SAMPLE CHAPTER

MANNING

EJB 3 IN ACTION

SECOND EDITION

Debu Panda
Reza Rahman
Ryan Cuprak
Michael Remijan

EJB 3 in Action

by Debu Panda
Reza Rahman
Ryan Cuprak
Michael Remijan

Chapter 13

brief contents

13 Packaging EJB 3 applications

This chapter covers

- The Java EE module system
- Class loading in EE applications
- Packaging EJBs and MDBx
- Packaging JPA
- Packaging CDI

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. But now that you have all this code, what do you do with it?

This chapter begins with a discussion of application packaging and deployment—the fundamentals needed to get your Enterprise application running. We'll explore how the various modules (JAR, EJB-JAR, WAR, CDI, and EAR) of an EE application fit together and interact when deployed to the EE server. We'll discuss how EE applications are configured both through annotations and XML. Finally, we'll cover best practices and some common deployment issues, especially when deploying the same application across different EE server implementations. Let's start by looking at the EE modules and how they're packaged together.

13.1 Packaging your applications

Your Enterprise Java application may contain hundreds of custom-developed Java classes. The code you develop will be of different types, such as EJBs, Servlets, and JSF managed beans, and persistence, helper, and utility classes. This code, in turn, will also be supported by dozens of external libraries, resulting in hundreds more classes. All of these classes are part of your application. In addition, applications will also typically contain non-Java code, such as JSPs, HTML, CSS, JavaScript, and images. At some point, everything needs to come together and be put on the EE server. This is known as *deployment*.

Recall from chapter 1 that the EJB container is part of the EE server and it's responsible for managing EJBs and MDBs. An EJB-JAR module must be created for deployment to the EE server. To understand how to package EJB-JAR modules, you must consider how it fits into the bigger picture of Java EE packaging and understand what constitutes a complete Enterprise Java application.

Up to this point, we've focused on using EJB components like session beans and MDBs to build business logic and JPA entities to implement database persistence code. But your application won't be complete without a presentation tier that accesses the business logic you built with EJBs. For example, the EJBs you built for ActionBazaar don't make sense unless you have a client application accessing them. Most likely, you've used standard technologies such as JSF to build the web tier of your applications. These web applications, together with EJBs, constitute an Enterprise application you can deploy to an application server.

To deploy and run an application, you have to package the EJB-JAR and WAR modules together into an EAR and deploy the EAR to an application server. Java EE defines a standard way of packaging these modules using the JAR file format. One of the advantages of having this format defined as part of the specification is that modules are portable across application servers.

Table 13.1 lists the modules supported by an EE server. Each module usually groups similar pieces of the application together. For instance, you may have multiple EJB-JAR modules, each responsible for different parts of your business logic. Similarly, you may have multiple WAR modules providing unique user interfaces into the business. The EAR is intended to be the über module containing all the other modules, so in the end you're deploying only one file. The application server will scan the contents of the EAR and deploy it. We'll discuss how an EAR is loaded by the server in section 13.1.2.

Table 13.1 Modules supported by an EE server

Description	Descriptor	Contents
EJB Java Archive (EJB-JAR)	`META-INF/ejb-jar.xml` or `WEB-INF/ejb-jar.xml`	Session and message-driven beans. Optionally JPA and CDI may be included as well.

Table 13.1 Modules supported by an EE server *(continued)*

Description	Descriptor	Contents
Web Application Archive (WAR)	`WEB-INF/web.xml`	Web application artifacts such as Servlets, JSPs, JSF, static images, and so on. Optionally EJBs, JPA, and CDI may be included as well.
Enterprise Application Archive (EAR)	`META-INF/application.xml`	Other Java EE modules such as EJB-JARs and WARs.
Resource Adapter Archive (RAR)	`META-INF/ra.xml`	Resource adapters.
Client Application Archives (CAR)	`META-INF/application-client.xml`	Standalone Java client for EJBs.
Java Persistence Archive (JPA)	`META-INF/persistence.xml` or `WEB-INF/persistence.xml`	Java EE standard ORM between applications and databases. May be included as part of the following archives: EJB-JAR, WAR, EAR, and CAR.
Context Dependency and Injection Bean Archive (CDI)	`META-INF/bean.xml` or `WEB-INF/bean.xml`	Java EE standard dependency injection. May be included as part of the following archives: EJB-JAR, WAR, EAR, and CAR.

Enterprise Java applications need to be assembled into specific types of modules. Then a master EAR module is assembled that can be deployed to an application server. These are the available module types as specified by Java EE.

All of these files are in the basic JDK-JAR file format. You can use the `jar` utility that comes with the JDK to build them. In reality, either your IDE or your build tool (Maven or ANT) will do the monotonous work of building the module for you. But for each module, you must still supply the code and the deployment descriptor to configure it. Once all the modules are assembled, the final step is to assemble the master module (that is, the EAR) for deployment.

Deployment descriptors versus annotations

For many cases, deployment descriptors in EE archives are optional. Convention-over-configuration as well as in-code annotation has taken over the heavy lifting that deployment descriptors used to do when it came to configuring EE modules. But there are still cases where deployment descriptors are not only useful but are required. For example, when using CDI, your archive must contain a META-INF/beans.xml or WEB-INF/beans.xml, even if the file is empty, to indicate that your application uses CDI. So this is a case where a deployment descriptor is required. Another example is wiring up remote EJBs in different environments to different servers. Sure, you can hard-code remote server information in an annotation, but it's easier to have different deployment descriptors for your archives for different environments.

In this chapter, we focus primarily on the EJB-JAR module and the EAR modules as well as JPA and CDI. JPA entities are special. They don't have a specific module type of their own. Instead, entities can be packaged as part of most module types. For example, the ability to package entities in WARs allows you to use the EJB 3 JPA in web applications. Entities can also be packaged inside EJB-JAR modules allowing business logic to retrieve and alter business data. Standalone EJB Java clients can also contain entities, though use of entities in EJB Java clients is usually limited to the standalone client's runtime and configuration data, not core business data—that's what the EJBs are for. Entities aren't supported in RARs.

If entities are special, why doesn't Java EE have a different module type to package entities? After all, 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've 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 Persistence Archive (PAR) was introduced, which mysteriously vanished in the proposed final draft. The EJB and Java EE expert groups fought a huge, emotional battle over whether to introduce a module type for a persistence module at the Java EE level, and they sought suggestions 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 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?

13.1.1 Dissecting the Java EE module system

The Java EE module system is based on the EAR file, which is the top-level module containing all other Java EE modules for deployment. So to understand how deployment works, let's take a closer look at the EAR file. We'll start with an example from ActionBazaar.

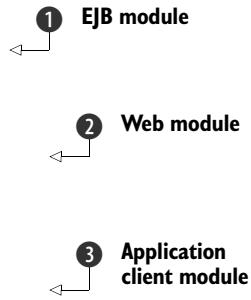
The ActionBazaar application contains an EJB-JAR module, a web module, a JAR containing helper classes, and an application client module. The file structure of the EAR module for ActionBazaar looks like this:

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

Here 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 the following listing.

Listing 13.1 ActionBazaar EAR module deployment descriptor

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



If you review the EAR module deployment descriptor in this listing, you'll see that it explicitly identifies each of the artifacts as a specific type of module. The EJB module ❶ has the EJBs with your application's business logic. The web module ❷ is the web-based version your application. The application client module ❸ is another version of your application, such as a thick-client GUI. 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 5.0 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. For more information on these conventions, see http://www.oracle.com/technetwork/java/namingconventions-139351.html. Next, we'll take a look at how application servers deploy an EAR module.

13.1.2 Loading a Java EE 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 as to exactly how to implement it. One area where server implementations stand out is in how fast they can deploy the archive.

Although 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 13.1 when attempting to load the EAR file that contains modules or archives from table 13.1.

When deploying an EAR containing multiple EJB-JARs, WARs, RARs, and other modules, the EAR module may easily contain thousands of individual classes. All these classes need to be resolved, loaded, and managed by the EE server, which isn't an easy

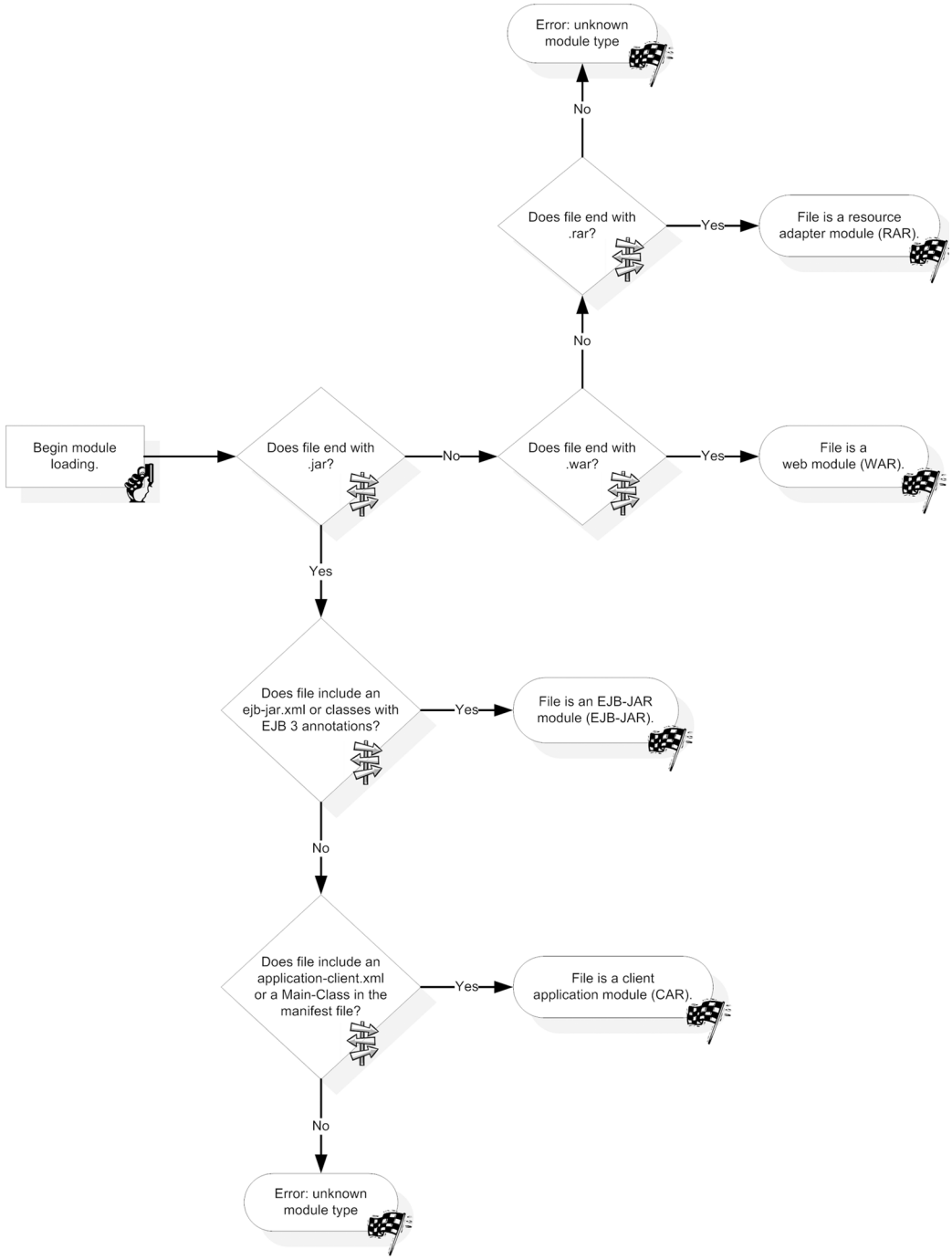


Figure 13.1 Rules followed by application servers to deploy an EAR module. Java EE doesn't require a deployment descriptor in the EAR module to identify the types of modules packaged. It's the responsibility of the Java EE server to determine the type of module based on its naming conventions (extension) and its content. It does so by following this algorithm.

task when different WARs may contain different versions of the same classes. We'll discuss next how an EE server handles this "JAR hell."

13.2 Exploring class loading

To explore EE server class loading, we'll first briefly review how Java class loaders work, and then we'll give a concise explanation of typical EE server class-loading strategies. Finally, we'll finish up by reviewing the EE specifications for dependencies between common EE modules.

13.2.1 Class-loading basics

Class loading in Java works based on a hierarchical structure of class loaders, with each class loader responsible for loading certain classes and each loader building on top of the previous one. This basic structure is depicted in figure 13.2.

Class loaders in Java follow a parent-first model when attempting to resolve a class. This means no matter what class loader you're in, the class loader will first ask its parent if it can load the class. If the parent can't load the class, the parent will ask its parent and so on until the bootstrap class loader is reached. If the bootstrap class loader can't

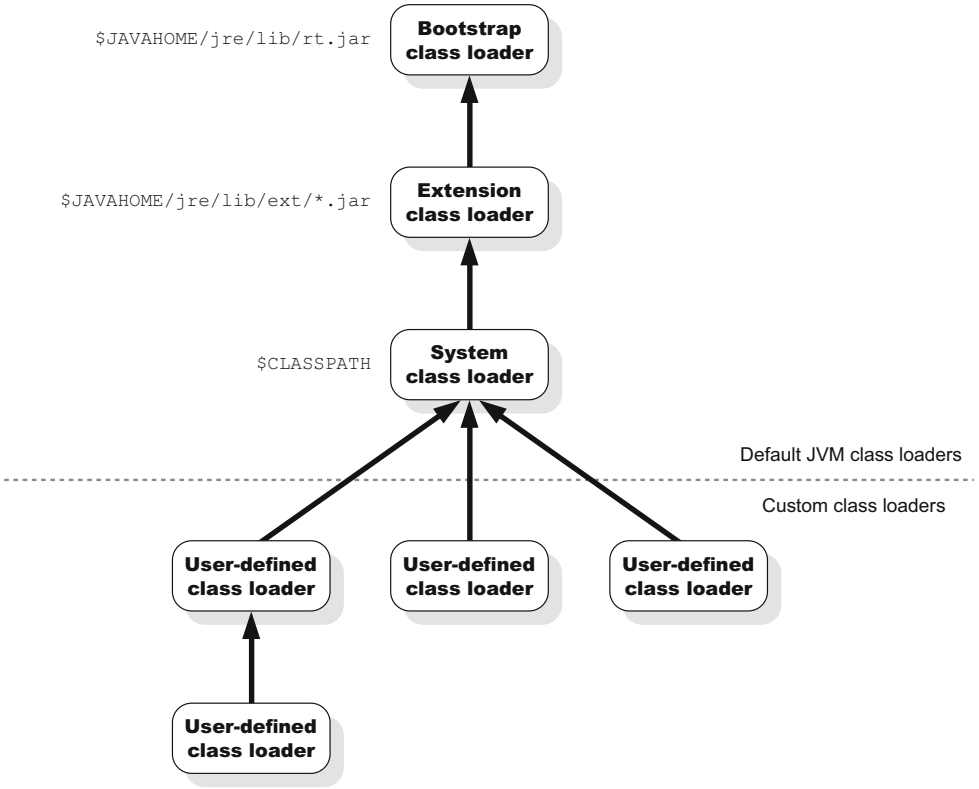


Figure 13.2 Basic class-loader structure for Java applications

load the class, you get a `java.lang.ClassNotFoundException`. Once a class is found and loaded, the class loader stores it in a local cache for quick reference. Now that you've had a quick refresher on basic class loading, let's look at how class loading gets a bit more complicated in Java EE applications.

13.2.2 Class loading in Java EE applications

An EAR module consisting of multiple EJB-JARs, WARs, and RARs, as well as supporting third-party libraries, will contain thousands of classes. It's up to the Java EE server to provide a class-loading strategy that ensures all the applications can resolve the classes they need.

Despite the importance of class loading, the Java EE specification doesn't provide implementation standards for EE server class loaders. The implementation is largely up to the EE server provider. What the Java EE specification does provide is the standards for the visibility and sharing of classes between different modules within an EAR. We'll look at these standards in the next section, but first, take a look at figure 13.3 to see the typical strategy most EE servers use to implement class loading.

As illustrated in figure 13.3, the application server class loader loads all of the JARs in the application server's /lib directory. These are all the libraries the application server is required to provide—libraries such as the Java EE API itself.

The EAR module class loader is extended from the application server class loader. This class loader loads the classes that are deployed at an EAR level. By default, they're the classes packaged inside JARs in the /lib directory of the EAR, but the default /lib directory can be overridden by the `library-directory` element in the application.xml deployment descriptor.

The EJB class loader is extended from the EAR module. Even though an EAR may contain multiple EJB-JAR modules, there's typically only a single EJB class loader that loads all the EJB-JAR modules. Finally, the WAR class loader is extended from the EJB class loader. By extending the EJB class loader, the WAR gains access to all EJBs deployed as part of the EAR. Each WAR will get its own class loader.

This gives you an idea of how EE servers typically implement class loading for their Enterprise applications. Why EE servers use this implementation is driven by the EE standards for the visibility and sharing of classes between different modules within an EAR. We'll look at these standards next.

13.2.3 Dependencies between Java EE modules

The Java EE specification defines the requirements for the visibility of classes for each EE module. For example, the specification defines what classes are visible to a WAR module but doesn't define how the EE server class loaders provide that visibility. We have an explanation in section 13.2.2 of how EE servers typically implement class loaders.

Suppose listing 13.2 describes what is deployed to your EE server. We'll use the example EE server deployment of this listing to explore the most common requirements for the visibility of classes for the EJB-JAR and WAR modules. For more visibility

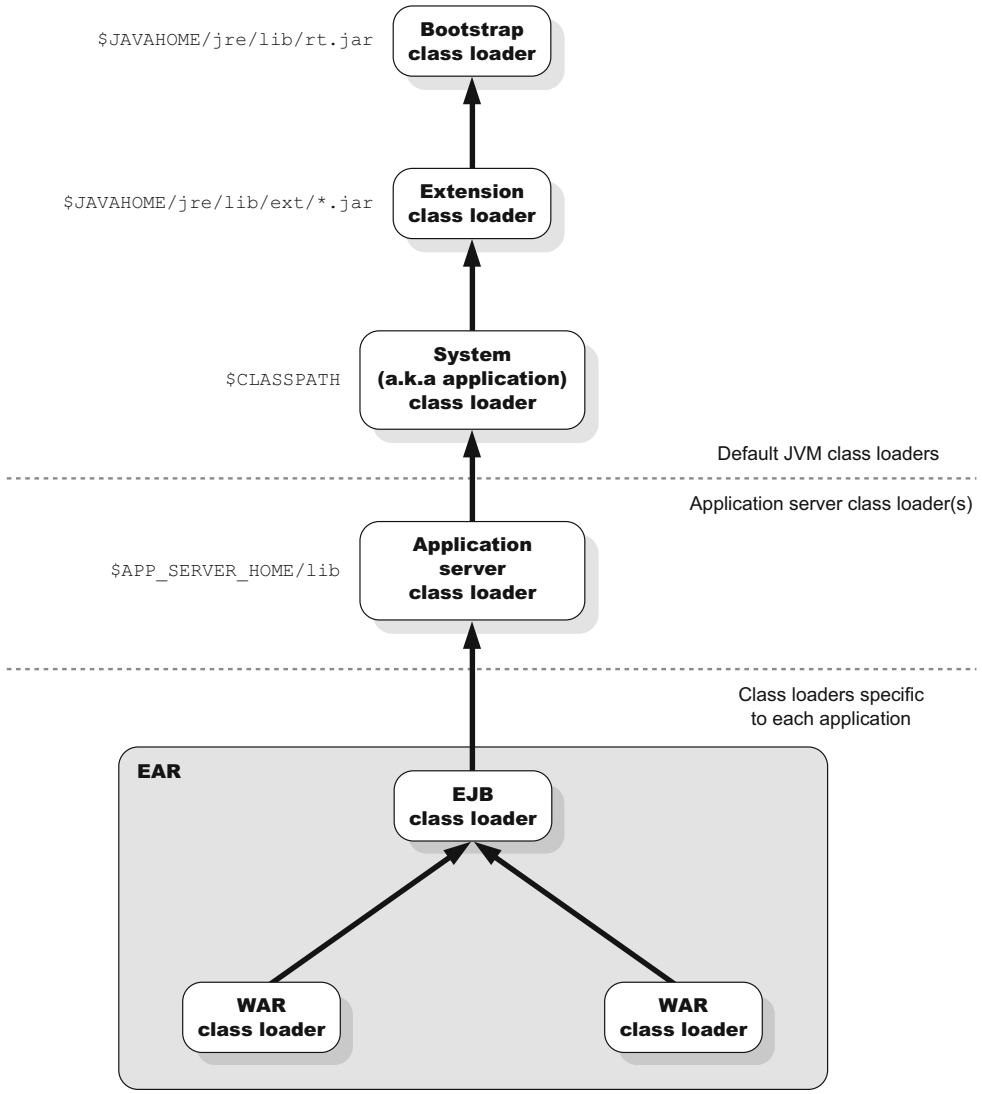


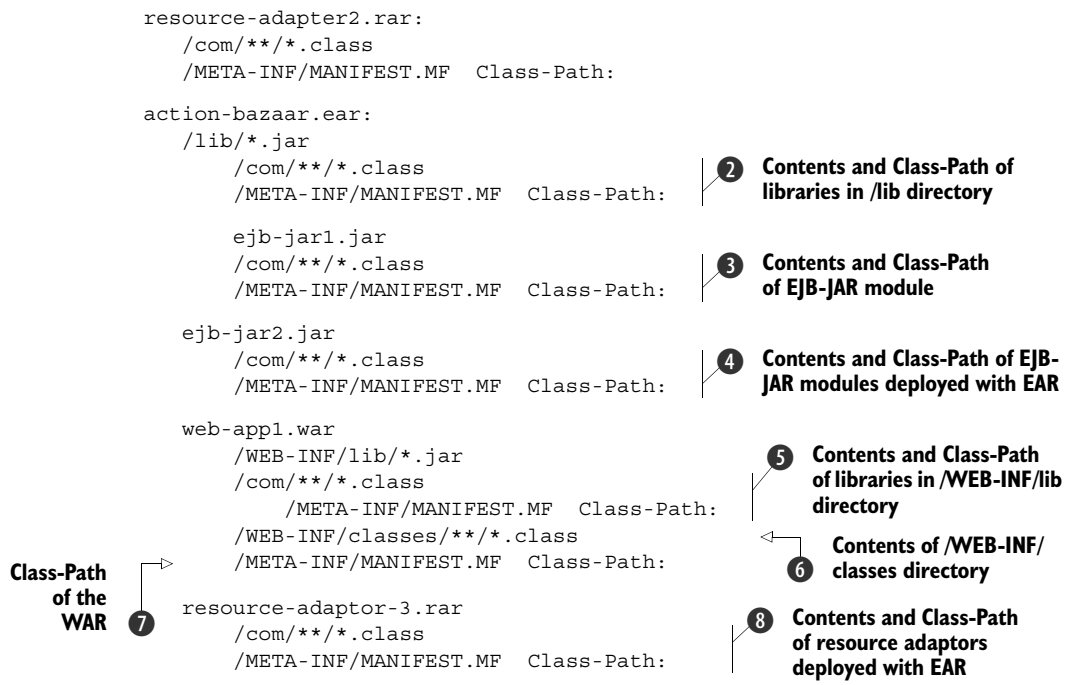
Figure 13.3 Typical class-loader structure for Java EE applications

requirements, or visibility requirements of other modules, refer to the Java EE platform specification.

Listing 13.2 Example EE server deployment

```
EE Server:
    resource-adapter1.rar:
        /com/**/*.class
        /META-INF/MANIFEST.MF  Class-Path:
```

❶ **Contents and Class-Path of external resource adaptors**



EJB-JAR

Given the example EE server deployment in listing 13.2, an EJB-JAR module must have access to the following:

- The contents and `Class-Path` of any external resource adaptor ❶
- The contents and `Class-Path` of each library in the /lib directory of the EAR ❷
- The contents and `Class-Path` of the EJB-JAR module itself ❸
- The contents and `Class-Path` of additional EJB-JAR modules deployed with the EAR ❹
- The contents and `Class-Path` of any resource adaptor deployed with the EAR ❽

WAR

Given the example EE server deployment in listing 13.2, a WAR module must have access to the following:

- The contents and `Class-Path` of any external resource adaptor ❶
- The contents and `Class-Path` of each library in the /lib directory of the EAR ❷
- The contents and `Class-Path` of each library in the /WEB-INF/lib directory of the WAR ❺
- The contents of the /WEB-INF/classes directory of the WAR ❻
- The `Class-Path` of the WAR ❼

- The contents and `Class-Path` of all EJB-JAR modules deployed with the EAR ❸, ❹
- The contents and `Class-Path` of any resource adaptor deployed with the EAR ❽

As you can see, the EE server class loader has a big job to do. With so many classes at so many different levels, class version conflicts become inevitable. Understanding the visibility requirements and remembering the parent-first delegation model that class loaders use to find classes will help you package your applications effectively. Next, let's take a look at how to package session and message-driven beans.

13.3 Packaging session and message-driven beans

Before creating an EAR to deploy your application to the EE server, you must create the modules the EAR will contain. Now that you understand class visibility between the different modules of your application, you're ready to look at how to package these modules and get them ready for deployment. Specifically, we're going to look at how to package session and message-driven beans. The Java EE specification allows session and message-driven beans to be packaged in an EJB-JAR module or the WAR module. When packaged in the EJB-JAR module, the beans will run in a full EJB container. When packaged inside a WAR module, the beans will run in an EJB Lite container, which has most but not all of the features of a full EJB container. We'll now look at packaging EJB-JAR and WAR modules, comment on the pros and cons of deployment descriptors versus annotations, and then finish by looking at configuring default interceptors.

13.3.1 Packaging EJB-JAR

An EJB-JAR module is really nothing more than a Java JAR archive. With Java EE's emphasis on convention-over-configuration and its use of annotations, the EJB-JAR module doesn't even need to include the META-INF/ejb-jar.xml deployment descriptor. When the EE server deploys an EAR, it'll automatically scan through the JAR and look for either EJB 3 annotations or the META-INF/ejb.jar.xml file to determine if the JAR is an EJB-JAR module (refer back to section 13.1.2 for more information on this scanning process). So to create an EJB-JAR module, create a normal Java JAR archive with the following structure:

```
ActionBazaar.jar:
  com/actionbazaar/ejb/BidServiceEjb.class
  com/actionbazaar/mdb/BidServiceMdb.class
  META-INF/ejb.jar.xml
```

Session bean

Message-driven bean

Deployment descriptor (optional)

How you create this JAR file is really a question of what technology you're using. For Java development, there are a lot of options when it comes to technology, so there's something out there to fit your needs. Let's explore options using NetBeans and Maven.

NETBEANS

Using NetBeans, create a new EJB module and add the source code for your session and message-driven beans. Figure 13.4 shows what the Projects tab view of the module may look like.

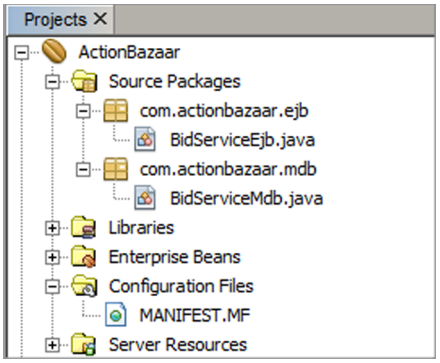


Figure 13.4 NetBeans Projects view of an EJB module

Once you've coded your beans, right-click on the project and choose Build. NetBeans will automatically compile the source code and generate the EJB-JAR module. Find the EJB-JAR module in the project's /dist directory. Looking inside the JAR file (figure 13.5), you'll see the classes and configuration files in the right places.

```
c:\>jar -tf ActionBazaar.jar
META-INF/
META-INF/MANIFEST.MF
com/
com/actionbazaar/
com/actionbazaar/ejb/
com/actionbazaar/mdb/
com/actionbazaar/ejb/BidServiceEjb.class
com/actionbazaar/mdb/BidServiceMdb.class
```

Figure 13.5 Contents of EJB-JAR module

You may notice that there's no META-INF/ejb-jar.xml. This is because the project doesn't have one yet. To add one, right-click the project, choose New, and then choose Standard Deployment Descriptor. NetBeans will then add an empty ejb-jar.xml file to the project. When you rebuild the project, the META-INF/ejb-jar.xml file will be automatically added to the EJB-JAR module.

MAVEN

When using Maven to create an EJB-JAR module, use `<packaging>ejb</packaging>` as the POM packaging type. To customize the configuration of the EJB-JAR module, use the `maven-ejb-plugin` to set options to change how Maven creates the JAR. The following listing shows a minimal POM file that will create an EJB-JAR module.

Listing 13.3 POM file to build an EJB-JAR

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.actionbazaar</groupId>
  <artifactId>actionbazzar-ejb</artifactId>
  <version>1.0.0</version>
  <packaging>ejb</packaging>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-ejb-plugin</artifactId>
        <version>2.3</version>
```

Packaging type as "ejb" will create an EJB-JAR module.

Use maven-ejb-plugin to configure creation of EJB-JAR module.

```
        <configuration>
          <ejbVersion>3.1</ejbVersion>
          <archive>
            <addMavenDescriptor>false</addMavenDescriptor>
            <manifest>
              <addClasspath>true</addClasspath>
            </manifest>
          </archive>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
      <groupId>javax</groupId>
      <artifactId>javaee-api</artifactId>
      <version>6.0</version>
      <type>jar</type>
      <scope>provided</scope>
    </dependency>
  </dependencies>
</project>
```

This option will add Class-Path: to META-INF/MANIFEST.MF.

Maven will create the EJB-JAR module in the /target directory. This will create an EJB-JAR module with no META-INF/ejb-jar.xml. To include the deployment descriptor, you can create /src/main/resources/META-INF/ejb-jar.xml. By putting the deployment descriptor in the standard Maven resources directory, Maven will automatically include it when the EJB-JAR module is built.

These are two options for building an EJB-JAR module. Now we'll look at packaging session and message-driven beans inside a WAR module.

13.3.2 Packaging EJB in WAR

The Java EE 6 specification allows for session and message-driven beans to be included inside a WAR module instead of having to deploy them separately in an EJB-JAR module. When included in a WAR module, the beans run in the EJB Lite container. This means that not all functionality of a full EJB container is available, but simple applications or prototypes will find this sufficient.

A WAR module is a Java JAR archive with contents conforming to the requirements of a WAR module specified by the Java EE specification. Typically, a WAR module will contain classes and other resources in a /classes subdirectory, third-party dependencies in a /lib subdirectory, and the configuration in /WEB-INF/web.xml. A WEB module's structure may look like this:

```
ActionBazaar.war:
  classes/
    com/actionbazaar/web/ejb/BidServiceEjb.class
    com/actionbazaar/web/mdb/BidServiceMdb.class
  WEB-INF/web.xml
```

Session bean inside a WAR

Message-driven bean inside a WAR

WAR deployment descriptor (optional)

Similar to creating EJB-JAR modules, how you actually create a WAR module is a question of what technology your development team is using. Next we'll look at examples using NetBeans and Maven.

NETBEANS

Using NetBeans, create a new Web Application module and add the source code for your session and message-driven beans. Figure 13.6 shows what the Projects view of the module may look like.

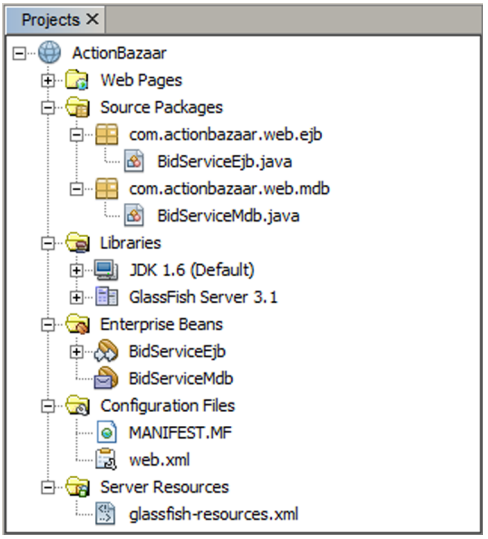


Figure 13.6 NetBeans Projects view of a Web Application module

This ActionBazaar WEB module contains a session bean and a message-driven bean. To build the project, right-click it and choose Build. NetBeans will create the WAR module for you and put it in the /dist subdirectory. You can see the structure of the WAR module in figure 13.7.

Similar to creating an EJB-JAR module with NetBeans, you'll notice there's no ejb-jar.xml. This is because the project doesn't have one yet. To add one, right-click the project, choose New, and then choose Standard Deployment Descriptor. NetBeans will then add an empty ejb-jar.xml file to the project. Unlike the EJB-JAR module, for a WEB module, NetBeans will create WEB-INF/ejb-jar.xml. When packaging session and message-driven beans in a WAR module, the ejb-jar.xml deployment descriptor goes in the WEB-INF subdirectory, not META-INF.

MAVEN

When using Maven to create a WAR module, use `<packaging>war</packaging>` as the POM packaging type. Use the `maven-war-plugin` to customize how Maven creates the WAR. The next listing shows a minimal POM file that will create a WAR module.

```
c:\>jar -tf ActionBazaar.war
META-INF/
META-INF/MANIFEST.MF
WEB-INF/
WEB-INF/classes/
WEB-INF/classes/com/
WEB-INF/classes/com/actionbazaar/
WEB-INF/classes/com/actionbazaar/web/
WEB-INF/classes/com/actionbazaar/web/ejb/
WEB-INF/classes/com/actionbazaar/web/mdb/
WEB-INF/classes/com/actionbazaar/web/ejb/BidServiceEjb.class
WEB-INF/classes/com/actionbazaar/web/mdb/BidServiceMdb.class
WEB-INF/web.xml
index.xhtml

c:\>
```

Figure 13.7 Contents of WAR module

Listing 13.4 POM file to build a WAR

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.actionbazaar</groupId>
  <artifactId>actionbazzar-web</artifactId>
  <version>1.0.0</version>
  <packaging>web</packaging>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-war-plugin</artifactId>
        <version>2.1</version>
        <configuration>
          <warName>ActionBazaar</warName>
          <archive>
            <addMavenDescriptor>false</addMavenDescriptor>
          </archive>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
      <groupId>javax</groupId>
      <artifactId>javaee-api</artifactId>
      <version>6.0</version>
      <type>jar</type>
      <scope>provided</scope>
    </dependency>
  </dependencies>
</project>
```

Packaging type as "war" will create a WAR module.

Use maven-war-plugin to configure creation of WAR module.

This option will set the name of the WAR module to ActionBazaar.war.

Maven will create the WAR module in the /target directory. The WAR module will contain a WEB-INF/web.xml but no ejb-jar.xml. To include ejb-jar.xml, you can create /src/main/webapp/WEB-INF/ejb-jar.xml. By putting it here, Maven will automatically include it when the EJB-JAR module is built.

Using remote EJBs in WAR modules

Although including EJBs directly inside a WAR module is a great convenience started with the Java EE 6 spec, it's not the only way to include EJBs inside a WAR module. It's very typical (particularly in high-use and high-availability applications) to need much more business rules processing power than web front-end display power. Therefore, the architecture of the application is divided into a small cluster of web front-end servers running the WAR module and a much larger cluster of back-end servers running the EJB modules. With two separate clusters, the WAR modules on the front end talk to the EJB modules on the back end remotely through `@Remote` EJBs.

(continued)

If your application is architected like this, the Maven configuration of the maven-ejb-plugin can include the `<generateClient>true</generateClient>` option. With this option, Maven will package up all interfaces for the EJBs into its own JAR file. Your WAR modules can then include this dependency, which has only the interfaces instead of the entire EJB-JAR.

These are two options for building a WAR module. Now we'll look at whether deployment descriptors are even needed.

13.3.3 XML versus 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's 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's common in large companies for different people to be involved for each environment (development, test, production, and so on). For instance, your application requires such resources as DataSource or JMS objects, and the JNDI names for these resources change between these environments. It doesn't make sense to hardcode 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 (see section 13.3.5). 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`.

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. The following listing shows a simple example of a deployment descriptor for the BazaarAdmin EJB.

Nonstandard deployment descriptors

In addition to the standard deployment descriptor (ejb-jar.xml), most application servers also have extensions that provide application server–specific configuration options. For example, GlassFish has the glassfish-ejb-jar.xml file and WebLogic has weblogic-ejb-jar.xml. It's important to remember that these aren't part of the Java EE specification standards. They'll work only on their respective application servers. These application-specific configuration files usually give you direct access to features of the server that are nonstandard and extend the default behavior of an EE server in some way. Although these extensions are sometimes useful, they make your application less portable between servers. In some cases, portability may become impossible.

Listing 13.5 A simple `ejb-jar.xml`

```
<ejb-jar version="3.2">
  <enterprise-beans>
    <session>
      <ejb-name>BazaarAdmin</ejb-name>
      <remote>actionbazaar.buslogic.BazaarAdmin</remote>
      <ejb-class>actionbazaar.buslogic.BazaarAdminBean</ejb-class>
      <session-type>stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
...
  <assembly-descriptor>
    <container-transaction>
      <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>
    </security-role>
  </assembly-descriptor>
</ejb-jar>
```

1. EJB specification version, in this case 3.2
2. Identifier for EJB
3. Type of bean
4. Transaction type for bean
5. Contains settings for container-based transactions
6. Specifies which roles are able to access BazaarAdmin bean

If you're using deployment descriptors for your EJBs, make sure that you set the `ejb-jar` version to 3.2 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're 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 ❻.

Table 13.2 lists commonly used annotations and their corresponding descriptor tags. Note that as we mentioned earlier there's an element for every annotation. You'll need only those that 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 13.2 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`
		`name`	`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`
`@Transaction-Attribute`	Transaction settings method		`container-transaction trans-attribute`
`@Interceptors`	Interceptors		`interceptor-binding interceptor-class`
`@ExcludeClass-Interceptors`	Interceptors		`exclude-classinterceptor`
`@ExcludeDefault-Interceptors`	Interceptors		`exclude-defaultinterceptors`
`@AroundInvoke`	Custom interceptor		`around-invoke`
`@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`

Table 13.2 One-to-one mapping between annotations and XML descriptor elements *(continued)*

Annotation	Type	Annotation element	Corresponding descriptor element
`@RunAs`	Security setting		`security-identity` `run-as`

`@Resource`	Resource references (`DataSource`, JMS, environment, mail, and so on)		`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`
`@PersistenceUnit`	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.

13.3.4 Overriding annotations with XML

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 don't 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're overriding an annotation setting with your deployment descriptor, the `ejb-name` element must match the bean class name.

Suppose you 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 don't specify the `name` element, the container will use the name `BazaarAdminBean` as the name of the bean class, and to override annotations you'll have to use that name in the deployment descriptor:

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

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

```
<assembly-descriptor>
  <container-transaction>
    <method>
      <ejb-name>BazaarAdmin</ejb-name>
      <method-name>getUserWithItems</method-name>
      <method-params></method-params>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
</assembly-descriptor>
```

❶ Identifier for EJB

❷ Specify transaction level

In this example, the `assembly-descriptor` element is used 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.

13.3.5 Specifying default interceptors

Interceptors (as you'll recall from chapter 5) allow you to implement crosscutting 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 ejb-jar.xml deployment descriptor. The following listing shows how to specify default interceptors for an EJB module.

Listing 13.6 Default interceptor setting in ejb-jar.xml

```
<interceptor-binding>
  <ejb-name>*</ejb-name>
  <interceptor-class>
    actionbazaar.buslogic.CheckPermissionInterceptor
  </interceptor-class>
  <interceptor-class>
    actionbazaar.buslogic.ActionBazaarDefaultInterceptor
  </interceptor-class>
</interceptor-binding>
```

❶ Defines interceptor binding

❷ Applies binding to all EJBs

The `interceptor-binding` ❶ tag defines the binding of interceptors to a particular EJB with the `ejb-name` element. If you want to define the default interceptor or an

interceptor binding for all EJBs in the EJB module, then you can specify * as the value for ejb-name ❷. You specify a class to use as the interceptor with the <interceptor-class> tag. As is evident from the listing, you can specify multiple interceptors in the same binding, and the order in which they're specified in the deployment descriptor determines the order of execution for the interceptor. In the 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.

13.4 JPA packaging

The Java Persistence Architecture (JPA) was part of the EJB 3 specification, but as of EJB 3.2, it has been spun off into its own specification (http://jcp.org/en/jsr/detail?id=317). The main reason for this is JPA isn't limited to Java EE. It can be used in a Java Standard application as well. Because JPA is now its own specification, we'll highlight the most important parts as they relate to EJBs. Refer back to chapter 9 for more details about JPA.

13.4.1 Persistence module

Because the specification allows it to be used in any Java application, when it comes to Java Enterprise application development, JPA entities are simply part of the EJB-JAR or WAR modules. JPA entities may also be packaged as a regular JAR archive and deployed in the root of the EAR module. The key is the META-INF/persistence.xml file that designates the JAR archive, EJB-JAR module, or WAR module as containing one or more persistence units. We'll take a quick look at how to properly package JPA classes and the persistence.xml file.

JAR

The following listing shows how to package JPA inside a plain JAR archive.

Listing 13.7 Structure of a JAR containing JPA entities

```
ActionBazaar.jar:
  META-INF/
    persistence.xml
  actionbazaar/
    persistence/
      Category.class
      Item.class
```

❶ JPA configuration file

❷ JPA entity classes in persistence package

The configuration file exists as META-INF/persistence.xml ❶ and the entity classes are in their package subdirectory ❷.

EJB-JAR

If you want to include JPA entities in your EJB-JAR module, packaging is identical to how it's done for a plain JAR archive. The following listing shows how to package JPA inside an EJB-JAR module.

Listing 13.8 Structure of an EJB-JAR containing JPA entities

```
ActionBazaar-ejb.jar:
  META-INF/
    persistence.xml
  actionbazaar/
    buslogic/
      BazaarAdminBean.class
    persistence/
      Category.class
      Item.class
      BazaarAdmin.class
```

❶ JPA configuration file

❷ JPA entity classes in persistence package

The configuration file exists as META-INF/persistence.xml ❶ and the entity classes are in their package subdirectory ❷.

WAR

Packaging JPA entities in a WAR module can be a bit tricky. There are two ways to do it. The first and easiest way is to put the JAR containing your JPA entities into the WAR module WEB-INF/lib directory. The next listing shows how to package JPA inside an EAR module (assuming the JAR file from listing 13.7 is used).

Listing 13.9 Structure of a WAR containing JPA entities in JARs

```
ActionBazaar-web.war:
  WEB-INF/
    classes/
      ...
    lib/
      ActionBazaar.jar
    web.xml
  ...
```

But if the JPA entities aren't in a separate JAR but are instead directly part of the WAR module itself, the following listing shows how this should be done.

Listing 13.10 Structure of a WAR containing JPA entities

```
ActionBazaar-web.war:
  WEB-INF/
    classes/
      META-INF/
        persistence.xml
      persistence/
        Category.class
        Item.class
        BazaarAdmin.class
    lib/
      ...
    web.xml
```

❶ JPA configuration file

❷ JPA entity classes in persistence package

The persistence.xml file must go under /classes/META-INF/ ❶. The JPA entities are copied as *.class files in the subdirectory of their package ❷ as they usually are for a

WAR. Now that you know the structure of a persistence module, we'll teach you a little more about persistence.xml.

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's visible only to that specific module. But if you define the unit by placing a JAR file in the root or 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
```

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

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

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

But 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;
```

13.4.2 Describing the persistence module with persistence.xml

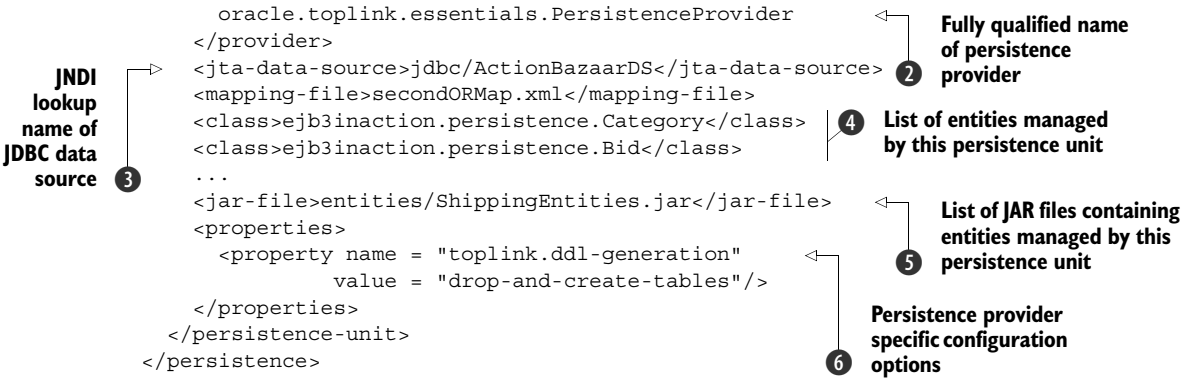
In chapter 9 we showed you how to group entities as a persistence unit and how to configure that unit using persistence.xml. Now that you know how to package entities, it's time to learn more about persistence.xml, the descriptor that transforms any JAR module into a persistence module. It's worth mentioning that persistence.xml is the only mandatory deployment descriptor that you have to deal with.

The following listing is an example of a simple persistence.xml that can be used with the ActionBazaar application. It'll successfully deploy to any Java EE container that supports JPA.

Listing 13.11 An example persistence.xml

```
<persistence>
  <persistence-unit name = "actionBazaar" transaction-type = "JTA"> <1>
    <provider>
```

<1> Defines a JPA persistence unit



Let's run through a quick review of the code. You define a persistence unit by using the `<persistence-unit>` element ❶. You can specify an optional factory class for the persistence provider ❷. If a persistence provider isn't specified, the default provider for the Enterprise server will be used. You specify the data source for the persistence provider ❸ so JPA knows how to connect to the database. If you have multiple persistence units in a single archive, you may want to identify the entity classes that compose the persistence unit ❹. If you have entities in another JAR file that you want to include in this persistence unit, use `<jar-file>` ❺ with a path to the JAR that's relative to this JAR file. Optionally, you can specify vendor-specific configuration using the `properties` element ❻.

JPA O/R mapping

Typically, JPA uses annotations for all of its O/R mappings. Sometimes annotations aren't a sufficient solution. An example is if the details of your object model change in different environments. In this case, JPA has orm.xml for specifying O/R mappings outside of code. When packaging your EE application, orm.xml goes side by side with persistence.xml. It can also be packaged with a location and name specified by `<mapping-file>` in persistence.xml. As with all deployment descriptors, if it exits, its configuration takes precedence over any annotations.

Because JPA is now its own specification, we won't go into any more detail here. We've given the requirements of packaging JPA entities in an EE application and described the basics of a persistence.xml file. Refer to the JPA specification or another Manning book for more information on JPA.

13.5 CDI packaging

Similar to JPA, CDI doesn't define any special deployment module because dependency injection isn't limited to a Java EE environment. CDI 1.1 for Java EE 7 is defined in JSR 346 (http://jcp.org/en/jsr/detail?id=346), which extends JSR 299 and JSR 330. CDI 1.1 adds specific requirements for dependency injection in a Java EE environment. But dependency injection may also be used in a Java Standard application.

Therefore, like JPA, CDI doesn't have its own EE module but is simply included as a part of other modules. In this section we'll quickly review how CDI is packaged as part of other EE modules, specifically JAR archives, EJB-JAR modules, and WAR modules. Refer to chapter 12 for more details about CDI.

13.5.1 CDI modules

When an EE application is deployed, CDI goes through bean discovery. Bean discovery is a process of determining which artifacts inside the EE application use CDI. These artifacts contain beans that the EE server must manage. An artifact may be a regular JAR archive or any of the Enterprise module types (see table 13.1). Bean discovery is simply looking for the beans.xml file in the following locations:

- META-INF/beans.xml in any JAR, EJB-JAR, application client JAR, or RAR in the EAR or in any JAR archive or directory referred to by any of them by the `Class-Path` of their META-INF/MANIFEST.MF
- WEB-INF/beans.xml of a WAR
- Any directory on the JVM class path with META-INF/beans.xml

If any of these locations has the beans.xml file, the EE server will scan all the classes in the archive and manage any managed beans it finds.

13.5.2 Using the beans.xml deployment descriptor

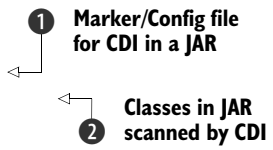
Chapter 12 covered CDI and the beans.xml file in detail so we won't cover it again here. But we'll quickly look at how to use the beans.xml file to package CDI in JAR archives, EJB-JAR modules, and WAR.

JAR

Suppose you're working on a non-EE application, or you're working on a supporting utility project to an EE application. In either case, you'll probably be creating a regular JAR archive to distribute your code. For the classes in your archive to use CDI, add the META-INF/beans.xml file. The following listing shows an example.

Listing 13.12 Structure of a JAR archive marked for CDI bean discovery

```
ActionBazaar-Utils.jar:
  META-INF/
    beans.xml
  com/actionbazaar/util/BidServiceUtil.class
```



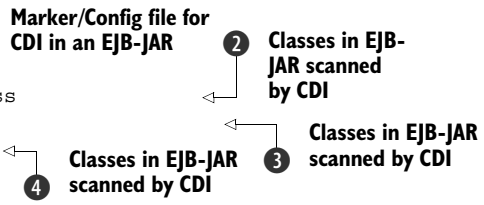
The beans.xml ❶ marks the archive for CDI bean discovery; ❷ is a class scanned by CDI, and if it contains CDI annotations, it becomes a bean managed by CDI.

EJB-JAR

EJB-JAR modules are exactly the same as JAR archives. In the module, add the META-INF/beans.xml file. The next listing shows an example.

Listing 13.13 Structure of an EJB-JAR marked for CDI bean discovery

```
ActionBazaar.jar:
  META-INF/
    bean.xml
  com/actionbazaar/ejb/BidServiceEjb.class
  com/actionbazaar/dao/BidDao.class
  com/actionbazaar/dao/BidDaoJdbc.class
```



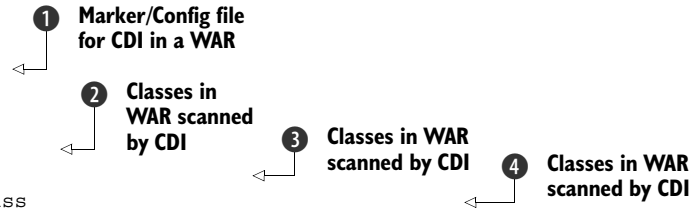
The beans.xml ❶ marks the module for CDI bean discovery; ❷–❹ are classes scanned by CDI, and if they contain CDI annotations, they're managed by CDI.

WAR

For WAR modules, add the WEB-INF/beans.xml file. The following listing shows an example.

Listing 13.14 Structure of a WAR marked for CDI bean discovery

```
ActionBazaar-web.war:
  WEB-INF/
    beans.xml
    classes/
      persistence/
        Category.class
        Item.class
        BazaarAdmin.class
```



The beans.xml ❶ marks the module for CDI bean discovery; ❷–❹ are classes scanned by CDI, and if they contain CDI annotations, they're managed by CDI. A WAR module also contains a WEB-INF/lib/ directory, which may contain any number of JAR archives. Remember, those JAR archives themselves may be marked for CDI by following the example of listing 13.12.

The beans.xml file has been vital to CDI up until Java EE 7. Now the Java EE 7 specification has made the beans.xml file optional. It's been replaced with the `bean-discovery-mode` annotation in the deployment descriptor. We'll look at this annotation next.

13.5.3 Using the bean-discovery-mode annotation

Once the beans.xml file could remain empty, and it served only as a marker for CDI; requests have been made to remove it altogether. The Java EE 7 specification has done just that. The `bean-discovery-mode` annotation has defined three modes that CDI uses to scan your classes in Java EE 7 applications. Table 13.3 describes these modes.

Table 13.3 Values for the bean-discovery-mode annotation

Value	Description
ALL	All types are processed. This behavior is the same as including the beans.xml in a Java EE 6 application.

Table 13.3 Values for the bean-discovery-mode annotation *(continued)*

Value	Description
ANNOTATED	Only types with bean-defining annotations are processed. This is the default Java EE 7 behavior.
NONE	All types in the archive (JAR) will be ignored.

In a Java EE 7 application, there's no need to include a beans.xml file. With no beans.xml file, CDI 1.1 will default to a beans discovery mode of ANNOTATED as described in table 13.3. This default behavior can be overridden by including a beans.xml and specifying a value for bean-discovery-mode, as shown in the following listing.

Listing 13.15 Specifying a value for bean-discovery-mode

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
       http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       bean-discovery-mode="all">
</beans>
```

❶ Change value for bean-discovery-mode

In this listing the value of bean-discovery-mode is set to "all" ❶, overriding the default Java EE 7 behavior and allowing all types to be processed as in Java EE 6.

Packaging your CDI code inside of JAR archives or EE modules is pretty easy. Next we'll go over some best practices for packaging and common problems you may run into.

13.6 Best practices and common deployment issues

After reading this chapter, it may appear that a lot of little pieces are required to deploy EJB 3 components. That may not be 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.

13.6.1 Packaging and deployment best practices

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

- *Start small.* Even if you're an experienced EE developer, start small when working on packaging your application and deploying it. Don't work for a month, generate hundreds of beans, and then try to package and deploy your application for the first time. If there are problems with the packaging, it's easier to solve them on a small deployment than a larger one.

- *Use a constant server environment.* For many, GlassFish is the choice for EE servers, but there are a lot of other options out there. When at all possible, make sure everyone on the team and all of your development environments are consistently using the same version of the same EE server. If EE servers must be different, avoid packaging problems by separating code from packaging.
- *Separate code from configuration packaging.* It's common for your applications to contain both your code and your configuration. It's also common for this to cause problems because packaging for development, test, and production environments may all differ. Avoid this headache by creating separate projects for packaging and move the packaging configuration (deployment descriptors!) to those projects. If you're using a build tool like Maven, declaring dependencies and combining projects is made easier. You may think it's overkill to have different projects for packaging to different environments, but it'll save you from packaging problems in the long run.
- *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's 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 so 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're depending on proprietary behavior, check whether you can take advantage of a proprietary deployment descriptor.
- *Leverage your database administrator (DBA).* Work with your DBA to automate the creation of any database schemas for your application. Avoid depending on the automatic table creation feature for entities, because it may not meet your production deployment requirement. Make sure that the database is configured properly and that it doesn't 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.

- *Use your build tools.* Most likely you'll be using Maven to build your applications, but whatever tools you use, make sure you understand how to use them well and take complete advantage of them. Avoid any manual intervention to package your application. If you find yourself with manual steps, you're either not using the tool to its full potential or your tool of choice isn't adequate and you should replace it with something that will better suit your needs.

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.

13.6.2 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 can't 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 you use `Thread.currentThread().getContextClassLoader().getResourceAsStream()`.
- `NoClassDefFoundException` is thrown when code tries to instantiate an object or when dependencies of a previously loaded class can't be resolved. Typically you run into this issue when all dependent libraries aren't at the same classloader 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 doesn't exist. The stack trace for this exception gives the details about which lookup is failing. Make sure that your dependencies on data sources, EJBs, and other resources resolve properly.
- `NotSerializableException` is thrown when an object needs to be moved from in-memory to some kind of `byte[]` form but the object doesn't support this conversion. This can happen if stateful session beans need to be passivated and saved to disk to free up memory, or it can happen if session beans are accessed remotely and the objects they return need to be transferred over the network. Whatever the reason, if the object isn't serializable, you'll get this exception. The best way to avoid this is to add a JUnit test to assert the object is serializable. Typically objects start life with the ability to be serialized, but as time goes on and the objects are updated, nonserializable stuff creeps in.

- 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.

13.7 Summary

At the heart of Java EE applications lies the art of assembling and packaging Enterprise applications. This chapter briefly introduced the concepts of class loading and how the dependencies between classes in an EE application are specified by the Java Enterprise specification. We looked at how to properly package an EJB-JAR module as both a standalone module and inside a WAR, making sure the deployment descriptor for the EJB-JAR module gets packaged in the proper location. After that we covered packaging JPA and CDI, which don't have their own specific EE module but instead are packed inside of existing EE modules. Finally, we looked at some best practices and common errors and how to avoid them.

JAVA

EJB 3 IN ACTION, Second Edition

Panda • Rahman • Cuprak • Remijan

The EJB 3 framework provides a standard way to capture business logic in manageable server-side modules, making it easier to write, maintain, and extend Java EE applications. EJB 3.2 provides more enhancements and intelligent defaults and integrates more fully with other Java technologies, such as CDI, to make development even easier.

EJB 3 in Action, Second Edition is a fast-paced tutorial for Java EE business component developers using EJB 3.2, JPA, and CDI. It tackles EJB head-on through numerous code samples, real-life scenarios, and illustrations. Beyond the basics, this book includes internal implementation details, best practices, design patterns, performance tuning tips, and various means of access including Web Services, REST Services, and WebSockets.

What's Inside

- Fully revised for EJB 3.2
- POJO persistence with JPA 2.1
- Dependency injection and bean management with CDI 1.1
- Interactive application with WebSocket 1.0

Readers need to know Java. No prior experience with EJB or Java EE is assumed.

Debu Panda, **Reza Rahman**, **Ryan Cuprak**, and **Michael Remijan** are seasoned Java architects, developers, authors, and community leaders. Debu and Reza coauthored the first edition of *EJB 3 in Action*.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/EJB3inActionSecondEdition

"The ultimate tutorial for EJB 3."
—Luis Peña, HP

"Reza sits on the EJB 3.2 Expert Group and is a great instructor. That's all you need to know."
—John Griffin, Progrexion ASG

"Thoroughly and clearly explains how to leverage the full power of the JEE platform."
—Rick Wagner, Red Hat, Inc.

"Provides a rock-solid foundation to EJB novice and expert alike."
—Jeet Marwah, gen-E

"If you have EJB troubles, this is your cure."
—Jürgen De Commer, Imtech ICT

Free eBook
SEE INSERT

ISBN 13: 978-1-935182-99-3
ISBN 10: 1-935182-99-4
55499
9 781935 182993

MANNING $54.99 / Can $57.99 [INCLUDING eBOOK]