EAR module.

11.1.2 *Loading the EAR module*

During the deployment process, the application server determines the module types, validates them, and takes appropriate steps so that the application is available to users. Although all application servers have to accomplish these goals, it's up to the individual vendor exactly how to implement it. One area where server implementations stand out is in how fast they can deploy the archives.

While vendors are free to optimize their specific implementation, they all follow the specification's rules when it comes to what is required to be supported and in what order the loading occurs. This means that your application server will use the algorithm from figure 11.1 when attempting to load the EAR file that contains modules or archives from table 1.1.

Before we delve into how EJB components and entities are packaged, let's briefly discuss what class loading is and how it works in the Java EE environment.

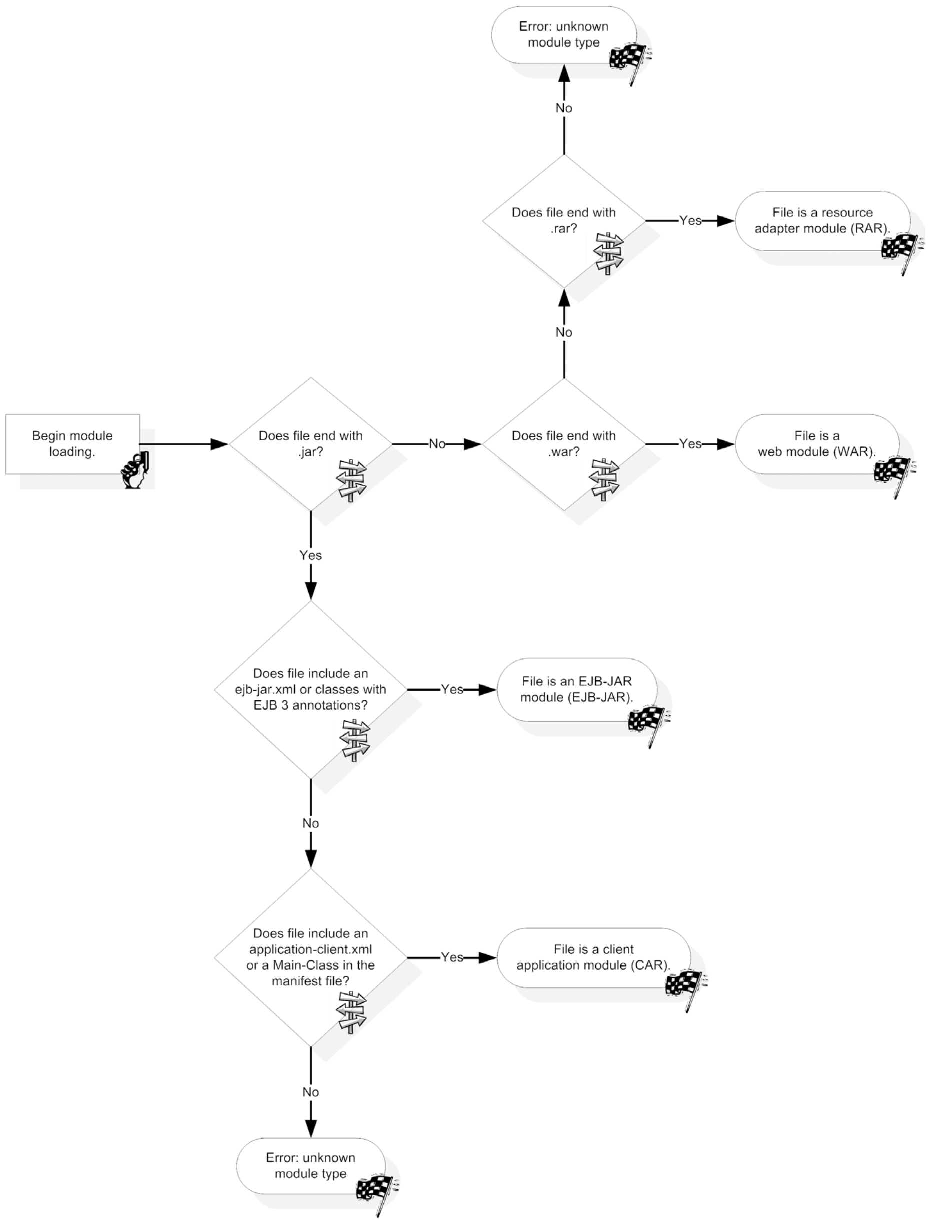
Error: unknown module type
No
Does file end with .rar?
Yes
File is a resource adapter module (RAR).
No
Begin module loading.
Does file end with .jar?
No
Does file end with .war?
Yes
File is a web module (WAR).
Yes
Does file include an ejb-jar.xml or classes with EJB 3 annotations?
Yes
File is an EJB-JAR module (EJB-JAR).
No
Does file include an application-client.xml or a Main-Class in the manifest file?
Yes
File is a client application module (CAR).
No
Error: unknown module type

Figure 11.1 Rules followed by application servers to deploy an EAR module. Java EE 5 does not require a deployment descriptor in the EAR module that identifies the type of modules packaged. It is the responsibility of Java EE container to determine the type of module based on its name (extension) and its content. It does so by following this algorithm.

11.2 Exploring class loading

There is a misconception among many developers that all classes are loaded into memory when the JVM starts up; *this is not true*. Classes are loaded dynamically as and when they are needed at runtime. This process of locating the byte code for a given class name and converting that code into a Java *class* instance is known as *class loading*. Your application may have hundreds of EJBs and other resources; loading all these classes into the JVM consumes a lot of memory. Most application servers use a sophisticated mechanism to load classes as and when needed. Therefore, your EJB class will be loaded into memory only when a client accesses it. However, it is implementation specific. Application servers support the bean pooling mechanism, so EJB classes would be loaded into memory while some instances would be instantiated and put into the pool during deployment time.

When you build an application using EJB 3, you may use third-party libraries such as Log4J or you may depend on an in-house shared library configured in the application server. You may have web applications that depend on your EJB components and entities. As you can see, a complex application may depend on libraries available at several places. This means that you may run into many deployment errors such as `ClassNotFoundException` or `ClassNoDefException`. Understanding the class-loading concepts will educate you on effectively packaging your EJB 3 applications and help you troubleshoot any deployment-related issues.

In this section, we introduce the concept of class loading and look at the class-loader hierarchy in an application server. We then expose the parent delegation model. Finally, we examine class loading in Java EE and explore the dependencies between different modules.

11.2.1 Class-loading basics

If you've built simple applications with Java, you must be aware that when you run your application, the classes that make it up (often packaged in a standard JAR file) are made available to the JVM through the `CLASSPATH` environment variable. When a particular class is invoked, the JVM loads that class into memory by locating it from the available byte code files provided either via JAR files in the `CLASSATH` or a specified directory structure.

Class loading is initially performed by the JVM when it starts up. It loads the essential classes required, and then subclasses of the `java.lang.ClassLoader` class take the lead. These class loaders allow applications to load classes dynamically that may not be required during the compilation process. By default, the JVM utilizes a few different class loaders. As an illustration, the Sun JVM has a hierarchy of three loaders, as shown in figure 11.2

The boot class loader loads all platform classes that the Java language requires, such as classes in the `java.lang` or `java.util` package. You can optionally use the `bootclasspath` command-line option of the JVM to instruct the boot class loader to load additional classes from other JAR files.

The extension class loader is a child class loader of the boot class loader, and loads classes from any JARs placed in the `$JAVA_HOME/jre/lib/ext` directory, or in a separate directory specified with the `-Djava.ext.dir` system property. By default, it loads the Java cryptography library, as well as the security classes.

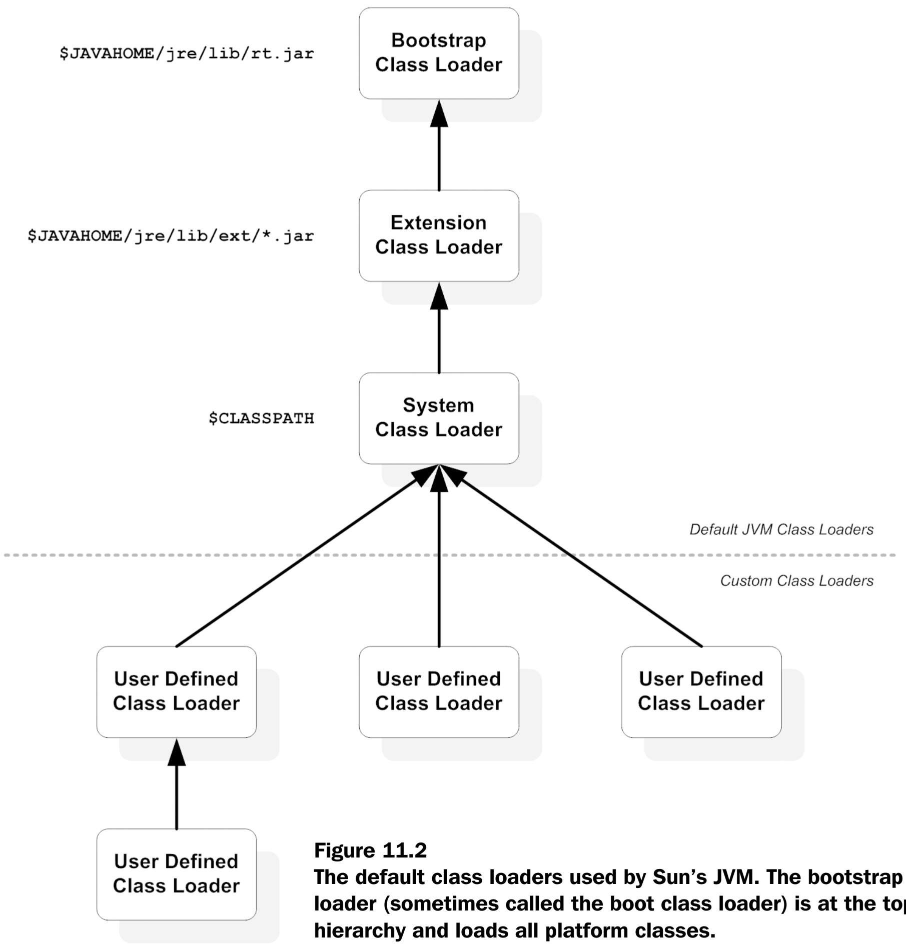
$JAVAHOME/jre/lib/rt.jar
Bootstrap Class Loader
$JAVAHOME/jre/lib/ext/*.jar
Extension Class Loader
$CLASSPATH
System Class Loader
Default JVM Class Loaders
Custom Class Loaders
User Defined Class Loader
User Defined Class Loader
User Defined Class Loader
User Defined Class Loader

Figure 11.2
The default class loaders used by Sun's JVM. The bootstrap class loader (sometimes called the boot class loader) is at the top of the hierarchy and loads all platform classes.

The system class loader actually loads application classes as specified by an application, and is also known as the application class loader. You can use several mechanisms to specify the location from which the system class loader loads classes. One way is to specify the CLASSPATH environment variable. Another is to specify the manifest Class-Path entry of a JAR file that is being executed or that is in the CLASSPATH.

For example, the JAR file actionBazaar-client.jar has a Manifest.mf file in the META-INF directory that has this entry:

```
Class-Path:  lib/actionBazaar-utility.jar.
```

When the class loader loads the classes, it will search not only for the required class in the actionBazaar-client.jar, but also in the actionBazaar-utility.jar. The location of the JAR specified in the manifest Class-Path is relative to the JAR file that contains it.

For a simple Java application, this process is probably as simple as packaging the classes in a JAR and making the file available in the CLASSPATH. However, in a sophisticated environment such as Java EE, the application servers utilize several mechanisms to load the classes from a variety of locations, such as an application module, a library module, or a shared library configured in the application server environment.

When you start up an application server, a Java process starts loading classes required for the application server. When you deploy and execute an application in a Java application server, the application server loads the classes dynamically by creating new instances of class loaders.

11.2.2 Exposing the classic parent delegation model

You must be curious as to why JVM always loads the class from the parent class loader. In this section we will uncover the reason.

Let's review the scenario for ActionBazaar in order to understand the class-loading delegation model. The ActionBazaar website is built with JSP pages that invoke EJBs. When a user visits the ActionBazaar website and browses the items listed for auction, the application server uses class loaders to dynamically load required classes from application modules. All class loaders follow a standard algorithm to load classes, as illustrated in figure 11.3.

A class loader loads a class dynamically on an as-needed basis. It first looks at its local cache to see if it was loaded earlier. If not, it asks its parent to load the class. If its parent cannot load the class, it attempts to load it from its local code sources. Simply put, a code source is a base location, such as a JAR file,

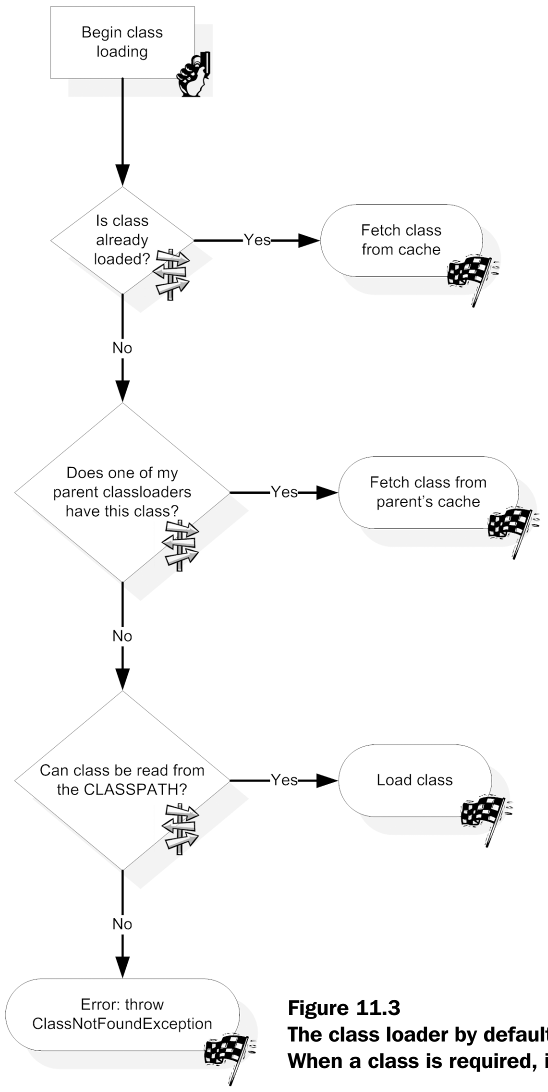
Begin class loading
Is class already loaded?
Yes
Fetch class from cache
No
Does one of my parent classloaders have this class?
Yes
Fetch class from parent's cache
No
Can class be read from the CLASSPATH?
Yes
Load class
No
Error: throw ClassNotFoundException

Figure 11.3
The class loader by default follows Parent First Delegation model. When a class is required, it first asks its parent to load the class.

which the JVM searches for classes. This approach is called the Parent First delegation model.

Now that we've reviewed the basics of Java class loading, let's quickly review how class loading works in a Java EE application.

11.2.3 Class loading in Java EE applications

As we discussed earlier, an EJB application may make use of third-party libraries. In order to enable that, most Java EE containers use sophisticated mechanisms to load classes from a variety of places. You may remember from previous discussions that we follow standard practices to package our application components into standard-compliant archives such as EAR, EJB-JAR, WAR, and so forth. Table 11.2 lists the code sources for commonly used Java EE modules. For simplicity we are ignoring resource adapter (RAR) modules.

Table 11.2 A standard archive may load classes either packaged inside it or from any other archives it is dependent on.

Module	Code Sources
EAR	1. All JARs in the `/lib` directory of the EAR 2. Manifest `Class-Path` of any JARs in 1
EJB-JAR	1. EJB-JAR file itself 2. JARs referenced by manifest `Class-Path` of EJB-JAR 3. JARs referenced by manifest `Class-Path` of above JARs (in 2)
WAR	1. `WEB-INF/classes` 2. JARs in `WEB-INF/lib` 3. JARs referenced by manifest `Class-Path` of WAR 4. JARs referenced by manifest `Class-Path` of JARs in 2 and 3

The sooner you develop a good understanding of how the packaging standards work, the easier the whole packaging and deployment process will be.

11.2.4 Dependencies between Java EE modules

Unfortunately, no Java EE specification provides a standard for class loading, and each application server implements class loaders in whatever way seems best to the vendor. However, Java EE defines the visibility and sharing of classes between different modules, and we can depict the dependency between different modules as shown in figure 11.4.

As illustrated in figure 11.4, the EAR class loader loads all JARs in the `lib` directory that is shared between multiple modules. Typically a single EJB class loader loads all EJB classes packaged in all EJB-JAR modules. The EJB class loader is often the child of the application class loader, and loads all EJB classes. Because the EJB is a child to the EAR class loader, all classes loaded at the EAR level will be visible to the EJBs.

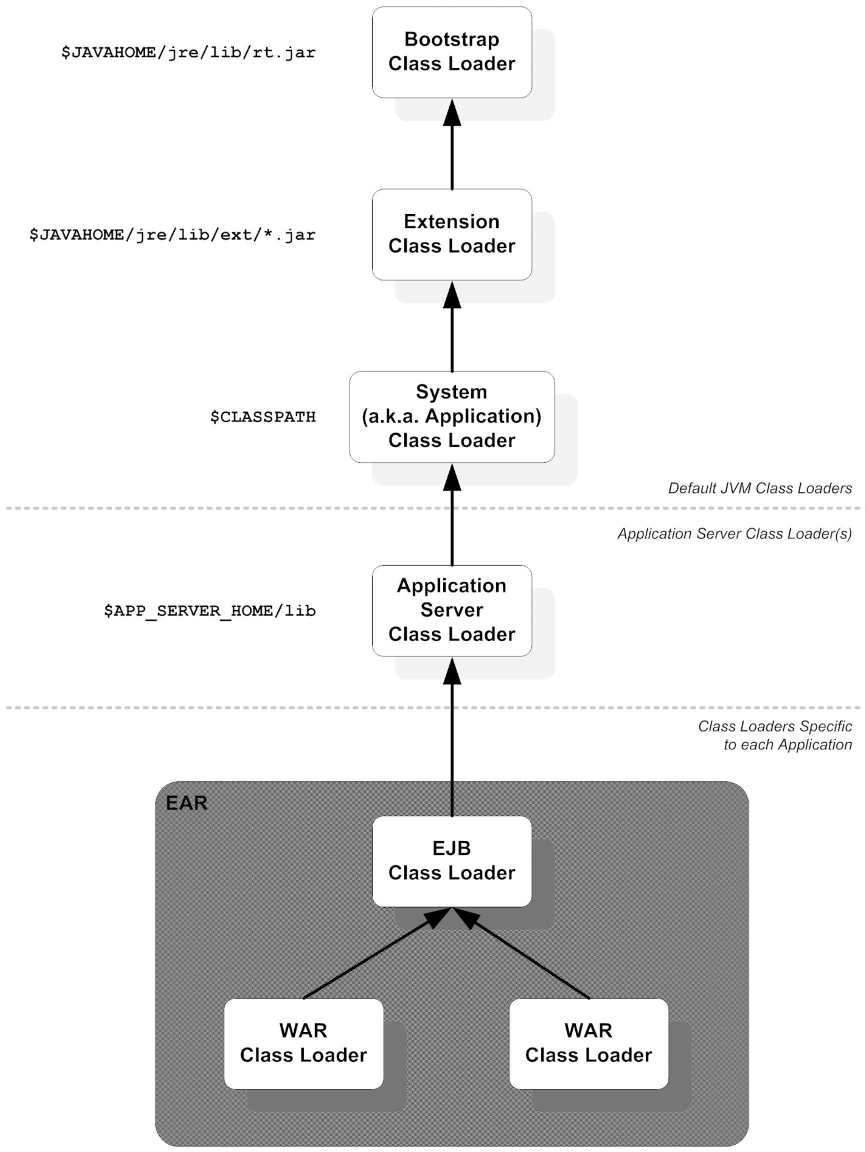
$JAVAHOME/jre/lib/rt.jar
Bootstrap Class Loader
$JAVAHOME/jre/lib/ext/*.jar
Extension Class Loader
$CLASSPATH
System (a.k.a. Application) Class Loader
Default JVM Class Loaders
Application Server Class Loader(s)
$APP_SERVER_HOME/lib
Application Server Class Loader
Class Loaders Specific to each Application
EAR
EJB Class Loader
WAR Class Loader
WAR Class Loader

Figure 11.4 Illustration of class visibility of an EAR file containing multiple web modules, EJBs, and shared library modules. The EAR class loader loads the classes in the JARs packaged as library modules, and all classes loaded by the EAR class loader are visible to the EJBs. The classes loaded by EJB class loader are typically visible to the web module in most containers because the WAR class loader is a child of the EJB class loader.

EJBs are accessible from WAR modules. Furthermore, the EJB class loader is the parent of the WAR application class loader, and all EJB classes will be visible to the WAR module by default.

So before we move on to packaging EJBs, let's recap how this is going to help in packaging EJB 3 applications. If you package classes in a specific EJB module, it will probably be visible to only that module. If you want your classes (helper and utility) to be visible to all modules in the EAR file, you can package them as a library module in the EAR.

Armed with this knowledge on class loading, we can now return to the discussion on packaging EJBs. First we'll talk about the packaging of session and message-driven beans, and quickly proceed to the packaging of persistence entities.

11.3 Packaging session and message-driven beans

A car manufacturer has to assemble all essential parts of a car before it can run. As an EJB developer you build core classes that make your application, and you have to assemble them as an EJB-JAR and deploy them into your application server before your customers can execute the application.

Throughout this book we have used annotations and avoided deployment descriptors. The EJB deployment descriptor (`ejb-jar.xml`) describes the contents of an EJB-JAR, such as beans, interceptors, the resource they use, security, transaction settings, and so forth. For every annotation we have discussed in this book there is an element in the descriptor. You'll recall from chapter 2 that deployment descriptors can be used to override settings in metadata annotations. Let's now uncover the elements of `ejb-jar.xml` and explain how you can define default interceptors. We'll conclude this section with a discussion on vendor-specific descriptors and annotations.

11.3.1 Packaging EJB-JAR

Session beans and MDBs can be packaged in a Java standard JAR file as defined in the Java Archive specification at http://java.sun.com/j2se/1.5.0/docs/guide/jar/. To create an EJB-JAR file to package your EJB components, you have to compile your EJB classes and then create a JAR file using the `jar` tool supplied by JDK. For example, you can use the following command to create the `adventure-ejb.jar`:

```
jar cvf adventure-ejb.jar *
```

This will create a JAR file containing all class files in the current directory, and any subdirectories below the current directory. You can automate building JAR files using several tools. Most modern IDEs support building EJB-JAR modules, and make the creation of JAR modules somewhat transparent to you. A number of specialized utilities in addition to IDEs also support the build process. Today, the most frequently used tool to assist with builds is Apache Ant (http://ant.apache.org/), although there is a strong movement toward Apache Maven (http://maven.apache.org/). Listing 11.2 shows a sample Ant build script that was created to automate building an EJB-JAR module. Ant build scripts are provided with our code examples and can be downloaded from this book's website (www.manning.com/panda).

Listing 11.2 Sample script for building an EJB-JAR file

```
...
  <target name="compile-ejb-classes" depends="setup">     <-- Compiles EJB classes
    <echo message="-----> Compiling EJBs"/>
    <javac srcdir="${src.ejb.dir}"
      destdir="${bld.ejb.dir}"
      debug="on">
    <classpath>
      <pathelement path="${common.j2ee.class.path}"/>
      <pathelement location="${bld.ejb.dir}"/>
      <pathelement location="${lib.dir}/${ejb.name}.jar"/>
    </classpath>
   </javac>
  </target>

  <target name="ejb-descriptor" depends="setup">     <-- Copies deployment descriptors
    <copy todir="${bld.ejb.dir}/META-INF">
      <fileset dir="${etc.dir}"
               includes="ejb-jar.xml, persistence.xml"/>
    </copy>
  </target>

  <target name="package-ejb"
          depends="compile-ejb-classes,ejb-descriptor">     <-- Builds EJB-JAR
    <echo message="-----> Create EJB JAR file"/>
    <jar jarfile="${bld.ear.dir}/${ejb.name}.jar">
      <fileset dir="${bld.ejb.dir}" includes="**"/>
    </jar>
  </target>
...
```

The EJB-JAR file must include the interfaces and bean classes. It may also include any helper classes. Optionally the helper classes may be packaged in a separate JAR file in the EAR file. You have two options:

- The JAR containing helper classes may be packaged in the `lib` directory of the EAR file. Using this approach, the packaged classes will be automatically visible to all modules in the EAR module.
- If you want to limit the visibility to only a specific EJB-JAR or WAR module, you can create an entry in the `Manifest.mf` file of the module that contains a `Class-Path` attribute to the JAR file.

Now that you know the structure of EJB-JAR and how to package it, let's look at the elements of `ejb-jar.xml`.

11.3.2 Deployment descriptors vs. annotations

An EJB deployment descriptor (`ejb-jar.xml`) describes the contents of an EJB module, any resources used by it, and security transaction settings. The deployment descriptor is written in XML, and because it is external to the Java byte code, it allows you to separate concerns for development and deployment.

The deployment descriptor is optional and you could use annotations instead, but we don't advise using annotations in all cases for several reasons. Annotations are great for development, but may not be well suited for deployments where settings may change frequently. During deployment it is common in large companies for different people to be involved for each environment (development, test, production, etc.). For instance, your application requires such resources as `DataSource` or JMS objects, and the JNDI names for these resources change between these environments. It does not make sense to hard-code these names in the code using annotations. The deployment descriptor allows the deployers to understand the contents and take appropriate action. Keep in mind that even if the deployment descriptor is optional, certain settings such as default interceptors for an EJB-JAR module require a deployment descriptor. An EJB-JAR module may contain

- A deployment descriptor (`ejb-jar.xml`)
- A vendor-specific deployment descriptor, which is required to perform certain configuration settings in a particular EJB container

The good news is that you can mix and match annotations with descriptors by specifying some settings in annotations and others in the deployment descriptor. Be aware that the deployment descriptor is the final source and overrides settings provided through metadata annotations. To clarify, you could set the `TransactionAttribute` for an EJB method as `REQUIRES_NEW` using an annotation, and if you set it to `REQUIRED` in the deployment descriptor, the final effect will be `REQUIRED`.

Annotations vs. XML descriptors: the endless debate

Sugar or sugar substitute? It’s a matter of choice. Zero calories versus the risk of cancer? The debate may well be endless, and the same applies to the debate between annotations and deployment descriptors. Some people find annotations elegant, while they see XML as verbose, ugly, and hard to maintain. Others find annotations unsightly, and complain that annotations complicate things by making configurations reside closer to the code. The good thing is that you have a choice, and Java EE allows you to override annotation settings in the code with deployment descriptors if you desire. We suggest you weigh the pros and cons of these options with a clear mind.

Although we won’t delve deeply into deployment descriptors, let’s look at some quick examples to see what deployment descriptors look like so that you can package a deployment descriptor in your EJB module if you need to. Listing 11.3 shows a simple example of a deployment descriptor for the `BazaarAdmin` EJB.

Listing 11.3 A simple ejb-jar.xml

```
<ejb-jar version="3.0">       <-❶ Specifies version element (must be 3.0)
  <enterprise-beans>
    <session>                                  ❷ Identifies EJB
      <ejb-name>BazaarAdmin</ejb-name>     <-┘
      <remote>actionbazaar.buslogic.BazaarAdmin</remote>
      <ejb-class>actionbazaar.buslogic.BazaarAdminBean</ejb-class>
      <session-type>stateless</session-type>                <-┐ Specifies
      <transaction-type>Container</transaction-type>   <-┐    ❸ bean type
    </session>                                          │
  </enterprise-beans>                        Specifies   │
...                                   transaction type ❹
  <assembly-descriptor>
    <container-transaction>     <-❺ Contains transaction attribute setting
      <method>
        <ejb-name>BazaarAdmin</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
    <security-role>
      <role-name>users</role-name>     <-❻ Specifies security setting
    </security-role>
  </assembly-descriptor>
</ejb-JAR>
```

If you are familiar with EJB 2, you may have noticed that the only notable difference between this deployment descriptor and one in EJB 2 is that the `version` attribute must be set to `3.0`, and the `home` element is missing because EJB 3 does not require a home interface.

If you are using deployment descriptors for your EJBs, make sure that you set the ejb-jar `version` to `3.0` ❶ because this will be used by the Java EE server to determine the version of the EJBs being packaged in an archive. The `name` element ❷ identifies an EJB and is the same as the `name` element in the `@Stateless` annotation. These must match if you are overriding any values specified in the annotation with a descriptor. The `session-type` element ❸ determines the type of session bean. This value can be either `stateless` or `stateful`. You can use `transaction-type` ❹ to specify whether the bean uses CMT (`Container`) or BMT (`Bean`). The transaction, security, and other assembly details are set using the `assembly-descriptor` tag of the deployment descriptor ❺ and ❻.

Table 11.3 lists commonly used annotations and their corresponding descriptor tags. Note that as we mentioned earlier there is an element for every annotation. You will need only those which make sense for your development environment. Some of the descriptor elements you'll probably need are for resource references, interceptor binding, and declarative security. We encourage you to explore these on your own.

Table 11.3 One-to-one mapping between annotations and XML descriptor elements

Annotation	Type	Annotation Element	Corresponding Descriptor Element
`@Stateless`	EJB type		`<session-type>Stateless`
		`name`	`ejb-name`
`@Stateful`	EJB type		`<session-type>Stateful`
			`ejb-name`
`@MessageDriven`	EJB type		`message-driven`
		`name`	`ejb-name`
`@Remote`	Interface type		`remote`
`@Local`	Interface type		`local`
`@Transaction-Management`	Transaction management type at bean level		`transaction-type`

continued on next page

Table 11.3 One-to-one mapping between annotations and XML descriptor elements *(continued)*

Annotation	Type	Annotation Element	Corresponding Descriptor Element
`@Transaction-Attribute`	Transaction settings method		`container-transaction` `trans-attribute`
`@Interceptors`	Interceptors		`interceptor-binding` `interceptor-class`
`@ExcludeClass-Interceptors`	Interceptors		`exclude-class-interceptor`
`@ExcludeDefault-Interceptors`	Interceptors		`exclude-default-interceptors`
`@AroundInvoke`	Custom interceptor		`around-invoke`
`@PreConstruct`	Lifecycle method		`pre-construct`
`@PostDestroy`	Lifecycle method		`post-destroy`
`@PostActivate`	Lifecycle method		`post-activate`
`@PrePassivate`	Lifecycle method		`pre-passivate`
`@DeclareRoles`	Security setting		`security-role`
`@RolesAllowed`	Security setting		`method-permission`
`@PermitAll`	Security setting		`unchecked`
`@DenyAll`	Security setting		`exclude-list`
`@RunAs`	Security setting		`security-identity` `run-as`
`@Resource`	Resource references (`DataSource`, JMS, Environment, mail, etc.)		`resource-ref` `resource-env-ref` `message-destination-ref` `env-ref`
	Resource injection	Setter/field injection	`injection-target`
`@EJB`	EJB references		`ejb-ref` `ejb-local-ref`
`@Persistence-Context`	Persistence context reference		`persistence-context-ref`
`@PresistenceUnit`	Persistence unit reference		`persistence-unit-ref`

You can find the XML schema for the EJB 3 deployment descriptor at http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd.

11.3.3 Overriding annotations with deployment descriptors

As we explained, you can mix and match deployment descriptors with annotations and use descriptors to override settings originally specified using annotations. Keep in mind that the more you mix the two, the more likely you are to make mistakes and create a debugging nightmare.

NOTE The basic rule to remember is that the `name` element in stateless, stateful, and message-driven annotations is the same as the `ejb-name` element in the descriptor. If you do not specify the `name` element with these annotations, the name of the bean class is understood to be the `ejb-name` element. This means that when you are overriding an annotation setting with your deployment descriptor, the `ejb-name` element must match the bean class name.

Suppose we have a stateless session bean that uses these annotations:

```
@Stateless(name = "BazaarAdmin")
public class BazaarAdminBean implements BazaarAdmin {
...
@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
public Item addItem() {
  }
}
```

The value for the `name` element specified is `BazaarAdmin`, which is the same as the value of the `ejb-name` element specified in the deployment descriptor:

```
<ejb-name>BazaarAdmin</ejb-name>
```

If you do not specify the `name` element, the container will use the name of `BazaarAdminBean` as the name of the bean class, and in order to override annotations you have to use that name in the deployment descriptor:

```
<ejb-name>BazaarAdminBean</ejb-name>
```

We used `@TransactionAttribute` to specify that the transaction attribute for a bean method be `REQUIRES_NEW`. If we want to override it to use `REQUIRED`,[1] then we use the following descriptor:

[1] Keep in mind the impact of changing a transaction attribute from `RequiresNew` to `Required`, as shown in this example. We investigated this effect in greater detail in chapter 6.

```
<assembly-descriptor>
  <container-transaction>
    <method>                                        ❶ Specifies ejb-name
      <ejb-name>BazaarAdmin</ejb-name>         <─┘
      <method-name>getUserWithItems</method-name>
      <method-params></method-params>
    </method>
    <trans-attribute>Required</trans-attribute>   <─┐ Changes transaction
  </container-transaction>                          ❷ attribute setting
</assembly-descriptor>
```

In this example, we used the `assembly-descriptor` element to specify a transaction attribute ❷. In addition, the `ejb-name` element ❶ in the `assembly-descriptor` matches the original `name` specified with the `@Stateless` annotation in the bean class.

11.3.4 Specifying default interceptor settings

Interceptors (as you'll recall from chapter 5) allow you to implement cross-cutting code in an elegant manner. An interceptor can be defined at the class or method level, or a default interceptor can be defined at the module level for all EJB classes in the EJB-JAR. We mentioned that default interceptors for an EJB module can only be defined in the deployment descriptor (`ejb-jar.xml`). Listing 11.4 shows how to specify default interceptors for an EJB module.

Listing 11.4 Default interceptor setting in ejb-jar.xml

```
...                          ❶ Defines interceptor binding
<interceptor-binding>   <─┘
  <ejb-name>*</ejb-name>   <─❷ Applies binding to all EJBs
  <interceptor-class>
    actionbazaar.buslogic.CheckPermissionInterceptor
  </interceptor-class>
  <interceptor-class>
    actionbazaar.buslogic.ActionBazaarDefaultInterceptor
  </interceptor-class>
</interceptor-binding>
...
```

The `interceptor-binding` ❶ tag defines the binding of interceptors to a particular EJB with the `ejb-name` element. If we want to define the default interceptor or an interceptor binding for all EJBs in the EJB module, then we can specify `*` as the value for `ejb-name` ❷. We specify a class to use as the interceptor with the `<interceptor-class>` tag. As evident from the listing, you can specify multiple interceptors in the same binding, and the order in which they are specified in the deployment descriptor determines the order of execution for the interceptor. In our example, `CheckPermissionInterceptor` will be executed prior to `ActionBazaarDefaultInterceptor` when any EJB method is executed.

If you want a refresher on how interceptors work, make a quick detour back to chapter 5 and then rejoin us here. We'll wait…

11.3.5 Using vendor-specific annotations and descriptors

We've already explained that stateless session beans and MDBs may be pooled. In addition, you can configure passivation for stateful session beans, and you can set up the handling of poisonous messages for MDBs. However, we have not discussed configuration details for either of these scenarios. Unfortunately, these configurations are left to the vendors as proprietary features, and they can be supported with proprietary annotations, proprietary deployment descriptors, or both. Table 11.4 lists the name of the deployment descriptor file for some popular application servers.

Table 11.4 Vendor-specific deployment descriptors for popular application servers

Application Server	Vendor-Specific Deployment Descriptor
BEA WebLogic	`weblogic-ejb-jar.xml`
IBM WebSphere	`ibm-ejb-jar.xml`
JBoss	`jboss.xml`
Oracle Application Server	`orion-ejb-jar.xml`
Sun GlassFish	`sun-ejb-jar.xml`

Many developers shun deployment descriptors as a matter of inconvenience. Application server vendors will continue to provide support for annotations that match deployment descriptor elements, as developers voice their preference for these features. Chances are that each vendor has a set of proprietary annotations to set configuration information with the code.

For example, you can use the `oracle.j2ee.ejb.StatelessDeployment` proprietary annotation to provide configuration information such as pooling and transaction management for stateless session beans. Look at the following code, which configures pooling with Oracle's proprietary annotation:

```
import oracle.j2ee.ejb.StatelessDeployment;

@StatelessDeployment(
  minInstances = 100, maxInstances = 500, poolCacheTimeout = 120)
@Stateless(name = "BazaarAdmin")
public class BazaarAdminBean implements BazaarAdmin {
}
```

As other Java EE vendors create their implementations of EJB 3, we anticipate that each vendor will devise its own subset of corresponding annotations as well.

You should review these proprietary annotations with caution for a couple of reasons. First, adding configuration information in the code is not a good idea, although application servers provide the ability to override this information with their proprietary deployment descriptors. This is not desirable because in order to make a change to the setting, the code must be edited and compiled, and in most organizations it must go through a significant quality assurance effort before being released to production. Another reason is that as the code is promoted across different environments (Development, Test, Production, etc.), the deployer may change the configuration to accommodate different servers and environmental configurations.

Second, this defeats the goal of portability of applications. Deployment descriptors serve as a guideline to the deployer to understand the contents, the applications, and the suggested configurations. Deployers manage the deployment to each environment by tweaking the configuration. We recommend using the proprietary deployment descriptors instead of using deployment annotations. If you're using Oracle, you could use the following element in Oracle's proprietary descriptor (`orion-ejb-jar.xml`) element as follows:

```
<session-deployment
  name = "BazaarAdmin"
  tx-retry-wait = "60"
  max-instances = "500"
  min-instances = "100"
  pool-cache-timeout = "120"
  location = "BazaarAdmin">
</session-deployment>
```

This concludes our discussion on packaging session beans and message-driven beans. Next we take a peek at packaging entities. Can you feel the anticipation building?

11.4 Packaging entities

Can't you package EJB 3 entities in the same way? Afraid not. We're sure you've noticed that while session and message-driven beans share a lot of characteristics, entities are quite another beast. You may remember from our discussion in chapter 1 that JPA can be used directly from the web container. That means entities will need some additional care and feeding with respect to packaging, so that deployment will work as expected.

This section covers some new deployment files, `persistence.xml` and `orm.xml`, and provides a slew of tips and information on how to position your entities for maximum deployment enjoyment. You do want deployment enjoyment, don't you? We know we do. Let's begin by looking at the packaging structure for entities.

11.4.1 Exposing the persistence module

With EJB 3, entities can be used inside either the EJB or web container, or in a Java SE application. Thus, entities may be packaged in a standard Java EE module such as an EJB-JAR, WAR, or JAR file in the root of the EAR module or as a library module in an EAR. When using entities in your applications, you have to package entity classes into a Java EE module such as an EJB-JAR or WAR of simple JAR files, with a simple deployment descriptor named `persistence.xml`.

If you are using entities within an EJB module, then the EJB (session beans, MDBs) classes and entities need to be packaged together in the same EJB module. Therefore, the EJB-JAR module will contain a `persistence.xml` file to designate that the module contains one or more persistence units. Recall from our discussion in chapter 9 that a persistence unit is a logical group of entities used together. For example, you may create a persistence unit for all entities in the ActionBazaar application.

Let's look at the structure of a JAR that contains a simple persistence unit, as shown in listing 11.5.

Listing 11.5 Structure of a sample EJB-JAR file containing entities

```
ActionBazaar-ejb.jar:
META-INF/
  persistence.xml
orm.xml (optional)    <-❶ Default O/R mapping file
actionbazaar/
  persistence/
    Category.class
    Item.class
    ...
    BazaarAdmin.class
...
    secondORMap.xml      <-2  Additional O/R mapping file
```

persistence.xml is the deployment descriptor for a persistence module, which is discussed in the next section. The orm.xml ❶ file defines the object-relational mapping (if you use XML mapping). You may package an additional mapping file ❷ that defines O/R mapping for entities that was not defined in orm.xml. We discuss O/R mapping with XML in section 11.5.2. The JAR also contains entity classes—Category.class and Item.class—and another class, BazaarAdmin.class, that is needed in order to make persistence work. Now that you know the structure of a persistence module, let's drill down and learn more about persistence.xml.

11.4.2 Describing the persistence module with persistence.xml

In chapter 9 we showed you how to group entities as a persistence unit and how to configure that unit using persistence.xml. Now that you know how to package entities, it's time to learn more about persistence.xml, the descriptor that transforms any JAR module into a persistence module. It's worth mentioning that persistence.xml is the only mandatory deployment descriptor that you have to deal with. We hope the Java EE specification will ease this requirement in future releases of the specification.

At the time of this writing, some EJB containers such as Oracle and JBoss support proprietary extensions of persistence modules without persistence.xml in EJB-JAR modules. Although user-friendly, this feature will not be portable across EJB 3 containers. You can find the schema for persistence.xml online at http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd.

Listing 11.6 is an example of a simple persistence.xml that we can use with the ActionBazaar application; it should successfully deploy to any Java EE 5 container that supports JPA. The first thing the file does is define a persistence unit and package it to your deployment archive—for example, WAR or EJB-JAR.

Listing 11.6 An example persistence.xml

```
<persistence>
  <persistence-unit name = "actionBazaar"    <-1  Persistence unit
                    transaction-type = "JTA">
    <provider>                                 <-2  Factory class for JPA provider
      oracle.toplink.essentials.PersistenceProvider
    </provider>
    <jta-data-source>jdbc/ActionBazaarDS          ❸ DataSource used by
        </jta-data-source>                          persistence unit
    <mapping-file>secondORMap.xml</mapping-file>
    <jar-file>entities/ShippingEntities.jar</jar-file>
    <class>ejb3inaction.persistence.Category</class>   ❹ Entity classes
    <class>ejb3inaction.persistence.Bid</class>.          included in unit
...
    <properties>      <property name = "toplink.ddl-generation"
            value = "drop-and-create-tables"/>   <-
                                                    Vendor-specific
    </properties>                                ❺ properties
  </persistence-unit>
</persistence>
```

Let's run through a quick review of the code before we drill down into the details. We define a persistence unit by using the `persistence-unit` element ❶. We can specify an optional factory class for the persistence provider ❷. The JPA provider connects to the database to store retrieved entities; we specified the data source for the persistence provider ❸. If you have multiple persistence units in a single archive, you may want to identify the entity classes that comprise the persistence unit ❹. Optionally, you can specify vendor-specific configuration using the `properties` element ❺.

We hope you're ready for a detailed exploration on the use of each of the elements in `persistence.xml` from listing 11.6—you'll find it pretty straightforward. After reading the next few pages, you should be fairly comfortable with this new part of the EJB standard.

Naming the persistence unit

Each persistence unit must have a `name`, and that `name` must be unique across the Java EE module. The `name` is important because the container uses it to create an entity manager factory, and then again to create the entity manager instances using the factory to access the entities specified inside the unit. Also, you access the persistence unit by using its `name` when you attempt to perform CRUD operations with the entities packaged in the module. All other elements in a persistence unit can be defaulted.

You can define more than one persistence unit for an application module in `persistence.xml`. All entities identified in a persistence unit are managed by a single set of entity instances. Thus, a `persistence.xml` may have multiple persistence units in a particular JAR module as follows:

```
<persistence>
  <persistence-unit name = "actionBazaar">
...
  </persistence-unit>
  <persistence-unit name = "humanResources">
...
  </persistence-unit>
</persistence>
```

Persistence unit scoping

You can define a persistence unit in a WAR, EJB-JAR, or JAR at the EAR level. If you define a persistence unit in a module, it is only visible to that specific module. However, if you define the unit by placing a JAR file in the `lib` directory of the EAR, the persistence unit will automatically be visible to all modules in the EAR. For this to work, you must remember the restriction that if the same name is used by a persistence unit in the EAR level and at the module level, the persistence unit in the module level will win.

Assume you have an EAR file structure like this:

```
lib/actionBazaar-common.jar
actionBazaar-ejb.jar
actionBazaar-web.war
```

`actionBazaar-common.jar` has a persistence unit with the name `actionBazaar` and `actionBazaar-ejb.jar` has also a persistence unit with the name `actionBazaar`.

The `actionBazaar` persistence unit is automatically visible to the web module, and you can use as follows:

```
@PersistenceUnit(unitName = "actionBazaar")
private EntityManagerFactory emf;
```

However, if you use this code in the EJB module, the local persistence unit will be accessed because the local persistence unit has precedence. If you want to access the persistence unit defined at the EAR level, you have to reference it with the specific name as follows:

```
PersistenceUnit(unitName =
 "lib/actionBazaar-common.jar#actionBazaar")
private EntityManagerFactory emf;
```

Again, the `name` element is important because it is what you use to access the entities. As shown in chapter 9, we use `unitName` to inject a container-managed `EntityManager` as follows:

```
@PersistenceContext(unitName = "actionBazaar")
private EntityManager entityManager;
```

Refer to the sidebar "Persistence unit scoping" for more on how a persistence unit is scoped depending on its presence.

Specifying the transaction type

You can specify `transaction-type` in `persistence.xml` (as in listing 11.6) by using the `transaction-type` attribute. `transaction-type` can either be `JTA` or `RESOURCE_LOCAL`. If you do not specify `transaction-type`, the container will assume the default `transaction-type` is `JTA`. You must utilize `JTA` as the `transaction-type` for a persistence unit packaged in a Java EE module. `RESOURCE_LOCAL` should be specified as a transaction type only when you're exercising JPA outside a Java EE container. As you may recall, we discussed the `javax.persistence.EntityTransaction` interface in chapter 9; we recommend you avail yourself of `EntityTransaction` only when you use EJB 3 persistence outside of a Java EE environment.

Using a specific persistence provider

The `provider` element specifies the factory class of the EJB 3 persistence provider, such as Hibernate or TopLink. You do not have to specify the persistence provider if you're using the default persistence provider integrated with your Java EE 5 container. For example, if you want Hibernate's persistence provider in the JBoss Application Server or TopLink Essentials persistence provider with Sun GlassFish or the Oracle Application Server, you don't have to define the `provider` element in `persistence.xml`. But if you decide to go with the EJB 3 persistence provider from the GlassFish project with either JBoss or Apache Geronimo, then you must specify the `provider` element as follows:

```
<provider>oracle.toplink.essentials.PersistenceProvider</provider>
```

Obviously this example specifies Oracle TopLink as the persistence provider; you can specify the `provider` element for Hibernate as follows:

```
<provider>org.hibernate.ejb.HibernatePersistence</provider>
```

This is helpful when using JPA outside the container.

Setting up a DataSource

Our entities are persistence objects that access databases. Chapters 7 through 10 discussed how O/R mappings are defined with metadata annotations, and how an entity interacts with one or more database tables. We have not, however, broached the subject of how entities interact with a database connection. Back in chapters 3 and 4 we briefly discussed what a `DataSource` is and how it can be used in an application server by accessing it through JNDI. In addition, you saw examples of session and message-driven beans accessing a `DataSource` using resource injection. In spite of this, entities cannot use injection, connect to the database themselves, or perform any operation directly; the persistence provider does all that magic behind the scenes. When you persist an instance of an entity, the persistence provider will open or reuse a pooled connection to the database and execute the SQL on your behalf.

To configure a persistence unit to connect to a database, you first have to create a `DataSource` in your Java EE container. For scalability, each `DataSource` is commonly associated with a connection pool, and the connection pool contains the information for connecting to the database.

Configuring an application DataSource

Every Java EE application server provides the ability to create and manage `DataSources` and connection pools. Here is an example of a `DataSource` and a connection pool used by Sun's GlassFish open source project:

```
<jdbc-connection-pool
    connection-validation-method = "auto-commit"
    datasource-classname = "oracle.jdbc.pool.OracleDataSource"
    max-pool-size = "32"
    max-wait-time-in-millis = "60000"
    name = "ActionBazaarDS"
    res-type = "javax.sql.DataSource"
    steady-pool-size = "8">
  <property name = "user" value = "ejb3ina"/>
  <property name = "port" value = "1521"/>
  <property name = "password" value = "ejb3ina"/>
  <property name = "networkProtocol" value = "thin"/>
  <property name = "databaseName" value = "ORCL"/>
  <property name = "serverName" value = "localhost"/>
</jdbc-connection-pool>

<jdbc-resource enabled = "true"
               jndi-name = "jdbc/ActionBazaarDS"
               pool-name = "ActionBazaarDS"/>
```

The `DataSource` uses the JNDI name and connection pool information for the specified database instance. In this example, the `DataSource` has a `jndi-name` of `jdbc/ActionBazaarDS`. Two common naming techniques are to name the pool either the DataSource name without the JNDI reference (ActionBazaarDS), or to use the pool in the DataSource name (ActionBazaarPooledDS). We'll illustrate the first approach here.

Telling the persistence unit about the DataSource

You can specify the DataSource for a persistence unit using either the jta-data-source or non-jta-data-source element in the persistence.xml (as we did in listing 11.6). Typically, Java EE containers support two types of DataSources: Java Transaction API (JTA) and non-JTA. A JTA (or global) DataSource is one that supports JTA or distributed transactions. A non-JTA (or local) DataSource only supports local transactions that are limited to the process/server where they begin. For example, we can specify the name of the JTA DataSource we created earlier using the jta-data-source element as follows:

```
<jta-data-source>jdbc/ActionBazaarDS</jta-data-source>
```

NOTE You have to specify global JNDI names for the data source in the jta-data-source and non-jta-data-source elements of persistence.xml. If you do not specify a DataSource for the persistence unit, the persistence unit will try to use the default DataSource for the application server. The default DataSource for a Java EE application is typically specified using a proprietary mechanism.

Many application servers such as BEA WebLogic Server and Oracle Application Server also allow the packaging of DataSource configurations in an EAR.

Identifying entity classes

If you are using JPA within a Java EE container, the persistence provider reads the module and determines which entity classes are annotated with the @Entity annotation. You can identify the entity classes that constitute a persistence unit (as we did in listing 11.6). This is useful when you want to divide the packaged entities into more than one persistence unit as follows:

```
<persistence>
  <persistence-unit name = "actionBazaar">
    <class>ejb3inaction.persistence.Category</class>
    <class>ejb3inaction.persistence.Bid</class>
 ...
  </persistence-unit>
  <persistence-unit name = "humanResources">
    <class>ejb3inaction.persistence.Employee</class>
    <class>ejb3inaction.persistence.Department</class>
  ...
   </persistence-unit>
</persistence>
```

Packaging at this more granular level may seem like more work at first glance. In reality, it makes sharing the persistence units across applications much easier.

Specifying vendor-specific extensions

Most JPA providers will provide extensions such as caching, logging, and automatic table creation. You can use the `property` element in `persistence.xml` to specify these vendor-specific extensions. The persistence provider will read such configurations while creating the entity manager factory and configure the persistence unit accordingly.

In listing 11.6 we enabled automatic schema generation for the persistence unit when using TopLink:

```
<properties>
      <property name = "toplink.ddl-generation"
                value = "drop-and-create-tables"/>
      <property name =
           "toplink.ddl-generation.output-mode"
          value = "database"/>
    </properties>
```

Remember that automatic schema generation is a developer-friendly feature and, when it's turned on, the JPA provider creates the underlying database schema (tables, sequences, etc.) when the persistence unit is deployed. If you want to turn on automatic schema generation for Hibernate, you can do so by adding the following in `persistence.xml`:

```
<property name="hibernate.hbm2ddl.auto" value="create-drop"/>
```

Similarly you can pass configuration parameters for caching, logging, and JDBC configuration when using an outside container as a property. Check your vendor documentation for details.

Specifying additional mapping and JAR files

There may be times when you want to use multiple O/R mapping files for your project. Doing this supports the packaging of smaller functional units into separate JAR files to allow a more granular deployment scheme. Of course, regardless of how many JARs make up your application, they will all need to be in the classpath of the application in order for all the components to be found by the class loader.

For example, if you have mapping information in a separate XML file named `secondORMap.xml`, you can specify as much by using the `mapping-file` element that we saw in listing 11.6. It is vital to remember that additional mapping files are not packaged in the `META-INF` directory of the persistence module, but also that these files may be packaged inside the JAR as a resource (as shown in listing 11.5).

You can include additional JAR files (such as `ShippingEntities.jar` in listing 11.6). The JAR file location is relative to the persistence module; that is, the JAR file that contains the `persistence.xml`.

11.4.3 Performing O/R mapping with orm.xml

Chapter 8 discussed how to perform O/R mapping using metadata annotations. Believe it or not, for a large application the use of O/R mapping metadata *within* the code is not a good idea. Using O/R mapping annotations hardwires your relational schema to your object model. Some folks feel it's perfectly okay to hardcode schema information, because they see it as being similar to JDBC. Others consider it a very bad idea. It is also quite possible that for certain projects you may be directed to implement O/R mapping with an XML file. As mentioned earlier (in listing 11.1), you can specify O/R mapping information in a file named `orm.xml` packaged in the `META-INF` directory of the persistence module, or in a separate file packaged as a resource and defined in `persistence.xml` with the `mapping-file` element.

The source that takes precedence is always the deployment descriptor. EJB 3 persistence specifies that the deployment descriptor can override O/R mapping specified using annotations, the `orm.xml` file, or any other XML mapping. Listing 11.7 shows an example of an O/R mapping file (`orm.xml`) used in ActionBazaar.

Listing 11.7 An orm.xml that specifies default values for a persistence unit and O/R mapping information

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings version="1.0"
  xmlns=http://java.sun.com/xml/ns/persistence/orm
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
  orm_1_0.xsd">

  <persistence-unit-metadata>
    <persistence-unit-defaults>     <-❶ Defines persistence unit defaults
      <schema>ACTIONBAZAAR</schema>
      <access>PROPERTY</access>
        <entity-listeners>         <-❷ Specifies default entity listeners
          <entity-listener
```

```
            class = "actionbazaar.persistence.DefaultListener">
         ...
          </entity-listener>
        </entity-listeners>
    </persistence-unit-defaults>
  </persistence-unit-metadata>

  <package>actionbazaar.persistence</package>
  <access>PROPERTY</access>

  <named-query name = "findAllCategories">
    <query>SELECT c FROM Category AS c</query>
    <hint name = "refresh" value = "true"/>
  </named-query>

 <entity name = "Category" class = "Category" metadata-complete = "false">    Specifies entity mapping ❸
    <table name = "CATEGORIES" />
    <sequence-generator name = "CATEGORY_SEQ_GEN"
                        sequence-name = "CATEGORY_SEQ"
                        allocation-size = "1"
                        initial-value = "1"/>
    <exclude-default-listeners/>
    <exclude-superclass-listeners/>
    <attributes>
      <id name = "categoryId">
        <column name = "CATEGORY_ID"/>
        <generated-value strategy = "SEQUENCE"
                        generator = "CATEGORY_SEQ_GEN"/>
      </id>
      <basic name = "categoryName">
        <column name = "CATEGORY_NAME"/>
      </basic>
      <basic name = "createDate">
        <column name = "CREATE_DATE"/>
      </basic>
      <many-to-many name = "items" target-entity = "Item">
        <cascade>
          <cascade-all/>
        </cascade>
        <join-table name = "CATEGORY_ITEMS">
          <join-column name = "CATEGORY_ID"
                       referenced-column-name = "CATEGORY_ID"/>
          <inverse-join-column name = "ITEM_ID"
                               referenced-column-name = "ITEM_ID"/>
        </join-table>
      </many-to-many>
    </attributes>
  </entity>
</entity-mappings>
```

The orm.xml file defines the actual O/R mapping with XML for the entities packaged in an EAR. Listing 11.7 ❶ shows how to define defaults for a persistence unit using the persistence-unit-defaults element. This element defines schema, catalog, and access, default entity listeners, and cascade type. We mentioned schema and catalog types in chapter 8 when we discussed the @Table and @SecondaryTable annotations. You can define the default values for the schema and catalog type in persistence-unit-defaults, and this can be overridden by each entity.

The access type may either be FIELD or PROPERTY.

In chapter 9 you learned that entity listeners can be defined to handle lifecycle callbacks for entities, and that a default listener for all entities in a persistence module can be defined by using the entity-listeners subelement in persistence-unit-defaults ❷. Use @ExcludeDefaultListener on the entity or a mapped superclass if you need to exclude the default entity listener. The name element ❸ identifies the name of the entity and is the equivalent of the name in @Entity. This value is used in the from clause in JPQL queries.

The other O/R mapping elements in orm.xml are somewhat self-explanatory, and we won't discuss them in detail.

Table 11.5 lists the one-to-one mapping between the most often used annotations and their associated deployment descriptors. You'll probably notice immediately that the XML element is usually quite similar to its annotation cousin.

Table 11.5 Mapping of persistence annotations to associated deployment descriptor elements

Annotations Grouped by Type	XML Element
Object type	
@Entity	entity
@MappedSuperClass	mapped-superclass
@Embedded	embedded
@Embeddable	embeddable
Table mapping	
@Table	table
@SecondaryTable	secondary-table

continued on next page

Table 11.5 Mapping of persistence annotations to associated deployment descriptor elements *(continued)*

Annotations Grouped by Type	XML Element
Query	
`@NamedQuery`	`named-query`
`@NamedNativeQuery`	`named-native-query`
`@SqlResultsetMapping`	`sql-result-set-mapping`
Primary key and column mapping	
`@Id`	`id`
`@IdClass`	`id-class`
`@EmbeddedId`	`embedded-id`
`@TableGenerator`	`table-generator`
`@SequenceGenerator`	`sequence-generator`
`@Column`	`column`
`@PrimaryKeyJoinColumn`	`primary-key-join-column`
`@GeneratedValue`	`generated-value`
Relationship mapping	
`@ManyToMany`	`many-to-many`
`@OneToOne`	`one-to-one`
`@OneToMany`	`one-to-many`
`@ManyToOne`	`many-to-one`
`@JoinTable`	`join-table`
`@JoinColumn`	`join-column`
`@InverseColumn`	`inverse-join-column`
Listeners	
`@ExcludeDefaultListeners`	`exclude-default-listeners`
`@ExcludeSuperClassListeners`	`exclude-superclass-listeners`
`@PreUpdate`	`pre-update`

continued on next page

Table 11.5 Mapping of persistence annotations to associated deployment descriptor elements *(continued)*

Annotations Grouped by Type	XML Element
Listeners *(continued)*	
`@PostUpdate`	`post-update`
`@PrePersist`	`pre-persist`
`@PostPersist`	`post-persist`
`@PreRemove`	`pre-remove`
`@PostRemove`	`post-remove`
`@PostLoad`	`post-load`

Manually performing O/R mapping using XML can be quite arduous, error-prone, and difficult to troubleshoot. You may want to investigate tools that will assist with this effort.

NOTE The goal of the Eclipse Dali project (http://wiki.eclipse.org/index.php/Dali_Project) is to provide developer support for O/R mapping of EJB 3 persistence objects, and help generate O/R mappings with annotations and XML descriptors.

Our trek through packaging EJB 3 is nearing the end. Before we finish, we want to highlight some things you should keep in mind when packaging your shiny new EJB 3 applications.

11.5 Best practices and common deployment issues

After reading this chapter it may appear that a lot of little pieces are required in order to deploy EJB 3 components. That may not be all that far from the truth. The reality, though, is that you don't have to keep track of all the pieces yourself; tools provided by the application servers help, and much of the glue code can be automated. You need to keep in mind some key principles, regardless of which components your application makes use of and which server you plan to deploy it to.

11.5.1 Packaging and deployment best practices

The following list of best practices can make your life easier while you're building and deploying your applications:

- *Understand your application and its dependencies.* Make sure that resources are configured before you deploy the application in your target environment. If an application requires a lot of resources, it is a good idea to use the deployment descriptor to communicate the dependencies for the deployer to resolve before attempting to deploy the application. Improper packaging of classes and libraries causes a lot of class-loading issues. You also need to understand the dependency of your applications on helper classes and third-party libraries and package them accordingly. Avoid duplication of libraries in multiple places. Instead, find a way to package your applications, and configure your application server such that you can share common libraries from multiple modules within the same application.
- *Avoid using proprietary APIs and annotations.* Don't use vendor-specific tags or annotations unless it's the only way to accomplish your task. Weigh doing so against the disadvantages, such as making your code less portable. If you are depending on proprietary behavior, check whether you can take advantage of a proprietary deployment descriptor.
- *Leverage your DBA.* Work with your DBA to automate creation of any database schemas for your application. Avoid depending on the automatic table creation feature for entities, as it may not meet your production deployment requirement. Make sure that the database is configured properly, and that it does not become a bottleneck for your application. Past experience indicates that making friends with the DBA assigned to your project really helps! If your application requires other resources such as a JMS provider or LDAP-compliant security provider, then work with the appropriate administrators to configure them correctly. Again, using O/R mapping with XML and resource dependencies with XML descriptors can help you troubleshoot configuration issues without having to fiddle with the code.

Now that you have some best practices in place, what do you do when that's still not enough? We'll let you in on a few secrets from the trenches that will make solving those packaging problems easier.

11.5.2 Troubleshooting common deployment problems

This section examines some common deployment problems that you may run into. Most can be addressed by properly assembling your application.

- `ClassNotFoundException` occurs when you're attempting to dynamically load a resource that cannot be found. The reason for this exception can be a missing library at the correct loader level; you know, the JAR file containing the class that can't be found. If you're loading a resource or property file in your application, make sure that you use `Thread.currentThread().getContextClassLoader().getResourceAsStream()`.

- `NoClassDefFoundException` is thrown when code tries to instantiate an object, or when dependencies of a previously loaded class cannot be resolved. Typically, you run into this issue when all dependent libraries are not at the same class loader level.
- `ClassCastException` normally is the result of duplication of classes at different levels. This occurs in the same-class, different-loader situation; that is, you try to cast a class loaded by class loader (L1) with another class instance loaded by class loader (L2).
- `NamingException` is typically thrown when a JNDI lookup fails, because the container tries to inject a resource for an EJB that does not exist. The stack trace for this exception gives the details about which lookup is failing. Make sure that your dependencies on `DataSource`s, EJBs, and other resources resolve properly.
- Your deployment may fail due to an invalid XML deployment descriptor. Make sure that your descriptors comply with the schema. You can do this by using an IDE to build your applications instead of manually editing XML descriptor files.

11.6 Summary

At the heart of Java EE applications lies the art of assembly and packaging enterprise applications. This chapter briefly introduced the concepts of class loading and code sources used by various application archives. We also explained how to package all of the EJB types, including persistence entities. You learned about the deployment descriptor of an EJB-JAR module, and how you can use descriptors to override settings specified in metadata annotations. You saw that `persistence.xml` is the only required deployment descriptor in Java EE 5. We also tackled persistence modules and the various configurations required for a persistence unit, as well as O/R mapping with XML.

Finally, we provided some best practices on packaging, and identified some common deployment issues that you may run into. In the next chapter we discuss how you can use EJBs across tiers.

JAVA

EJB 3 IN ACTION

Debu Panda, Reza Rahman, Derek Lane

EJB 2 is widely used but it comes at a cost—procedural, redundant code. EJB 3 is a different animal. By adopting a POJO programming model and Java 5 annotations, it dramatically simplifies enterprise development. A cool new feature, its Java Persistence API, creates a standard for object-relational mapping. You can use it for any Java application, whether inside or outside the EJB container. With EJB 3 you will create true object-oriented applications that are easy to write, maintain and extend.

EJB 3 in Action is a fast-paced tutorial for both novice and experienced Java developers. It will help you learn EJB 3 and the JPA quickly and easily. This comprehensive, entirely new EJB 3 book starts with a tour of the EJB 3 landscape. It then moves quickly into core topics like building business logic with session and message-driven beans. You'll find four full chapters on the JPA along with practical code samples, design patterns, performance tuning tips, and best practices for building and deploying scalable applications.

What's Inside

- Dependency Injection and Interceptors
- Domain modeling and persisting entities with the JPA and its query language
- Using EJB 3 across application tiers and web services
- Integrating with Spring
- Migrating from EJB 2, JDBC, and other O-R frameworks

Debu Panda, a member of the EJB 3.0 Expert Group, is a lead product manager of the Oracle Application Server development team focused on EJB. **Reza Rahman** is a JavaEE5 architect who works with EJB, Spring and Hibernate. **Derek Lane** is the CTO of a technology company and the founder of the Dallas and Oklahoma City Java User Groups.

For more information, code samples, and ebook visit www.manning.com/EJB3inAction

"... this technical book is surprisingly entertaining."
—King Y. Wang, Oracle Canada

"... well written, easy, and fun."
—Patrick Dennis
Management Dynamics Inc.

"... this is *the* [EJB] book to read. Don't miss its practical advice!"
—Jeanne Boyarsky
JavaRanch.com

"Great book—covers everything relating to EJB 3.0."
—Awais Bajwa
Expert Group Member
JSR 243 Java Data Objects

MANNING $44.99 US/$58.99 Canada

ISBN-10: 1-933988-34-7
ISBN-13: 978-1-933988-34-4
54499
9 781933 988344