Chapter 2

An Introduction to the Java 2 Platform, Enterprise Edition

This chapter provides an introduction to the Java 2 Platform, Enterprise Edition (J2EE platform). When describing a new technology, it is always useful to provide a familiar context within which that technology is explained. To provide such a context, we start by discussing the concerns that commonly arise when building enterprise systems, and then describe the architectural solutions that have been proven to address these concerns. It is these general architectural solutions that are implemented by the J2EE platform. Following an overview of the J2EE platform, we discuss each of the J2EE technologies in detail.

Enterprise Concerns

Some development concerns are common to every project. For example, system attributes such as performance, scalability, reliability, maintainability, security, ease of incorporating new functionality, and the ability to integrate the system with existing systems are always on the list. There are also other concerns that are not directly related to the solution's properties, but have everything to do with the project's overall success. Examples include product quality, time to market, cost of development, team productivity, dependence on unique or hard-to-find skills, and dependence on a single technology or vendor.

Although we could discuss the J2EE platform in terms of satisfying such general concerns, we have chosen to concentrate on the concerns that are built on this platform. In particular, the J2EE platform has been designed to support the development and execution of a specific type of software system that

we call *enterprise systems*. Enterprise systems exhibit a number of common concerns, which are described in this section.

Business Concerns

Enterprise systems implement business processes. Thus, they must contain some representation of the "business reality" that they work within. In fact, enterprise systems are an essential element of many businesses (and in some cases, the software *is* the business) and therefore contain a rich representation of domain concepts. For example, even a simple order-processing system must keep track of products, customers, orders, inventory, and so on.

Enterprise systems, therefore, collect and process large amounts of structured information. They may manage thousands of business data types and structures, and millions of instances of these types (and associations between them). Consider an order-processing system that has thousands of customers, tens of thousands of products, and hundreds of thousands of orders (both in progress and fulfilled).

Although enterprise systems usually perform only a small number of complex computations, they do perform complex data manipulation. For example, an order-processing system doesn't extrapolate trajectories of flying objects, switch mobile phones from cell to cell, or analyze images. However, such a system is very strict about the integrity of the data it maintains and imposes specific rules about how the data is changed.

On a similar note, an enterprise system does manage complex interactions with its many concurrent users. For example, the process of placing an order requires that a customer make him- or herself known to the system, select one or more products, specify appropriate quantities, provide relevant shipping details, and so on. This complexity is made manageable by ensuring that the system imposes very specific interactions with its users.

Integration Concerns

A fundamental aspect of many enterprise systems is their integration with other systems. Even the apparently simple task of placing an order can result in electronic interactions with a warehouse system (to request delivery of the purchased products) and a bank system (to ensure that payment is made). As a result, enterprise systems often run on complex, distributed technical infrastructures that reflect the physical distribution of the organizations participating in the business processes supported by them.

Moreover, even though an enterprise system may interact with similar systems, these systems may have been implemented at different times and may use

different technologies. This is simply a consequence of the rapid pace at which business systems change in today's environment. This constant change must therefore be acknowledged and planned for.

The Internet revolution has lead many organizations to make their valuable information available to third parties through a variety of means, including the electronic exchange of information between businesses. This information often represents an organization's most valuable asset. An emphasis must therefore be placed on the security of the enterprise systems providing that information.

Development Concerns

We should also acknowledge that the development and maintenance of enterprise systems is logistically complex for all sorts of reasons. For example, the development of enterprise systems must keep pace with changing business conditions. Even gathering the requirements of the enterprise system can be a complex task when there are a number of stakeholders involved (including end users and business partners, as well as internal staff).

A related concern is that large enterprise systems are developed over long periods of time (sometimes decades), even though an initial version of the system may be made available quickly to address time-to-market concerns. We must therefore take into account the longevity of the system. Large systems are often implemented through a set of concurrent related projects, leading to systems whose elements are at different levels of articulation and maturity.

Multitier Architectures and the J2EE Platform

It has been known for some time that the best approach to developing systems is to divide their responsibilities across a number of *tiers*[1], resulting in common architectural styles known as *multitier architectures*. An example of a multitier architecture is the 3-tier structure shown in Figure 2.1. The presentation tier is

Figure 2.1 3-Tier Architecture

1 A tier is an architectural layer that has a particular responsibility.

responsible for handling interactions with the end user. The business tier is responsible for performing any business processing. The integration tier is responsible for providing access to backend resources, including databases and external systems. Such division allows the content of each tier to be developed and changed independently.

From a historical perspective, the database technology of the integration tier was the first to mature, resulting in powerful relational databases[2]. The technologies in the presentation tier and business tier matured later, resulting in user interface frameworks and transaction-processing monitors, respectively.

For many years, architects of enterprise systems took the concept of multitier architectures and the available technologies, and produced custom-made platforms for their solutions. This is, of course, a very costly and complex effort. One of the key objectives of the J2EE platform is to provide a standard environment on which to develop enterprise systems.

The J2EE platform has been influenced by many earlier initiatives. One of those was Microsoft's successful integration of technologies for all three tiers: Visual Basic, Microsoft Transaction Server (MTS), and SQL Server. Microsoft also promoted and exemplified the concept of container-based computing with its MTS, which itself was influenced by BEA's Tuxedo and other transaction-processing monitors. The concept of container-based computing is central to the J2EE platform, and allows components to execute in an environment that provides the services they require.

Another major influence on the J2EE platform has been the Internet, which has created a high demand for a class of enterprise system known as *online enterprise systems* and the technology needed to implement them. Internet technology has changed and unified the way that user interactions are supported. For example, the stateless nature of the Hypertext Transfer Protocol (HTTP), whereby a client does not have a permanent connection with the server, has implications on how state is managed between invocations.

Finally, the advent of Java and its "write once, run anywhere" (WORA) philosophy has provided a basis for tying together the many technologies required to develop and deploy enterprise systems.

So, in summary, the J2EE platform is best considered as a set of technologies for developing and deploying multitier enterprise systems. As such, it contains an expected set of services (so expected, in fact, that Microsoft's .NET initiative provides an almost identical set). We introduce these technologies in the next section.

The J2EE platform also has the desirable characteristic of being an open specification. There are many commercial and open-source implementations of

2 The foundation of relational database technology was created over two decades ago.

the specification. A free reference implementation of the J2EE platform is also available.

We now consider the specific technologies that the J2EE platform provides in supporting a multitier architecture.

J2EE Platform Overview

Let us start from putting the J2EE technologies together to show the context within which they operate, as well as the relationships between them. Figure 2.2 positions each of the technologies in terms of the tiers discussed earlier. It also shows the physical location of each of the elements in terms of client and server. The details contained in Figure 2.2[3] are discussed throughout

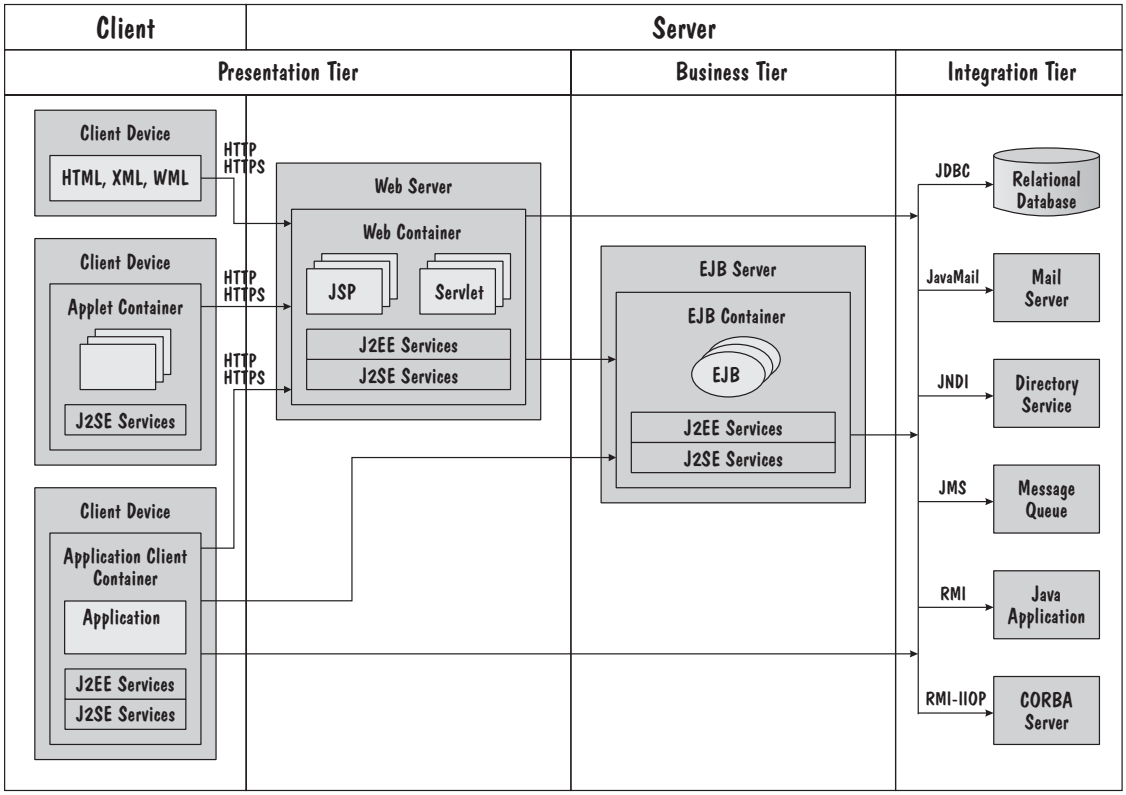

Figure 2.2 The J2EE Platform and Technologies

3 Figure 2.2 shows a multitier deployment configuration that is discussed later in this chapter.

this chapter. Although it is not shown in Figure 2.2, it is also worth mentioning that all containers explicitly use a Java virtual machine (JVM) when executing any compiled Java code.

J2EE Technology Overview

A summary of the various J2EE technologies is shown in Figure 2.3 and described in the following sections.

J2EE Application Component Technologies

The *application component technologies* are those that we use to build the components of the solution. Each of these technologies is discussed in detail later in this chapter. The J2EE application component technologies are described in the following list.

- **Applets**. An applet is primarily used to provide some form of rendering in the user interface, where performance is key.
- **Application clients**. An application client is a standalone Java application that provides an alternate means of accessing a J2EE application, other than through the use of a markup language such as Hypertext Markup Language (HTML).
- **Java Servlets ("servlets")**. A servlet defines how a request from the client is processed and how a response is generated.

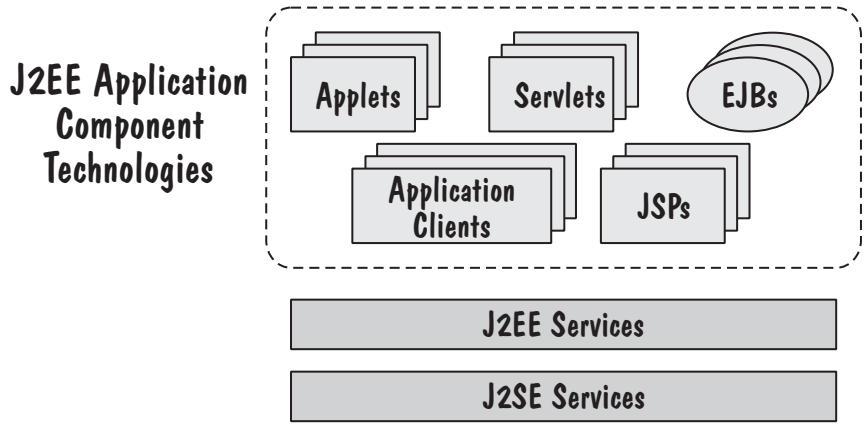

Figure 2.3 J2EE Technology Summary

- **JavaServer Pages (JSPs)**. A JSP is a text document that, like a servlet, describes how a request is processed and a response generated. JSPs provide an alternative to servlets when the generation of statements in a markup language (such as HTML) is required.
- **Enterprise JavaBeans (EJBs)**. An EJB is responsible for implementing an aspect of the business logic of a J2EE application.

J2EE Services

J2EE services, as the name implies, are the services made available by the J2EE platform. They are described in the following list.

- **Java API for XML Parsing (JAXP)**. JAXP provides a standard service that supports the parsing and manipulation of XML documents. JAXP provides a further abstraction when using an external standard such as the Simple API for XML Parsing (SAX), the Document Object Model (DOM) and eXtensible Stylesheet Language Transformations (XSLT).
- **Java DataBase Connectivity (JDBC)**. JDBC provides programmatic access to a relational database.
- **Java Message Service (JMS)**. JMS provides a standard interface to reliable asynchronous messaging implementations (such as IBM's MQSeries or Tibco's Rendezvous).
- **Java Authentication and Authorization Service (JAAS)**. JAAS allows J2EE applications to authenticate users (to reliably and securely determine who is currently executing Java code) and authorize users (to ensure that they have the permissions required to perform the necessary actions).
- **Java Transaction API (JTA)**. In circumstances where programmatic control of transactions is required, JTA provides a standard interface to the transaction services.
- **JavaMail API (JavaMail)**. The JavaMail API allows application components to send mail using a standard interface. Typical implementations of the JavaMail API interface to a number of protocols and specifications, such as the Simple Mail Transfer Protocol (SMTP), Multipurpose Internet Mail Extensions (MIME) and Post Office Protocol 3 (POP3).
- **J2EE Connector Architecture (JCA)**. One of the common requirements of an enterprise application is the ability to connect to enterprise information system (EIS) resources. Some of these resources may be external applications that are accessed using a vendor-specific protocol. JCA provides a standard means for providing resource adapters (more commonly known as "connectors").

J2SE Services

The J2EE platform is dependent upon services provided by the Java 2 Platform, Standard Edition (J2SE)[4]. The J2SE services include support for collections, internationalization (support for multiple human languages), input/output, Java Archive (JAR) files, user interfaces, math, networking, object serialization, Remote Method Invocation (RMI), security, and sound. The J2SE technologies in the following list are considered essential parts of the J2EE platform.

- **Hypertext Transfer Protocol (HTTP) API**. The HTTP API is a client-side API that supports interaction with server-side presentation tier elements using HTTP, the standard protocol for communication on the Web. In many applications, this API is not used since the use of HTTP is handled entirely by the client device and requires no programmatic involvement. For example, the submission of an HTML form to a Web server is handled entirely by the Web browser.
- **HTTPS API**. HTTPS is the use of HTTP over the Secure Socket Layer (SSL). SSL is a security protocol used by Web servers and client devices (such as a Web browser) to establish a secure communication channel over HTTP. This API is the secure equivalent of the HTTP API.
- **Remote Method Invocation over Internet Inter-Orb Protocol (RMI-IIOP)**. RMI is a Java standard for providing distributed object communication between two Java objects. In order to provide maximum interoperability between elements that may not be written in Java (such as an EJB container), the J2EE platform stipulates that the language-independent Internet Inter-Orb Protocol (IIOP) be used. IIOP is an element of the CORBA (Common Object Request Broker Architecture) standard that is defined by the Object Management Group (OMG).
- **Java Naming and Directory Interface (JNDI)**. JNDI provides a uniform interface to a number of directory and naming services, which support the locating of resources on a network. For example, JNDI can be used to obtain a reference to the home interface of a remote EJB.

Containers

The concept of a container is central to the J2EE platform. A container provides runtime support for application components (such as JSPs, servlets, or EJBs) that execute within it. For example, an EJB container provides component life cycle management (the creation and removal of application components, as

4 A third Java edition, Java 2 Micro Edition (J2ME), is not a prerequisite of J2EE.

required), transaction management, security, and persistence support to the EJBs that execute within it.

The fact that a component executes inside a container is transparent to its clients. For example, in Figure 2.4 there is a client interacting with an EJB. The client request is handled by the EJB container housing the EJB, rather than directly by the EJB itself. Based upon configuration information held in a deployment descriptor associated with the EJB, the container can interject appropriate transaction characteristics before passing the request on to the EJB for processing.

There is a two-way contract between a container and an application component. From the perspective of the container, the application component must conform to certain interfaces so that the container can manage the component appropriately. For example, an EJB must provide operations to support its removal from memory (known as "passivation"). From the perspective of the application component, the container must make certain services available to the component. For example, an EJB container must provide the Java DataBase Connectivity (JDBC) API.

As shown in Figure 2.2, there are four types of containers.

- An *applet container,* which provides services required by Java applets
- An *application client container,* which provides services required by Java application clients
- A *Web container,* which provides services to JSPs and servlets
- An *EJB container,* which provides services to EJBs

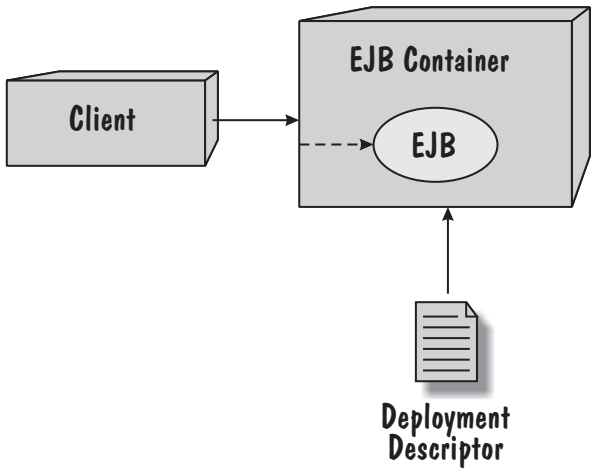

Figure 2.4 An EJB Container

Containers and Servers

In Figure 2.2, we see that a Web container executes inside a Web server, and an EJB container executes inside an EJB server. Some J2EE platform implementations also include the concept of a J2EE application server (or simply "J2EE server"), which includes both a Web container and an EJB container.

Containers and servers are considered to be logical concepts, and the J2EE platform specification does not state how they should be implemented. As a result, a server can be interpreted as the pool of resources, such as operating system processes and memory, which the container implementers can use as they see fit. Therefore, different J2EE platform implementations have taken different approaches to implementing containers, while trying to improve their scalability and reliability properties. For example, load-balancing containers can run in several processes potentially distributed across a number of machines.

Presentation Tier

The presentation tier contains elements that reside on both the client and the server.

The client-side elements are responsible for rendering the user interface and for handling user interactions. In Figure 2.2, we see three clients, each executing in its own device. The first client is processing a markup language. Examples of such a client include a Web browser that processes HTML, an XML-aware device that processes XML, and a Wireless Access Protocol (WAP) device, such as a mobile phone, that processes Wireless Markup Language (WML). The second client houses an applet container that supports the execution of *applets*. An applet (discussed in detail later) is a Java program that typically provides some form of high-performance user interface rendering. The third client houses an application client container that supports the execution of a J2EE application client. A J2EE application client (discussed in detail later) is a standalone Java application that typically provides access to elements in the business tier and integration tier. For example, an application client may be used to provide an administrative interface to the J2EE application.

The server-side elements are responsible for processing client-side requests and providing appropriate responses. A response is typically delivered to the client in the form of a markup language, such as HTML, XML or WML. The response is often dependent on the data held by an EJB or an enterprise information system (EIS), such as a mainframe transaction-processing system or a legacy database. Therefore, the presentation tier application components on the server (the JSPs and the servlets) interact with the components in the business tier or directly with the integration tier. These elements may also be responsible for aspects of user session management, data validation, and application control logic.

Business Tier

The business tier is responsible for an application's business logic. In the most common case, business tier components (EJBs) provide business logic services to the server-side presentation tier application components. However, they can also provide services to standalone Java application clients. EJBs and EJB containers have been designed to simplify communication between the presentation tier and the integration tier.

Integration Tier

The integration tier is responsible for providing access to EIS resources. Figure 2.2 identifies specific types of EIS resources, including a relational database, mail server, directory service, message queue, Java application, and CORBA server. We have labeled each connection to an EIS resource with the technology used to access that resource. For example, JDBC is used to access a relational database.

J2EE Deployment Configurations

A deployment configuration is a mapping of application functionality to application components and then to J2EE containers and services. In other words, it is a way of structuring and distributing the application functionality between tiers, containers, and components. Although there are a number of deployment configurations, there are a few common structures. We briefly look at these common structures in the following sections and discuss the pros and cons of each.

Standalone Deployment Configuration

In the standalone deployment configuration, shown in Figure 2.5, there is not a Web container or an EJB container. The client accesses EIS resources directly and is responsible for handling all presentation logic, business logic, and integration logic.

This configuration may seem like an attractive proposition for applications that provide simple manipulation of data held in the EIS resources. However, this configuration has a number of drawbacks.

Changes to the EIS resource can have a major impact on the implementation of the client, because it is directly dependent on the internal structure of that EIS resource (such as the structure of database tables). In addition, any change to the application itself requires a rollout to every user.

Also, the configuration does not encourage a division of responsibility. For example, often presentation logic and business logic are tightly coupled, making it difficult to support application evolution and maintenance.

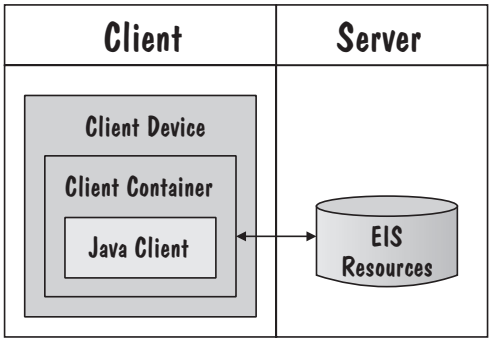

Figure 2.5 Standalone Deployment Configuration

However, the real issues with this deployment configuration start to surface when we want to scale the application to support a large number of concurrent users. When we attempt to provide concurrent client access to an EIS resource, we may find that we are constrained by the EIS resource itself. For example, a database may limit the number of concurrent database connections. However, since there is no coordinated access to the EIS resource, it is not possible to provide an efficient access mechanism (such as a managed pool of database connections).

EJB-Centric Deployment Configuration

In the EJB-centric deployment configuration, shown in Figure 2.6, there is no Web container, and an EJB container sits between the client container and the EIS resources. The presentation logic is in the client, with business logic residing in the EJBs on the server.

Rather than accessing EIS resources directly, all requests from the clients are managed by the appropriate EJBs. Clients are therefore shielded from changes in EIS resources (unless the extent of the change requires additional information to be supplied by the client, for example).

The EJB-centric deployment configuration is designed to address a number of the issues present in the standalone deployment configuration. From a scalability perspective, an EJB container is responsible for ensuring efficient use of limited resources, such as database connections. From an application evolution and maintenance perspective, this configuration also encourages a separation of presentation logic and business logic.

However, one of the drawbacks of the EJB-centric deployment configuration is that any change to the presentation logic requires a rollout to every client.

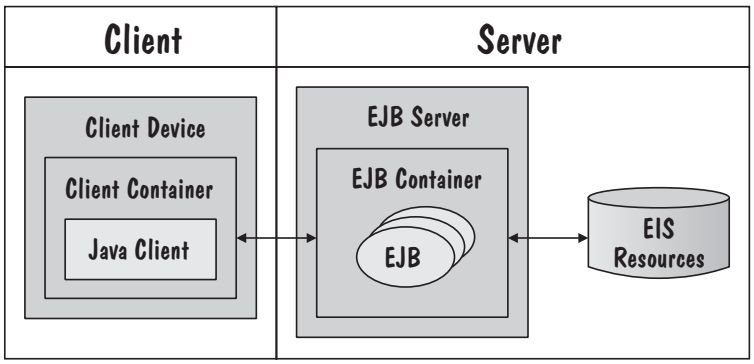

Figure 2.6 EJB-Centric Deployment Configuration

Web-Centric Deployment Configuration

In the Web-centric deployment configuration, shown in Figure 2.7, a Web container sits between the client and the EIS resources, and there is no EJB container. Both presentation logic and business logic are handled by components in the Web container. A Web-centric deployment configuration typically results in an emphasis on the look and feel of the application, with less emphasis on supporting the business logic.

This configuration provides a number of benefits. Clients aren't affected by changes to EIS resources, since clients don't access these resources directly (again, unless the extent of the change requires additional information to be supplied by the client, for example). It is also easier to redeploy the entire application, since all of the application logic resides on the server.

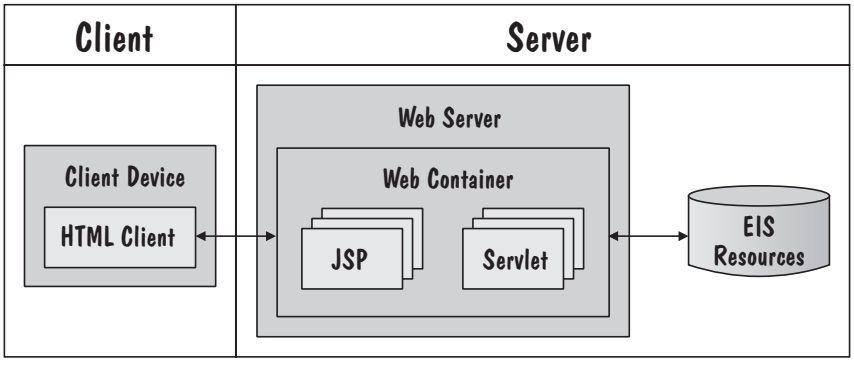

Figure 2.7 Web-Centric Deployment Configuration

However, although the use of EJBs is sometimes considered to be overkill, the omission of EJBs results in some of the same issues raised for the standalone deployment configuration. Specifically, this configuration does not encourage a clear division of responsibility between presentation logic and business logic, often resulting in tightly coupled elements that impede application evolution and maintenance. Also, from a scalability perspective, it is the developer's responsibility to ensure the efficient use of limited resources, such as database connections.

Multitier Deployment Configuration

The multitier deployment configuration is shown in Figure 2.8 and was also shown in Figure 2.2. This configuration includes both a Web container and an EJB container. Presentation logic is handled by elements in the Web container, with business logic handled by EJBs in the EJB container.

In this configuration, clients aren't affected by changes to EIS resources since these resources aren't accessed directly by the clients (again, unless the extent of the change requires additional information to be supplied by the client, for example).

It is also possible to redeploy the entire application without requiring any rollout to clients, since the application resides wholly on the server. From a scalability perspective, the EJB container is responsible for ensuring efficient use of limited resources, such as database connections.

From an application evolution and maintenance perspective, this configuration encourages a clean separation of responsibilities. The presentation logic is decoupled from EIS resources, and the business logic is decoupled from the look and feel. This separation helps when allocating work to developers with

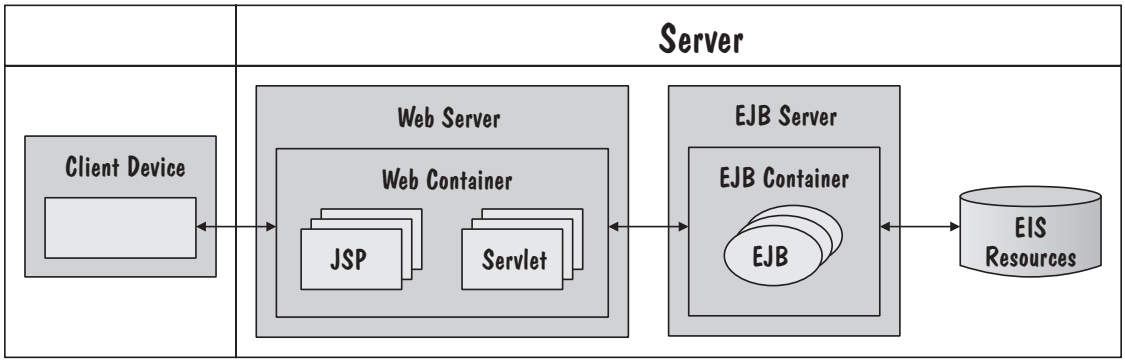

Figure 2.8 Multitier Deployment Configuration

different skills. It also allows for concurrent development, testing, and deployment of presentation logic and business logic elements. The decoupling of presentation logic and business logic also increases the reuse potential of the business logic elements.

The multitier deployment configuration can also ease the migration from one client device (such as a Web browser) to another (such as a PDA). A complete rewrite of the application isn't required since the business logic encapsulated in the EJBs remains unchanged.

To summarize, there are a number of deployment configurations, each with its pros and cons. One of the objectives of the J2EE platform is to be flexible enough to support whatever configuration best fits an organization, while addressing the development concerns we discussed earlier in this chapter.

J2EE Component Technologies

Let's now take a closer look at the J2EE application component technologies.

Applets

A Java applet is a Java program that executes within an applet container that is contained within a client device, such as a Web browser. An applet is primarily used to provide some form of rendering in the user interface where performance is key. For example, supporting graphical manipulation in a Web browser may be best achieved by using an applet. An applet is specified using the OBJECT tag in HTML, as shown in the code fragment below.

```
<html>
  <body>
    <object codetype="application/java" code="TestApplet.class"
width=300 height=100>
    ....
    </object>
  </body>
</html>
```

This tag tells the browser to load the applet whose compiled code is in the file TestApplet.class. The code fragment below is the source code of the applet. This code shows the paint method that is invoked whenever the applet must repaint itself.

```
import java.applet.Applet;
import java.awt.Graphics;
```

```
public class TestApplet extends Applet
{
  public void paint(Graphics g)
  {
    ....
  }
}
```

Application Clients

An application client is a standalone Java application that can contain presentation logic, business logic, and integration logic, and as a result is sometimes referred to as a "fat client". In order to perform its processing, an application client may access the server-side elements of the presentation tier, the elements of the business tier, and the elements of the integration tier. Application clients are often used where a more sophisticated user interface is required than can be provided using a markup language such as HTML. An application client may be used to provide an administration interface to a J2EE application.

Java Servlets

A servlet is a Java class that is used to implement presentation logic on the server. A servlet defines the way in which a request is processed and the way in which a response is generated. Servlets are often accessed directly from a client device, such as a Web browser, either by using a URL or through the use of an HTML form, as shown in the HTML fragment below. When the form represented by this HTML code is submitted to the Web server for processing, the Web server identifies the target servlet, based on the name specified in the form's "action" attribute; identifies the appropriate servlet method, based on the form's "method" attribute; constructs an appropriate request; and invokes the servlet method, passing the request as an argument.

```
<html>
  <body>
    <form method=post action="/auction/main">
      ....
    </form>
  </body>
</html>
```

The code fragment below is the source code of the servlet used in the example above. This code shows aspects of the implementation of the doPost method that is invoked by the Web server when the HTML form is submitted. This method takes two parameters. The first parameter is an HttpServletRequest, which provides the content of the request. The second parameter is

an HttpServletResponse, which is used to return the response. In implementing the required presentation logic, a servlet often interacts with other servlets, EJBs, and JSPs. Specific design patterns that describe such interactions are discussed in Chapter 8, Design.

```
package com.pearlcircle;

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

public class PresentationRequestController extends HttpServlet
{
  public void doPost(HttpServletRequest req, HttpServletResponse resp)
                    throws ServletException, IOException
  {
    ....
  }
}
```

It is often necessary for a servlet to produce output that can be rendered in the client device, such as HTML (if this is the output expected). This requires writing the doPost method so that it places HTML in the HttpServletResponse object. However, when it comes to Web page layout, the J2EE platform provides an alternative technology, JavaServer Pages (JSP), to render the markup language required.

JavaServer Pages (JSP)

A JSP is a text document that, like a servlet, describes how a request is processed and a response is generated. A JSP is often accessed directly from a client device, such as a Web browser, using a URL. For example, accessing the URL http://www.pearlcircle.com/utils/getServerDate.jsp will result in the Web server executing the getServerData.jsp file and returning the response generated by this JSP. One way to think about JSPs is that they provide an alternative to servlets for generating statements in a markup language. Internally, the Web server automatically compiles JSPs into servlets before they are executed. This does raise the question of when to use servlets and when to use JSPs, since they can provide equivalent functionality. This is a design decision that is touched upon in Chapter 8.

The content of a simple JSP file that returns the date associated with the machine on which the Web server is executing is shown below. We can see that the JSP contains two types of statements. The first type of statement is a markup language to be returned in the response. In the example, the markup language is HTML. The second type of statement is a command language that

supports the generation of dynamic content when the JSP file is executed. In the example, the text *new Date().toString()* is a command that will create a new Java Date object, and return its current value as a string. All command statements are enclosed within "<% … %>" pairs. The statement *@page import="java.util.Date"* is included to declare the location of the Date class.

```
<%@page import="java.util.Date"%>
<html>
  <body>
    <h2>Web server information</h2>
    <table border=1>
      <tr>
        <td>Date:</td>
        <td><%= new Date().toString()%></td>
      </tr>
    </table>
  </body>
</html>
```

When this JSP file is executed by the Web server, it produces output similar to that shown below. All of the HTML statements are placed in the response as-is. However, the command statements in the original JSP file are executed and the result included in the response. In particular, we can see that executing the statement *new Date().toString()* produced the value "Wed Mar 03 17:02:50 GMT+00:00 2002".

```
<html>
  <body>
    <h2>Web server information</h2>
    <table border=1>
      <tr>
        <td>Date:</td>
        <td>Wed Mar 03 17:02:50 GMT+00:00 2002</td>
      </tr>
    </table>
  </body>
</html>
```

When rendered, this HTML produces the result shown in Figure 2.9.

JSP technology encourages an interface-based contract between the provider of the JSP pages and the provider of any application components used by the JSP pages (which may be EJBs, as well as simple Java classes like the Date class used in the above example). This division of responsibility is something that we shall revisit later in this book when we discuss user-experience modeling in Chapter 7, Analysis.

Figure 2.9 The Output from a JSP Rendered in a Browser

Enterprise JavaBeans (EJB)

Enterprise JavaBeans (EJBs) reside in the business tier and are typically responsible for implementing the business logic of J2EE applications. In this section we discuss the different types of EJBs and the interfaces and classes that constitute an EJB.

A client's access to an EJB is provided through interfaces. The interfaces provided by an EJB are dependent on the manner in which the EJB is intended to be invoked. An EJB can offer interfaces that allow the EJB to be invoked remotely (known as the *home interface* and the *remote interface*) or interfaces that allow the EJB to be invoked locally (known as the *local home interface* and the *local interface*)[5].

Figure 2.10 shows a client accessing the home interface and remote interface of an EJB. These interfaces are considered remote from the perspective of the client in that the EJB providing these interfaces may physically reside in a different JVM than the client (possibly on another machine). Figure 2.10 also shows the single bean class that implements the operations defined in the EJB

5 Local interfaces were introduced in EJB 2.0, which is part of J2EE 1.3.

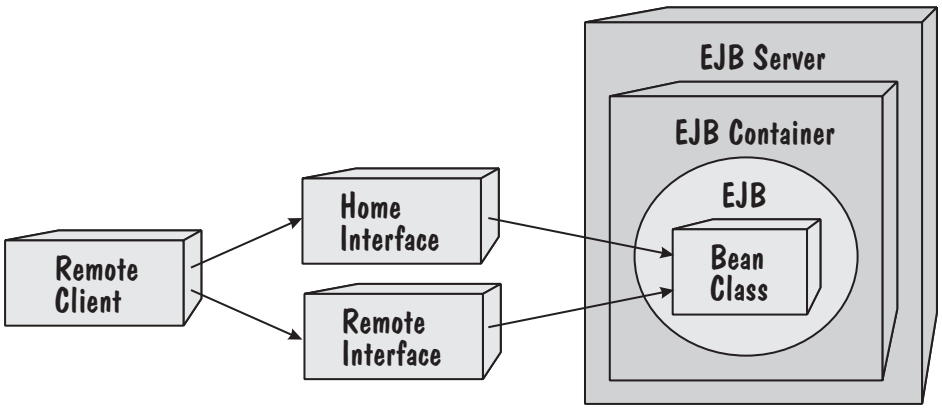

Figure 2.10 An EJB Exposing Home and Remote Interfaces

interfaces (even though an EJB may be implemented by any number of classes). This class is an internal implementation class that is not directly accessed by client objects (it is invoked indirectly by the EJB container).

Figure 2.11 shows a client accessing the local home interface and local interface of an EJB. These interfaces are considered local from the perspective of the client in that the EJB providing these interfaces always resides in the same JVM as the client.

One of the advantages of using remote interfaces is that the application developer need not be concerned with the physical location of the target EJB. However, one of the disadvantages is that, even though the target EJB may reside in the same JVM, there is an overhead in treating it as being *potentially* remote.

Local interfaces are used in situations where *co-location* of source and target is both required and known. Use of these interfaces allows the EJB container to optimize the messaging. One of the disadvantages of local interfaces is that it is the responsibility of the client to determine whether the target EJB should be treated as local or potentially remote.

Some guidance is therefore required in making the decision of whether to use local or remote interfaces. This is briefly discussed in Chapter 8.

Home Interface

The home interface of an EJB declares operations that pertain to the management of the elements represented by the EJB. For example, there are typically operations to create, remove, and find these elements. Consider the code fragment below, where we see the home interface of a UserAccount EJB. The create operation supports the creation of a new UserAccount.

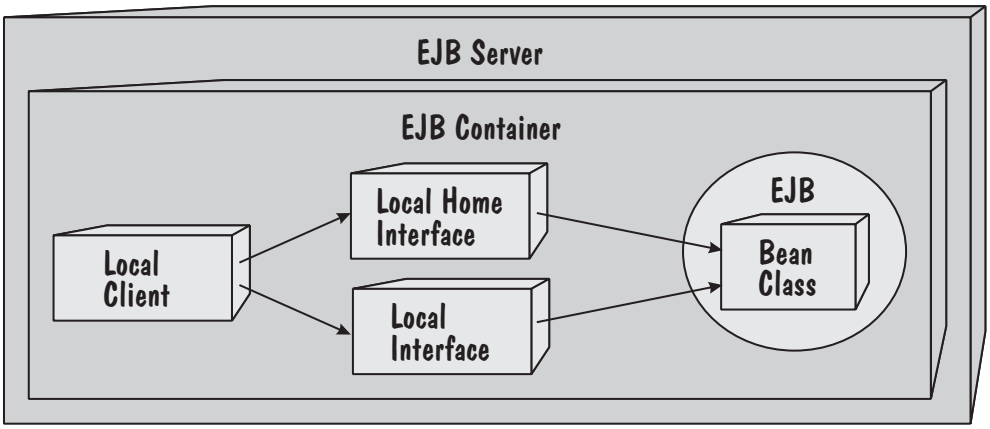

Figure 2.11 An EJB Exposing Local Home and Local Interfaces

```
package com.pearlcircle;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;
....

public interface UserAccountHome extends EJBHome
{
  public UserAccount create() throws CreateException, RemoteExcep-
tion;
  ....
}
```

Remote Interface

The remote interface of an EJB declares business operations supported by the EJB. In the code fragment below, we see an operation to set the password of a UserAccount.

```
package com.pearlcircle;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;
....

public interface UserAccount extends EJBObject
{
  public void setPassword(String password) throws RemoteException;
  ....
}
```

Local Home Interface

Should the UserAccount EJB support local interfaces rather than remote interfaces, then the definition of the local home interface would be as shown in the code fragment below. As you can see, the definition of the local home interface is identical to that of a remote home interface, with the exception that the interface extends EJBLocalHome rather than EJBHome, and that RemoteException is not thrown. The convention used here is to prefix the interface name with the word "Local".

```
package com.pearlcircle;

import javax.ejb.CreateException;
import javax.ejb.EJBLocalHome;
....

public interface LocalUserAccountHome extends EJBLocalHome
{
  public LocalUserAccount create() throws CreateException;
  ....
}
```

Local Interface

Likewise, the definition of the local interface is identical to that of the remote interface, with the exception that the interface extends EJBLocalObject rather than EJBObject, and that RemoteException is not thrown. Again, the convention used here is to prefix the interface name with the word "Local".

```
package com.pearlcircle;

import javax.ejb.EJBLocalObject;
....

public interface LocalUserAccount extends EJBLocalObject
{
  public void setPassword(String password);
  ....
}
```

Bean Class

A code fragment of the bean class for the UserAccount EJB (irrespective of whether it supports remote or local interfaces) is shown below.

```
package com.pearlcircle;

import javax.util.*;
import javax.ejb.*;
```

```
....

public abstract class UserAccountBean implements EntityBean
{
  // Instance variables
  private String password;
  ....

  // Business operations
  public void setPassword(String password) { this.password = pass-
word; }
  public String getPassword() { return password; }
  ....

  // Container operations
  public void ejbCreate() throws CreateException { .... }
  public void ejbRemove()                        { .... }
  public void ejbActivate()                      { .... }
  public void ejbPassivate()                     { .... }
  ....
}
```

This example has been kept deliberately simple to help us concentrate on specific EJB features. You can see that the bean class contains instance variables that represent the state of the objects implemented by this EJB. You can also see that the bean class implements the business operations defined in the remote interface (or local interface), such as setPassword. Finally, you can see that the bean class implements operations that are required as part of the two-way contract between the bean and the container. For example, when a client invokes the create operation on the home interface (or local home interface), this eventually results in the container calling the ejbCreate method of the bean class.

There are three distinct "flavors" of EJBs: session beans, entity beans, and message-driven beans.

Session Beans

Session beans, as the name suggests, are beans whose state is valid in the context of a "user session." For example, if you were to access a Web site that provided a "shopping cart" capability then, in most circumstances, the content of the cart would be "lost" were you to exit the site before placing the order. This occurs because the content of the cart is not, in this scenario, maintained beyond the life of the user session[6]. The content of the cart is often referred to as "conversational state," since it is available during the "conversation" the user has

6 Depending on the J2EE platform implementation, the state may reside in memory or on disk.

with the Web site. The J2EE platform specifies two types of session beans: stateless session beans and stateful session beans.

A **stateless session bean** is intended to be very lightweight, in that it maintains no conversational state whatsoever. Stateless session beans are often used as "controllers" that coordinate a series of interactions between other EJBs, but don't actually maintain any state of their own. A good example would be a stateless session bean that handles the checkout of the shopping cart we've just mentioned. In implementing the checkout process, the session bean determines the items in the cart, ships the items, debits the bank account of the buyer, credits the bank account of the seller, and then empties the shopping cart. Although the stateless session bean may store intermediate values (in program variables) during the execution of the checkout operation, it does not maintain these values outside of this operation. This is why it is called "stateless."

A **stateful session bean**, on the other hand, does maintain the state between one invocation and the next, within the context of the user session. For example, a shopping cart could be implemented as a stateful session bean[7]. Another common example of conversational state is login information, such as username and password. If this information weren't maintained with the session, then the user would have to log in with every request made. Hence, a stateful session bean could be used in this circumstance also.

Entity Beans

Entity beans represent coarse-grained elements that are considered to be multiuser and generally long-lived. They are, therefore, provided with support for persistence. Examples of entity beans are Customer, Order, and Product.

An entity bean has an associated *primary key class*. This is a Java class that is used to represent the primary key of the entity and may be a user-defined class[8]. For example, if we were implementing a Product entity bean, then we might have a ProductPrimaryKey class that holds the product manufacturer and product model as attributes, since these two attributes are what make a product unique. This class is used, for example, as a parameter to the findByPrimaryKey method on a home interface. The J2EE platform specifies two types of entity beans: container-managed persistence (CMP) entity beans and bean-managed persistence (BMP) entity beans.

---

7 HttpSession objects, available to servlets, provide an alternative mechanism for managing conversational state. The choice is discussed briefly in Chapter 8, Design.

8 Any Java class that implements the Serializable interface can be used as a primary key class.

A **CMP entity bean** is one that delegates the storage and retrieval of its persistent attributes to the EJB container. This is only possible if the container knows which attributes are to be made persistent. This is specified in the deployment descriptor associated with the entity bean (deployment descriptors are discussed later in this chapter). The deployment descriptor can also specify the relationships that a CMP entity bean has with other local CMP entity beans. If specified, the EJB container will manage such relationships. The J2EE platform also defines an *EJB Query Language (EJB QL)* that a developer uses to specify the queries used by a CMP entity bean within its finder methods[9].

A **BMP entity bean** is one that handles its own persistence, rather than delegating this responsibility to the EJB container. BMP entity beans are typically used in situations where the persistence facilities available to CMP entity beans are insufficient. A developer is therefore responsible for writing a certain amount of database access code when creating a BMP entity bean, and will make use of the Java DataBase Connectivity (JDBC) API.

Message-Driven Beans

In addition to session beans and entity beans, the J2EE platform specifies message-driven beans[10]. Message-driven beans are designed to support asynchronous communication. A client sending messages to message-driven beans does not block waiting for a response after sending a message.

A client of a message-driven bean uses the Java Message Service (JMS) to deliver a message to either a queue or a topic. A *queue* represents a list of messages that are processed by a single message-driven bean, whereas a *topic* represents a list of messages that are processed by potentially many message-driven beans. From the perspective of a client, message-driven beans are anonymous and have no client-visible identity (all interactions occur via queues and topics).

A message-driven bean is automatically instantiated by the container within which it resides and consumes messages from a JMS destination (a message queue or topic), as shown in Figure 2.12.

A code fragment of the bean class for a CloseAuction EJB (that e-mails the buyer and seller when an auction closes) is shown below. We can see from this example that the core of the bean implementation is the onMessage operation that is invoked by the EJB container when a message is received on the queue

9 EJB QL was introduced in EJB 2.0, which is part of J2EE 1.3.

10 Message-driven beans were introduced in EJB 2.0, which is part of J2EE 1.3.

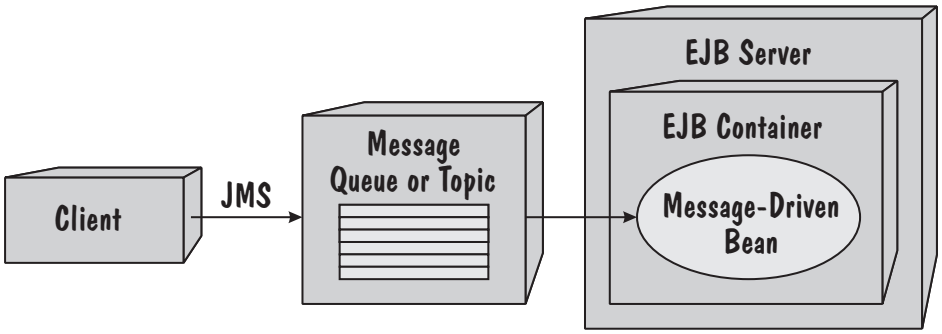

Figure 2.12 Interactions Involving a Message-Driven Bean

or topic associated with this bean (this association is set up when the bean is deployed).

```
package com.pearlcircle;

import javax.ejb.*;
import javax.jms.*;
....

public class CloseAuctionBean implements MessageDrivenBean, Message-
Listener
{
  // Process a message
  public void onMessage(Message msg)
  {
    ....
  }

  // Container operations
  public void ejbCreate() throws CreateException { .... }
  public void ejbRemove()                        { .... }
  ....
}
```

Assembly and Deployment

In this section, we discuss various aspects of J2EE Assembly and Deployment, specifically J2EE Modules.

J2EE Modules

An overview of J2EE modules is provided in Figure 2.13.

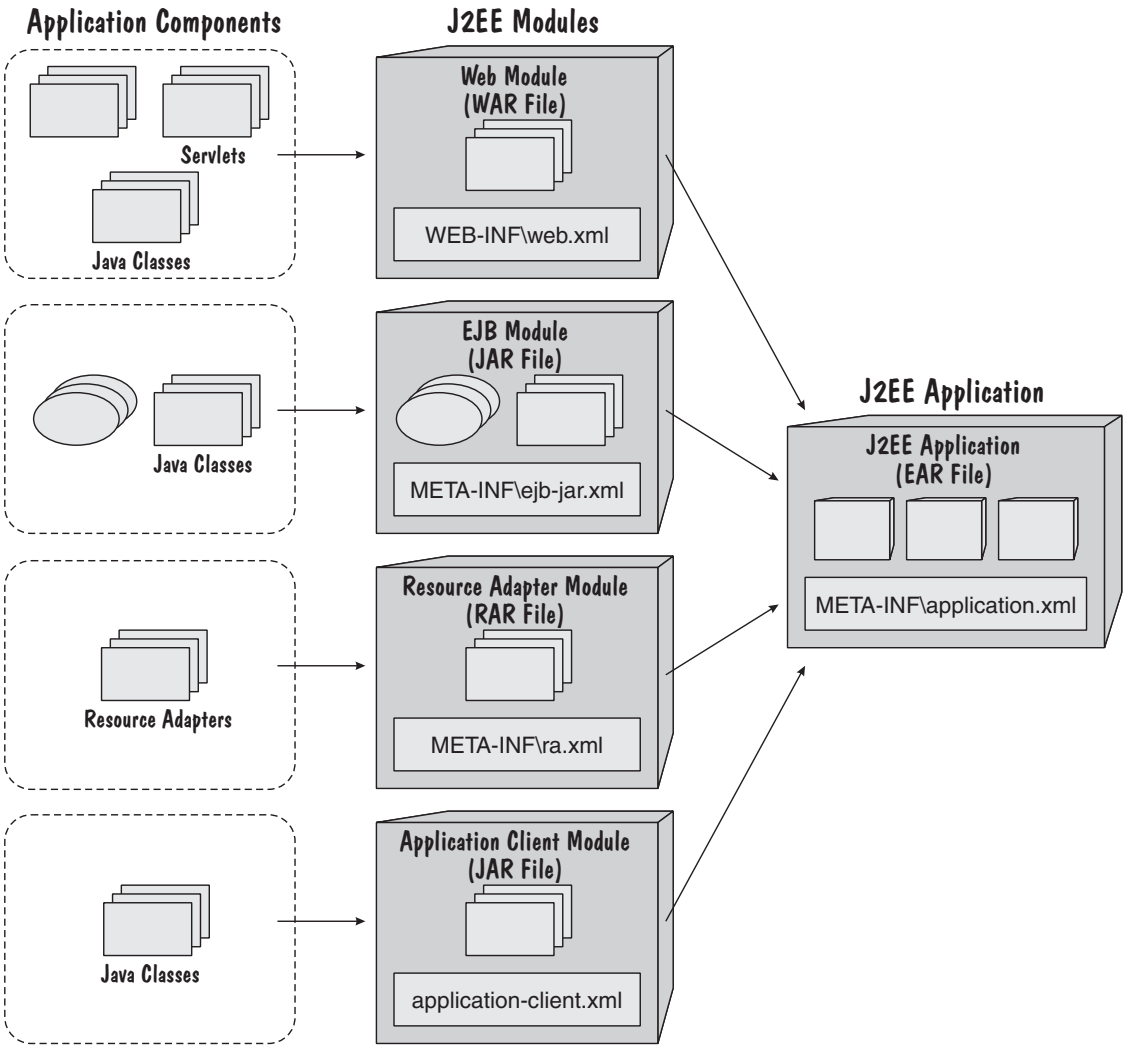

Figure 2.13 J2EE Module Overview

After their initial creation, all application components (such as JSPs, servlets, and EJBs) are packaged within a J2EE module, which is physically represented as a Java Archive (JAR) file. A JAR file contains one or more files, usually in a compressed form (much like ZIP files). The use of JAR files allows related files to be deployed as a unit. For example, a Web module is used to package the presentation tier components of a J2EE application, such as JSP files, servlets, and required Java classes.

Table 2.1 J2EE Modules

J2EE Module	*Content*	*File Type*	*Deployment Descriptor*
Web module	JSPs, servlets, image files, static HTML files, Java classes	Web Archive (WAR)	WEB-INF\web.xml
EJB module	EJBs, Java classes	Java Archive (JAR)	META-INF\ejb-jar.xml
Resource adapter module	Resource adapters	Resource adapter Archive (RAR)	META-INF\ra.xml
Application client module	Java classes	Java Archive (JAR)	application-client.xml
J2EE application module	J2EE modules	Enterprise Archive (EAR)	META-INF\application.xml

A J2EE module can be deployed as is, or it can be assembled, along with other J2EE modules, into a larger module that represents the J2EE application, which is then deployed. A summary of the different J2EE modules is shown in Table 2.1. This table includes the name of the deployment descriptor associated with each J2EE module, and its location within that module.

Every J2EE module includes a description supplied in a deployment descriptor, which is an XML file. The example below shows the elements describing a UserAccount EJB, which is a CMP entity bean. We have chosen to show the description of a single EJB, although a deployment descriptor can describe any number of items.

This descriptor introduces the UserAccount entity EJB to the container in which it will reside. It declares the bean's name, the names of the interfaces supported by this bean (and their type in terms of home, remote, local home, or local), the name of the bean class, the name of the primary key class, the persistence type (container-managed or bean-managed), and the names of all persistent fields.

```
<ejb-jar>
  <description>Online Auction</description>
  <display-name>OnlineAuction</display-name>
  <enterprise-beans>
    <entity>
      <ejb-name>UserAccount</ejb-name>
      <home>com.pearlcircle.UserAccountHome</home>
      <remote>com.pearlcircle.UserAccount</remote>
      <ejb-class>com.pearlcircle.UserAccountBean</ejb-class>
      <prim-key-class>com.pearlcircle.UserAccountPK</prim-key-class>
      <persistence-type>Container</persistence-type>
      <cmp-field><field-name>userId</field-name></cmp-field>
      <cmp-field><field-name>password</field-name></cmp-field>
```

```
      <cmp-field><field-name>firstName</field-name></cmp-field>
      <cmp-field><field-name>lastName</field-name></cmp-field>
      <cmp-field><field-name>address</field-name></cmp-field>
      <cmp-field><field-name>city</field-name></cmp-field>
      <cmp-field><field-name>state</field-name></cmp-field>
      <cmp-field><field-name>zipcode</field-name></cmp-field>
      <cmp-field><field-name>country</field-name></cmp-field>
      <cmp-field><field-name>phone</field-name></cmp-field>
      <cmp-field><field-name>email</field-name></cmp-field>
      ....
    </entity>
    ....
  </enterprise-beans>
</ejb-jar>
```

Summary

In this chapter, we have provided a brief introduction to the Java 2 Platform, Enterprise Edition.

Although the J2EE platform provides a good starting point for developing enterprise systems, it only provides a part of the solution required. In particular, the J2EE platform does not provide a complete solution since we still need to define the application logic that executes using the J2EE platform. J2EE is also a complex platform, and we need to look for ways to simplify the developers' perception of these complexities.

Both of these concerns can be addressed by following the process described in this book. Before introducing this process in Chapter 4, An Introduction to the J2EE Developer Roadmap, we discuss the foundation upon which this roadmap is based in Chapter 3, An Introduction to the Rational Unified Process.