#### 2.1.2.2 JavaServer Pages Technology

The JavaServer Pages (JSP) technology provides an extensible way to generate dynamic content for a Web client. A JSP page is a text-based document that describes how to process a request to create a response. A JSP page contains:

- Template data to format the Web document. Typically the template data uses HTML or XML elements. Document designers can edit and work with these elements on the JSP page without affecting the dynamic content. This approach simplifies development because it separates presentation from dynamic content generation.
- JSP elements and scriptlets to generate the dynamic content in the Web document. Most JSP pages use JavaBeans and/or Enterprise JavaBeans components to perform the more complex processing required of the application. Standard JSP actions can access and instantiate beans, set or retrieve bean attributes, and download applets. JSP technology is extensible through the development of custom actions, or tags, which are encapsulated in tag libraries.

#### 2.1.2.3 Web Component Containers

Web components are hosted by servlet containers, JSP containers, and Web containers. In addition to standard container services, a *servlet container* provides the network services by which requests and responses are sent. It also decodes requests and formats responses. All servlet containers must support HTTP as a protocol for requests and responses; they may also support other request-response protocols such as HTTPS. A *JSP container* provides the same services as a servlet container. Servlet and JSP containers are collectively referred to as *Web containers*.

### 2.1.3 Enterprise JavaBeans Components

The Enterprise JavaBeans architecture is a server-side technology for developing and deploying components containing the business logic of an enterprise application. Enterprise JavaBeans components, also referred to as *enterprise beans*, are

scalable, transactional, and multi-user secure. There are three types of enterprise beans: session beans, entity beans, and message-driven beans. Session and entity beans have two types of interfaces: a component interface and a home interface. The home interface defines methods to create, find, remove, and access metadata for the bean. The component interfaces define the bean's business logic methods. Message-driven beans do not have component and home interfaces.

An enterprise bean's component and home interfaces are required to be either local or remote. Remote interfaces are RMI interfaces provided to allow the clients of a bean to be location independent. Regardless of whether the client of a bean that implements a remote interface is located on the same VM or a different VM, the client uses the same API to access the bean's methods. Arguments and return results are passed by value between a client and a remote enterprise bean, and thus there is a serialization overhead.

A client of an enterprise bean that implements a local interface must be located in the same VM as the bean. Because object arguments and return results are passed by reference between a client and a local enterprise bean, there is no serialization overhead.

The following sections give an overview of enterprise beans. Enterprise beans are discussed in detail in Chapter 5.

#### 2.1.3.1 Session Beans

A *session bean* is created to provide some service on behalf of a client and usually exists only for the duration of a single client-server session. A session bean performs operations such as calculations or accessing a database for the client. While a session bean may be transactional, it is not recoverable should its container crash.

Session beans can be stateless or can maintain conversational state across methods and transactions. If they do maintain state, the EJB container manages this state if the object must be removed from memory. However, the session bean object itself must manage its own persistent data.

#### 2.1.3.2 Entity Beans

An *entity bean* is a persistent object that represents data maintained in a data store; its focus is data-centric. An entity bean is identified by a primary key. An entity bean can manage its own persistence or it can delegate this function to its container.

An entity bean can live as long as the data it represents. Persistence is handled in one of two ways:

- **Bean-managed persistence**—The developer handles persistence as part of the entity bean's source code.
- **Container-managed persistence**—The developer specifies the bean fields that need to be persistent and lets the EJB container manage persistence.

Beans with container-managed persistence are more portable across databases. In addition, entity beans with container-managed persistence can maintain relationships among themselves. This feature enables queries that join multiple database tables. With bean-managed persistence, a change in the underlying database may require the developer to change the entity bean's source code to conform to the SQL implemented by the new database.

#### 2.1.3.3 Message-Driven Beans

A message-driven bean enables asynchronous clients to access the business logic in the EJB tier. Message-driven beans are activated only by asynchronous messages received from a JMS queue to which they listen. A client does not directly access a message-driven bean; instead, a client asynchronously sends a message to a JMS queue or topic. Because message-driven beans have no need to expose their methods to clients, they do not implement component or home interfaces. They also do not maintain state on behalf of a client.

#### 2.1.3.4 EJB Component Containers

Enterprise beans are hosted by an *EJB container*. In addition to standard container services, an EJB container provides a range of transaction and persistence services and access to the J2EE service and communication APIs.

### 2.1.4 Components, Containers, and Services

The J2EE component types and their containers are illustrated in Figure 2.1.

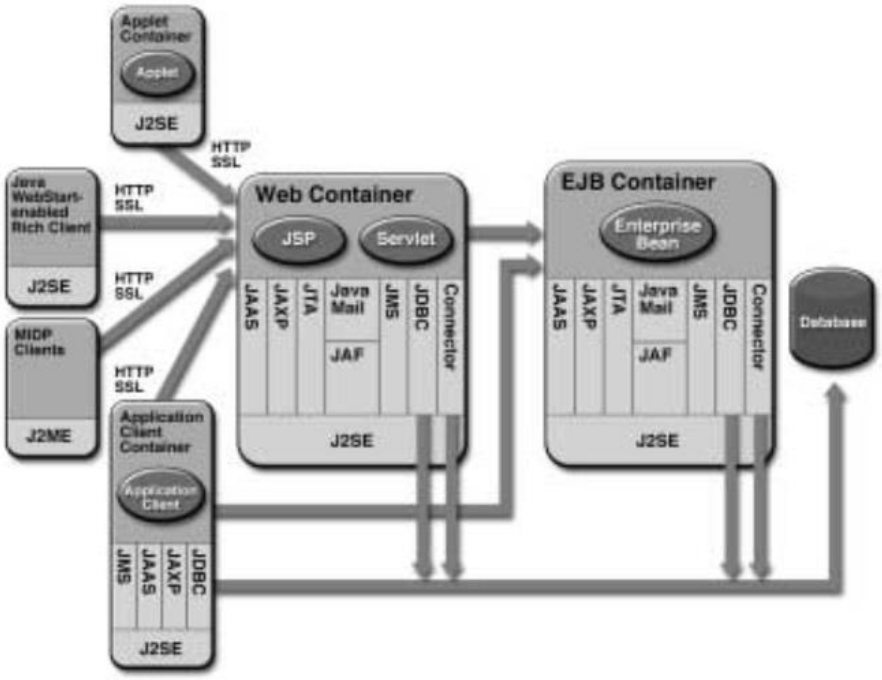


**Figure 2.1** J2EE Components and Containers

Containers provide all application components with the J2SE platform APIs, which include the Java IDL and JDBC 2.0 core enterprise APIs. Table 2.1 lists the Standard Extension APIs that are available in each type of container. The J2EE platform APIs are described in Section 2.4 on page 42 and Section 2.5 on page 45.

**Table 2.1** J2EE Required Standard Extension APIs

| API | Applet | Application Client | Web | EJB |
|---|---|---|---|---|
| JDBC 2.0 Extension | N | Y | Y | Y |
| JTA 1.0 | N | N | Y | Y |

**Table 2.1** J2EE Required Standard Extension APIs (continued)

| API | Applet | Application Client | Web | EJB |
|---|---|---|---|---|
| JNDI 1.2 | N | Y | Y | Y |
| Servlet 2.3 | N | N | Y | N |
| JSP 1.2 | N | N | Y | N |
| EJB 2.0 | N | Y[a] | Y[b] | Y |
| RMI-IIOP 1.0 | N | Y | Y | Y |
| JMS 1.0 | N | Y | Y | Y |
| JavaMail 1.2 | N | N | Y | Y |
| JAF 1.0 | N | N | Y | Y |
| JAXP 1.1 | N | Y | Y | Y |
| JAAS 1.0 | N | Y | Y | Y |
| Connector 1.0 | N | N | Y | Y |

[a] Application clients can only make use of the enterprise bean client APIs.

[b] Servlets and JSP pages can only make use of the enterprise bean client APIs.

## 2.2 Platform Roles

The J2EE platform defines several distinct roles in the application development and deployment life cycle: J2EE product provider, application component provider, application assembler, deployer, system administrator, and tool provider. In general, the roles are defined to aid in identifying the tasks performed by various parties during the development, deployment, and running of a J2EE application. However, while some of these roles, such as system administrator and tool provider, perform tasks that are common to non-J2EE platforms, other roles have a meaning specific to the J2EE platform, because the tasks those roles perform are specific to J2EE technology. In particular, application component providers, application assemblers, and deployers must configure J2EE components and applications to use J2EE platform services (described in Section 2.3 on page 35).