23

J2EE Packaging, Enhanced EARs, and the Application Server Toolkit

As you learned about in Chapter 1, WebSphere Application Server V6 supports the full Java 2 Platform, Enterprise Edition (J2EE) 1.4 programming model. The J2EE specification consists of several functional subspecifications. However, it isn't always obvious how you should put together the different elements to form a complete J2EE application. In this chapter, we take a closer look at J2EE packaging and explain the role IBM Enhanced EAR files and the WebSphere Application Server Toolkit (AST) play in creating a J2EE application.

J2EE Packaging at a Glance

The J2EE specification provides guidelines for the structuring and creation of J2EE applications, and one of the major ones relates to *packaging*. Individual specifications provide guidelines for the packaging of individual components, such as Enterprise JavaBeans (EJBs), Java Server Pages (JSPs), and servlets. The J2EE specification then dictates how these heterogeneous components are themselves to be packaged together.

This section provides an analysis of the J2EE packaging mechanism, focusing on the relationships these components have within an *Enterprise Application Archive (EAR)* file and the process involved in building EAR files. Some of the questions we'll ask are

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

As we answer these questions, you'll learn

- how J2EE class loading schemes work
- how to create 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, you must create a *J2EE application*. When building a J2EE application, you must take into account many considerations, including

- the types of J2EE components you can package into a J2EE application
- the roles people play when creating J2EE packages
- the current limitations of J2EE packaging
- the class loading approaches 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 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 runtime containers and deployment modules. *Runtime 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 runtime container. Recall how J2EE containers are structured:

- *EJB container* — The EJB container provides containment and request-level interception for business logic. The EJB container lets EJBs access Java Message Service (JMS), Java Authentication and Authorization Service (JAAS), the Java Transaction API (JTA), JavaMail (which uses JavaBeans Activation

Framework, or JAF), the Java API for XML Processing (JAXP), Java Database Connectivity (JDBC), and the Connector architecture.

- *Web container* — The Web container provides interception for requests sent over HTTP, File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP), and other protocols. Most Web containers support only HTTP (and HTTPS) but could support a broader range of protocols. The Web application container lets JSPs and servlets have access to the same resources the EJB container provides.
- *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 the EJB container operate.

  A program running in an application client container is similar to a Java program with a main() method. However, instead of a JVM controlling the application, a wrapper controls the program. This wrapper is the application client container. Application client containers are a new concept in the J2EE specification, your application server provider should provide them.

  An application client container can optimize access to a Web container and an EJB container by providing direct authentication, performing load balancing, allowing failover routines, providing access to server-side environment variables, and properly propagating transaction contexts. Programs that run within an application client container have access to JAAS, JAXP, JDBC, and JMS resources on a remote application server.
- *Applet container* — An applet container is a special type of container that provides request-level interception for Java programs running in a browser. An important point to remember is that an applet container doesn't provide access to any additional resources, such as JDBC or JMS.

  Applets running within an applet container are expected to request resources directly from 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 regulate how an applet should communicate with an EJB container, but the J2EE specification does. J2EE requires applets that want to directly use an EJB to use the HTTP(S) protocol and tunnel Remote Method Invocation (RMI) invocations. Many application server vendors support a form of HTTP tunneling to support this functionality.

The components you can package into a J2EE EAR file don't directly correlate to the components that contain containers. There are no basic requirements for what an EAR file must minimally include. An EAR file consists 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. Because 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's intended to run within an application client container. The application client JAR file contains a specialized deployment descriptor and is composed similarly to an EJB JAR file. The JAR file also contains the classes required to run the standalone client as well as any client libraries needed to access JAAS, JAXP, JDBC, JMS, 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 don't execute within a container; rather, they're designed to execute as a bridge between an application server and an external enterprise information system.

Each of these components is developed and packaged individually apart from the EAR file and its own deployment descriptor. A J2EE EAR file combines 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 individuals assume during the development and deployment process, these roles are nothing more than logical constructs that let you better plan and execute an application. It's likely (and expected) that a single individual or organization will perform multiple roles.

The common roles involved in building, deploying, and using an EAR file include the following:

- *J2EE product provider* — The J2EE product provider supplies 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. Many roles within the J2EE specification can also be characterized as application component providers, including document developers, JSP authors, enterprise bean developers, and resource adapter developers.
- *Application assembler* — The application assembler is responsible for combining one or more J2EE components into an EAR file to create a J2EE application. This person is also responsible for creating the J2EE application deployment descriptor and identifying any external resources (e.g., class libraries, security roles, naming environments) on which the application may depend. The application assembler will commonly use tools provided by the J2EE product provider and the tool provider (described next).
- *Tool provider* — A tool provider furnishes 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 supplied by a tool provider can be either platform-independent (i.e., work with all EAR files irrespective of the environment) or 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, producing container-ready Web applications, EJB applications, applets, and application clients that have been customized for the target environment of the application server.

  The deployer isn't responsible for deploying a resource adapter archive or an application client archive but may be responsible for additional configuration of these components. These components, although packaged as part of a J2EE EAR file, aren't considered when the enterprise application is deployed. They're part of the J2EE application but don't group through the runtime "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 packaged as part of a J2EE EAR file, they don't operate within the context of a J2EE container. Therefore, because resource adapter archives don't have a J2EE container, they don't need to have a J2EE deployer involved with their activation.

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

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

In this chapter, when we discuss the creation of EAR files and the resolution of conflicts, we'll be acting in the roles of application assembler and deployer.

The Limitations of Packaging

EAR files meet the basic requirements for packaging an application because most Web-based J2EE applications consist solely of Web and EJB applications. However, EAR files lack the capability to package complicated J2EE applications. For example, you can't declare the following components in an EAR file, but they are often used in J2EE applications:

- JDBC DataSource objects
- JMS ConnectionFactory and Destination objects
- Java Management Extensions (JMX) MBeans
- some JMS consumers that run within an application server, such as a MessageConsumer, which runs as part of a ServerSession
- classes triggered when an application is deployed or undeployed (these classes are vendor-provided proprietary extensions not defined in the J2EE specification; however, all vendors generally supply them)

At present, these components must be manually configured and deployed via an administration interface provided by the implementation vendor and are the system administrator's responsibility. The use of these items will increase over time, and it will be important for EAR files to support the packaging of these components to enable application portability. Starting with WebSphere V6, you can include these components in an EAR. This form of EAR file is called an *Enhanced EAR*.

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 to load classes into memory. A class loader is a Java class that's responsible for loading Java classes from a source. Java classes can be loaded from disk 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 (i.e., can use) classes loaded by any of the parent class loaders. Classes loaded by a parent class loader don't have visibility to classes loaded by any of the parent's children's class loaders. Class loaders and EAR files are important because 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 class loaders to resolve these dependencies.

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 won't be visible to the classes loaded by other class loaders. This circumstance 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 aren't just a convenient packaging format; they also provide a special class loading scheme that lets applications within the EAR file access the classes of other applications.

The J2EE 1.3 specification makes no specific requirements as to how an EAR class loader should work, giving application server vendors the flexibility to determine how to load classes. Before implementing an EAR class loader, a vendor must decide the following questions:

- 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 on 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 the JAR file is visible to both applications?
- 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 don't. So, will EJB applications be loaded differently from Web applications so that Web application integrity can be maintained?

Class Loading Starting with EJB 2.0

The EJB 2.0 Public Final Draft 2 specification introduced the concept of local interfaces and placed an interesting twist on the EAR class loading problem. Local interfaces let colocated clients and EJBs 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 before EJB 2.0 won't 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.

In this model, Web application class loaders are children of the EJB class loader. This arrangement enables 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 because it doesn't require the EJB class loader to export any files to the EAR class loader.

An Ambiguity in the J2EE Specification

Certain implementations have exposed an ambiguity in the J2EE specification. The ambiguity arises because the J2EE specification is unclear about how dependency libraries of a Web application should be loaded. It's 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, the specification doesn't state whether a utility library specified as a dependency library of the Web application should be loaded by the Web application's class loader or exported to the EAR class loader. This distinction can have a behavioral impact. If it's known that a dependency utility library will be loaded only once for all Web applications, the Web applications can take advantage of knowing that a singleton class will create one object that all the Web applications can share. But if each Web application's class loader isolated the utility library, a singleton class (which is a class intended to create only a single instance in the virtual machine) will create one object in each Web application.

At present, WebSphere loads any utility library specified as a dependency library of a Web application at the EAR class loader level. This approach makes sense because you can always achieve Web application isolation by placing utility libraries in WEB-INF\lib. This solution 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 you have a basic grasp of how the J2EE architecture is implemented — specifically, of the different roles and the behavior of class loaders — you're ready to configure and deploy enterprise applications. To do so, you need to understand the process of EAR file creation and the contents of the deployment descriptors that describe the EAR file contents.

The Enterprise Application Development Process

The overall process used to build an enterprise application is as follows:

1. Developers build individual components. These components can be EJBs, JSP pages, servlets, and resource adapters.
2. Some number of components is 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 contain more than one EJB, a Web application module can consist of multiple JSP pages and servlets, and a resource adapter archive can consist of 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. Multiple J2EE modules make up more complicated J2EE applications. A complex J2EE application consists of 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. You install the application on the J2EE platform and then integrate it 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 file with a my.jar and a my.war contained within it, when you deploy the application, 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 module is deployed successfully, the J2EE application is considered to have been deployed successfully.

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. You can optionally 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
```

Issues with Dependency Packages

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

In the first approach, you can place dependency libraries that are packaged as JAR files in the WEB-INF\lib directory of a Web application. In general, you should use WEB-INF\lib primarily for the storage of servlet classes, but servlets and JSP pages will look for classes in this directory when loading new ones. If only your servlets and JSP pages need the utility libraries you're using, this solution will be sufficient. However, if EJBs, JMS consumers, or startup and shutdown classes also need the same libraries, this option won't work because the WEB-INF\lib directory isn't visible to these items.

In the second approach, you place a complete copy of all the utility libraries in each EJB JAR file as well as in the WEB-INF\lib directory. When you deploy an EJB, an EJB class loader will look only within its own JAR file for any utility classes that are referenced. It won't look in the JAR files of other EJB applications that have been deployed or in WEB-INF\lib. If all your EJB applications require the use of the same library, placing a copy of that library's classes in each JAR file will meet your needs. The utility classes will be redeployable along with the EJB.

Although the second scenario achieves redeployability of dependency libraries, it is incredibly inefficient. The purpose of having multiple JAR files for packaging is to promote application modularity, and placing the same class in multiple JAR files destroys this benefit. In addition, having multiple copies of the same classes unnecessarily bloats your applications. Last, the build process requires an extra step because you need to rebuild every JAR file if you want to change even a single library.

With the release of Java Development Kit (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 on other JAR files, enabling an application to consist of multiple modules.
- Class loaders are modified to search optional packages and application paths for classes.

In addition, the J2EE 1.3 specification mandates that application servers support the extension mechanism as defined for JAR files. This requires any deployment tool that references a JAR file to be capable of loading any optional libraries defined through the extension mechanism. It also implies that if an application server or deployment tool supports runtime undeployment and redeployment of EJB applications that use libraries via the extension mechanism, that tool or application server must also support undeployment and redeployment of any dependent libraries.

Support for the extension mechanism doesn't exist for EAR or resource adapter applications as defined in the J2EE specification because these applications aren't directly loaded by an instance of ClassLoader. Web applications have the freedom to use the extension mechanism or the WEB-INF\lib directory when specifying a dependency library. As we discussed earlier, the way a dependency library is loaded can vary depending on whether you specify the library using the extension mechanism or WEB-INF\lib.

Enterprise applications need to repackage any libraries required by the Web application or EJB application as part of the EAR file. After this repackaging, the extension mechanism provides a standard way for Web application WAR files and EJB application JAR files to specify which dependency libraries in the enterprise application EAR file they need.

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 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. You can edit this file to include a Class-Path attribute entry in addition to the other entries that already exist in the 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). You can specify multiple URLs 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
```

Inside a Sample EAR File

WebSphere Application Server provides an example of a packaged J2EE application named WebSphere Bank, delivered as a sample in the WebSphere SamplesGallery. Like every J2EE application, the WebSphere Bank sample is packaged in an EAR. Go ahead and open it with a utility such as WinZip. You'll see that the EAR contains these J2EE modules:

Web modules:
BankCMRQLWeb.war, DepositJCAWeb.war, BankGallery.war

EJB modules:
BankCMRQLEJB.jar

Connector modules:
BankRA.rar

Dependent JAR modules:
BankAdapterInterface.jar, WsaEJBDeployUtility.jar

Application client JAR modules:
TransferJMSClient.jar, GetAccounts.jar, FindAccounts.jar

Remember that each module contains deployment descriptor files you can locate and view. Another thing to note is that EARs configured for WebSphere can have IBM proprietary deployment descriptor files. These files are always named ibm-application-bnd.xmi and ibm-application-ext.xmi. These files, when present, contain proprietary parameters for the WebSphere Application Server platform.

If you're viewing the contents of WebSphereBank.ear, you'll notice many files packaged in a directory called Database. These files contain the Cloudscape files to support the WebSphere Bank sample. This approach is a common practice when packaging an EAR. The technique lets any files be packaged with the EAR and be available relative to where the EAR is deployed.

Configure an Enhanced EAR

One oversight, some might say, of J2EE packaging is that there is no standard way to package all the parameters for all the J2EE resources a J2EE application might require. If there were, these parameters could be used to automatically create the resources when you deploy the application. Instead, you must script this step (or perform it manually), and the process differs for each J2EE environment (e.g., WebSphere, WebLogic).

IBM has tried to address this issue with the advent of Enhanced EARs. Using the WebSphere Application Server Toolkit, you can enhance your EAR with files that will be used to create the resources for the EAR when you deploy it.

Within the AST, you use the WebSphere Enhanced EAR editor to edit server configurations for WebSphere Application Server V6. The server configuration data you specify in this editor is embedded within the application itself. By preserving the existing server configuration, this technique improves the administration process of publishing to WebSphere when you install a new application to an existing local or remote WebSphere server.

As of Version 6, server-specific configurations for WebSphere Application Server are set in the WebSphere administrative console. You can use the Enhanced EAR editor to set the configuration settings specific to an enterprise application. The Enhanced EAR editor is available on the deployment page of the AST's Application Deployment Descriptor editor. Use this editor to configure the following elements that are specific to an enterprise application:

- data sources
- resource adapters and connection factories
- substitution variables
- authentications
- shared libraries
- virtual hosts
- class loader policies

When you enhance an EAR with the above configurations, several XML files are added to your EAR directory. Table 23-1 lists these files.

Table 23-1: XML files added to the EAR directory

File name	EAR directory
deployment.xml	<AST_workspace>\<EAR_PROJECT>\META-INF\ibmconfig\cells\defaultCell\applications\defaultApp\deployments\defaultApp
resources.xml	<AST_workspace>\<EAR_PROJECT>\META-INF\ibmconfig\cells\defaultCell\applications\defaultApp\deployments\defaultApp
variables.xml	<AST_workspace>\<EAR_PROJECT>\META-INF\ibmconfig\cells\defaultCell\applications\defaultApp\deployments\defaultApp
libraries.xml	<AST_workspace>\<EAR_PROJECT>\META-INF\ibmconfig\cells\defaultCell\nodes\defaultNode\servers\defaultServer
security.xml	<AST_workspace>\<EAR_PROJECT>\\META-INF\ibmconfig\cells\defaultCell
virtualhosts.xml	<AST_workspace>\<EAR_PROJECT>\\META-INF\ibmconfig\cells\defaultCell

To illustrate how to locate the WebSphere Enhanced EAR editor, let's test a sample data source:

1. In the Application Server Toolkit, switch to the J2EE perspective.
2. In the Project Explorer view, expand the **Enterprise Applications** folder.

3. Under the enterprise application project folder for which you want to test the data source, double-click **Deployment Descriptor** to open the Application Deployment Descriptor editor. (We're doing this to get to the Enhanced EAR editor.)
4. At the bottom of the editor window, select the **Deployment** tab to open the WebSphere Enhanced EAR editor.

Note: Before adding or removing J2EE modules using the Application Deployment Descriptor editor's **Module** page, you must first click the **Deployment** tab to activate the functions in the **Deployment** page. Then add or remove your modules from the **Module** page. You must complete this task for each Application Deployment Descriptor editor session that you want to have add or remove modules from the **Module** page.
If the WebSphere Enhanced EAR editor is opened and you make changes to its dependent files either on the file system or using another editor, the changes aren't reloaded on the **Deployment** page. To refresh the changes on this page, you must close and reopen the Enhanced EAR editor.

Let's step through the process of configuring an Enhanced EAR with the configuration information for a new data source for the WebSphere Bank sample application. To begin, you must start the AST (you can do so from the Windows **Start** menu). Then import the EAR you want to enhance:

1. Select **File|Import**.

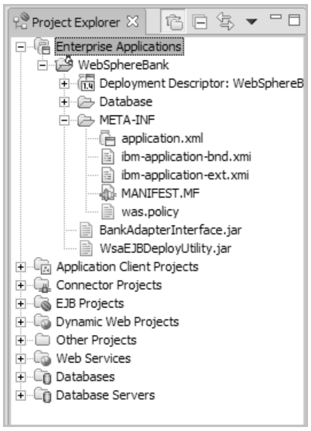
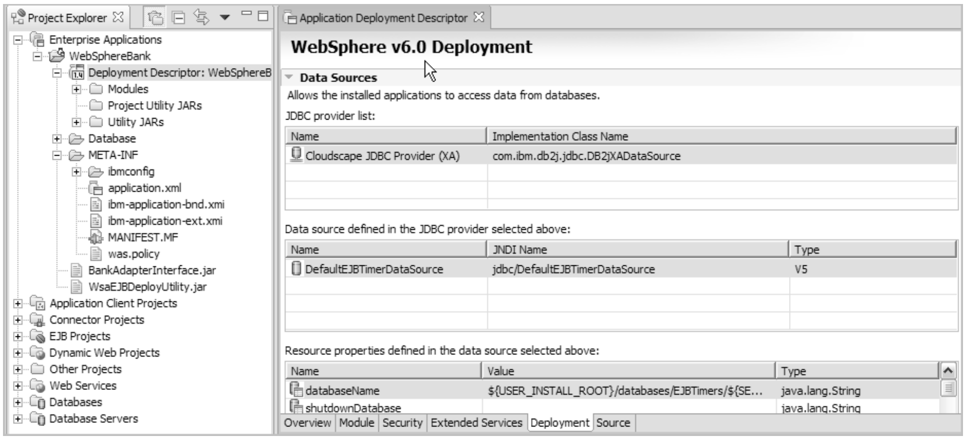


Figure 23-1: WebSphereBank imported into an AST workspace

2. Select **EAR File**, and click **Next**.
3. Browse to the WebSphere Bank EAR file (<WASV6-ROOT>\samples\lib\WebSphereBank\WebSphereBank.ear), and click **Finish**.



```
Enterprise Applications
  WebSphereBank
    Deployment Descriptor: WebSphereB
      Modules
      Project Utility JARs
      Utility JARs
    Database
    META-INF
      ibmconfig
      application.xml
      ibm-application-bnd.xmi
      ibm-application-ext.xmi
      MANIFEST.MF
      was.policy
    BankAdapterInterface.jar
    WsaEJBDeployUtility.jar
Application Client Projects
Connector Projects
EJB Projects
Dynamic Web Projects
Other Projects
Web Services
Databases
Database Servers
```

WebSphere v6.0 Deployment

Data Sources

Allows the installed applications to access data from databases.

JDBC provider list:

Name	Implementation Class Name
Cloudscape JDBC Provider (XA)	com.ibm.db2j.jdbc.DB2jXADataSource

Data source defined in the JDBC provider selected above:

Name	JNDI Name	Type
DefaultEJBTimerDataSource	jdbc/DefaultEJBTimerDataSource	V5

Resource properties defined in the data source selected above:

Name	Value	Type
databaseName	${USER_INSTALL_ROOT}/databases/EJBTimers/${SE...	java.lang.String
shutdownDatabase		java.lang.String

Figure 23-2: WebSphereBank EAR's deployment descriptor opened in editor

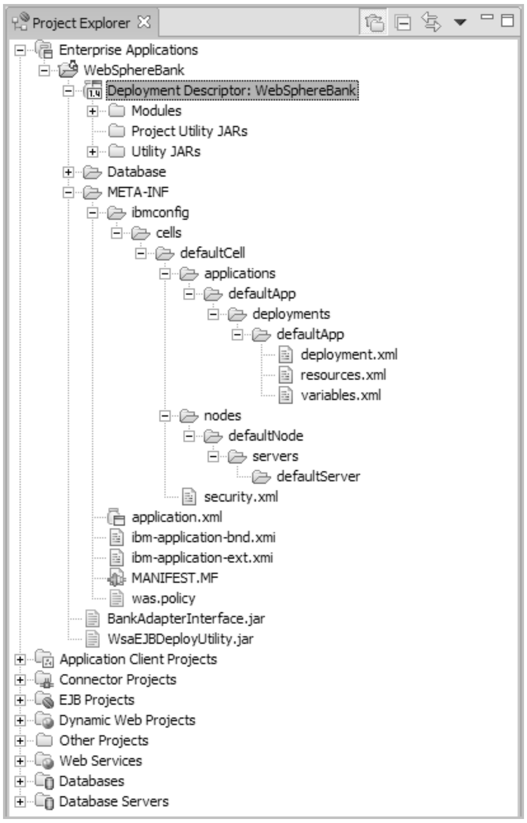
After you import the EAR, your Project Explorer should look as shown in Figure 23-1.

Now, double-click the WebSphere Bank EAR's deployment descriptor to open up the Application Deployment Descriptor editor. Then go to the **Deployment** tab. This is where you'll access the Enhanced EAR editor. Figure 23-2 shows the WebSphereBank EAR's deployment descriptor opened in the editor.

As a test, go ahead and add a new data source:

1. To the right of the panel labeled "Data source defined in the JDBC provider selected above," click the **Add** button (note that this button doesn't appear in the figure).
2. Select **Cloudscape JDBC Provider (XA)**, and then click **Finish**, accepting all defaults.

After adding enhanced parameters (as you just did), new enhanced EAR files will appear in your Project Explorer; look for deployment.xml, resources.xml, variables.xml, and security.xml. Depending on what you added (or didn't add), some files may not be created. Figure 23-3 shows sample results for our example.



```
Project Explorer
Enterprise Applications
  WebSphereBank
    Deployment Descriptor: WebSphereBank
      Modules
      Project Utility JARs
      Utility JARs
    Database
    META-INF
      ibmconfig
        cells
          defaultCell
            applications
              defaultApp
                deployments
                  defaultApp
                    deployment.xml
                    resources.xml
                    variables.xml
            nodes
              defaultNode
                servers
                  defaultServer
            security.xml
      application.xml
      ibm-application-bnd.xmi
      ibm-application-ext.xmi
      MANIFEST.MF
      was.policy
    BankAdapterInterface.jar
    WsaEJBDeployUtility.jar
Application Client Projects
Connector Projects
EJB Projects
Dynamic Web Projects
Other Projects
Web Services
Databases
Database Servers
```

Figure 23-3: Enhanced EAR's added files

Once you've finished with the AST, you can export your EAR and deploy it as normal. During the deployment, the enhanced EAR files will be detected, and the resources will be created.

24

Manually Install WebSphere Bank

In Chapter 4, you saw how to use a WebSphere –samples script from the command line to install the WebSphere SamplesGallery application, one element of which is the WebSphere Bank sample you learned about in Chapter 24. What you didn't see are all the resources that were created for you at that time. In this chapter, we review the resources created for the WebSphere Bank application and explain how you can create them yourself using the WebSphere administrative console.

Installing resources manually is a common requirement to successfully deploy any J2EE application because the J2EE specification doesn't define how to package this information in the Enterprise Archive (EAR) file. As you learned in Chapter 23, IBM has solved this issue in a proprietary way with Enhanced EARs. In that chapter, you used the WebSphere Application Server Toolkit (AST) to create an enhanced WebSphere Bank EAR. In most cases, however, the J2EE developer/packager will tell you which resources the application requires, and, as the system administrator, you'll need to create these resources using either WebSphere's admin console or the wsadmin command-line tool.

As it turns out, WebSphere Bank needs the following resources to run:

- a J2C resource adapter called *WebSphere Relational Resource Adapter* that, luckily, is created automatically when you install WebSphere
- a J2C authentication alias named *IBM-79D6XZF0P9FNode01Cell/samples* (note that the first part of this name contains the name of the node/cell where the alias is created and so will vary from node to node)
- a JDBC provider named *Samples Cloudscape JDBC Provider (XA)*

- a data source named *BANKDS*
- a connection factory named *BANKDS_CF* (which, as you'll see, is also created automatically)
- a service integration bus (SIBus) named *IBM-79D6XZF0P9FNode01SamplesBus* (again, the first part of the name is the node name where the bus is created and so will vary from node to node)
- an SIBus member for the SIBus named *IBM-79D6XZF0P9FNode01SamplesBus*
- an SIB Java Message Service (JMS) connection factory named *BankJMSConnFactory*
- an SIB JMS queue named *BankJMSQueue*
- an SIB queue named *BankJSQueue*
- an SIB JMS activation specification named *BankActivationSpec*

Last, but not least, you'll need to enable the SIB service for the application server (server1). Let's walk through the steps to set up each required resource.

Verify the Existence of the J2C Resource Adapter

First, take the following steps to make sure the required J2C resource adapter, WebSphere Relational Resource Adapter, has been created.

1. In the WebSphere admin console's navigation tree, expand **Resources**, and click **Resource Adapters**.
2. Verify that WebSphere Relational Resource Adapter appears in the list of installed resource adapters.

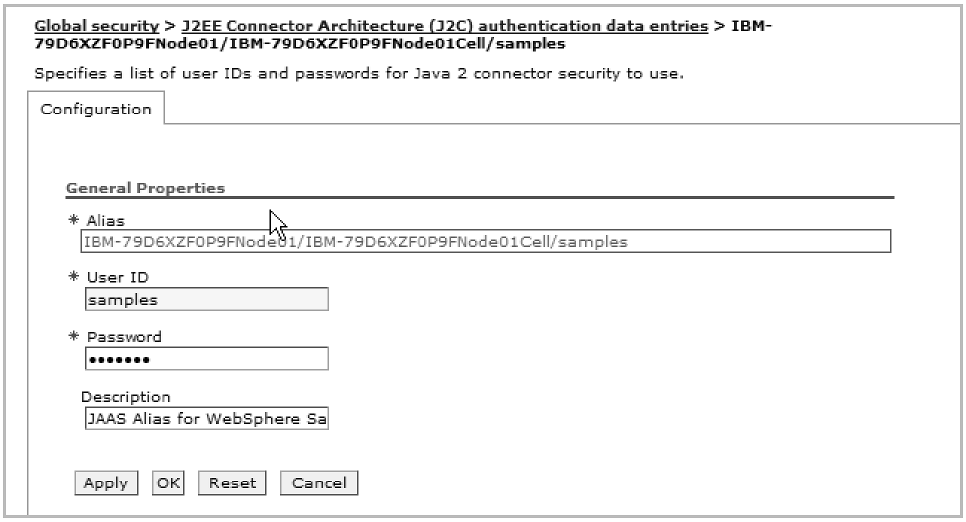
Create the J2C Authentication Alias

Next, create the required J2C authentication alias, IBM-79D6XZF0P9FNode01Cell/samples:

1. In the console, navigate to **Security|Global Security** to display the **Global Security** panel.
2. Under the **Authentication** heading, select **JAAS configuration|J2C authentication data** to display the **J2C Authentication Data Entries** panel.
3. Click **New**.

4. On the resulting configuration panel (shown in Figure 24-1), enter the required values: a unique alias name (IBM-79D6XZF0P9FNode01Cell/samples), a valid user ID (samples), a valid password (samples), and a short description (JAAS Alias for WebSphere Samples).

5. Click **Apply** or **OK**.



Global security > J2EE Connector Architecture (J2C) authentication data entries > IBM-79D6XZF0P9FNode01/IBM-79D6XZF0P9FNode01Cell/samples

Specifies a list of user IDs and passwords for Java 2 connector security to use.

Configuration

General Properties

* Alias
IBM-79D6XZF0P9FNode01/IBM-79D6XZF0P9FNode01Cell/samples

* User ID
samples

* Password
•••••••

Description
JAAS Alias for WebSphere Sa

Figure 24-1: Creating a J2C authentication alias

Create the JDBC Provider

Next, create the JDBC provider resource, Samples Cloudscape JDBC Provider (XA):

1. Navigate to **Resources|JDBC Providers**.
2. On the resulting panel, make sure the scope is set to Server.
3. Click **New**.
4. Enter the values as shown in Figure 24-2 to create the JDBC provider.