Chapter

22

SECTION V: ENTERPRISE JAVA BEANS

Introduction To Enterprise Java Beans

Previously an enterprise application used to be large and monolithic. Developing, maintaining and enhancing the functionality of individual sections of such large monolithic applications used to be a difficult and time-consuming task.

Today enterprise applications are in the form of a distributed system, where the whole system is broken into layers. Each layer is independent and serves a distinct purpose. Each layer is further made up of one or more logical components. Thus, component based programming approach towards developing an enterprise application evolved as a trend in software engineering industry.

This approach helps the developer in:

- Managing the complexity of software projects in a more easier way
- Having great discipline to the software development process
- Cutting the development time drastically

A component is simply a pre-built piece of programming code designed to perform a specific function. The component lives in a container provided by the component based architecture. The crucial system level functionalities are provided by the container itself and leaving the component just to serve the clients by executing the business logic.

The important difference between components and classes is that component based architectures aim to minimize dependencies between components, hence making components more reusable than classes. Components can be successfully reused when they have packaging, integration, legacy support and robustness. Thus, with the concept of components, larger chunks of functionalities can be reusable.

Enterprise Bean Architecture

The EJB Architecture is composed of:

- An Enterprise Bean **Server**
- Enterprise Bean **Containers** that runs on these servers
- **Enterprise Beans** that run in these containers
- Enterprise Bean **Clients**
- Other systems such as the Java Naming and Directory Interface [JNDI] and the Java Transaction Service [JTS]

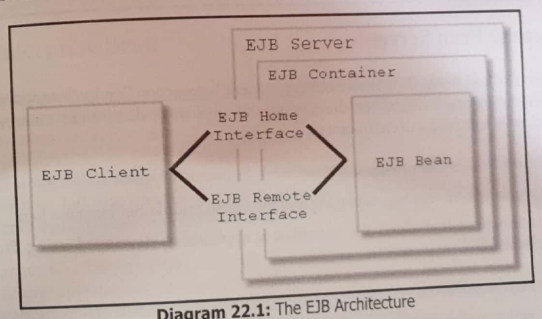

Diagram 22.1: The EJB Architecture

Scanned by CamScanner

REMINDER

Since an EJB client does not communicate directly with an EJB component, proxy objects such as home and remote interfaces that implement the components are used. The component's remote interface defines the business methods that can be called by the client. The client calls the home interface methods to create and destroy proxies for the remote interface.

How Does The Communication Takes Place

- Client object makes a request for a method on the bean
- Container comes into picture and checks whether the client is in the approved list for calling a method on the bean
- If the client is authorized, container either creates new instance or activates the requested bean from the pool
- Container ensures the bean gets its own new transaction
- Container informs the EJB object [wrapper class generated by container] that the bean is ready and passes the client's method request to the bean

HINT

Nobody can talk to the bean except the container.

Enterprise Bean Server

An Enterprise Java Beans [EJB] Server is a Component Transaction Server. It supports the EJB server side component model for developing and deploying distributed enterprise-level applications in a multi-tiered environment.

An Enterprise Bean Server provides:

- The framework for creating, deploying and managing middle-tier business logic
- An environment that allows the execution of applications developed using Enterprise Java Beans [EJB] components

In a three-tier environment:

- The client provides the **User Interface** logic
- The **Business Rules** are separated to the middle tier
- The **Database** is the information repository

The client does not access the database directly. Instead, the client makes a call to the EJB Server on the middle tier, which then accesses the database.

An EJB server takes care of:

- Managing and coordinating the allocation of resources to the applications
- Security
- Threads
- Connection pooling
- Access to a distributed transaction management service

The EJB server provides one or more containers for the enterprise beans, which is called an EJB container.

An EJB **container** manages the **enterprise beans** contained within it.

Enterprise Beans [or components] are reusable modules of code that combine related tasks [methods] into a well-defined interface. These enterprise beans EJB components contain the methods that execute business logic and access data sources.

These components [i.e. the executable code] are installed on an EJB Server. Any number of independent Java or EJB applications [clients] can use the EJBs.

An Enterprise Bean

An **Enterprise Bean** [EJB] is a **server-side component** that encapsulates the code that fulfills the purpose of the application. They can be combined with other components and rapidly produce a custom application.

An EJB is not a GUI component instead it sits behind the GUI and performs all the business logic such as database operations. GUIs such as rich web-based clients and web services are the clients that can make connections to an EJB.

Enterprise Java Beans [EJB] simplify the process of building enterprise, distributed component applications in Java. It defines an agreement between Components and Application Servers that enables any component to run in any complaint application server. An application written in EJB is scalable, transactional, reliable and secure.

The main purpose of introducing EJB was for building distributed components. EJB was introduced with a promise to solve all issues and complexities of CORBA.

Scanned by CamScanner

Over the past few years, EJB specification has evolved significantly. In the early days of EJB, application developers faced a burden of overwhelming complexity i.e. they had to manage several component interfaces, deployment descriptors and unnecessary callback methods and learn and to implement the design patterns used to overcome the limitations of the specification.

Enterprise Java Bean architecture defines a model for the development and deployment of reusable Java server components. The EJB component architecture is the backbone of the Java EE 5 platform.

Business components developed using the EJB architecture are called as **Enterprise Java Beans components** or simply **Enterprise Beans**.

The EJB technology is powerful and sophisticated. The technology helps developers to build business applications that support very large number of users simultaneously. The applications developed with EJB are capable of maintaining data integrity even though its data is processed concurrently by multiple users, thus making them transaction enabled.

The core of a Java EE 5 application is comprised of one or several enterprise beans that perform the application's business operations and hides the business logic of an application.

The EJB components can be assembled and reassembled into different distributed applications as per the requirement. One of the goals of Enterprise Java Beans architecture, EJB 3.0, is to make writing distributed component based applications easy.

For example: A Customer bean can be assembled in an accounting program, an e-commerce shopping cart system or virtually into any other application that needs representing a customer.

An Order bean, a Payroll bean, a Shopping-Cart bean, a Credit Card Validator bean are some common components, a bean developer builds.

Enterprise Bean Containers

An EJB container is basically a software implementing the EJB specification. As the name suggests, it provides an environment within which EJB component lives and breathes. It does everything on behalf of the enterprise bean by stepping in each time a client makes a request for a method on the bean.

For each enterprise bean, the container is responsible for:

- Registering the object

- Providing a remote interface for the object
- Creating and destroying object instances
- Checking security for the object
- Managing the active state for the object
- Coordinating distributed transactions

Optionally, the container can also manage all persistent data within the object.

Following is what a container does:

- Allocates resources such as threads, sockets and database connections on behalf of an enterprise bean
- Manages the life and death of enterprise beans. For example:
  - Creates the bean instances
  - Destroys them
  - Serializes them to a secondary storage
  - Activates them by reading their serialized state from the secondary stage
- Converts the network less beans into distributed, network-aware objects in order to service the remote clients
- Serializes the requests when multiple clients call the methods on a bean's instance, thereby allowing only one client to call the bean instance at a time without making the bean provider write multithreading code to handle them
- Saves the conversational state of an object between two method calls
- Gives security clearance to call a method on the bean
- Manages the start, enrollment, commitment and rollback processes of each transaction of the enterprise bean
- Maintains persistent data by synchronizing the data updates with beans and databases
- Helps registering and deploying components

Enterprise Bean Clients

An EJB client is a stand-alone application that provides the User Interface logic on a client machine.

Scanned by CamScanner

The EJB client makes calls to remote EJB components on a Server. The EJB client needs to be informed about:

- How to find the EJB server
- How to interact with the EJB components

HINT

An EJB component can act as an EJB client by calling methods in another EJB component.

An EJB client does not communicate directly with an EJB component. The container provides proxy objects that implement the components home and remote interfaces. The component's remote interface defines the business methods that can be called by the client. The client calls the home interface methods to create and destroy proxies for the remote interface.

An EJB client such as a **Java Servlets** or **Java Server Pages** residing on a Web Server can communicate with Enterprise bean for business logic and generating dynamic web pages.

HINT

An EJB client could be **remote** or **local**.

A remote client runs on a different machine and a different **J**ava **V**irtual **M**achine [JVM] than the enterprise bean it accesses. The location of the enterprise bean is transparent to a remote client.

A **local client** runs in the same JVM as the enterprise bean it accesses.

Features / Benefits Of Enterprise Bean

EJB provides the following features:

Complete Focus Only On Business Logic

Enterprise Beans live and run in the server under the control of an EJB container.

The EJB container provides all the big infrastructure services such as Security, Concurrency, Transaction management, Networking, Resource management, Persistence, Massaging and Customization during deployment.

Developer can use these services with minimal effort and time, thus making writing an enterprise bean as simple as writing a Java class.

EJB model separates system level services from the business logic. This allows the server vendor to concentrate on system level functionalities, while the developer can concentrate more on only the business logic for the domain specific application. Developer need not code for these hardcore services. The results are that applications get developed more quickly and the code is of better quality.

Reusable Components

Each enterprise bean is a reusable building block. An enterprise bean can be reused by assembling them in several different applications. For each application, making simple changes in deployment descriptor with out changing the source code can customize its behavior with the underlying services.

For example: The security access for a bean can be controlled by declaring in XML document at deploy-time instead of doing it programmatically.

In new EJB 3.0 version the bean behavior with the underlying system of the container can be written in bean class itself by annotation.

Other applications can reuse the deployed enterprise bean, by making calls to its client-view interfaces. Multiple applications can make calls to the deployed bean.

Portable

Enterprise Java Beans basically use Java language, which is portable across multiple platforms. With EJB's **write-once-deploy-anywhere** concept, the developers and customers are free from being at the mercy of application server vendor.

These components can run on any platform and are completely portable across any vendor's EJB-compliant application server. The EJB environment automatically maps the component to the underlying vendor-specific infrastructure services.

Fast Building Of Applications

The EJB architecture simplifies building complex enterprise applications. Customized EJB components can be assembled from a set off-the-shelf business objects to the desired working system.

Scanned by CamScanner

With the component-based EJB architecture, the development, enhancing the functionalities and maintenance of complex enterprise applications becomes easier.

With its clear definition of roles and well-defined interfaces, the EJB architecture promotes and supports team-based development and lessens the demands on individual developers.

One Business Logic Having Many Presentation Logics

An enterprise bean typically performs a business process or represents a business entity and it is independent of the presentation logic. The EJB model allows the business programmer to concentrate on business logic while Web page designer concentrates on formatting the output.

In addition, this separation makes it possible to develop multiple presentation logic for the same business process or to change the presentation logic of a business process without needing to modify the code that implements the business process.

Distributed Deployment

The Enterprise beans, the business components of distributed system are deployed across multiple servers on a network. The client on the same machine as enterprise bean or client located on different system on the network can call the methods on the bean. The EJB architecture provides infrastructure where one bean can communicate with another bean object or client situated on different machines working in different environment successfully.

The bean developer does not have to be aware of the deployment procedure when developing enterprise beans. The same code is written whether the client of an enterprise bean is on the same machine or a different one.

Application Interoperability

EJB architecture is mapped to standards followed by CORBA, which is an industrial standard. Hence, it's relatively simple to craft an EJB and make it work with components developed in a different language like VC++ and so on using CORBA. The EJB's, client-view interface serves as a well-defined integration point between components built using different programming languages.

Customized Deployment

Each customer works in its own unique operational environment. The component typically needs to be customized at deployment time to work in such operational environment and sometimes component needs to represent different database schemas. This is possible only because the EJB architecture provides the following features such as:

- The EJB model permits the developer to separate common application business logic from the customization logic which is the information used while deploying the EJB
- The code written for developing the entity bean is not limited to a single type of database management system. It means that an entity bean can be bound to different database schemas. The choice of persistence binding can be done at deployment
- There are deployment standards defined for the deployment of an application crafted using EJB's. These deployment standards are bound to things like data source lookup, other application dependencies, data security configuration so on and so forth. Since these standards exist, EJB deployment tools can be used for EJB deployment. These tools make the jobs of EJB developers and the EJB deployer comfortable and almost error free

Simplified EJB 3.0 API

With EJB 3.0 specification, writing beans are made much simpler and easier. In this model the bean provider has to write simple Java Interface known as Business Interface [POJI], consisting of business methods it wants to expose to the client of the bean.

The business provider then writes bean class [POJO] that implements this interface and it consists of the main business logic of an application.

New EJB 3.0 API is simplified as compared to the previous version in the following ways:

- Home and Object interfaces are not required
- Component interface is not required
- Use of Java Annotations
- Simplified access to environment

Types Of Enterprise Bean

The EJB architecture defines three types of enterprise beans:

Scanned by CamScanner

Session Beans

A session bean object is a short-lived object that executes on behalf of a single client. It is a temporary, logical extension of a client application that runs on the Application Server.

A session bean does not represent shared data in a database, but obtains a data snapshot. However, a session bean can update data.

Session beans:

- Execute for a single client
- Can be transaction aware
- Do not represent directly shared data in an underlying database, although they may access and update this data
- Are short lived
- Are not persistent in a database
- Are removed if the container crashes and the client has to establish a new session
- May be **Stateless** and **Stateful**

Verifying credit card, closing a bank account, online shopping are some of the examples of session beans.

REMINDER

Session beans **implement** javax.ejb.SessionBean.
javax.ejb.SessionBean **extends** javax.ejb.EnterpriseBean.

Stateless Session Bean

Stateless beans simply do not remember anything about the client between the calls of the method. It forgets about the client once the business process completes or executes. The bean's state has no data regarding a specific client.

Because they are stateless, these bean types are lightweight and provide a high degree of scalability. Additionally, since they do not retain the client state, stateless session beans can be used across multiple clients.

Typically, an application requires fewer stateless session beans than stateful session beans to support the same number of clients.

A stateless session bean can implement a web service, but other types of enterprise beans cannot.

Unlike stateful session beans, the server, improving overall application performance, can pool stateless session beans.

Life Cycle Of Stateless Session Bean

A stateless session bean is never passivated. Its life cycle therefore has two stages:

1. Non-existent
2. Ready for business methods contained within the bean to be invoked

The client initiates the life cycle by obtaining a reference to a stateless session bean. The bean container performs dependency injections and then invokes the method named @PostConstruct, if it exists. The bean is now ready to have its business methods invoked by the client.

At the end of the EJB's life cycle, the EJB container calls the method named @PreDestroy. The bean's instance is then ready for garbage collection.

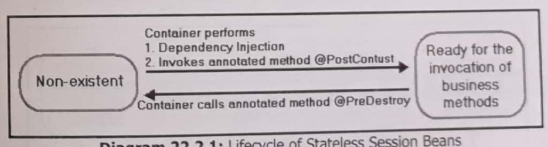

Diagram 22.2.1: Lifecycle of Stateless Session Beans

Stateful Session Bean

Stateful session bean can remember conversations between the client of the application and the application itself across method calls. They store data that is client specific. When a client calls the method again the bean remembers the client's previous method call.

The container typically maintains a Stateful session bean's object state in main memory, even across transactions, although the container may swap that state to secondary storage when deactivating the session bean. Only a single client can use a stateful session bean at a time.

The state is retained for the duration of the client bean session. If the client removes the bean or terminates, the session ends and the state disappears. Shopping cart is a typical example of Stateful Session Beans.

Scanned by CamScanner

Life Cycle Of Stateful Session Bean

The client initiates the life cycle by obtaining a reference to a stateful session bean. The container performs dependency injections and then invokes the method named @PostConstruct, if it exists. The bean is now ready to have its business methods invoked by the client.

While in this ready stage, the EJB container may decide to deactivate or passivate the EJB by moving it from memory to secondary storage. Typically, the EJB container uses a least-recently-used algorithm to select a bean for passivation. The EJB container invokes the method named @PrePassivate, if it exists, immediately before passivating the EJB. If a client invokes a business method belonging to the EJB while it is passivated, the EJB container activates the EJB, calls the method named @PostActivate, if it exists and then flags it as ready.

At the end of the life cycle, when a client application invokes a method named @Remove the EJB container calls the method named @PreDestroy. The bean's instance is then ready for garbage collection.

Application developer code spec controls the invocation of only one life-cycle method i.e. the method named @Remove. All other methods are invoked by the EJB container itself.

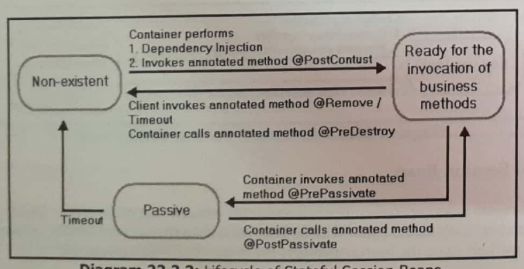

Diagram 22.2.2: Lifecycle of Stateful Session Beans

Entity Beans

Entity bean represents data maintained by an enterprise, typically in a database. An instance of the entity bean represents a row in a table.

In Bean-managed persistence, more control is given to the EJB developer to save and retrieve data, which is harder to build, as more coding is required.

In Container-managed persistence, container is responsible for saving and retrieving the data from the underlying database.

Multiple clients can share an entity object. A client may pass an object reference to another client. An entity object, survives a failure and restart its container. If a client holds a reference to the entity object before the container failure, the client can continue to use this reference after the container restarts.

An entity object is transact-able and lives as long as database entries.

A typical entity example is a Customer, one entity represents Mr. Shah [ID 12] and other might represent Mr. Bayross [ID 42].

REMINDER

Entity Beans **implement** javax.ejb.EntityBean.
javax.ejb.EntityBean **extends** javax.ejb.EnterpriseBean.

Message Driven Beans

A message-driven bean is an enterprise bean that allows enterprise applications to process messages asynchronously. It acts as a JMS message listener, which is similar to an event listener except that it receives messages instead of events.

The messages may be sent by any component, an application client, another enterprise bean or a Web component, a JMS application or system that does not use Java EE technology.

Message-driven beans:

- Do not have home or component interfaces
- Do not have business methods but define a message listener method, which the EJB container invokes to deliver messages
- Do not hold any state between calls of the message listener method
- Could be a New Customer Notification subscriber
- Are relatively short-lived
- Do not represent directly shared data in the database, but they can access and update this data

Scanned by CamScanner

- Can be transaction-aware

In order to get a message-driven bean to do something, a client sends message to a messaging service [JSM].

REMINDER

Message Driven Beans **implement** javax.ejb.MessageDrivenBean.
javax.ejb.MessageDrivenBean **extends** javax.ejb.EnterpriseBean.

Life Cycle Of Message-Driven Bean

The EJB container usually creates a pool of message-driven EJB instances. For each EJB instance, the EJB container does the following:

1. If the message-driven bean requires dependency injections, the EJB container injects these references before instantiating the instance
2. The EJB container calls the method named @PostConstruct, if it exists

Like a stateless session bean, a message-driven bean is never passivated and therefore has only two states:

1. Non-existent
2. Ready to receive messages

At the end of the life cycle, the EJB container calls the method named @PreDestroy, if it exists. The EJB's instance is then ready for garbage collection.

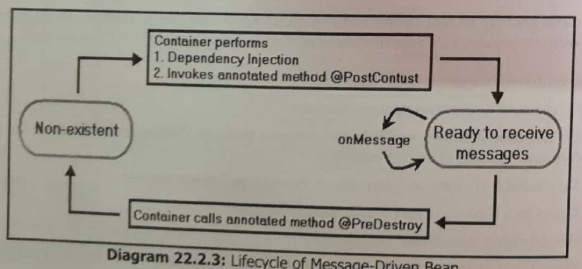

Diagram 22.2.3: Lifecycle of Message-Driven Bean

Writing Enterprise Bean

Before getting to know the beans in depth, let's develop, deploy and test a bean from start to end. This will give an idea about how a simple bean is built and run.

The bean that is being developed is named HelloAdviceBean and is a remote stateless service, which returns a greeting with a piece of advice to the client.

Following are the steps involved in creating this bean:

- Write Java code spec that forms the business interface
- Code the bean class with all its business methods
- Compile the Java source
- Create an XML deployment descriptor
- Create the EJB-jar file, containing the classes generated by compiling the business interface and bean class, using the jar utility or tools provided by the container
- Deploy the EJB-jar file on the container, either using auto-deploy feature or by using a container provided deployment tool
- Check the container to verify the EJB's deployment
- Write and execute a client to check the functioning of the deployed bean

Writing Business Interface

The business methods that are coded in the bean class and needs to be exposed to the client are declared in business interface. In other words, these are the methods that the client wants to invoke in the enterprise bean. Hence, it is a simple Plain Old Java Interface [POJI].

HINT

POJI is used to enforce the idea of developing an API against interfaces instead of implementations. POJI is an interface that a user defines. In EJB 3.0, a business interface is usually a POJI.

The business interface can be further declared as local business interface or a remote business interface depending on the client view supported by the bean.

Code spec for HelloAdvice.java:

```
package myFirstBeans;

/* This is HelloAdvice business Interface */
```

Scanned by CamScanner

```
public interface HelloAdvice
{
    public String getHelloAdvice();
}
```

Writing Bean Class

The bean class, HelloAdviceBean, is a **P**lain **O**ld **J**ava **O**bject [POJO] consisting of an implementation of getHelloAdvice(). This is where the actual business logic is written. In addition, it also has annotations that configure the bean appropriately.

HINT

POJO is a Java class that the user defines. In EJB 3.0, an Entity bean is usually a POJO.

Code spec for HelloAdviceBean.java:

```
package myFirstBeans;
import javax.ejb.Remote;
import javax.ejb.Stateless;

/* This is a remote stateless session bean */
  @Stateless
  @Remote(HelloAdvice.class)

  public class HelloAdviceBean implements myFirstBeans.HelloAdvice
  {
      private String{} adviceStrings = {"The present is the point at which time touches
                                        eternity", "Silence is sometimes an answer",
                                        "Knowledge of what is possible is the beginning of
                                        happiness", "The best way to keep good deeds in
                                        memory is to keep them refreshed with new ones", "It
                                        is not where we are, but what we are that creates our
                                        happiness"};
      public String getHelloAdvice()
      {
          System.out.println("This is getHelloAdvice() method");
          Int count = (int) (Math.random() * adviceStrings.lenght);
          Return "Hello! Here is an advice for the day: " + adviceStrings[count];
      }
  }
```

Points To Be Remembered While Writing Bean Class

- Import **javax.ejb.Remote** package [for remote bean]

- Import **javax.ejb.Stateless** package [for stateless bean]
- Bean must implement the component interface
- Ensure that the methods in the component interface matches the methods in bean class
- Annotation **@Stateless** configures the bean as stateless while deploying
- Annotation **@Remote** informs the container that the bean supports remote client view with remote business interface **HelloAdvice** while deploying
- The additional run time behavior of the bean can also be defined in XML Deployment Descriptor, instead of annotations
- Transaction and security behavior and the method of accessing the resources can also be declared using annotations

Writing Deployment Descriptor

The **ejb-jar.xml** file contains information regarding middleware requirements such as security, transaction services for the bean to live and serve the clients. During the deployment of the bean, the container will look into the ejb-jar.xml file and provides the required run time behavior services for the bean as mentioned in the file.

But in new simplified EJB 3.0 version the information about middleware service requirements of the bean can also be incorporated in bean class itself along with the business code with a new approach called **deployment annotations**.

In previous versions of EJB API, the container will get to know the runtime requirements of the bean only through deployment descriptor while in version 3.0 the container knows the information through deployment annotations in bean class or through deployment descriptor.

In EJB 3.0, writing ejb-jar.xml is optional. Bean can be successfully deployed even without the Deployment Descriptor file.

The following Deployment Descriptor is being used just to show that ejb-jar.xml is not compulsory as the bean can be configured by predefined deployment annotations.

Syntax for ejb-jar.xml:

```
<ejb-jar>
    <enterprise-beans><The Enterprise Bean Name></enterprise-beans>
</ejb-jar>
```

Scanned by CamScanner

Packaging Bean

To craft an EJB, the following files must be defined and available:

- **Enterprise bean class:** Implements the methods defined in the business interface and any lifecycle call back methods
- **Business interface:** Defines the methods implemented by the EJB class
- **Helper classes:** Are the classes needed by the EJB class. Example: Exception and Utility classes

After coding the **bean class** and its **interface**, package the bean class file, interface file and other helper files including its XML deployment descriptor into an **ejb-jar** file. An ejb-jar file is the module that stores the EJB.

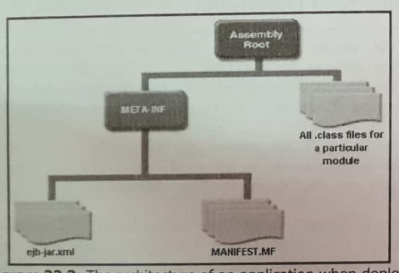

Diagram 22.3: The architecture of an application when deployed

If the bean class and business interface are dependent on other user classes, then ejb-jar should also contain the files containing these user classes.

HINT

ejb-jar can also contain EJB-Home interface file and EJB-Object interface file, old-style bean classes file as EJB 3.0 supports pre-Ejb 3.0 beans along with new beans.

An ejb-jar.xml file is a convenient way of distributing the bean component and can be used for developing different applications. After creating ejb-jar.xml, it has to be deployed within the container to offer its service to the clients.

To assemble a Java EE application, package one or more modules, i.e. **ejb-jar** files into an **EAR** file, which is an archive file that holds the application. When the EAR file [which contains the bean's ejb-jar file] is deployed, the enterprise bean is also deployed within the Application Server.

Deployment of Bean

EJB deployment tool bundled with container or IDE comes with a facility to package the files into ejb-jar file, which is much easier and less error prone.

The ejb-jar file can be deployed separately or it can be assembled with one or more ejb-jar files into a Java EE 5 application, which is EAR file. When EAR file is deployed, the enterprise beans assembled also gets deployed in Application Server.

During deployment, the container reads the **deployment descriptor** or the **deployment annotations** to generate an environment with services requested by the provider. The container creates stubs, skeletons, wrapper classes which are required for the bean to survive and to service the clients.

While deploying, the deployer gives a logical name to the bean and the bean gets registered in the server with this name. The client who wants to call methods on the bean refers the bean with this name.

Writing the Client

Following is the code spec that forms a Java based client application and invokes the **HelloAdvice()** method in the deployed bean.

Code spec for HelloAdviceClient.java:

```
package myFirstBean;

import javax.naming.Context;
import javax.naming.InitialContext;
/* This is client code which invokes HelloAdvice on remote
stateless session bean */
  public class HelloAdviceClient
  {
      public static void main(String[] args) throws Exception
      {
          Context ctx = new InitialContext();
      /* Initial context is the entry point for connecting to a
      JNDI tree */
          HelloAdvice Adv = (HelloAdvice) ctx.lookup("myFirstBean.HelloAdvice");
```

Scanned by CamScanner

```
    /* Lookup the HelloAdvice bean using JNDI */
    System.out.println(Adv.getHelloAdvice());

    /* Finally the actual business method is available. The
    method declared in the Business Interface(HelloAdvice) and
    implemented in the bean class */
    }
}
```

Here, an object of the HelloAdvice class is created. Using this object the getHelloAdvice() method is invoked.

Chapter

23

SECTION V: ENTERPRISE JAVA BEANS

Beginning With Enterprise Java Beans

This chapter shows how to develop, deploy and run a simple Java EE application named currencyconverter. The purpose of currencyconverter is to calculate currency conversions between Indian Rupees [INR] and United States Dollar [USD].

currencyconverter is a stateless session bean. This bean performs conversions across two currencies:

- INR to USD
- USD to INR

To demonstrate an EJB:

1. Create a business interface named **CurrencyConverter**
2. Create an enterprise bean named **CurrencyConverterBean**
3. Create a web client using JSP named index.jsp
4. Compile and package

Scanned by CamScanner

5. Deploy the web application using asant
6. Run the application using a web browser

Creating A Business Interface

The business interface defines the business methods that a client can call. The business methods are implemented in the enterprise bean class.

Code spec for CurrencyConverter.java:

```
package ejb;

import java.math.BigDecimal;
import javax.ejb.Remote;

@Remote

public interface CurrencyConverter
{
   public BigDecimal convert(String from, String to, BigDecimal amount);
   public BigDecimal findRate(String to);
}
```

Explanation:
In the above code spec:

```
package ejb;
```

A package named **ejb** is declared. This creates a folder named **ejb** and the **CurrencyConverter.class** file is placed in the ejb folder when deployed.

```
import java.math.BigDecimal;
```

BigDecimal consists of an arbitrary precision integer unscaled value and a non-negative 32-bit integer scale. It represents the number of digits to the right of the decimal point.

The number represented by the BigDecimal is **unscaledValue/10scale**.

BigDecimal allows performing following operations:

- Basic arithmetic
- Scale manipulation
- Comparison
- Hashing

- Format conversion

Since Currency conversions are performed in this example, BigDecimal is used.

```
import javax.ejb.Remote;
```

Remote interface is imported so that the Remote annotation can be used.

```
@Remote
```

The @Remote annotation enhances the interface definition. This informs the container that CurrencyConverterBean is accessed by remote clients.

REMINDER

Unlike EJB 2.1, the business methods do not have to throw java.rmi.RemoteException.

```
public interface CurrencyConverter
{
   public BigDecimal convert(String from, String to, BigDecimal amount);
   public BigDecimal findRate(String to);
}
```

A business interface named CurrencyConverter is declared with two methods i.e. convert and findRate.

Creating An Enterprise Bean

The enterprise bean class is named CurrencyConverterBean. This class implements the two business methods named convert and findRate, which is defined in the CurrencyConverter remote business interface.

Code spec for CurrencyConverterBean:

```
package ejb;

import java.math.BigDecimal;
import javax.ejb.Stateless;

@Stateless

public class CurrencyConverterBean implements ejb.CurrencyConverter
{
   private BigDecimal USD = new BigDecimal("0.0229137");
   private BigDecimal INR = new BigDecimal("46.589100");
```

Scanned by CamScanner

```
public BigDecimal convert(String from, String to, BigDecimal amount)
{
    if(from.equals(to))
    {
        return amount;
    }
    else
    {
        BigDecimal toRate = findRate(to);
        BigDecimal result = toRate.multiply(amount);
        return result.setScale(2, BigDecimal.ROUND_UP);
    }
}

public BigDecimal findRate(String to)
{
    BigDecimal returnValue = null;
    if(to.equals("INR"))
    {
        returnValue = INR;
    }
    if(to.equals("USD"))
    {
        returnValue = USD;
    }
    return returnValue;
}
}
```

Explanation:
In the above code spec:

```
import javax.ejb.Stateless;
```

Stateless interface is imported so that the Stateless annotation can be used.

```
@Stateless
```

The @Stateless annotation enhances the enterprise bean class. This informs the container that CurrencyConverterBean is a stateless session bean.

```
private BigDecimal USD = new BigDecimal("0.0229137");
private BigDecimal INR = new BigDecimal("46.589100");
```

Two private memory variables of type BigDecimal are declared and initialized with the currency rates.


```
public BigDecimal convert(String from, String to, BigDecimal amount)
{
    if(from.equals(to))
    {
        return amount;
    }
    else
    {
        BigDecimal toRate = findRate(to);
        BigDecimal result = toRate.multiply(amount);
        return result.setScale(2, BigDecimal.ROUND_UP);
    }
}
```

This method accepts the source currency, the destination currency and the amount. Based on these three values converts the amount from the source currency to the destination currency.

To perform the conversion a method named findRate() is invoked, which returns the rate of destination currency.

```
public BigDecimal findRate(String to)
{
    BigDecimal returnValue = null;
    if(to.equals("INR"))
    {
        returnValue = INR;
    }
    if(to.equals("USD"))
    {
        returnValue = USD;
    }
    return returnValue;
}
```

This method is invoked by the convert() method to fetch the rate of destination currency.

Creating A Web Client Using JSP

Once the CurrencyConverter EJB is coded, a client needs to be coded that can utilize the services of EJB. To do so a web client is created using JSP.

Code spec for CurrencyConverterBean:

```
<%@ page import="ejb.CurrencyConverter, java.math.BigDecimal,
                 javax.naming.InitialContext"%>
<%!
  private CurrencyConverter converter = null;
```

Scanned by CamScanner

```
    public void jspInit()
    {
        try
        {
            InitialContext ic = new InitialContext();
            converter = (CurrencyConverter) ic.lookup(CurrencyConverter.class.getName());
        }
        catch (Exception ex)
        {
            System.out.println("Could not create currency converter stateless session bean."+
                                ex.getMessage());
        }
    }

    public void jspDestroy()
    {
        converter = null;
    }
%>
<HTML>
  <HEAD>
      <TITLE>Currency Converter</TITLE>
  </HEAD>
  <BODY BGCOLOR="pink">
      <H1>Currency Converter in 4 easy steps</H1>
      <HR>
      <P>Enter an amount to convert:</P>
      <FORM METHOD="get">
          <TABLE WIDTH="100%" BORDER="0" CELLSPACING="0"
                 CELLPADDING="0">
              <TR>
                  <TD WIDTH="5%" ALIGN="center"><H1>1</H1></TD>
                  <TD> Convert: <BR />
                      <INPUT TYPE="text" NAME="amount" VALUE="1" SIZE="10"
                             TABINDEX="1" />
                      <DIV>Enter an amount</DIV>
                  </TD>
              </TR>
              <TR>
                  <TD WIDTH="5%" ALIGN="center"><H1>2</H1></TD>
                  <TD> From this currency:<BR />
                      <SELECT NAME="From" SIZE="3" TABINDEX="2">
                          <OPTION VALUE="USD">America (United States), Dollar (USD)
                          </OPTION>
                          <OPTION VALUE="INR">India, Rupee (INR)</OPTION>
                      </SELECT>
                  </TD>
              </TR>
```



```
              <TR>
                  <TD WIDTH="5%" ALIGN="center"><H1>3</H1></TD>
                  <TD> To this currency:<BR />
                      <SELECT NAME="To" SIZE="3" TABINDEX="3">
                          <OPTION VALUE="USD">America (United States), Dollar (USD)
                          </OPTION>
                          <OPTION VALUE="INR">India, Rupee (INR)</OPTION>
                      </SELECT>
                  </TD>
              </TR>
              <TR>
                  <TD WIDTH="5%" ALIGN="center" CLASS="title"><H1>4</H1></TD>
                  <TD><INPUT NAME="cmdSubmit" TYPE="submit" VALUE="Submit"
                             ALT="Convert" TABINDEX="4" /></TD>
              </TR>
          </TABLE>
      </FORM>
      <%
          String amount = request.getParameter("amount");
          if (amount != null && amount.length() > 0)
          {
              BigDecimal d = new BigDecimal(amount);
              BigDecimal convertedAmount = converter.convert(request
                      .getParameter("From"), request.getParameter("To"), d);
      %>
              <HR>
              <%= amount %> <%= request.getParameter("From")%> =
              <%= convertedAmount %> <%= request.getParameter("To")%>
      <%
          }
      %>
  </BODY>
</HTML>
```

Explanation:

In the above code spec:

```
<%@ page import="ejb.CurrencyConverter, java.math.BigDecimal,
                 javax.naming.InitialContext"%>
```

The classes needed by the web client are declared using a JSP **page** directive, which is enclosed within the <%@ %> characters.

Following classes are imported:

- ejb.CurrencyConverter: This is the stateless session bean that was created and explained earlier

Scanned by CamScanner

- java.math.BigDecimal: BigDecimal consists of an arbitrary precision integer unscaled value and a non-negative 32-bit integer scale
- javax.naming.InitialContext: This class is the starting context for performing naming operations. All naming operations are relative to a context. The initial context implements the Context interface and provides the starting point for resolution of names

```
private CurrencyConverter converter = null;
```

An object named converter of type CurrencyConverter is declared and initialized to null.

```
public void jspInit()
{
    try
    {
        InitialContext ic = new InitialContext();
        converter = (CurrencyConverter) ic.lookup(CurrencyConverter.class.getName());
    }
    catch (Exception ex)
    {
        System.out.println("Could not create currency converter stateless session bean."+
                                ex.getMessage());
    }
}
```

Since locating the business interface i.e. **CurrencyConverter** and creating the enterprise bean are performed only once, this code appears within the jspInit() method inside JSP declaration enclosed within the <%! %> characters.

An object named ic of type InitialContext is declared and initialized.

Using the lookup() method of the InitialContext object, the named object of the CurrencyConverter class is retrieved. If name is empty, a new instance of this context, which represents the same naming context as this context is retrieved.

The named object that is returned by the lookup() method is type casted to **CurrencyConverter** and then assigned to the **CurrencyConverter** class object created earlier named converter.

```
public void jspDestroy()
{
    converter = null;
}
```

The **CurrencyConverter** class object created earlier named converter is destroyed in the jspDestroy() method of the JSP page.

The declaration is followed by standard HTML markup for creating a form that visually appears as shown in diagram 23.1.

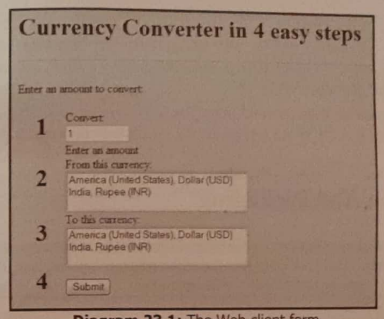

Diagram 23.1: The Web client form

This form holds one text box, two list boxes and a command button.

HINT

For demonstration purpose, only two currencies are populated in the list box, which can be increased as desired.

```
<%
    String amount = request.getParameter("amount");
    if (amount != null && amount.length() > 0)
    {
        BigDecimal d = new BigDecimal(amount);
        BigDecimal convertedAmount = converter.convert(request.getParameter("From"),
                                                       request.getParameter("To"), d);
%>
```

A scriptlet enclosed within the <% %> characters retrieves a parameter named amount from the request object.

Only if the amount holds a valid value, the convert() method of the stateless session bean named CurrencyConverter's object is invoked with the following parameters:

- request.getParameter("From"): This represents the value selected by the user from the first list box

Scanned by CamScanner

- request.getParameter("To"): This represents the value selected by the user from the second list box
- d: This represents the amount that the user has entered

```
<HR>
<%= amount %> <%= request.getParameter("From")%> = <%= convertedAmount %>
<%= request.getParameter("To")%>
```

Finally, a JSP scriptlet invokes the enterprise bean's business methods and JSP expressions enclosed within the <%= %> characters insert the results into the stream of data returned to the client.

Compilation And Packaging

Now the remote business interface i.e. CurrencyConverter.java, the enterprise bean class i.e. CurrencyConverterBean.java and the web client i.e. index.jsp are in place, its time to compile and package and thus make it available for deployment.

To proceed with the above exercise, the directory structure as shown in diagram 23.2 is created.

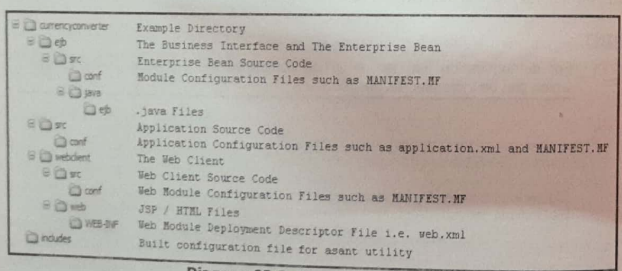

Diagram 23.2: Directory structure

- Create a directory structure identical to the one shown in diagram 23.2
- Place the CurrencyConvert.java and CurrencyConverterBean.java files in the \currencyconvert\ejb\src\java\ejb directory
- Place the index.jsp file in the \currencyconvert\webclient\web directory

- Create a file named MANIFEST.MF with the following contents:

```
Manifest-Version: 1.0
```

Place this file in the following locations:

- \currencyconvert\ejb\src\conf
- \currencyconvert\src\conf
- \currencyconvert\webclient\src\conf

REMINDER

JAR files support a wide range of functionality, including electronic signing, version control, package sealing, extensions and so on. JAR file's manifest gives JAR files the ability to be so versatile.

The manifest is a special file that contains information about the files packaged in a JAR file. By tailoring this **meta** information that the manifest contains, enable the JAR file to be used for a variety of purposes.

- Create a file named application.xml with the following contents and place this file in \currencyconvert\src\conf:

```
<?xml version="1.0" encoding="UTF-8"?>
<application version="5" xmlns="http://java.sun.com/xml/ns/javaee"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                                http://java.sun.com/xml/ns/javaee/application_5.xsd">
  <display-name>currencyconverter</display-name>
  <module>
      <web>
          <web-uri>webclient.war</web-uri>
          <context-root>/currencyconverter</context-root>
      </web>
  </module>
  <module>
      <ejb>ejb.jar</ejb>
  </module>
</application>
```

The application.xml file usually holds:

- Icons, a description and a name used by tools
- The modules contained within their location within the EAR, the context root for each web application module and alternative deployment descriptor references
- Security role names and descriptions

Scanned by CamScanner

- Create a file named web.xml with the following contents and place this file in \currencyconvert\webclient\web\WEB-INF:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5">
  <session-config>
      <session-timeout>30</session-timeout>
  </session-config>
  <welcome-file-list>
      <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

The web.xml file provides configuration and deployment information such as Servlet parameters, Servlet and Java Server Pages [JSP] definitions and Uniform Resource Locators [URL] mappings for the Web components that comprise a Web application.

- Create a file named most-common-targets.xml with the following contents and place this file in \includes:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="app-server-ant" default="compile">
  <property name="javaee.home" value="c:/Sun/AppServer"/>
  <property name="javaee.server.name" value="localhost"/>
  <property name="javaee.server.port" value="8080"/>
  <property name="javaee.adminserver.port" value="4848"/>

  <property name="home.dir" value="."/>
  <property name="build.dir" value="${home.dir}/build"/>
  <property name="build.classes.dir" value="${build.dir}/classes"/>
  <property name="src.dir" value="${home.dir}/src/java"/>
  <property name="src.conf" value="${home.dir}/src/conf"/>
  <property name="dist.dir" value="${home.dir}/dist"/>
  <property name="module.name" value="${ant.project.name}"/>

<!-- Cleans the build directory -->
  <target name="clean">
      <delete dir="${build.dir}" />
      <delete dir="${dist.dir}" />
      <delete dir="${client.jar.dir}" />
      <iterate target="clean"/>
  </target>

  <available property="has-java-sources" file="${src.dir}"/>
```



```
  <dirname property="antfile.dir" file="${ant.file}"/>

  <path id="classpath">
      <fileset dir="${javaee.home}" includes="lib/javaee.jar, lib/appserv-rt.jar,
                lib/appserv-ws.jar, lib/activation.jar"/>
      <dirset dir="${antfile.dir}/..">
          <include name="**/build/classes"/>
      </dirset>
      <pathelement location="${build.classes.dir}"/>
      <pathelement path="${extra.classpath}"/>
      <path refid="javaee.classpath" />
  </path>

  <target name="compile" depends="-pre-compile,-post-compile" if="has-java-sources">
      <mkdir dir="${build.classes.dir}"/>
      <javac srcdir="${src.dir}" excludes="${javac.excludes}" destdir="${build.classes.dir}"
            includes="**">
          <classpath refid="classpath"/>
      </javac>
  </target>

  <target name="getBrowser" unless="default.browser">
      <condition property="browser" value="rundll32">
          <os family="windows"/>
      </condition>
      <condition property="browser" value="/usr/bin/open">
          <and>
              <os family="mac"/>
          </and>
      </condition>
      <property name="default.browser" value="${browser}"/>
  </target>

<!-- This target launches a browser for the specified URL -->
  <target name="launch" depends="getBrowser">
      <echo>
          Trying to launch the browser with the url
          http://${javaee.server.name}:${javaee.server.port}/${module.name}
      </echo>
      <condition property="extra-args" value="url.dll,FileProtocolHandler">
          <os family="windows"/>
      </condition>
      <condition property="extra-args" value="">
          <not>
              <os family="windows"/>
          </not>
      </condition>
      <echo>Launching ${module.name}.</echo>
      <exec executable="${default.browser}" spawn="true">
```

Scanned by CamScanner

```
        <arg line ="${extra-args} http://${javaee.server.name}:${javaee.server.port}
                    /${module.name}" />
    </exec>
</target>

<target name="init" depends="-pre-init"/>
<target name="-pre-init"/>
<target name="-pre-compile"/>
<target name="-post-compile"/>
<target name="-pre-setup"/>

<target name="default" depends="default-ear,default-not-ear"/>

<target name="default-ear" if="is.ear.module" depends="init">
    <iterate target="default"/>
</target>

<target name="default-not-ear" unless="is.ear.module" depends="init,compile,package,
                copy-dist"/>

<target name="copy-dist">
    <mkdir dir="${dist.dir}"/>
    <copy todir="${dist.dir}">
        <fileset file="${app.module}"/>
    </copy>
</target>

<target name="run" depends="default,deploy,launch" description="builds, packages and
                runs the application"/>

<target name="tools">
    <condition property="javaee-script-suffix" value=".bat">
        <os family="windows"/>
    </condition>
    <condition property="javaee-script-suffix" value="">
        <not>
            <os family="windows"/>
        </not>
    </condition>
    <condition property="path.separator" value=";">
        <os family="windows"/>
    </condition>
    <condition property="path.separator" value=":">
        <not>
            <os family="windows"/>
        </not>
    </condition>
```



```
<!-- setup properties for various Java EE tools -->
    <property name="asadmin" value="${javaee.home}/bin/asadmin
                    ${javaee-script-suffix}"/>
    <property name="appclient" value="${javaee.home}/bin/appclient
                    ${javaee-script-suffix}"/>

</target>

<!-- Construct Java EE classpath -->
<path id="javaee.classpath">
    <pathelement location="${javaee.home}/lib/javaee.jar"/>
    <pathelement location="${javaee.home}/bin"/>
</path>

<target name="undeploy" depends="tools">
    <echo message="Undeploying ${module.name}." />
    <exec executable="${asadmin}">
        <arg line="undeploy"/>
        <arg line=" --host ${javaee.server.name}" />
        <arg line=" ${module.name} " />
    </exec>
</target>

<target name="deploy" depends="tools">
    <fail unless="app.module" message="app.module must be set before invoking this
            target. Otherwise there is no application to deploy."/>
    <fail unless="module.name" message="app.name must be set before invoking this
            target. Otherwise there is no application to deploy."/>
    <condition property="app.url" value="http://${javaee.server.name}:
                    ${javaee.server.port}/${module.name}">
        <not>
            <isset property="app.endpointuri"/>
        </not>
    </condition>
    <property name="failonerror" value="true"/>
    <exec executable="${asadmin}" failonerror="${failonerror}">
        <arg line=" deploy "/>
        <arg line=" --host ${javaee.server.name}" />
        <arg line=" --port ${javaee.adminserver.port}" />
        <arg line=" --name ${module.name}"/>
        <arg line=" --force=true "/>
        <arg line=" --upload=true "/>
        <arg line="${app.module}" />
    </exec>
    <echo message="Application Deployed at: ${app.url}"/>
</target>
</project>
```

Scanned by CamScanner

This file holds following Ant targets:

Targets	Description
clean	Removes the generated directories such as build and dist
compile	Compiles the project
default	Compiles and packages the archive
deploy	Deploys the application
launch	Launches the application in a browser
package	Packages the archive
run	Builds, packages and runs the application
undeploy	Undeploys the application

Additionally, it also holds common properties that represent information such as:

- The installation directory of the Java EE 5 SDK
- Host name of the server where the Java EE 5 SDK is installed
- The port number for the server while installing Java EE 5 SDK
- The port number for admin server while installing the Java EE 5 SDK

- Create a file named build.xml with the following contents and place this file in \currencyconvert:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="currencyconverter" default="default" basedir=".">
  <import file="../includes/most-common-targets.xml"/>

  <property name="is.ear.module" value="true"/>
  <property name="app.module" value="${build.dir}/${module.name}.ear"/>
  <property name="ear.build.dir" value="${build.dir}/ear"/>

  <path id="ear-components">
      <filelist dir="./ejb" files="build.xml"/>
      <filelist dir="./webclient" files="build.xml"/>
  </path>

  <macrodef name="iterate">
      <attribute name="target"/>
      <sequential>
          <subant target="@{target}" failonerror="false">
              <buildpath refid="ear-components"/>
          </subant>
      </sequential>
  </macrodef>

  <target name="package">
```



```
      <mkdir dir="${ear.build.dir}"/>
      <mkdir dir="${ear.build.dir}/META-INF"/>
      <copy file="${src.conf}/application.xml" todir="${ear.build.dir}/META-INF/"
              failonerror="false"/>
      <dirname property="ear.dir" file="${ear.build.dir}/MANIFEST/"/>
      <subant target="copy-dist" failonerror="false">
          <property name="dist.dir" value="${ear.dir}"/>
          <buildpath refid="ear-components"/>
      </subant>
      <delete file="${app.module}"/>
      <jar jarfile="${app.module}" basedir="${ear.build.dir}" />
      <delete dir="${ear.build.dir}"/>
  </target>
</project>
```

REMINDER

The build.xml is an Ant build file that holds targets to create a build subdirectory and to copy and compile packaged war file into that directory.

Build files are always crafted using XML. Each build file holds a project and at least one [i.e. the default] target. A target contains task elements. Each task element within the build file can have an **id** attribute. Task elements can later be referenced by its id attribute. The id attributes assigned to task elements must be unique across the file.

This is the master build file, which in turn refers to two child build files for both the modules [ejb and webclient] in this example.

```
<import file="../includes/most-common-targets.xml"/>
```

A file named **most-common-targets.xml** is imported to refer to the most common targets required to compile and package a web application.

```
<property name="is.ear.module" value="true"/>
```

A property named is.ear.module is declared and set to true. This helps **most-common-targets.xml** to identify whilst processing that this is an ear module.

EAR files are a way of creating a portable collection of logically related Java Platform, Enterprise Edition components.

There are a number of benefits for doing this, including:

- Synchronized security role-names
- Dependencies

Scanned by CamScanner

- Versioning
- Web context configuration
- Application-level configuration
- Alternate deployment descriptors

An EAR file usually contains the following:

1. Enterprise Java Bean [EJB] component modules [.jar]
2. Web application modules [.war]
3. Application client modules [.jar]
4. Resource adapter modules [.rar]

In the CurrencyConverter example, point 1 and 2 are used i.e. EJB module and Web client module.

```
<property name="app.module" value="${build.dir}/${module.name}.ear"/>
```

A property named **app.module** is declared and holds the path to the .ear file.

```
<property name="ear.build.dir" value="${build.dir}/ear"/>
```

A property named **ear.build.dir** is declared and holds the path where the ear file will be placed.

```
<path id="ear-components">
    <filelist dir="./ejb" files="build.xml"/>
    <filelist dir="./webclient" files="build.xml"/>
</path>
```

An element named **path** is defined to hold paths to the two child modules i.e. ejb and webclient.

Using **filelist** an ordered list of files in this case build.xml of both the child modules are added to the path element.

```
<macrodef name="iterate">
    <attribute name="target"/>
    <sequential>
        <subant target="@{target}" failonerror="false">
            <buildpath refid="ear-components"/>
        </subant>
    </sequential>
</macrodef>
```


A macro named **iterate** is defined to provide the ability to transparently iterate over sub-projects [in this case two child modules] calling a target in each of their build files.

The <subant> target is used to uniformly call an Ant target on multiple sub-projects [in this case two child modules] without having to maintain a list of those sub-projects. This is especially helpful when in a master build file is used to build the child modules.

```
<target name="package">
    <mkdir dir="${ear.build.dir}"/>
    <mkdir dir="${ear.build.dir}/META-INF"/>
    <copy file="${src.conf}/application.xml" todir="${ear.build.dir}/META-INF/"
            failonerror="false"/>
    <dirname property="ear.dir" file="${ear.build.dir}/MANIFEST/"/>
    <subant target="copy-dist" failonerror="false">
        <property name="dist.dir" value="${ear.dir}"/>
        <buildpath refid="ear-components"/>
    </subant>
    <delete file="${app.module}"/>
    <jar jarfile="${app.module}" basedir="${ear.build.dir}" />
    <delete dir="${ear.build.dir}"/>
</target>
```

A target named package is defined to perform the following actions:

- Create a directory named **ear** under \currencyconverter\build\
- Create a directory named **META-INF** under \currencyconverter\build\ear\
- Copy **application.xml** to \currencyconverter\build\ear\META-INF
- Copy the **child module packages** to \currencyconverter\build\ear
- Built **currencyconverter.ear** using jar and places it under \currencyconverter\build
- Delete the directory named **ear** from \currencyconverter\build
- Create the directory named **dist** under \currencyconverter
- Copy **currencyconverter.ear** to \currencyconverter\dist

- Create a file named build.xml with the following contents and place this file in \currencyconvert\ejb:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="ejb" default="default" basedir=".">
    <import file="../../includes/most-common-targets.xml"/>

    <property name="app.module" value="${build.dir}/${module.name}.jar"/>
    <property name="jar.build.dir" value="${build.dir}/jar"/>
```

Scanned by CamScanner

```
  <patternset id="all.nonjava.files">
  <!-- All conf files [including test files] -->
      <include name="**/*.txt"/>
      <include name="**/*.xml"/>
      <include name="**/*.properties"/>
  </patternset>

  <target name="package">
      <mkdir dir="${jar.build.dir}"/>
  <!-- Copy in the compiled Classes -->
      <copy todir="${jar.build.dir}">
          <fileset dir="${build.classes.dir}"/>
          <fileset dir="${src.dir}">
              <patternset refid="all.nonjava.files"/>
          </fileset>
      </copy>
  <!-- Copy in any deployment descriptors -->
      <copy todir="${jar.build.dir}/META-INF">
          <fileset dir="${src.conf}">
              <include name="*.xml"/>
          </fileset>
      </copy>
      <delete file="${app.module}"/>
      <jar jarfile="${app.module}" basedir="${jar.build.dir}"/>
      <delete dir="${jar.build.dir}"/>
  </target>
</project>
```

This build file is invoked by the master build file as explained earlier and perform the following actions:

```
<property name="jar.build.dir" value="${build.dir}/jar"/>
```

A property named **jar.build.dir** is declared and holds the path where the jar file is placed.

```
<patternset id="all.nonjava.files">
    <include name="**/*.txt"/>
    <include name="**/*.xml"/>
    <include name="**/*.properties"/>
</patternset>
```

A patternset element is used to define following kinds of files:

- Text files
- XML files
- Property files

This pattern is referenced by the package target to include all the non-java files whilst performing copy operation from the source directory [src] to the jar directory [currencyconverter\ejb\build\jar]

```
<target name="package">
    <mkdir dir="${jar.build.dir}"/>
    <copy todir="${jar.build.dir}">
        <fileset dir="${build.classes.dir}"/>
        <fileset dir="${src.dir}">
            <patternset refid="all.nonjava.files"/>
        </fileset>
    </copy>
    <copy todir="${jar.build.dir}/META-INF">
        <fileset dir="${src.conf}">
            <include name="*.xml"/>
        </fileset>
    </copy>
    <delete file="${app.module}"/>
    <jar jarfile="${app.module}" basedir="${jar.build.dir}"/>
    <delete dir="${jar.build.dir}"/>
</target>
```

A target named package is defined to perform the following actions:

- Create the directory named **classes** under \currencyconverter\ejb\build
- Compile **2 source files** i.e. CurrencyConverter.java and CurrencyConverterBean.java to \currencyconverter\ejb\build\classes
- Create the directory named **jar** under \currencyconverter\ejb\build
- Copy **2 compiled class files** to \currencyconverter\ejb\build\jar
- Built **ejb.jar** using jar and places it under \currencyconverter\ejb\build
- Delete the directory named **jar** from \currencyconverter\ejb\build
- Create the directory named **dist** under \currencyconverter\ejb
- Copy **ejb.jar** to \currencyconverter\ejb\dist

- Create a file named build.xml with the following contents and place this file in \currencyconvert\webclient:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="webclient" default="default" basedir=".">
  <import file="../../includes/most-common-targets.xml"/>

  <property name="app.module" value="${build.dir}/${module.name}.war"/>
  <property name="web.docbase.dir" value="${home.dir}/web"/>
  <property name="war.build.dir" value="${build.dir}/war"/>
  <property name="build.web.dir" value="${build.dir}/web"/>
```

Scanned by CamScanner

```
<path id="common.jars">
    <filelist dir="../ejb" files="build.xml"/>
</path>

<macrodef name="iterate">
    <attribute name="target"/>
    <sequential>
        <subant target="@{target}" failonerror="false">
            <buildpath refid="common.jars"/>
        </subant>
    </sequential>
</macrodef>

<target name="package" description="packages the archive">
    <mkdir dir="${war.build.dir}"/>
<!-- Copy in the content -->
    <mkdir dir="${build.web.dir}"/>
        <copy todir="${build.web.dir}">
            <fileset dir="${web.docbase.dir}"/>
        </copy>
    <copy todir="${war.build.dir}">
        <fileset dir="${build.web.dir}"/>
    </copy>
    <delete file="${app.module}"/>
    <jar jarfile="${app.module}" basedir="${war.build.dir}"/>
    <delete dir="${war.build.dir}"/>
</target>
</project>
```

This build file is invoked by the master build file as explained earlier and perform the following actions:

```
<property name="web.docbase.dir" value="${home.dir}/web"/>
```

A property named **web.docbase.dir** is declared and holds the path where the web documents is placed.

```
<property name="war.build.dir" value="${build.dir}/war"/>
```

A property named **war.build.dir** is declared and holds the path where the war file is placed.

```
<property name="build.web.dir" value="${build.dir}/web"/>
```

A property named **build.web.dir** is declared and holds the path where the web documents is placed.

```
<path id="common.jars">
    <filelist dir="../ejb" files="build.xml"/>
```



```
</path>
```

An element named **path** is defined to hold path to the common.jars.

Using **filelist** an ordered list of file in this case build.xml of the EJB child module is added to the path element.

```
<macrodef name="iterate">
    <attribute name="target"/>
    <sequential>
        <subant target="@{target}" failonerror="false">
            <buildpath refid="common.jars"/>
        </subant>
    </sequential>
</macrodef>
```

A macro named **iterate** is defined to provide the ability to transparently iterate over EJB module calling a target in its build file.

The <subant> target is used to uniformly call an Ant target on EJB module without having to maintain a list of the EJB module.

```
<target name="package" description="packages the archive">
    <mkdir dir="${war.build.dir}"/>
    <mkdir dir="${build.web.dir}"/>
        <copy todir="${build.web.dir}">
            <fileset dir="${web.docbase.dir}"/>
        </copy>
    <copy todir="${war.build.dir}">
        <fileset dir="${build.web.dir}"/>
    </copy>
    <delete file="${app.module}"/>
    <jar jarfile="${app.module}" basedir="${war.build.dir}"/>
    <delete dir="${war.build.dir}"/>
</target>
```

A target named package is defined to perform the following actions:

- Create the directory named **war** under \currencyconverter\webclient\build
- Create the directory named **web** under \currencyconverter\webclient\build
- Copy **2 files** [i.e. web.xml and index.jsp] to \currencyconverter\webclient\build
- Copy **2 files** [i.e. web.xml and index.jsp] to \currencyconverter\webclient\build
- Built **webclient.war** using jar under \currencyconverter\webclient\build
- Delete the directory named **war** from \currencyconverter\webclient\build
- Create the directory named **dist** under \currencyconverter\webclient

Scanned by CamScanner

- Copy **webclient.war** to \currencyconverter\webclient\dist

This completes placing all the code spec and the configuration files in the appropriate directories.

Using command prompt reach the directory that holds the application in this case currencyconverter.

Run the following command being in the same directory:

```
<Command Prompt>asant
```

Output:

```
C:\Java Codes\currencyconverter>asant
Buildfile: build.xml

-pre-init:

init:

default-ear:

-pre-init:

init:

default-ear:

-pre-compile:

-post-compile:

compile:
    [mkdir] Created dir: C:\Java Codes\currencyconverter\ejb\build\classes
    [javac] Compiling 2 source files to C:\Java Codes\currencyconverter\ejb\build\classes

package:
    [mkdir] Created dir: C:\Java Codes\currencyconverter\ejb\build\jar
     [copy] Copying 2 files to C:\Java Codes\currencyconverter\ejb\build\jar
      [jar] Building jar: C:\Java Codes\currencyconverter\ejb\build\ejb.jar
   [delete] Deleting directory C:\Java Codes\currencyconverter\ejb\build\jar

copy-dist:
    [mkdir] Created dir: C:\Java Codes\currencyconverter\ejb\dist
     [copy] Copying 1 file to C:\Java Codes\currencyconverter\ejb\dist

default-not-ear:

default:
Trying to override old definition of task iterate

-pre-init:

init:

default-ear:

-pre-compile:
```



```
-post-compile:

compile:

package:
    [mkdir] Created dir: C:\Java Codes\currencyconverter\webclient\build\war
    [mkdir] Created dir: C:\Java Codes\currencyconverter\webclient\build\web
     [copy] Copying 2 files to C:\Java Codes\currencyconverter\webclient\build\web
     [copy] Copying 2 files to C:\Java Codes\currencyconverter\webclient\build\war
      [jar] Building jar: C:\Java Codes\currencyconverter\webclient\build\webclient.war
   [delete] Deleting directory C:\Java Codes\currencyconverter\webclient\build\war

copy-dist:
    [mkdir] Created dir: C:\Java Codes\currencyconverter\webclient\dist
     [copy] Copying 1 file to C:\Java Codes\currencyconverter\webclient\dist

default-not-ear:

default:

-pre-compile:

-post-compile:

compile:

package:
    [mkdir] Created dir: C:\Java Codes\currencyconverter\build\ear
    [mkdir] Created dir: C:\Java Codes\currencyconverter\build\ear\META-INF
     [copy] Copying 1 file to C:\Java Codes\currencyconverter\build\ear\META-INF

copy-dist:
     [copy] Copying 1 file to C:\Java Codes\currencyconverter\build\ear
Trying to override old definition of task iterate

copy-dist:
     [copy] Copying 1 file to C:\Java Codes\currencyconverter\build\ear
      [jar] Building jar: C:\Java Codes\currencyconverter\build\currencyconverter.ear
   [delete] Deleting directory C:\Java Codes\currencyconverter\build\ear

copy-dist:
    [mkdir] Created dir: C:\Java Codes\currencyconverter\dist
     [copy] Copying 1 file to C:\Java Codes\currencyconverter\dist

default-not-ear:

default:

BUILD SUCCESSFUL
Total time: 4 seconds
C:\Java Codes\currencyconverter>
```

Once the package is built using the asant utility, the directory structure of the currencyconverter module changes as shown in diagram 23.3. Two new directories are created i.e. build and dist under every module and the master module. These directories hold the archive files for the appropriate modules.

Scanned by CamScanner

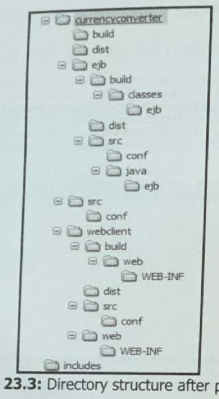

Diagram 23.3: Directory structure after packaging

Deploying The Web Application

Once the package has been built successfully, the same can be deployed using the asant utility.

Run the following command being in the same directory:

```
<Command Prompt>asant deploy
```

Output:

```
C:\Java Codes\currencyconverter>asant deploy
Buildfile: build.xml

tools:

deploy:
     [exec] Command deploy executed successfully.
     [echo] Application Deployed at: http://localhost:8080/currencyconverter

BUILD SUCCESSFUL
Total time: 4 seconds
C:\Java Codes\currencyconverter>
```


Running The Application

Once the package has been deployed successfully, the same can be executed using the asant utility or by pointing the web browser to http://localhost:8080/currencyconverter.

Run the following command being in the same directory:

```
<Command Prompt>asant run
```

HINT

run command of the asant utility will first compile, deploy and then execute the web application.

Output:

```
C:\Java Codes\currencyconverter>asant
Buildfile: build.xml

-pre-init:

init:

default-ear:

-pre-init:

init:

default-ear:

-pre-compile:

-post-compile:

compile:
    [mkdir] Created dir: C:\Java Codes\currencyconverter\ejb\build\classes
    [javac] Compiling 2 source files to C:\Java Codes\currencyconverter\ejb\build\classes

package:
    [mkdir] Created dir: C:\Java Codes\currencyconverter\ejb\build\jar
     [copy] Copying 2 files to C:\Java Codes\currencyconverter\ejb\build\jar
      [jar] Building jar: C:\Java Codes\currencyconverter\ejb\build\ejb.jar
   [delete] Deleting directory C:\Java Codes\currencyconverter\ejb\build\jar

copy-dist:
    [mkdir] Created dir: C:\Java Codes\currencyconverter\ejb\dist
     [copy] Copying 1 file to C:\Java Codes\currencyconverter\ejb\dist

default-not-ear:

default:
Trying to override old definition of task iterate

-pre-init:

init:
```

Scanned by CamScanner

```
default-ear:

-pre-compile:

-post-compile:

compile:

package:
     [mkdir] Created dir: C:\Java Codes\currencyconverter\webclient\build\war
     [mkdir] Created dir: C:\Java Codes\currencyconverter\webclient\build\web
      [copy] Copying 2 files to C:\Java Codes\currencyconverter\webclient\build\web
      [copy] Copying 2 files to C:\Java Codes\currencyconverter\webclient\build\war
       [jar] Building jar: C:\Java Codes\currencyconverter\webclient\build\webclient.war
    [delete] Deleting directory C:\Java Codes\currencyconverter\webclient\build\war

copy-dist:
     [mkdir] Created dir: C:\Java Codes\currencyconverter\webclient\dist
      [copy] Copying 1 file to C:\Java Codes\currencyconverter\webclient\dist

default-not-ear:

default:

-pre-compile:

-post-compile:

compile:

package:
     [mkdir] Created dir: C:\Java Codes\currencyconverter\build\ear
     [mkdir] Created dir: C:\Java Codes\currencyconverter\build\ear\META-INF
      [copy] Copying 1 file to C:\Java Codes\currencyconverter\build\ear\META-INF

copy-dist:
      [copy] Copying 1 file to C:\Java Codes\currencyconverter\build\ear
Trying to override old definition of task iterate

copy-dist:
      [copy] Copying 1 file to C:\Java Codes\currencyconverter\build\ear
       [jar] Building jar: C:\Java Codes\currencyconverter\build\currencyconverter.ear
    [delete] Deleting directory C:\Java Codes\currencyconverter\build\ear

copy-dist:
     [mkdir] Created dir: C:\Java Codes\currencyconverter\dist
      [copy] Copying 1 file to C:\Java Codes\currencyconverter\dist

default-not-ear:

default:

tools:

deploy:
      [exec] Command deploy executed successfully.
      [echo] Application Deployed at: http://localhost:8080/currencyconverter

getBrowser:

launch:
      [echo]
      [echo]
      [echo]              Trying to launch the browser with the url
      [echo]              http://localhost:8080/currencyconverter
```



```
      [echo] Launching currencyconverter.

run:

BUILD SUCCESSFUL
Total time: 6 seconds
C:\Java Codes\currencyconverter>
```

Once the above processing completes a web browser is automatically launched and the currencyconverter application is executed as shown in diagram 23.4.

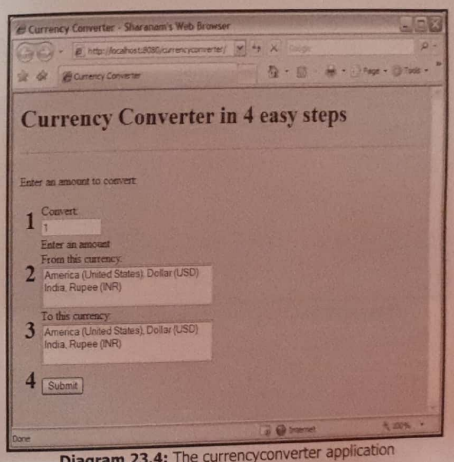

Diagram 23.4: The currencyconverter application

This form always loads with 1 as the default value in the amount text box.

As soon as the application is invoked, enter the desired value in the amount text box, select the appropriate currency from both list boxes and click Submit.

The output appears as shown in diagram 23.5.

Scanned by CamScanner

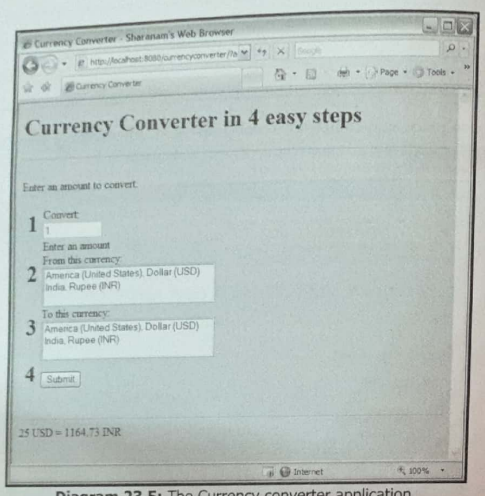

Diagram 23.5: The Currency converter application

Hands on Exercises

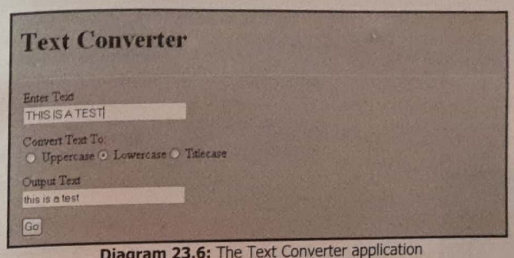

Diagram 23.6: The Text Converter application

Create & display the GUI as shown in diagram 23.6.

Now assume a user has entered some text in the "**Enter Text**" text box, as displayed.

- Assuming a convert text to radio button is selected; display the changed text in the "**Output Text**" text box.

Scanned by CamScanner