Programmer to Programmer ™

PROFESSIONAL Java Server Programming J2EE 1.3 Edition

Subrahmanyam Allamaraju, Cedric Buest, John Davies, Tyler Jewell, Rod Johnson, Andy Longshaw, Ramesh Nagappan, Dr P. G. Sarang, Alex Toussaint, Sameer Tyagi, Gary Watson, Mark Wilcox, Alan Williamson

wrox
Programmer to Programmer™

Summary of Contents

24

J2EE Packaging and Deployment

As you know, the J2EE specification comprises a number of functional sub-specifications. However, it is not always obvious how these should be put together to form a complete J2EE application. The J2EE specification provides guidelines for the structuring and the creation of J2EE applications and one of the major ones is related to **packaging**. Individual specifications provide guidelines for the packaging of individual components such as EJBs, JSP pages, and servlets. The J2EE specification then dictates how these heterogeneous components are themselves to be packaged together.

This chapter provides a thorough analysis of the J2EE packaging mechanism. We will not cover how to build EJB, web application, or resource adapter archive files. These aspects of J2EE are discussed in earlier chapters of their own. Instead, we will focus on the relationships that these components have within an EAR file, and the process involved in building EAR files.

Some of the questions we will ask are:

- What are the rules for using J2EE packaging as opposed to component packaging?
- What can be placed into a J2EE package?
- Is J2EE packaging necessary and are there behavioral changes that occur as a result of using J2EE packaging?

In answering these questions we will learn:

- How J2EE class loading schemes work
- How to create Enterprise Archive (EAR) files
- How to deal with dependency and utility classes

J2EE Packaging Overview

A J2EE application is composed of:

- One or more J2EE components
- A J2EE application deployment descriptor

When one or more heterogeneous J2EE components need to use one another, a **J2EE application** must be created. There are many considerations that must be taken into account when building a J2EE application, including:

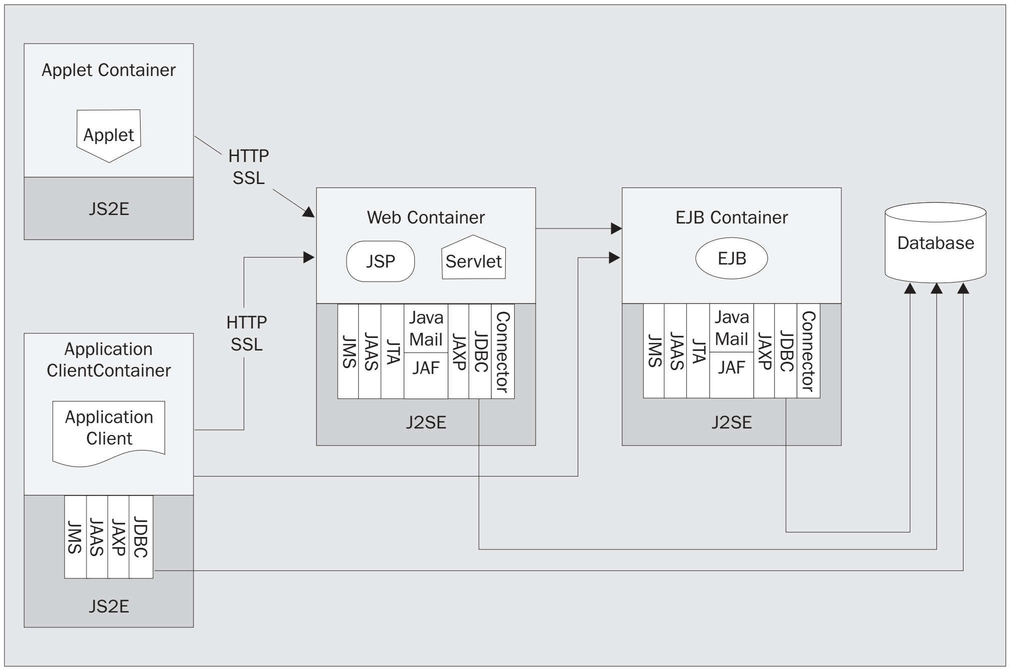
- The types of J2EE components that can be packaged into a J2EE application
- The roles that people play when creating J2EE packages
- The current limitations of J2EE packaging
- The class loading approaches that different vendors use to meet the needs of J2EE component interactions

What can be Packaged?

The J2EE specification differentiates between resources that run within a container and resources that can be packaged into a J2EE **Enterprise Application ARchive (EAR)** file.

An EAR file is used to package one or more J2EE modules into a single module so that they can have aligned classloading and deployment into a server.

J2EE clarifies the difference between run time containers and deployment modules. Run time containers are request-level interceptors that provide infrastructure services around components of the system. A deployment module is a packaging structure for components that will ultimately execute in a run time container. Recall how J2EE containers are structured:



- **The EJB Container**
  The EJB container provides containment and request-level interception for business logic. The EJB container allows EJBs to have access to JMS, JAAS, JTA, JavaMail (which uses JAF), JAXP, JDBC, and the Connector architecture.

- **The Web Container**
  The web container provides interception for requests sent over HTTP, FTP, SMTP, and other protocols. Most web containers only provide support for HTTP(S), but could support a broader range of protocols if they so chose. The web application container allows JSP pages and servlets to have access to the same resources as the EJB container provides.

- **The Application Client Container**
  An application client container provides request-level interception for standalone Java applications. These applications run remotely, in a different JVM from that in which the web container and EJB container operate.

  A program running in an application client container is very similar to a Java program with a `main()` method. However, instead of the application being controlled by a JVM, a wrapper controls the program. This wrapper is the application client container. Application client containers are a new concept in the J2EE specification and should be provided by your application server provider.

  An application client container can optimize access to a web container and EJB container through direct authentication, performing load balancing, allowing fail-over routines, providing access to server-side environment variables, and properly propagating transaction contexts.

  Programs that run within an application client container have access to JAXP, JDBC, JMS, and JAAS resources on a remote application server.

- **The Applet Container**
  An applet container is a special type of container that provides request-level interception for Java programs running within a browser. An important point to remember is that an applet container does not provide access to any additional resources such as JDBC or JMS.

  Applets running within an applet container are expected to make requests for resources directly to an application server (as opposed to making the request to the container and letting the container ask the application server). The EJB specification doesn't make any regulations as to how an applet should communicate with an EJB container, but the J2EE specification does. The J2EE specification requires that applets that want to directly use an EJB must use the HTTP(S) protocol and tunnel RMI invocations. Many application server vendors support a form of HTTP tunneling to allow for this.

The components that can be packaged into a J2EE EAR file do not directly correlate to those components that contain containers. There are no basic requirements for what must minimally be included into an EAR file. An EAR file is composed of any number of the following components:

- **EJB Application JAR Files**
  An EJB application JAR file contains one or more EJBs.
- **Web Application WAR Files**
  A WAR file contains a single web application. As an EAR file can contain multiple web applications, each web application in an EAR file must have a unique deployment context. The deployment mechanism for EAR files allows just such a specification of different contexts.
- **Application Client JAR Files**
  The application client JAR file contains a single, standalone Java application that is intended to run within an application client container. The application client JAR file contains a specialized deployment descriptor and is composed similarly to the way an EJB JAR file is composed.

  The JAR file contains the classes required to run the stand alone client, in addition to any client libraries needed to access JDBC, JMS, JAXP, JAAS, or an EJB client.
- **Resource Adapter RAR Files**
  The resource adapter RAR file contains Java classes and native libraries required to implement a Java Connector Architecture (JCA) resource adapter to an enterprise information system.

  Resource adapters do not execute within a container. Rather, they are designed to execute as a bridge between an application server and an external enterprise information system.

Each of these components are developed and packaged individually apart from the J2EE EAR file and own deployment descriptor. A J2EE EAR file is a combination of one or more of these components into a unified package with a custom deployment descriptor.

Packaging Roles

During the building, deployment, and use of an EJB, web application, or other component, different people will play different roles. The J2EE specification defines broad **platform roles** that developers play during the creation of an enterprise application. Even though there are many roles that individuals assume during the development and deployment process, these roles are nothing more than just logical constructs that allow an application to be better planned and executed. It is likely (and expected) that a single individual or organization will perform multiple roles. The common roles involved in building, deploying, or using an EAR file include:

- **J2EE Product Provider**
  The J2EE product provider provides an implementation of the J2EE platform including all appropriate J2EE APIs and other features defined in the specification. The J2EE product provider is typically an application-server, web-server, or database-system vendor who provides an appropriate implementation by mapping the specifications and components to network protocols.

- **Application Component Provider**
  The application component provider provides a J2EE component, for example an EJB application or a web application.

  There are also many roles within the J2EE specification that can be characterized as an application component provider. These include document developers, JSP authors, enterprise bean developers, and resource-adapter developers.

- **Application Assembler**
  The application assembler is responsible-combining one or more J2EE components into an EAR file to create a J2EE application. The application assembler is also responsible for creating the J2EE application deployment descriptor and identifying any external resources that the application may depend upon. These can include class libraries, security roles, and naming environments. The application assembler will commonly use tools provided by the J2EE product provider and the tool provider.

- **Tool Provider**
  A tool provider provides utilities to automate the creation, packaging, and deployment of a J2EE application. A tool provider can provide tools that automate the generation of deployment descriptors for an EAR file, the creation of an EAR file, and the deployment of an EAR file into an application server. Utilities provided by a tool provider can be either platform-independent (work with all EAR files irrespective of the environment) or be platform-dependent (working with the native capabilities of a particular environment).

- **Deployer**
  The deployer is responsible for deploying web applications and EJB applications into the server environment. The deployer is not responsible for deploying a resource-adapter archive or an application-client archive, but may be responsible for additional configuration of these components. These components, even though they are packaged as part of a J2EE EAR file, are not considered when the enterprise application is deployed. They are part of the J2EE application, but don't group through a run time "activation" process that web application and EJB containers go through during deployment. Resource adapter archives are simply libraries that are dropped into a valid JCA implementation. Although they are packaged as part of a J2EE EAR file they do not operate within the context of a J2EE container. Therefore, since resource-adapter archives do not have a J2EE container, the do not need to have a J2EE deployer involved with their activation.

  Application client programs do operate within the context of a J2EE container, but they are not deployed into an application server. Application client programs run standalone, and the deployer is not responsible for configuring the container environment for these programs.

  The deployer produces container-ready web applications, EJB applications, applets, and application clients that have been customized for the target environment of the application server.

- **System Administrator**
  The system administrator is responsible for configuring the networking and operational environment that application servers and J2EE applications execute within. The system administrator is also responsible for the monitoring and maintenance of J2EE applications.

In this chapter, during the discussion of the creation of EAR files and resolution of conflicts, we will be acting in the roles of application assembler and deployer.

The Limitations of Packaging

EAR files meet the basic requirements for packaging an application as most web-based J2EE applications are composed solely of web and EJB applications. However, EAR files lack the capability of packaging complicated J2EE applications. For example, the following components cannot be declared in an EAR file, but are often used in a J2EE application:

- JDBC `DataSource` objects.
- JMS `ConnectionFactory` and `Destination` objects.
- JMX `MBeans`.
- Some JMS consumers that run within an application server such as a `MessageConsumer` that runs as part of a `ServerSession`.
- Classes that are triggered when an application is deployed or un-deployed. (These classes are proprietary extensions provided by vendors not defined in the J2EE specification. However, all vendors generally supply them.)

At present, these components have to be manually configured and deployed via an administration interface provided by the implementation vendor and are the responsibility of the system administrator. Over time, the usage of these items will increase and consequently it will become important for EAR files to support the packaging of these components so that application portability is possible.

Understanding Class Loading Schemes

At runtime, when a class is referenced, it needs to be loaded by the Java Virtual Machine. The JVM uses a standard class loading structure for loading classes into memory. A class loader is a Java class that is responsible for loading Java classes from a source. Java classes can be loaded from disk, socket, or some other media; they can reside anywhere. Class loaders are hierarchical in the sense that they can be chained together in a parent-child relationship. Classes loaded by a child class loader have visibility (can use) classes loaded by any of the parent class loaders. Classes loaded by a parent class loader do not have visibility to classes loaded by any of its children's class loaders. Class loaders and EAR files are important since application server vendors can deploy application modules using common or different class loaders.

If, within an application, a web application needs to access an EJB, the web application will need to be able to load those classes it requires. Because of this implied dependency between different modules, application server vendors must consider different approaches for structuring EAR classloaders so that these dependencies are resolved.

A standalone application is deployed in its own class loader. This means that if you deploy a web application archive and an EJB application archive separately, the respective classes for each application will be loaded in different class loaders that are siblings of one another. The classes in the web application class loader will not be visible to the classes loaded by other class loaders. This creates a problem for web applications that want to use EJBs that have been deployed separately.

Before the advent of EAR files, many developers would deploy an EJB and then repackage the same EJB JAR file as part of the `WEB-INF\lib` directory of the web application. The same class files would exist in two different places so that the overall application could work correctly, a situation to be avoided. EAR applications were introduced to solve this problem. EAR files are not only a convenient packaging format; they also provide a special class loading scheme that allows applications within the EAR file to access the classes of other applications.

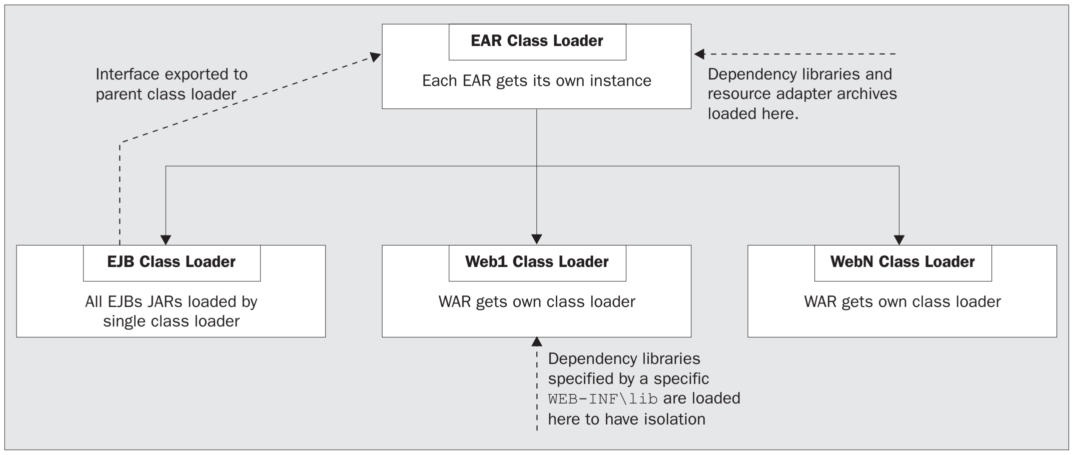
The J2EE 1.3 specification makes no specific requirements as to how an EAR class loader should work. This allows application server vendors flexibility in deciding how classes should be loaded. Before implementing an EAR class loader, a vendor needs to decide:

- ❑ Will all classes in all applications in the EAR file be loaded by a single class loader, or will separate files be loaded by different class loaders?
- ❑ Should there be any parent-child class loader relationships between different applications in the EAR file? For example, if two EJB applications depend upon `log4j.jar`, should appropriate visibility be maintained by loading `log4j.jar` in a parent class loader and loading the EJB applications in a child class loader so that?
- ❑ If a hierarchy of class loaders is created, to what depth will the hierarchy be allowed to extend?
- ❑ EJBs have inherent relationships with one another but web applications do not. So, will EJB applications be loaded differently from web applications so that web application integrity can be maintained?

The Pre-EJB 2.0 Option

Prior to the EJB 2.0 Public Final Draft 2, vendors had a lot of flexibility in choosing how to set up a class loading scheme. JSP pages and servlets that needed to make use of an EJB only needed to be able to load the home interface, remote interface, common classes, and stub classes of the EJB. The common classes such as exceptions and parameters should be placed into a dependency JAR file and loaded as a dependency package. This configuration requires vendors to determine a way for a web application that depends upon an EJB to load the home interface, remote interface, and stubs.

A vendor could implement this simple class loading scheme:


EAR Class Loader
Each EAR gets its own instance
Interface exported to parent class loader
Dependency libraries and resource adapter archives loaded here.
EJB Class Loader
All EJBs JARs loaded by single class loader
Web1 Class Loader
WAR gets own class loader
WebN Class Loader
WAR gets own class loader
Dependency libraries specified by a specific WEB-INF\lib are loaded here to have isolation

Client application class loading is not included in this model since a client application will execute within another virtual machine and will be isolated from all other components.

In this model, each EAR application would be loaded by a custom EAR class loader instance. EJB applications and web applications would each be loaded by custom class loaders that are children of the EAR class loader. Any class file that is to be shared by more than one application in the EAR will be loaded by the EAR class loader. (This includes any common dependency libraries and resource adapter archives.) Any files loaded at the EAR class loader level are automatically visible to all classes loaded by child class loaders.

All EJB applications are loaded by a single EJB class loader that is a child of the EAR class loader. Even if you have different EJB JAR files, they will all be loaded by the same class loader. This is done to facilitate EJB-to-EJB calls in different applications that are hosted on the same JVM.

Each web application is loaded in a different class loader to maintain class isolation. For example, if every web application contains a file called `index.jsp`, that servlet created from the JSP page could have the same class name as the equivalent servlets in other web applications. Each web application needs to be able to load its own version of that servlet, so each web application must be isolated in its own class loader.

In order for web applications to make use of the EJBs deployed in the same EAR file, the web applications need to have visibility to the external interfaces and stub implementation classes of those EJBs. Since the EJB and web application class loaders are siblings, the web applications do not have direct visibility to the right class files. The web application class loader and the EJB application class loader do have the same parent class loader, however. To allow the web application to be able to use the class files of the EJB, the EJB class loader can take all of the public interfaces of each of the EJBs and their stub implementation files and 'export' them to the EAR class loader in which they will be visible to all applications in the EAR. The public interfaces and stub implementation files are the classes needed by a client to make use of an EJB. The web applications will then be able to load the classes required to use any EJB.

Dependency utility libraries can be loaded in different places depending on where the library is specified. If a single web application lists a dependency library in its `WEB-INF\lib` directory, then that library is unique to that web application. There is no need for other applications to access the contents of that library and so the EAR class loader should not load that library. In this situation, the web application class loader will load the utility JAR file. Other web applications should include the same dependency in their own `WEB-INF\lib` to maintain this isolation.

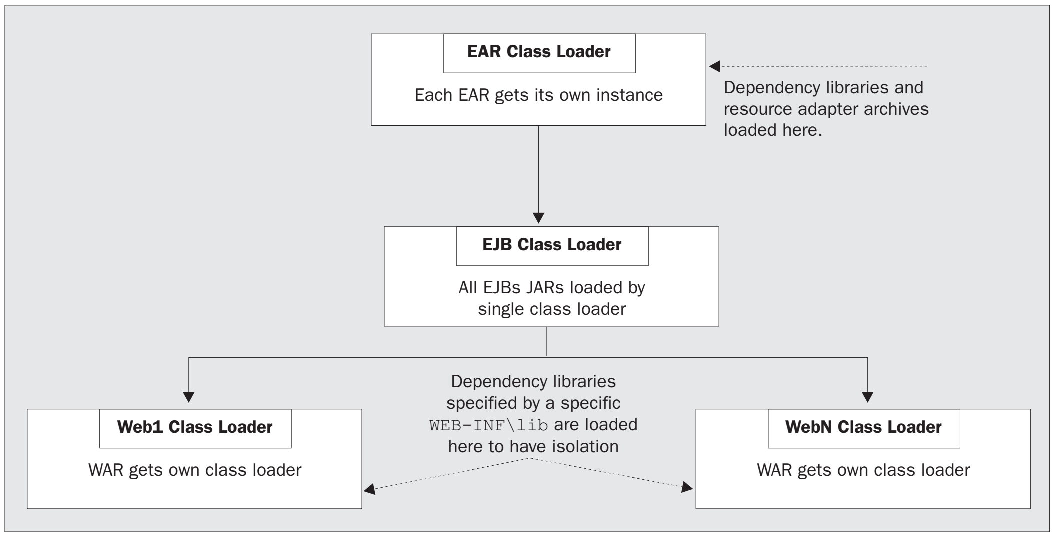
Dependency utility libraries that must be shared between EJB and web applications need to be loaded at the EAR class-loader level. It turns out that an EAR class loader will load any library specified as a dependency of an EJB in order to give the right visibility to those dependency classes. This allows an EJB developer to package any common exception classes, custom input parameter classes that are visible to a web application, and the EJB into a dependency library. This library is commonly called `common.jar` but does not have to be named that. In addition to the public interfaces and stub implementation classes, the common utility library will also be loaded at the EAR level, which allows a web application visibility to all of the classes used by the EJB.

Resource-adapter archives that are packaged within the EAR file along with EJBs and web applications are automatically loaded at the EAR class-loader level.

The Post-EJB 2.0 Option

The EJB 2.0 Public Final Draft 2 introduced the concept of local interfaces and placed an interesting twist on the EAR class loading problem. Local interfaces allow co-located clients and EJBs to be accessed using pass-by-reference semantics instead of pass-by-value semantics.

Having visibility to the public interfaces and stub implementation classes of an EJB is not sufficient for a client of an EJB to perform pass-by-reference invocations. The client needs to have a direct reference to the implementation classes of the EJB's container. With local interfaces, clients of EJBs need access to much more than before. This restriction means that the class loading scheme used pre-EJB 2.0 will not work. To solve this problem the class loaders of any applications that act as clients to an EJB must be loaded as children of the EJB class loader:


EAR Class Loader
Each EAR gets its own instance
Dependency libraries and resource adapter archives loaded here.
EJB Class Loader
All EJBs JARs loaded by single class loader
Dependency libraries specified by a specific WEB-INF\lib are loaded here to have isolation
Web1 Class Loader
WAR gets own class loader
WebN Class Loader
WAR gets own class loader

In this model, web application class loaders are children of the EJB class loader. This allows all web applications to have visibility to the files they need to allow them to behave as clients of the EJBs. Each web application is still loaded in a custom class loader to achieve isolation, though. The overall structure of this implementation is simpler to understand, as it does not require the EJB class loader to export any files to the EAR class loader.

An Ambiguity in the J2EE Specification

An ambiguity in the J2EE specification has been exposed by certain implementations. It arises because the J2EE specification is ambiguous as to how dependency libraries of a web application should be loaded. It is very clear that a utility library specified by `WEB-INF\lib` should remain isolated and be loaded by the class loader of the web application only. However, if a utility library is specified as a dependency library of the web application, it is not stated whether the library should be loaded by the web application's class loader or exported to the EAR class loader. This can have a behavioral impact. If it is known that a dependency utility library will only be loaded once for all web applications, the web applications can take advantage of knowing that a singleton class will only create a single object that can be shared among all web applications. However, if each web application's class loader isolated the utility library, a singleton class, which is a class that is intended to only create a single instance in the virtual machine, would create a single object in each web application.

Currently, Silverstream's application server and the J2EE Reference Implementation load utility libraries specified as a dependency library of a web application at the EAR class-loader level. WebLogic Server 6.0 loaded these libraries as part of the web application class loader. However, WebLogic Server 6.1 modified this approach to support the loading of web application dependency libraries at the EAR level. This makes sense since web application isolation can always be achieved by placing utility libraries in the `WEB-INF\lib` directory. This provides the best of both worlds: a dependency library loaded at the EAR class-loader level or a dependency library loaded at the web application class-loader level.

Configuring J2EE Packages

Now that we have a basic understanding of how the J2EE architecture is implemented, specifically the different roles and the behavior of class loaders, we are ready to configure and deploy enterprise applications. To do this we need to understand the process of EAR file creation, and the contents of the deployment descriptors that describe their contents.

The Enterprise Application Development Process

The overall process that is used to build an enterprise application is:

1. Developers build individual components; these can be EJBs, JSP pages, servlets, and resource adapters.

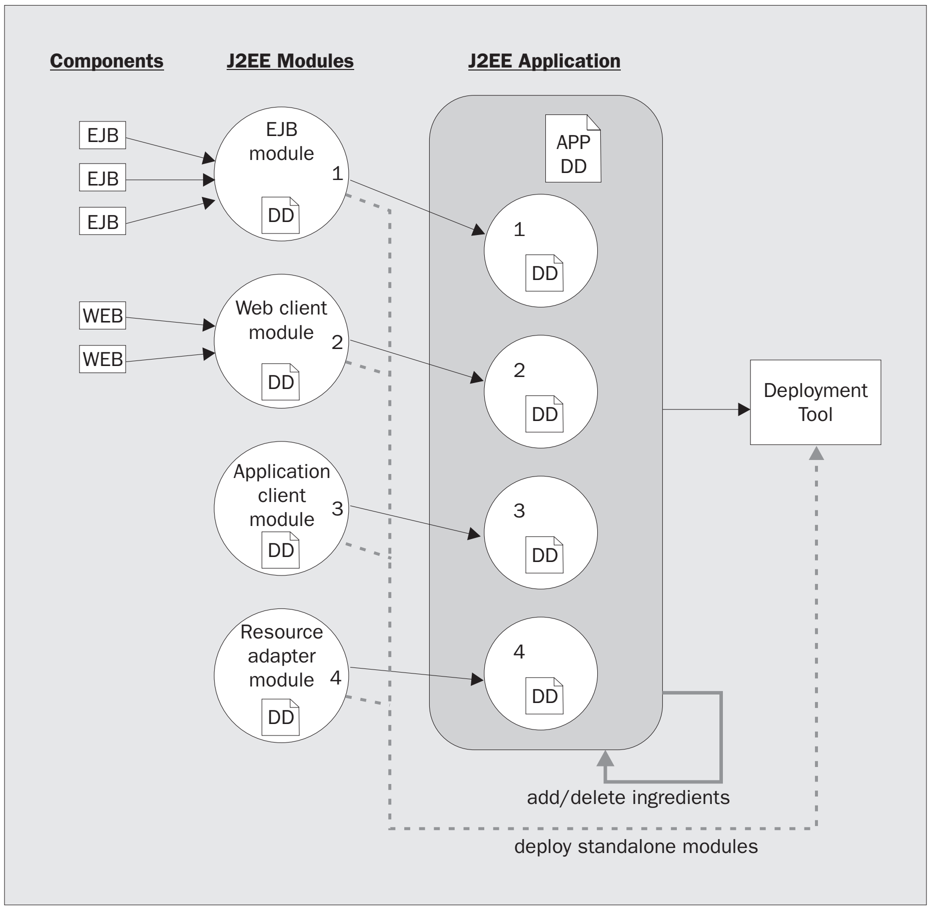
2. Some number of components are packaged into a JAR file along with a deployment descriptor to a J2EE module. A J2EE module is a collection of one or more J2EE components of the same component type, so an EJB module can comprise more than one EJB; a web application module can comprise multiple JSP pages and servlets; a resource adapter archive can comprise multiple resource adapters.

3. One or more J2EE modules are combined into an EAR file along with an enterprise application deployment descriptor to create a J2EE application. The simplest J2EE application is composed of a single J2EE module. More complicated J2EE applications are composed of multiple J2EE modules. A complex J2EE application comprise multiple J2EE modules, and dependency libraries that are used by the classes contained within the modules. A J2EE application may also contain help files and other documents to aid the deployer.

4. The J2EE application is deployed into a J2EE product. The J2EE application is installed on the J2EE platform and then integrated with any infrastructure that exists on an application server. As part of the J2EE application deployment process, each J2EE module is individually deployed according to the guidelines specified for deployment of that respective type. Each component must be deployed into the correct container that matches the type of the component.

   For example, if you have a `my.ear` with a `my.jar` and a `my.war` contained within the EAR file, when the application is deployed, the application server's deployment tool will copy the `my.ear` file into the application server. Next, the application server's deployment mechanism will extract the `my.jar` and `my.war` modules and deploy them separately following the class loading guidelines of that platform. If each of the modules deploys successfully, then the J2EE application is considered to have deployed successfully.

The J2EE enterprise application development and deployment process might work like this:



Components are built and packaged into a J2EE module with a deployment descriptor; a deployment tool can be used to create these J2EE modules. The deployment tool can also be used to deploy and un-deploy standalone J2EE modules; to take one or more J2EE modules and package them into a J2EE application with another deployment descriptor; to add or remove items from the J2EE application; or to deploy the entire application to an application server.

The Structure of a J2EE Package

The structure of a J2EE enterprise application package is straightforward; it is composed of one or more J2EE modules and a deployment descriptor named `application.xml` in a directory named `META-INF\`. The files are packaged using the JAR file format and stored in a file with an `.ear` extension. Optionally, you can include dependency libraries within the EAR file. The general structure of an EAR file is:

```
EJB .jar Files
Web Application .war Files
Resource Adapter .rar Files
Application Client .jar Files
Dependency Library .jar Files
```

```
META-INF\
         application.xml
```

An example EAR file that has an EJB module and a web application module with no dependency libraries might look like:

```
FirstEJB.jar
FirstWeb.war
META-INF\
         application.xml
```

J2EE modules that are stored in the EAR file do not necessarily have to be in the root directory of the structure. For example, the contents of an EAR file that has an EJB module and a resource adapter archive stored in subdirectories might look like:

```
ejbs\
     SecondEJB.jar
resources\
          LegacyAdapter.rar
META-INF\
         application.xml
```

Finally, an EAR file that has many components and dependency libraries may look like:

```
ejbs\
     ThirdEJB.jar
     FourthEJB.jar
resources\
          LegacyAdapter.rar
web\
    WebApp1.war
    WebApp2.war
lib\
    xmlx.jar
    common.jar
META-INF\
         application.xml
```

The EAR file is created using a deployment tool provided by a tool provider or alternatively by using the `jar` tool provided with the JDK. The creation steps are:

- ❑ Create a staging directory that will contain the contents of the EAR file
- ❑ Place all of the J2EE modules into the staging directory and create the `META-INF\` directory
- ❑ Create the `application.xml` deployment descriptor and place it into the `META-INF\` directory
- ❑ After you have built up the staging directory, go to the root of the directory and run the `jar` utility to create the EAR file

An example execution of the `jar` utility for the complex example discussed earlier might be:

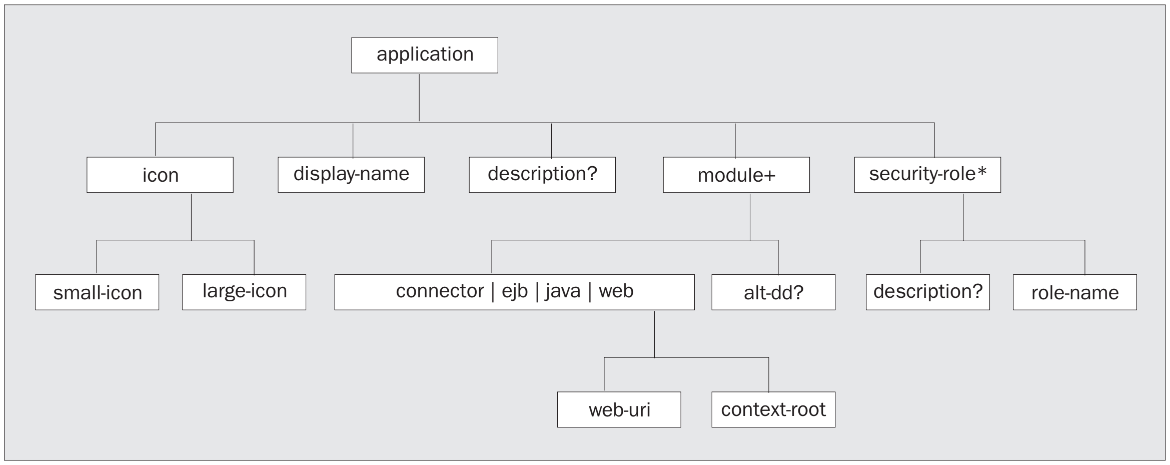
```
jar cvf Application.ear ejbs resources web lib META-INF
```

Once the EAR file has been created, you are free to deploy the J2EE application into the application server.

Working with the EAR Deployment Descriptor

Ideally, a graphical deployment tool would be used to build the `application.xml` files. However, there may be many situations where you need to construct or maintain `application.xml` manually, and so it is important to understand the tags that are used.

The `application.xml` deployment descriptor is straightforward. A valid descriptor doesn't require many tags. The possible tags contained within the DTD of the deployment descriptor are:



All valid J2EE application deployment descriptors must contain the following `DOCTYPE` declaration:

```
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application
1.3//EN" "http://java.sun.com/dtd/application_1_3.dtd">
```

Configuring a simple `application.xml` deployment descriptor takes only a few steps:

1. The `<application>` tag is used to declare an enterprise application. The `<application>` tag can contain an `<icon>`, `<display-name>`, and `<description>` for use by a deployment tool to provide descriptive information about the application. The content of these tags is the same as the content for the same tags in EJB, web application, and resource adapter deployment descriptors.

2. Each J2EE module included in the enterprise application must have an equivalent `<module>` tag describing the module. EJBs are described using the `<ejb>` tag, web applications are described using the `<web>` tag, resource adapters are described using the `<connector>` tag, and application client programs are described using the `<java>` tag. With the exception of the `<web>` tag, the content of the other tags is a relative URI naming the file that contains the J2EE module within the EAR file. The URI must be relative to the root of the EAR file.

3. If your enterprise application contains a web application J2EE module, you need to provide a `<web-uri>` and a `<context-root>`. The `<web-uri>` tag is a relative URI naming the file that contains the J2EE module within the EAR file. This is the same type of URI that is specified for the `<ejb>`, `<connector>`, and `<java>` tags. The `<context-root>` specifies the name of the context under which the web application will run. Subsequently, all requests for JSP pages and servlets for that web application must be preceded by this web context. For example, if you deploy a web application with:

   ```
   <context-root>web1</context-root>
   ```

   all HTTP requests for JSP pages and servlets will always be preceded with:

   ```
   http://host:port/web1/<AsSpecifiedInServletSpec>
   ```

   Each web application packaged within the EAR file must have a unique `<context-root>` value. Two web applications packaged in the same EAR file cannot have identical `<context-root>` values. If there is only one web application in the EAR file, the value of `<context-root>` may be an empty string.

Example

The most common use of EAR files will be the scenario where an enterprise application has a single EJB module and a single web application module that makes use of the EJB components deployed in the EJB module. In this example, the EJBs and web applications do not depend upon any dependency libraries. This section describes the steps involved in building this example.

Packaging the Components

This example has a servlet invoke an `invoke()` method on the remote interface of a stateless session EJB. The servlet and EJB print out statements to the console to indicate that they are successfully executing. If the console receives any exceptions, it is likely to be an indication that the packaging of the components was not done correctly. All of the EJB source files for this example are in the `wrox` package. The servlet is in the unnamed package. The Java files used to implement this example include:

- `EnterpriseServlet.java`
  The servlet implementation class that invokes the EJB
- `Enterprise.java`
  The remote interface of the EJB
- `EnterpriseHome.java`
  The home interface of the EJB
- `EnterpriseBean.java`
  The implementation class of the EJB

For complete information on how to author servlets and EJBs, please reference the appropriate chapters in this book. This section will only list the relevant code snippets to allow the reader to follow the example. Authoring EJB and web application deployment descriptors, creating JAR files, and creating web application WAR files are not demonstrated here and discussed in Chapters 7 to 21.

The sourcecode for the `EnterpriseBean.java` bean implementation class is:

```
package wrox;

import javax.ejb.*;

public class EnterpriseBean implements SessionBean {

private InitialContext ctx;

public void ejbCreate() {}
public void ejbRemove() {}
public void ejbActivate() {}
public void ejbPassivate() {}
public void setSessionContext(SessionContext c) {}

public void invoke() {
System.out.println("Executing in EJB.");
}
}
```

The sourcecode for the `EnterpriseServlet.java` servlet class is:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import javax.naming.*;

public class EnterpriseServlet extends HttpServlet {

  public void service(HttpServletRequest req, HttpServletResponse res)
              throws IOException{
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    try {
      System.out.println("Servlet Executing in Server");
      InitialContext ctx = new InitialContext();

      wrox.EnterpriseHome eHome = (wrox.EnterpriseHome)
                              ctx.lookup("EnterpriseEJB");
      wrox.Enterprise e = eHome.create();
      e.invoke();

    } catch(Exception e) {
      out.println("Exception: " + e);
      System.out.println("Exception: " + e);
    }

    out.println("<html><head><title>Title</title></head>");
    out.println("<body>");
    out.println("<h1>See console to ensure EJB was invoked.</h1>");
    out.println("</body></html>");
}
}
```

After the development of the EJB code and the relevant deployment descriptors (not listed here), the EJB should be packaged into a file named `EnterpriseBean.jar`. The EJB is configured to bind itself to `EnterpriseEJB` in the JNDI namespace.

After the development of the servlet code and the relevant deployment descriptors (also not listed here), the servlet should be packaged into a file named `WebApp.war`. The servlet class is registered to execute with the `/enterpriseservlet/` mapping.

Application Assembly

After completing the component build process, the enterprise application deployment descriptor must be developed. We need to register the EJB and web application as modules of the enterprise application. We also want the web application's components to execute under the `/web/` context root. The `application.xml` file for this example becomes:

```
<?xml version="1.0"  encoding="UTF-8"?>

<!DOCTYPE application PUBLIC '-//Sun Microsystems, Inc.//DTD J2EE Application
1.3//EN' 'http://java.sun.com/dtd/application_1_3.dtd'>

<application>
  <display-name>Enterprise Application</display-name>
  <module>
    <ejb>EnterpriseBean.jar</ejb>
  </module>
  <module>
    <web>
      <web-uri>WebApp.war</web-uri>
      <context-root>web</context-root>
    </web>
  </module>
</application>
```

After creating the `application.xml` deployment descriptor, the enterprise application build directory should resemble:

```
EnterpriseBean.jar
WebApp.war
META-INF\
         application.xml
```

To create an EAR file named `Enterprise.ear` using the `jar` utility, the following command should be entered on the console:

```
jar cvf Enterprise.ear EnterpriseBean.jar WebApp.war META-INF
```

What's interesting about the `deploytool` that is provided with the J2EE Reference Implementation is that developers are never expected to author `ejb-jar.xml`, `web.xml`, or `application.xml` deployment descriptors. These files are always automatically generated for you. In the case of this enterprise application, the `application.xml` that is placed into the EAR file is generated automatically for the developer.

Application Deployment

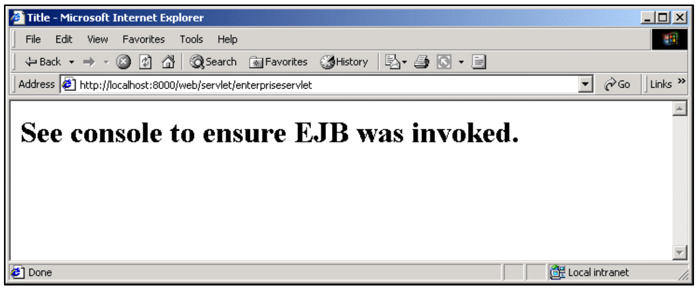
After the EAR file has been built, it needs to be deployed. Keep in mind that deployment is vendor specific and each vendor provides custom tools to allow this. The J2EE `deploytool` utility has an option to deploy an enterprise application to the Reference Implementation.

Running the Application

After the enterprise application has been successfully deployed, executing the client involves invoking the servlet that was deployed in the web application. Since the context root of the enterprise application is `/web`, the servlet is invoked as part of that. For example, to invoke the servlet, you would enter the following in a browser's window:

http://<server_name>:<server_port>/web/enterpriseservlet/

You should see the following appear in the browser after the servlet executes:


Title - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address
Go
Links
Done
Local intranet

http://localhost:8000/web/servlet/enterpriseservlet

See console to ensure EJB was invoked.

Optional Deployment Descriptor Tags

Two optional deployment descriptor tags can be used in certain scenarios. They are `<alt-dd>` and `<security-role>`.

`<alt-dd>` is a sub-tag of `<module>`. The value of this tag is a URI that would point to another deployment descriptor file for the module referenced from the root of the EAR file. The file does not have to be named the same as it is named inside the J2EE module. For example, all EJB module deployment descriptors must be named `ejb-jar.xml`. The value of this tag can be a file named other than `ejb-jar.xml` if it is referencing an alternative deployment descriptor for an EJB module.

The deployment descriptor file would override the one contained within the J2EE module. This is a type of post-assembly deployment descriptor. This tag can be used to reference an external version of the deployment descriptor that should be used if a deployer wants to use a deployment descriptor that is different from the one contained within an EJB, a web application, a resource adapter, or an application client module. If this value is not specified, then the deployment tool must use the values specified within the JAR, WAR, or RAR files provided in the EAR file. For example, to specify a web application with an external, alternative deployment descriptor that is located at the root of the EAR file, you would write:

```
<module>
  <web>
    <web-uri>web.war</web-uri>
    <context-root>web</context-root>
  </web>
```

```
    <alt-dd>external-web.xml</alt-dd>
  </module>
```

`<security-role>` allows the deployer to specify application-level security roles that should be used for all J2EE modules contained within the EAR file. If an EAR file contains multiple EJB modules and/or multiple web application modules, each of those modules may have its own security roles defined within. One of the deployer's responsibilities is to ensure that the names of all security roles contained within all J2EE modules are unique and have meaning for the application as a whole. Security roles can be 'pulled up' from the J2EE module level to the enterprise application level and included in this tag. If there is a duplicate security role value in one of the J2EE modules, that value can be removed if the value is provided at the enterprise application level.

This tag requires a `<role-name>` sub tag to actually provide the symbolic name of the security role. An example of configuring a `<security-role>` tag is:

```
  <security-role>
    <description>
      This is administrator's security role
    </description>
    <role-name>Administrator</role-name>
  </security-role>
```

Issues with the Ordering of Modules

The J2EE specification doesn't make any specifications for how J2EE modules contained within an EAR file should be deployed. In particular, the order in which modules must be deployed is not explicitly outlined in the specification. This can be an issue if a component in one module needs to make use of another component in another module that has yet to be deployed.

Most application servers will deploy EAR files using the same approach:

1. All resource adapters contained within the EAR file will be deployed into the connector infrastructure. If multiple resource adapters are configured, they will be deployed in the order that they are listed in the `application.xml` deployment descriptor.

2. All EJB modules will be deployed. EJBs are deployed after resource adapters since EJBs may make use of a particular resource adapter during their initialization stage. If multiple EJB modules are configured, they will be deployed in the order that they are listed in the `application.xml` deployment descriptor.

3. All web application modules will be deployed. Web applications are deployed after EJBs and resource adapters since web applications may make use of these resources during their initialization stage. If multiple web application modules are configured, they will be deployed in the order that they are listed in the `application.xml` deployment descriptor.

Issues with Dependency Packages

The most frequent question raised about J2EE packaging is about utility and support classes. When packaging a web application or an EJB application, where should these libraries be placed? If you place these classes into the standard classpath of your application server, they will likely lose any unloading ability that web and EJB applications have that are driven by the class loaders used to load them at deployment. If your web/EJB applications need to change the version of the libraries that they use, then the dependent library will need to be re-deployed when the web/EJB application is re-deployed. In this scenario, storing utility classes on the standard classpath is not a feasible option since the entire application server would have to be restarted for each deployment of a web/EJB application, which is clearly not ideal.

Given the standard definition of J2EE, where are dependency libraries supposed to be placed so that they can be re-deployed with an application at run time? There are two creative, yet ultimately undesirable, solutions:

- Dependency libraries that are packaged as JAR files can be placed in the `WEB-INF\lib` directory of a web application. Generally, the `WEB-INF\lib` directory should primarily be used for the storage of servlet classes, but servlets and JSP pages will look for classes in this directory when loading new ones. If the utility library that you are using is only needed by your servlets and JSP pages then this solution will be sufficient. However, if the same libraries are also needed by EJBs, JMS consumers, or startup and shutdown classes, then this option will not work as the `WEB-INF\lib` directory is not visible to these items.
- A complete copy of all of the utility libraries is placed in each EJB JAR file in addition to the `WEB-INF\lib` directory. When an EJB is deployed, an EJB class loader will only look within its own JAR file for any utility classes that are referenced. It will not look within the JAR files of other EJB applications that have been deployed or in the `WEB-INF\lib` directory. If all of your EJB applications require the use of the same library, then placing a copy of that library's classes in each JAR file will meet your needs. The utility classes will be re-deployable along with the EJB.

Although the second scenario achieves redeploy ability of dependency libraries, it is incredibly inefficient. The purpose of having multiple JAR files for packaging is to promote modularity of applications and placing the same class in multiple JAR files destroys this. In addition, having multiple copies of the same classes unnecessarily bloats your applications. Finally, there is an added step in the build process, as every JAR file will have to be rebuilt if you want to change even a single library.

Solutions

One of the possible solutions to this problem is to eliminate the need for multiple JARs in J2EE applications by converging all EJBs and their utility classes into a single, unified package. The EJB 2.0 specification is driving some projects to do this. This version of the specification mandates that entity EJBs participating in a relationship do so using local interfaces and so requires both of the EJBs in the relationship to be packaged into the same JAR file. Earlier drafts of EJB 2.0 allowed EJBs in different JAR files to participate in relationships, promoting greater modularity of the system, but ultimately limited the persistence optimizations available for CMP entity beans in a relationship.

Public Final Draft 2 eliminated remote relationships; so many vendors are thinking about providing tools that perform EJB JAR convergence. These tools will take as input two valid EJB JAR files and merge their classes and deployment descriptors into a single, unified package. You could potentially use one of these convergence tools to re-package your existing JAR applications to reduce redundancy of dependency libraries among EJB JAR files. At the time of writing, these convergence utilities were still being developed. Check with your application server provider to see if they have a convergence utility available for you to use.

Keep in mind that even if all of the EJBs are converged into a single JAR application, you will have eliminated copies of your dependency library among the EJBs, but a copy will still need to exist in a `WEB-INF\lib` library if a web application depends upon it. Additionally, the need for modularity of EJB applications still exists since many companies desire to re-deploy EJBs on an individual basis. Since every EJB in a JAR will be re-deployed when that JAR file is re-deployed, an unnecessary amount of deployment processing could occur if your only desire is to re-deploy a single EJB.

A Better Solution

With the release of JDK 1.3, Sun Microsystems redefined the "extension mechanism" which is the functionality necessary to support optional packages. The extension mechanism is designed to support two things:

- JAR files can declare their dependency upon other JAR files allowing an application to consist of multiple modules
- Class loaders are modified to search optional packages and application paths for classes

Additionally, the J2EE 1.3 specification mandates that application servers must support the extension mechanism as defined for JAR files. This requires any deployment tool that references a JAR file be capable of loading any optional libraries defined through the extension mechanism. It also implies that if an application server or deployment tool supports run time un-deployment and re-deployment of EJB applications that use libraries via the extension mechanism, then that tool or application server must also support un-deployment and re-deployment of any dependent libraries.

Support for the extension mechanism does not exist for EAR or resource adapter applications as defined in the J2EE specification, since these applications are not directly loaded by an instance of `ClassLoader`. Web applications have the freedom of using the extension mechanism or the `WEB-INF\lib` directory when specifying a dependency library. As we discussed earlier, how a dependency library is loaded can vary depending upon whether the library is specified using the extension mechanism or the `WEB-INF\lib` directory.

Enterprise applications need to re-package any libraries that are needed by the web application or EJB application as part of the EAR file. Once packaged, the extension mechanism provides a standard way for web application WAR files and EJB application JAR files to specify which dependency libraries that exist in the enterprise application EAR file are needed.

Understanding the Manifest Classpath

How does the extension mechanism work with EJB applications? A JAR file can reference a dependent JAR file by adding a `Class-Path:` attribute to the manifest file that is contained in every JAR file. The `jar` utility automatically creates a manifest file to place in a JAR file and names it `manifest.mf` by default. This file can be *edited* to include a `Class-Path:` attribute entry in addition to the other entries that already exist in the file. In fact, many EJB packaging tools that are being released by vendors are taking dependency packages into account as part of the packaging process and will automatically create an appropriate `manifest.mf` file that contains a correct `Class-Path:` attribute entry.

For example, if you create an EJB JAR file and modify the `manifest.mf` to include a `Class-Path:` attribute, the container generation utility provided by your application server vendor *must* preserve this entry when it generates a new EJB application file. With WebLogic Server 6.1, if you provide an EJB JAR utility that already contains a `Class-Path:` entry in the `manifest.mf` file, the `weblogic.ejbc` utility will preserve this entry when it generates a new EJB application with the container files. At the time of printing of this book, a tool that creates and inserts the `Class-Path:` entry into a `manifest.mf` file does not yet exist. Unfortunately, this task has to still be done by hand by editing the `manifest.mf` file of a JAR file.

The `Class-Path:` manifest attribute lists the relative URLs to search for utility libraries. The relative URL is always from the component that contains the `Class-Path:` entry (not the root of the EAR file). Multiple URLs can be specified in a single `Class-Path:` entry and a single manifest file can contain multiple `Class-Path:` entries. The general format for a `Class-Path:` entry is:

```
Class-Path: list-of-jar-files-separated-by-spaces
```

For example, a JAR file might have:

```
Class-Path: log4j.jar xmlx.jar foo/bar/util.jar
```

If you use the extension mechanism in a J2SE application, the `Class-Path:` manifest entry can reference directories too. However, for J2EE applications that are wholly contained within JAR files, the `Class-Path:` manifest entry can only reference other JAR files. Additionally, the `Class-Path:` entry must reside on a separate line apart from other attribute entries in the same manifest file.

The extension mechanism is a nice capability especially since it is designed to handle circular redundancies by creating a unified classpath containing all dependencies in first-parsed-based ordering. For example, if the first EJB application parsed is `EJB1.jar` and it references:

```
Class-Path: jaxp.jar EJB2.jar ..\xmlx.jar
```

a class loader will then parse `EJB2.jar` that references:

```
Class-Path: jaxp.jar EJB1.jar
```

The resulting "application" classpath that a class loader would ultimately use would be:

```
Class-Path: jaxp.jar EJB2.jar ..\xmlx.jar EJB1.jar
```

Dependency Example

This example is designed to demonstrate every possible loading configuration for an enterprise application. It demonstrates multiple EJB modules, multiple web applications, and dependency libraries that are unique and shared between these applications. Deploying and executing this example allows us to see how an application server loads different classes from different applications.

By printing out the `ClassLoader` hierarchy for different classes as they are executed we can see if all classes loaded in a single class loader, or different class loaders, and if so, what the hierarchy of the class loaders is.

Our example consists of two EJB modules, two web application modules, and seven dependency libraries that are used in different scenarios. The structure of the EAR file is:

```
Depend1-container.jar
Depend2-container.jar
WebApp1.war
WebApp2.war
Util1.jar
Util2.jar
Util3.jar
Util4.jar
Util5.jar
Util6.jar
Util7.jar
META-INF\
         application.xml
```

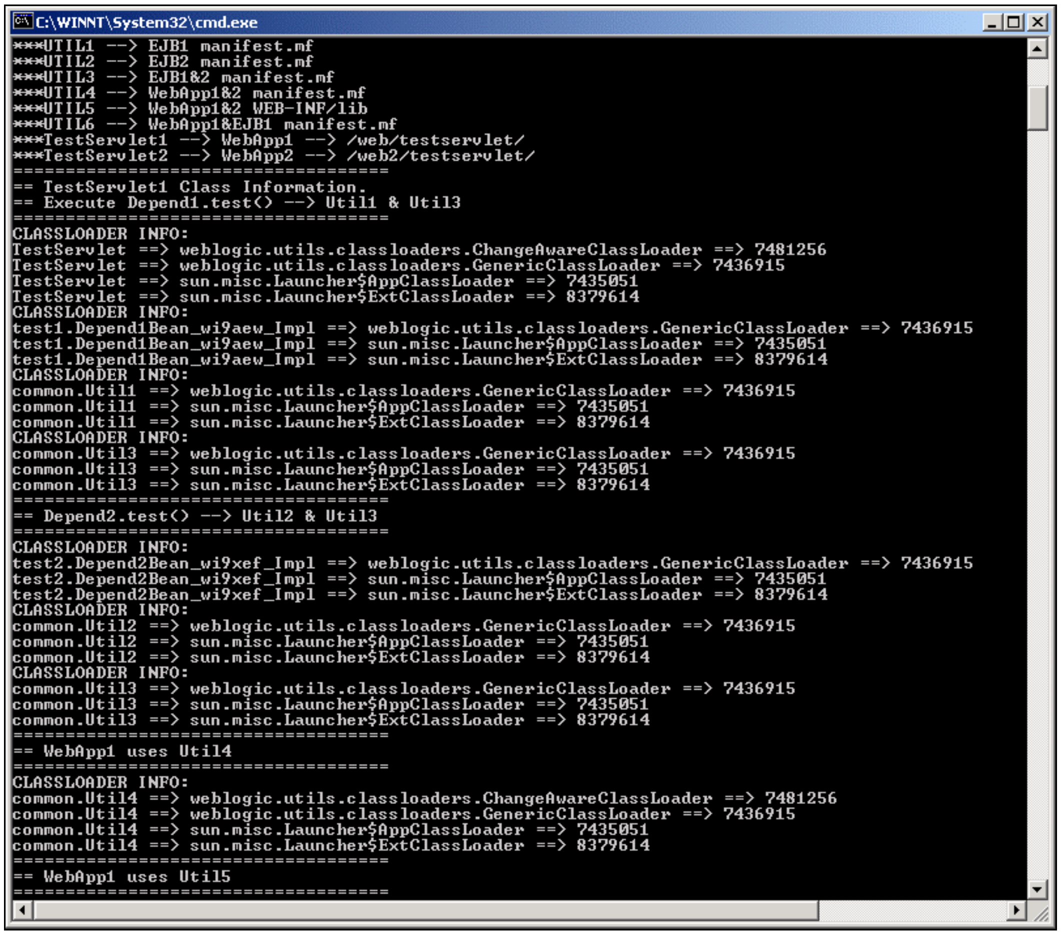
There is a single EJB in each of the `Depend` JAR files. There is also a servlet, `TestServlet`, which exists in each web application. Each of the dependency libraries contains a single class with a single method that prints out the `ClassLoader` hierarchy. This example causes different scenarios to occur to see how classes are loaded in the context of an EAR file. In particular, the following scenarios are tested:

- ❑ How is a dependency library loaded when used by an EJB and referenced in that EJB's manifest classpath?
- ❑ How is a dependency library loaded that is shared between EJBs in different EJB modules? The dependency library is specified in the manifest classpath of both EJBs.
- ❑ How is a dependency library loaded when it is referenced by a web application and stored in the manifest classpath of the web application module?
- ❑ How is a dependency library loaded when it is referenced by a web application and stored in the `WEB-INF\lib` directory of the web application module?
- ❑ How is a dependency library loaded when an EJB module and a web application module reference it in both of their manifest classpath?

To execute this example, deploy the `Depend.ear` file (available from http://www.wrox.com) and then run the `TestServlet` that is deployed in each of the web applications. The `TestServlet` will invoke the appropriate EJB methods that, in turn, invoke the methods on the classes in the dependency library. For example, to execute both servlets, you would use the following URIs in a browser:

http://<server_name>:<port_number>/web1/testservlet
http://<server_name>:<port_number>/web2/testservlet

You should see something like this appear in the server window:



```
***UTIL1 --> EJB1 manifest.mf
***UTIL2 --> EJB2 manifest.mf
***UTIL3 --> EJB1&2 manifest.mf
***UTIL4 --> WebApp1&2 manifest.mf
***UTIL5 --> WebApp1&2 WEB-INF/lib
***UTIL6 --> WebApp1&EJB1 manifest.mf
***TestServlet1 --> WebApp1 --> /web/testservlet/
***TestServlet2 --> WebApp2 --> /web2/testservlet/
====================================
== TestServlet1 Class Information.
== Execute Depend1.test() --> Util1 & Util3
====================================
CLASSLOADER INFO:
TestServlet ==> weblogic.utils.classloaders.ChangeAwareClassLoader ==> 7481256
TestServlet ==> weblogic.utils.classloaders.GenericClassLoader ==> 7436915
TestServlet ==> sun.misc.Launcher$AppClassLoader ==> 7435051
TestServlet ==> sun.misc.Launcher$ExtClassLoader ==> 8379614
CLASSLOADER INFO:
test1.Depend1Bean_wi9aew_Impl ==> weblogic.utils.classloaders.GenericClassLoader ==> 7436915
test1.Depend1Bean_wi9aew_Impl ==> sun.misc.Launcher$AppClassLoader ==> 7435051
test1.Depend1Bean_wi9aew_Impl ==> sun.misc.Launcher$ExtClassLoader ==> 8379614
CLASSLOADER INFO:
common.Util1 ==> weblogic.utils.classloaders.GenericClassLoader ==> 7436915
common.Util1 ==> sun.misc.Launcher$AppClassLoader ==> 7435051
common.Util1 ==> sun.misc.Launcher$ExtClassLoader ==> 8379614
CLASSLOADER INFO:
common.Util3 ==> weblogic.utils.classloaders.GenericClassLoader ==> 7436915
common.Util3 ==> sun.misc.Launcher$AppClassLoader ==> 7435051
common.Util3 ==> sun.misc.Launcher$ExtClassLoader ==> 8379614
====================================
== Depend2.test() --> Util2 & Util3
====================================
CLASSLOADER INFO:
test2.Depend2Bean_wi9xef_Impl ==> weblogic.utils.classloaders.GenericClassLoader ==> 7436915
test2.Depend2Bean_wi9xef_Impl ==> sun.misc.Launcher$AppClassLoader ==> 7435051
test2.Depend2Bean_wi9xef_Impl ==> sun.misc.Launcher$ExtClassLoader ==> 8379614
CLASSLOADER INFO:
common.Util2 ==> weblogic.utils.classloaders.GenericClassLoader ==> 7436915
common.Util2 ==> sun.misc.Launcher$AppClassLoader ==> 7435051
common.Util2 ==> sun.misc.Launcher$ExtClassLoader ==> 8379614
CLASSLOADER INFO:
common.Util3 ==> weblogic.utils.classloaders.GenericClassLoader ==> 7436915
common.Util3 ==> sun.misc.Launcher$AppClassLoader ==> 7435051
common.Util3 ==> sun.misc.Launcher$ExtClassLoader ==> 8379614
====================================
== WebApp1 uses Util4
====================================
CLASSLOADER INFO:
common.Util4 ==> weblogic.utils.classloaders.ChangeAwareClassLoader ==> 7481256
common.Util4 ==> weblogic.utils.classloaders.GenericClassLoader ==> 7436915
common.Util4 ==> sun.misc.Launcher$AppClassLoader ==> 7435051
common.Util4 ==> sun.misc.Launcher$ExtClassLoader ==> 8379614
====================================
== WebApp1 uses Util5
====================================
```

The `application.xml` deployment descriptor is:

```
<?xml version="1.0"  encoding="UTF-8"?>

<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application
1.3//EN" "http://java.sun.com/dtd/application_1_3.dtd">

<application>
  <display-name>Pretty Name</display-name>
  <module>
    <ejb>Depend1-container.jar</ejb>
  </module>
  <module>
    <ejb>Depend2-container.jar</ejb>
  </module>
  <module>
    <web>
      <web-uri>WebApp1.war</web-uri>
      <context-root>web1</context-root>
    </web>
    </module>
    <module>
      <web>
        <web-uri>WebApp2.war</web-uri>
        <context-root>web2</context-root>
      </web>
    </module>
  </application>
```

The manifest classpath for the first EJB module is:

```
  Class-Path: Util1.jar Util3.jar Util6.jar Util7.jar
```

The manifest classpath entries for the other EJB module and the web application modules vary and have different combinations of the seven dependency libraries contained in the EAR file. The servlets contained within each web application are fully documented outlining the thread of execution and which test case is being demonstrated for each dependency library.

The Impact of Dependency Libraries

The manifest classpath will definitely spur better modularity of J2EE packages in the future. Using this model, developers can employ a simple scheme for determining which EJBs should be packaged into a single JAR file and those that should be packaged in separate JAR files:

- Identify an entity EJB that participates in a CMR relationship. Identify all entity EJBs that are reachable via CMR relationships from the source entity EJB. This graph of entity EJBs must be packaged into a single EJB JAR application. This process should be repeated for each unique graph of entity EJB relationships.
- Package all remaining EJBs into separate JAR files.
- Analyze your business and technical requirements and "converge" multiple JAR files if it makes sense. Convergence of multiple EJBs into a single JAR file should occur if a scenario where re-deploying all of the EJBs when only a single one has been modified is acceptable.
- Each EJB JAR file should list its dependencies using the manifest `Class-Path:` attribute as described above. An intelligent class loader will resolve any circular or repeating dependencies. For example, in the dependency example provided, both EJB applications referenced the third library as a dependency. Despite this double referencing, the EAR class loader was still intelligent enough to load the library only once.

The only limitation that developers are now faced with is locating an application server that fully supports this packaging capability. Developers will need to look for application servers that run in a 1.3 JDK and fully support the J2EE 1.3 specification. This has the consequence that any J2EE 1.3-certified application server must only run in a 1.3 JRE environment.

Summary

This chapter covered J2EE applications in depth, including:

- How the J2EE packaging mechanism is ideal for providing better interoperability between multiple J2EE modules
- J2EE applications, modules, and EAR files
- The process of J2EE application development
- Application assembly and deployment
- An example application

We've now seen the entire range of J2EE technologies, from the basic building blocks of RMI, JDBC, JNDI, and JAXP, through the servlet and JSP web content APIs and Enterprise JavaBeans, to complete J2EE applications. We hope this book has inspired you to get stuck into creating your own J2EE applications – enjoy!