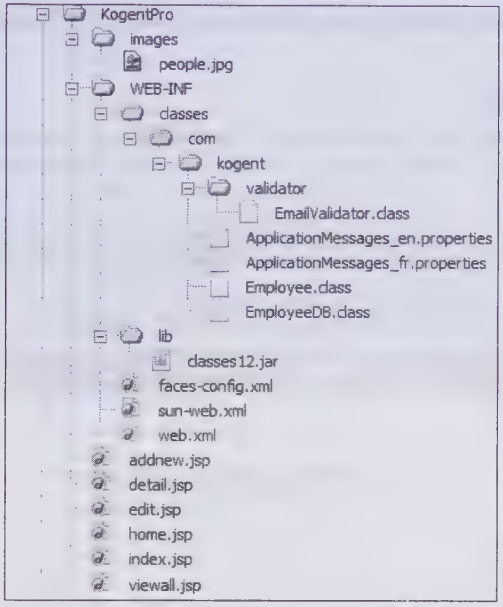


Figure 11.11: Displaying the Directory Structure of KogentPro Application

Let's now run the application.

Running the KogentPro Application

After successfully deploying the KogentPro application, you can launch the KogentPro application and access the home page, as shown in Figure 11.12:

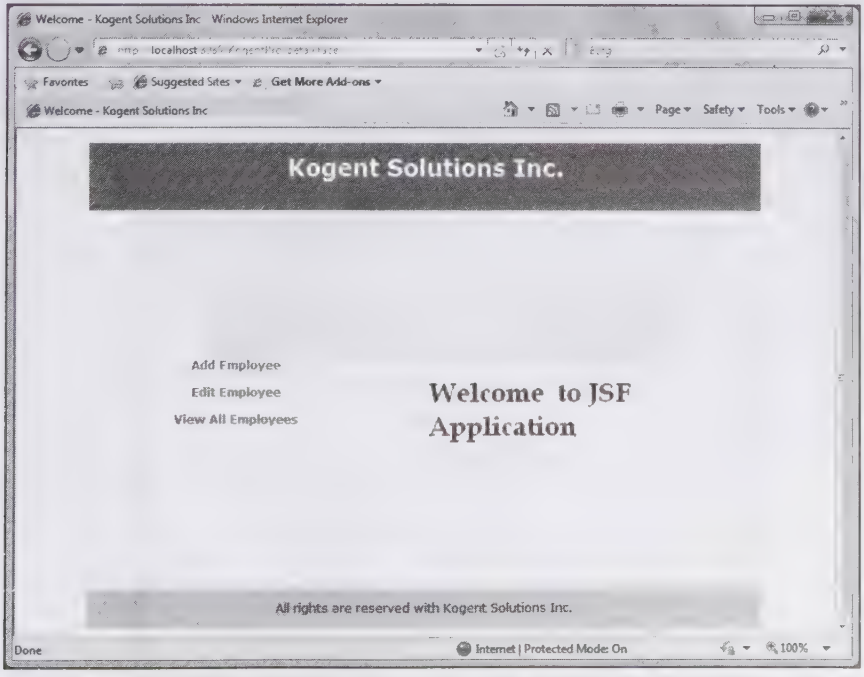


Figure 11.12: Displaying the Home Page of the KogentPro Application

You can also use the three `HtmlCommandLink` components created in the `home.jsp` page to navigate to other views.

Displaying All Employees

You can also click link `View All Employees` (Figure 11.12) to invoke the associated action method, `employee.getEmployees()`. This method returns `viewall` as the outcome string. Consequently, the navigation handler loads the `viewall.jsp` page, which displays all existing employees, as shown in Figure 11.13:

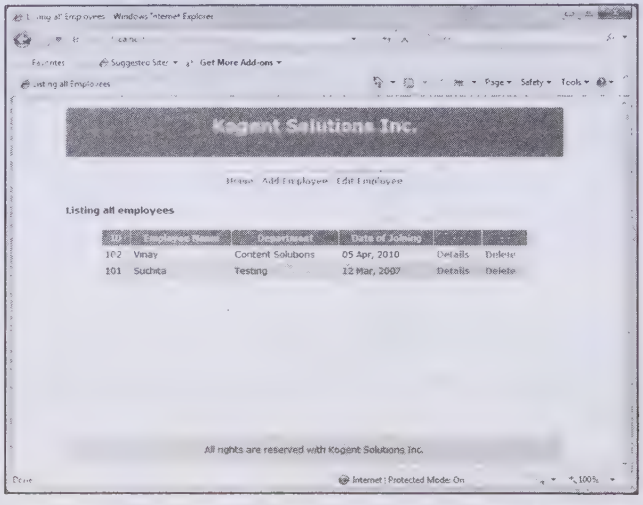


Figure 11.13: Showing the viewall.jsp Page Listing All Employees

In Figure 11.13, the `viewall.jsp` page uses an `HtmlDataTable` component, which iterates on an `ArrayList` of `Employee` objects and processes these objects to renderer rows of the table.

Getting Employee Detail

The `HtmlCommandLink` component (Figure 11.13) with the name, `Details` can be clicked to invoke the associated action method, `employee.getDetail()`. This method sets the given `employee id` in the request scope and returns `Details` as the outcome string. The output of the `details.jsp` page is Listing 11.11 shown in Figure 11.14:

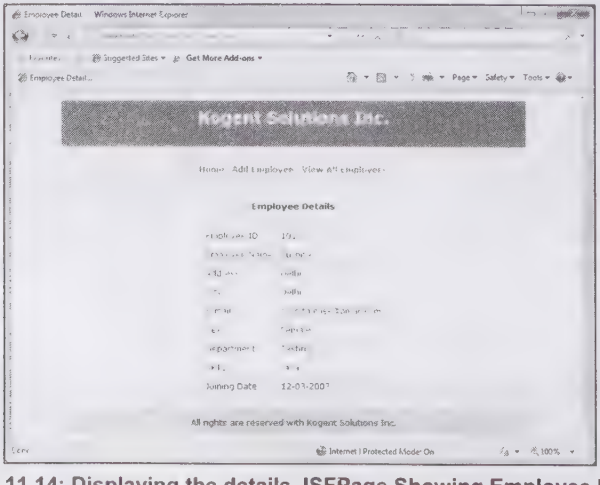


Figure 11.14: Displaying the details JSFPage Showing Employee Details

When you click the Details link of employee with id 101 (Figure 11.13), the details are displayed (Figure 11.14).

Adding New Employee

The `HtmlCommandLink` components is added on almost all pages created in the `KogentPro` application with the text, `Add Employee`, which can be clicked to navigate to the `addnew.jsp` page. This page provides a form with various input field (Figure 11.15). The `HtmlMessages` component used in the `addnew.jsp` page displays all validation and conversion error messages, as shown in Figure 11.15:

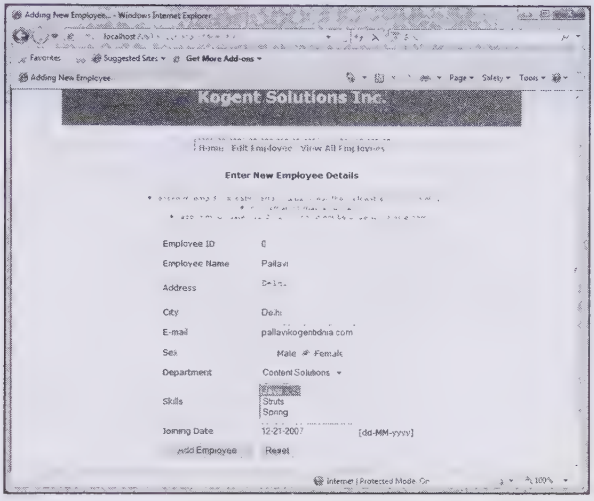


Figure 11.15: Displaying the addnew.jsp Page Showing Validation and Conversion Error Messages

You can submit the addNew form with valid employee id, valid email id, and date of joining for successful addition of new employee in the database. The successful addition of an employee displays the `viewall.jsp` page containing the details of the newly added employee.

Editing Employee Detail

You can access the `edit.jsp` page by clicking the `HtmlCommandLink` component showing the text, `Edit Employee`. The `viewall.jsp` page initially displays a form with single input field for entering employee id to be edited, as shown in Figure 11.16:

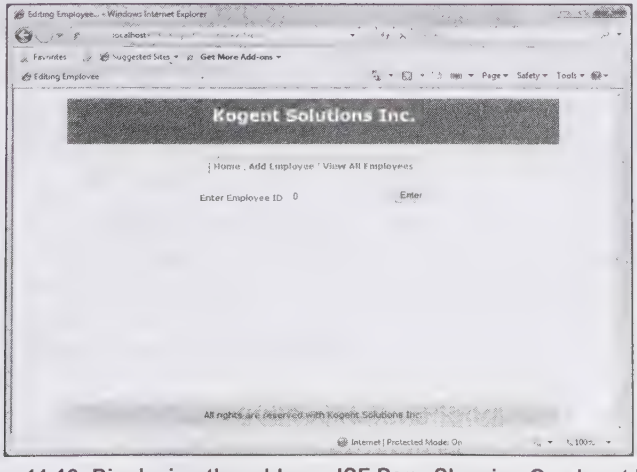


Figure 11.16: Displaying the addnew JSF Page Showing One Input Field

In Figure 11.16, we have used another `HtmlForm` component in the `edit.jsp` page, which has not been rendered, as its `render` attribute is set to `false`. The action listener method, `employee.getEmployee()` sets the `render` property of this `HtmlForm` component to `true` after setting the requested employee in the request scope. The editForm appears, as shown in Figure 11.17:

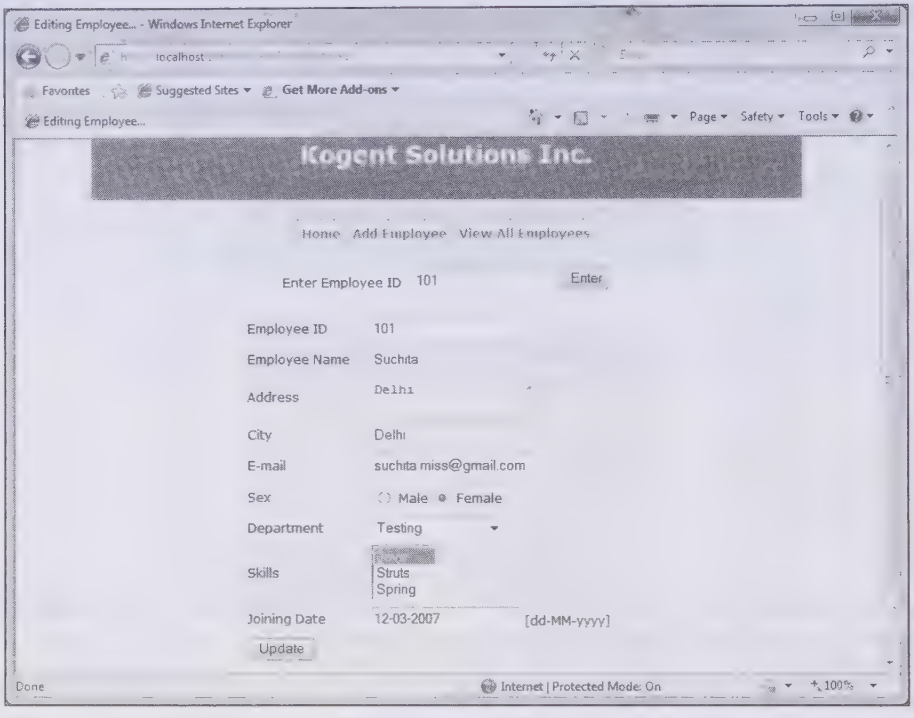


Figure 11.17: Displaying the edit.jsp Page Showing editForm Populated with the Employee Detail

You can modify the detail of an employee and submit the form by clicking the `Update` button, which executes the `employee.update()` method.

Deleting Employee

You can use the `HtmlCommandLink` components created in the `viewall.jsp` page corresponding to each row for individual employee to delete a particular employee. The associated backing bean method executed to delete an employee is `employee.deleteEmployee()`.

You have learned about various JSF pages using different UI components, implementing backing bean and action methods with all declaration of navigation rules, and managing bean in the `faces-config.xml` file.

Summary

In this chapter, we have discussed about the UI framework, JSF. The chapter describes all elements of the JSF framework, such as UI components, backing beans, validators, converters, and event handlers. Various phases of JSF request processing life cycle are also described in detail to clearly display the actual flow in the framework.

All JSF HTML and core tags have been explained with examples and their attributes. The backing bean concept and managed bean facility are the key features of JSF. Further, the chapter discusses JSF's support for input validation, type conversion, and Internationalization.

The chapter includes a full discussion on the development of various components, their implementation and configuration through a running Web application, named KogentPro.

The next chapter discusses JavaMail API in detail and demonstrates its implementation with the help of an application of sending and receiving mail.

Quick Revise

Q1. What is JSF?

Ans. JSF can be defined as a framework, which makes Web application development easy by providing rich, powerful, and ready to use UI components.

Q2. JSF architecture is designed based on ..... pattern.

Ans. MVC

Q3. What are the basic elements of JSF?

Ans. The following are the basic elements of JSF:

- UI Component
- Renderer
- Validator
- Backing Beans
- Converter
- Events and Listeners
- Message
- Navigation

Q4. List the different types of JSF standard actions.

Ans. The different types of JSF standard actions are as follows:

- Action events
- Value-change events
- Data model events
- Phase events

Q5. Write the element of the faces configuration file that describes the navigation rules of a JSF view.

Ans. The code that describes navigation rules is as follows:

```
<navigation-rule>
<from-view-id>/login.jsp</from-view-id>

<navigation-case>
<from-outcome>success</from-outcome>
<to-view-id>/welcome.jsp</to-view-id>

</navigation-case>
<navigation-case>
<from-outcome>failure</from-outcome>

<to-view-id>/login.jsp</to-view-id>
</navigation-case>
```

Q6. List the six phases of the JSF request processing life cycle.

Ans. The six phases of the JSF request-processing life cycle are as follow:

- Restore View
- Apply Request View
- Process Validation
- Update Model Values
- Invoke Application
- Render Response

Q7. Which JSF UI component's values are validated in the Apply Request Values phase of the JSF request processing?

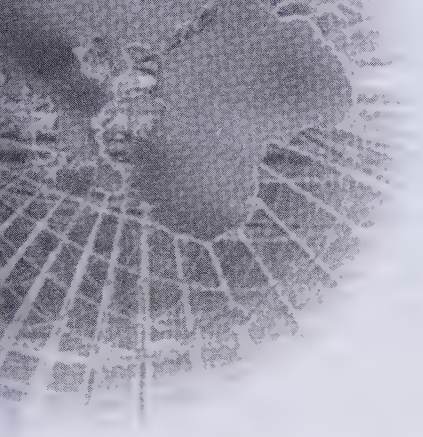
Ans. All the JSF UI components configured with immediate property set to true are validated in Apply Request Values phase of JSF request processing.

Q8. The commandButton element is described under the ..... JSF tag library.

Ans. HTML

Q9. Write the URI used to refer the JSF HTML tag library.

Ans. http://java.sun.com/jsf/html



12

Understanding JavaMail 1.4

JavaMail is a platform and protocol-independent technology, which is developed by Sun Microsystems to create E-mail applications with the help of the Java programming language. JavaMail provides JavaMail Application Programming Interface (API), which is used to create Java-based E-mail client applications. These E-mail client applications use various protocols, such as Simple Mail Transfer Protocol (SMTP) and Internet Message Access Protocol (IMAP) to transfer E-mails. The SMTP protocol is a mail transfer protocol used to send and receive an E-mail; whereas, IMAP is an Internet protocol that enables an E-mail client to retrieve the E-mails from a mail server. Apart from supporting various protocols, Java-based E-mail client applications provide various features, such as customized personal E-mail applications and the ability to add E-mail functionality to an enterprise application. The latest version of the JavaMail technology is 1.4 that is used in the Java EE 6 platform.

JavaMail consists of various components, such as abstract layer, Internet implementation layer, and JavaBeans Activation Framework (JAF). The abstract layer component consists of classes, interfaces, and abstract methods that perform E-mail handling functions, which are expected to be provided by all E-mail systems. The Internet implementation layer component provides implementation to the abstract layer component with the help of standards defined by RFC822, which is a standard for Internet text message format. Multipurpose Internet Mail Extensions (MIME) is a standard that enables you to include non-ASCII characters in E-mails, such as non-text attachments. ASCII stands for American Standard Code for Information Interchange. JAF encapsulates message data and manages commands needed to make data interaction possible.

In this chapter, you learn about JavaMail, its architecture, and the JavaMail API. The chapter also discusses classes, such as Folder, Message, Multipart, Address, and interfaces, such as Part and QuotaAware of the JavaMail API. In the end, you learn how to create E-mail client applications by using the JavaMail API.

Let's start with an overview of JavaMail.

Introducing JavaMail

JavaMail is a technology that provides a protocol-independent framework to create Java-based E-mail client applications. It supports various E-mail protocols, such as HyperText Transfer Protocol (HTTP) and Post Office Protocol (POP). Classes and interfaces of the JavaMail API are utilized to create E-mail client applications. The JavaMail API is broadly divided into the following two parts:

- The first part of the JavaMail API is concerned with how to send and receive messages in a provider/protocol independent manner
- The second part of the JavaMail API provides the classes and interfaces to use the communication protocols, such as SMTP, POP, IMAP, and News Network Transport Protocol (NNTP), used for transferring E-mails

To understand the JavaMail API in detail, you first need to understand few important terms related to E-mail, which are as follows:

- **E-mail client**—Refers to an application that is used to compose, read, and send E-mails.
- **Mail server**—Refers to an application that receives E-mails from E-mail client applications or other mail servers. It stores the messages until they are read or deleted by the E-mail client. A computer that is used especially to run applications which receive E-mails from E-mail client applications and various other mail servers is called a mail server. Some examples of mail servers are Eudora's mail server, Microsoft Exchange server, Microsoft Windows POP3 service, and WinGate.
- **Messaging system**—Provides an E-mail delivery system to send and receive E-mails from the mail server. This messaging system is made up of the following functional components:
  - **Mail User Agent (MUA or UA)**—Represents a client E-mail application, such as Outlook and Eudora, which sends or receives a message to or from a mail server. When a user sends a message, MUA sends the message to the Message Transfer Agent (MTA), discussed in next bullet. This MTA passes the message to another MTA. This process continues till the message reaches the mail server.
  - **Message Transfer Agent**—Serves as an agent who is responsible for sending and delivering E-mails between machines, as well as queuing of messages. MTA stores incoming E-mails and delivers them to the right recipient. The most commonly-used MTAs on the Internet are `Sendmail` and `PostFix`. A large organization may have several MTA servers for sending and delivering E-mails over the Internet.

The components (MUA and MTA) of the messaging system are packaged together to successfully send and receive E-mails from the mail server.

- **Message Delivery Agent (MDA)**—Stores a message into an E-mail client's mailbox. As the message reaches to its destination, i.e. correct mail server, an intermediate MTA gives the message to the final MTA. The message is then passed on to an MDA that adds it to the client's mailbox.
- **Message store (MS)**—Serves as a user mailbox that holds E-mails until they are read and deleted by the user.
- **SMTP**—Refers to a protocol that is used to send and receive an E-mail over the Internet.

Apart from the plain text content, an E-mail can also contain Uniform Resource Locator (URL) pages and files sent as attachments. To read such type of content, JavaMail API uses the JAF framework. This framework allows you to identify the type of data contained in an E-mail, access the data, and instantiate the relevant bean to perform the desired operation on the data.

Let's now discuss the protocols used with the JavaMail API.

Exploring the E-Mail Protocols

E-mail protocols can be defined as a set of some rules, formats, and functions used to exchange messages between the components of a messaging system. The JavaMail API provides five protocols, which are as follows:

- SMTP
- POP3
- IMAP
- NNTP
- MIME

Let's discuss these protocols in detail.

SMTP

SMTP is a standard protocol used to transfer E-mails over the Internet. It uses Transmission Control Protocol (TCP)/Internet Protocol (IP) port 25 or 2525 to send the messages to the recipients. SMTP acts just as a delivery agent to transfer messages and is open to abuse by spammers, who usually send numerous unsolicited mails over the Internet. Due to this reason, many system administrators have blocked or restricted the capability of their SMTP protocol of receiving E-mails from the E-mail clients.

In the context of JavaMail API, a JavaMail E-mail client application communicates with a mail server or SMTP server to send or receive E-mails. This SMTP server sends the message onto the recipient SMTP server with the help of POP or IMAP protocol.

POP3

The POP3 protocol allows E-mail client to collect, download, store, and remove the E-mails from the mail server. This protocol describes how E-mail clients interact with E-mail POP3 servers. The POP3 server is a mail server type used to store incoming E-mails, which are collected, downloaded, stored, and removed by the E-mail clients from the POP3 server. The E-mail client application usually connects to a POP3 server to download messages. Some of the popular POP3 servers are Mozilla Thunderbird, Opera, Eudora, KMail, and Novell Evolution.

IMAP

IMAP is a protocol that is used by many E-mail clients to access the E-mails from the mail server. The latest version of IMAP is 4.7. In POP3, the user is responsible for storing E-mails; whereas, in case of IMAP, the server is responsible for storing an E-mail. The main advantage of IMAP over POP3 is that it stores all the incoming E-mails on the mail server, so the E-mail client, having Internet connection, can connect to the mail server and access all his or her E-mails.

The client that uses the POP3 protocol to retrieve an E-mail from the mail server needs to download the E-mail from the mail server to read it. However, if an E-mail is sent through the IMAP protocol, it is stored on the server and the client can directly access it from the server instead of downloading it on his machine.

NNTP

NNTP is an Internet protocol that is used to view and post Usenet articles, which are articles posted to the worldwide discussion group Usenet, as well as share news among news servers. In other words, NNTP is a protocol used to transfer news and articles between a news server and a newsreader.

`Newsreader` is a client program that is used to read messages from discussion groups over the Internet. A newsreader can be a stand-alone application or part of an E-mail program or a Web browser. News server is a server that hosts newsgroups on the Internet.

Examples of some `newsgroup` servers are as follows:

- Giganews (http://www.giganews.com)
- Rhino News (http://www.rhinonewsgroups.com)
- Power Usenet (http://www.powerusenet.com)

MIME

The text content should always be converted into an ASCII format before it is transmitted over the network. Therefore, a standard format is required to encode the binary files (non-ASCII) into the ASCII text format and to decode them again from ASCII to the non-ASCII format. MIME is used in Internet communications to enable the users to attach formatted document in non-ASCII format, such as graphics, audio, video, spreadsheets, and Adobe Acrobat files, in the E-mails. For example, a user writes the message in non-ASCII format, which is then encoded by MIME into ASCII format for the successful transmission over the Internet. When an E-mail client receives the message in ASCII format, the message is again decoded into its original non-ASCII format so that the receiver could read the message. In messages with a Hypertext Markup Language (HTML) attachment, the MIME type header is set to text/html. In addition to the E-mail applications, Web browsers also support other MIME types, such as application/zip and image/gif.

After having a brief understanding about the mail protocols, let's discuss the role of these protocols in establishing communication between an E-mail client and E-mail server.

Establishing Communication between an E-mail Client and E-mail Server

Now, let's see how an E-mail client and an E-mail server interact with each other to send and receive mails. While sending an E-mail, the E-mail server first finds the sender's E-mail address by using Domain Name System (DNS), which is a hierarchical naming system used to identify computers, resources, and services over the Internet, and the SMTP protocol. Then, the E-mail server sends the mail to the E-mail client.

Figure 12.1 shows the process of interaction between the E-mail client and the E-mail server:

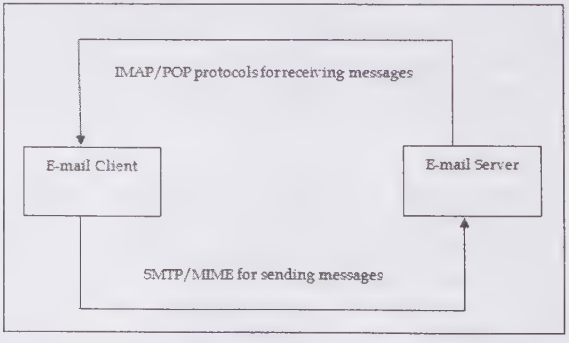


Figure 12.1: Showing the Process of Sending and Receiving Messages

Figure 12.1 shows how the SMTP and IMAP/POP protocols are used to maintain interaction between E-mail client and E-mail server. Let's now discuss the JavaMail architecture.

Exploring the JavaMail Architecture

The JavaMail architecture can be divided into three main layers, the application layer, the JavaMail API, and the implementation layer. This layered architecture permits clients to use the JavaMail API with different protocols, such as POP3, IMAP4, and SMTP, to create an E-mail client application.

Figure 12.2 shows the JavaMail architecture:

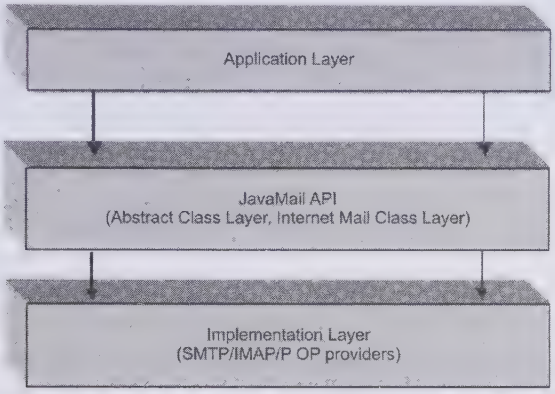


Figure 12.2: Showing the JavaMail Architecture

The application layer is the topmost layer. It uses the JavaMail API to communicate with the E-mail client application. The second layer is the JavaMail API, which is a set of classes and interfaces used to support E-mail client functionality. This layer helps to create E-mail client applications. The implementation layer is the bottom layer of the JavaMail architecture, which uses various protocols, such as POP3, IMAP, and SMTP, to transfer messages between the E-mail client and the mail server.

You can use the classes and interfaces of the JavaMail API to create a JavaMail application. Therefore, you should have a fair knowledge about the JavaMail API, which is discussed in the following section.

Exploring the JavaMail API

As discussed, the JavaMail API provides a set of classes and interfaces to create Java-based E-mail applications. JavaMail API supplies a set of classes and interfaces that enables you to create platform-independent and protocol-independent messaging applications. Some of the important classes and interfaces provided by JavaMail API are as follows:

- The Session class
- The Authenticator class
- The Message class
- The MimeMessage class
- The Part interface
- The MultiPart class
- The ContentType class
- The MimeBodyPart class
- The PreencodedMimeBodyPart class
- The MimeUtility class
- The InternetHeader class
- The ParameterList class
- The QuotaAwareStore interface
- The Resource class
- The Quota class

- The SharedFileInputStream class
- The SharedByteArrayInputStream class
- The ByteArrayDataSource class
- The Address class
- The Store class
- The Folder class
- The Transport class

Let's now discuss each of these classes in detail.

The Session Class

The `javax.mail.Session` class defines the mail session that is used to communicate with remote systems through its object. A session object is used to store information, such as mail server, username, password, and other data that can be shared across the entire E-mail client application. The JavaMail API supports some standard properties that can be set using the put method of the Properties object, which is used to create the `session` object.

Table 12.1 describes standard properties that can be accessed by the methods of the Session class:

Table 12.1: Explaining the Standard Properties that can be Accessed by the Session Class

Property	Description	Default Value
`mail.transport.protocol`	Represents the default transport protocol. An object that implements this transport protocol is returned when the getTransport() method of the Session class is called.	SMTP
`mail.store.protocol`	Represents the default store protocol. An object that implements this store protocol is returned when the getStore() method of the Session class is called.	POP3
`mail.host`	Represents the default host name of a mail server. Both the transport and host protocols use the host name specified in the E-mail client application, if their own host name is not specified.	Local Machine
`mail.user`	Represents the default user name to connect to a mail server. Both the transport and store protocols use the user name, which is specified in an E-Mail client application.	`user.name`
`mail.from`	Specifies an address of the current user.	`username@host`
`mail.protocol.host`	Specifies the IP address of the default mail server.	`mail.host`
`mail.protocol.user`	Overrides the mail.user standard property with respect to the specified protocol.	`mail.user`
`mail.debug`	Helps to print debug messages of the E-mail client application, when set to true. When the mail.debug property is set to false, the debug messages are not printed.	False

After learning about the standard properties that can be accessed by the Session class, let's learn to create a session object. The constructors of the Session class are private; however, you can get the instance of a session object with the help of the `getInstance()` static method of the Session class. A single default session, which can be shared among multiple applications running in the same Java Virtual Machine (JVM), is returned when the `getDefaultInstance()` static method of the Session class is invoked. The following code snippet shows the use of the getDefaultInstance() method:

```
Properties props = new Properties();
// filling props with information
props.put("mail.transport.protocol","smtp");
props.put("mail.smtp.host","yourmail.yourserver.com");
props.put("mail.smtp.port","25");
Session session = Session.getDefaultInstance(props);
```

You can create a private session by calling the `getInstance(Properties prop)` method in the same manner as given in the preceding code snippet.

Table 12.2 describes methods of the `Session` class:

Table 12.2: Describing the Methods of the Session Class

Method	Description
`void addProvider (Provider provider)`	Allows you to add a provider passed through the method to a session object.
`boolean getDebug()`	Returns the debug setting for a session object.
`static Session getDefaultInstance (java.util.Properties props)`	Returns the default session object. In case a default session object does not exist, a new session object is created and considered as default. The `props` parameter passed to the `getDefaultInstance()` method stores properties, such as host name, user name, and protocol used.
`static Session getDefaultInstance (java.util.Properties props, Authenticator authenticator)`	Returns the default session object. In case a default session object does not exist, a new session object is created and considered as default. The authenticator parameter passed to this method specifies that if the authenticator object, which is used to check the access permission, is not available in the session object, a new authenticator object is created and added to the session object. Otherwise, an existing authenticator object available in the session object is used. The props parameter used in this method stores properties, such as host name, user name, and protocol used.
`Folder getFolder(URLName url) throws MessagingException`	Returns a closed folder object, a state in which certain operations are allowed on the folder object, for the url parameter passed to it. If the requested folder cannot be obtained, null is returned. The url parameter represents the desired folder object. This method throws the `MessagingException` exception when a folder object cannot be located or created.
`static Session getInstance (java.util.Properties props)`	Returns a new instance of the Session class. The props parameter stores properties, such as host name, user name, and protocol used.
`static Session getInstance (java.util.Properties props, Authenticator authenticator)`	Returns a new instance of the Session class. The props parameter of the getInstance() method stores properties, such as host name, user name, and protocol used. The authenticator parameter of the getInstance() method defines an authenticator object.
`PasswordAuthentication getPasswordAuthentication (URLName url)`	Returns the PasswordAuthentication object for store or transport URLName. This method is used usually with the store or transport protocol.
`java.util.Properties getProperties()`	Retrieves the properties object related with the current session.
`String getProperty (String name)`	Returns a value of the property whose name is passed to it. Returns null if the property does not exist.

Table 12.2: Describing the Methods of the Session Class

Method	Description
`Transport getTransport (Address address) throws NoSuchProviderException`	Returns a transport object that can send a message to the address passed to this method.
`Transport getTransport (String protocol) throws NoSuchProviderException`	Returns a transport object that implements the specified protocol. In case the transport object is not found, null is returned. This method throws the `NoSuchProviderException` exception when a `protocol` is not found.
`void setProvider (Provider provider)`	Allows you to add the specified provider to the current session.
`void setProtocolForAddress (String addresstype, String protocol)`	Sets the protocol address in the currently used transport protocol.

The Authenticator Class

The javax.mail.`Authenticator` class is used to obtain authentication for a network connection. Authentication is the process used to verify the validity of a user with the help of the login credentials provided by the user. You need to create an authenticator object and call its getPasswordAuthentication() method to authenticate the user or find if the user is valid or not. After creating the authenticator object, you must register it with the session object. The session object must have necessary authentication details to send and receive an E-mail.

The following code snippet shows how to register an authenticator object with the session object:

```
static Session getInstance(Properties prop,Authenticator auth);
static Session getDefaultInstance(Properties prop,Authenticator auth);
```

In the preceding code snippet, the props parameter passed to the getInstance() method of the Session class is the properties object that contains relevant properties. The auth parameter passed to the getInstance() method is an authenticator object used by an application to authenticate a user. In the getDefaultInstance() method of the Session object, the prop and auth parameters are the same as in the getInstance() method.

The Message Class

The `javax.mail.Message` class is an abstract class used to create a new mail message. All the message structures are based on this class. A mail message consists of the following two main components:

- **Header**—Contains information about message properties, such as subject field, recipient, and sender of the message
- **Content**—Represents the content of the message

The Message class implements the `javax.mail.Part` interface. To attach a file with a message, the `javax.mail.internet` package provides the `MimeMessage` class, which extends the `Message` class.

The MimeMessage Class

The javax.mail.internet.MimeMessage class is used to create a MIME message, which accepts MIME types and headers. To create a new MIME message, you need to create an empty MimeMessage object and then provide suitable attributes and content to it. The MimeMessage class provides two constructors, which are used to create a MIME message, as shown in the following code snippet:

```
Constructor 1    public MimeMessage(Session session)
Constructor 2     public MimeMessage(MimeMessage msg)
```

In the preceding code snippet, constructor 1 creates an empty MimeMessage object that is created by passing a session object as a parameter to this constructor. Constructor 2 creates a new MimeMessage instance by passing the MimeMessage object as a parameter to this constructor.

Table 12.3 lists the methods of the `MimeMessage` class:

Table 12.3: Describing the Methods of the MimeMessage Class

Method	Description
`void addFrom(Address[] addresses)`	Allows you to add multiple addresses that are stored in the addresses parameter to the From property of the MimeMessage class.
`void setRecipients(Message.RecipientType type, Address[] addresses)`	Allows you to set the recipient type specified by the type parameter to the multiple addresses that are stored in the addresses parameter.
`void setSubject(String subject)`	Sets the Subject header field. The value of the subject parameter of the setSubject() method is assigned to the Subject header field.
`void setText(String text)`	Sets a string, which is passed as the text parameter, to the content of a message, whose MIME type is text/plain.
`void setText(String text, String charset)`	Sets a string, which is passed as the text parameter, to the content of a message having MIME type text/plain and the charset as passed to the method.
`void setText(String text, String charset, String subtype)`	Sets a string, which is passed as the text parameter, to the content of a message having the text MIME type and the MIME subtype similar to the subtype parameter passed to this method. Parameter charset represents the encoding scheme that specifies mapping of characters in text parameter string with 8 bits bytes.
`void setFileName (String filename)`	Sets the filename related to a message. In case, the value of the mail.mime.encodefilename System property is true, the MimeMessage and MimeBodyPart setFileName() methods invoke the MimeUtility.encodeText() method to encode the filename. When the mail.mime.encodefilename System property is false, they invoke the MimeUtility.decodeText() method to decode the filename. You should note that by default the value of both the properties are set to false.
`void updateMessageID() throws MessagingException:`	Allows you to update the Message-ID header. This method is called by the updateHeaders() method of the MimeMessage class, which makes it possible to override the algorithm written for selecting a Message-ID.
`MimeMessage createMimeMessage(Session session) throws MessagingException`	Allows you to create and get a MimeMessage object.

The most common properties used with the `MimeMessage` class are as follows:

- The From property
- The To, CC, BCC properties
- The Reply-To property
- The Subject property
- The other property

Let's now discuss each of these properties, one by one.

Using the From Property

The `From` property depicts the source of an E-mail message. The `Message` class has a set of methods, such as getFrom() and setFrom(), to set and get data from this property. The `getFrom()` method is used to get the E-mail addresses from the `From` property. The following code snippet shows the syntax to read the value of the `From` property:

```
void Address[] getFrom();
```

In the preceding code snippet, an array of E-mail addresses is retrieved from the `javax.mail.Address` object.

The setFrom() method is used to set an E-mail address in the From property, as shown in the following code snippet:

```
void setFrom(javax.mail.Address senderAddress)
```

Using To, CC, BCC Properties

The To property is used to specify recipients of the E-mail message. The carbon copy (CC) property is used to specify other recipients, which receive the copy of the original message. The Blind Carbon Copy (BCC) property is marked only to those recipients whose name you do not want to reveal to the other recipients of the E-mail.

Using the Reply-To Property

The Reply-To property is used to reply an E-mail. The following code snippet shows the implementation of the Reply-to property by using the setReplyTo() method:

```
void setReplyTo(Address myaddress);
```

If you do not make a call to the setReplyTo() method, then a null argument is set in this method, which means the Reply-To property is not included in the message header of the resulting mail message. The following code snippet shows how to retrieve the Reply-To header field:

```
Address[ ] getReplyTo().
```

If header is not present, the getReplyTo() method returns the From field value.

Using the Subject property

The Subject property represents the subject of an E-mail. You can set the value of the Subject property by using the setSubject() method. The following code snippet shows how to set the Subject property:

```
message.setSubject("MySubject");
```

In the preceding code snippet, the subject of an E-mail is set by using the setSubject() method.

Using the Other property

The other property is used to set or get a new header of an E-mail. The following code snippet shows how to set a new header:

```
void setHeader(String username,String passvalue);
```

The following code snippet shows how to get a new header:

```
String getHeader (String username,String delimiter);
```

The Part Interface

The javax.mail.Part interface is used to create the message body. The Part interface consists of attributes, such as content type and content. The getter and setter methods for the attributes of the Part interface gets and sets the values of these attributes. The content attribute of the Part instance uses a MIME type, such as text/plain and text/html, to represent the type of data of the message.

Table 12.4 describes the methods of the *Part* interface:

Table 12.4: Describing the Methods of the Part Interface

Method	Description
String getContentType()	Returns content type of the content of a Part instance. Content type of an E-mail is the type of data it contains.
Object getContent()	Gets the content of the current Part instance as a Java object
InputStream getInputStream()	Returns an input stream of text content of an E-mail
void writeTo(Outputstream os)	Writes the data passed to it to the output stream of an E-mail
boolean isMimeType (String mimeType)	Allows you to determine whether the MIME content type of the current Part instance is same as the argument mimeType passed to it
String getDescription()	Allows you to get descriptive information, such as MIME type and file attachment, about the Part instance.
void setDescription (String desc)	Allows you to set descriptive notes about the MimePart instance

Table 12.4: Describing the Methods of the Part Interface	
Method	**Description**
`void setDisposition (String disp)`	Allows you to set the file attachment in a body part of an E-mail
`String getFileName()`	Gets the name of the attachment file of the current Part instance
`void setFileName(String fname)`	Sets the name of the attachment file of the current Part instance
`int getSize()`	Retrieves the length of the content of the current BodyPart object in bytes

The JavaMail API provides the `MultiPart` class that implements the Part interface. A mail message is constructed using the Part interface as well as the MultiPart and BodyPart classes.

Figure 12.3 shows the use of the Part interface, the MultiPart class, and the BodyPart class in a complex mail message:

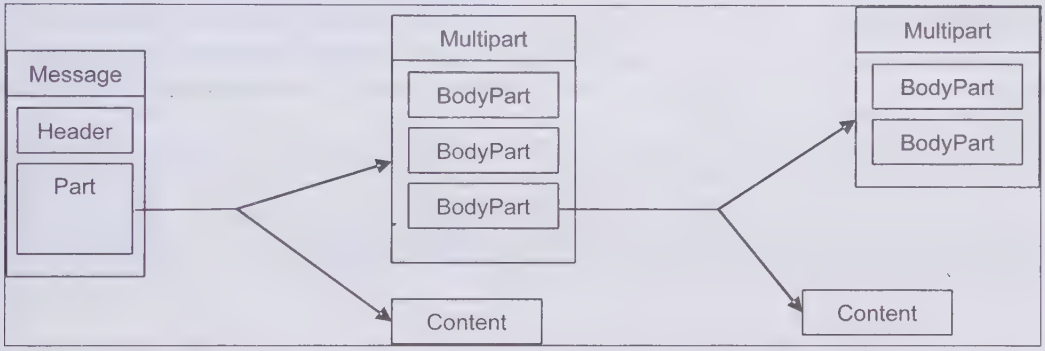


Figure 12.3: Showing the Structure of a Complex Mail Message

Figure 12.3 shows the concept of a complex E-mail implemented by the Part interface. In the first block, a message is built by the Header and Part components. The Header component contains information about message properties, such as subject field and recipient. The body of the message is represented by a customized BodyPart class that implements the Part interface. The Part component represents the message body that contains file attachments, which are represented by the `MultiPart` class and Content, as shown in the second block. In the second block, the Multipart class contains many BodyParts. Each of these BodyParts can further contain a file attachment that is represented using the `MultiPart` class and Content. As body of the E-mail message can only contain HTML content; therefore, JavaMail uses JAF to process E-mail content, when it contains non-HTML attachments, such as portable document format (PDF) file or graphics files.

JAF provides services to determine the type of data (PDF or graphic file), determine the operations available on the data, and instantiate the appropriate bean, which is a Java class that performs data access and storage functions to transfer the data to a recipient.

For example, if a browser obtains a PDF file, JAF would enable the browser to use MIME type to determine the type of the received data. This MIME type helps the browser to determine whether it can display the file itself or whether it has to invoke another helper application, such as acrobat reader for reading a PDF file.

JAF provides the `DataHandler` class that acts as the main interface of non-HTML data and handles necessary operations on that data. When the `Part` interface handles the content of an E-mail, all operations are performed through the `DataHandler` class.

Most common MIME types provided by the `DataHandler` class are as follows:

- **text/plain**—Helps in processing only ASCII text messages
- **text/html**—Helps in processing HTML text messages
- **multipart/mixed**—Helps in processing non-HTML data

Now, let's use the `getContentType()` method to return the MIME type of the `DataHandler` object, as shown in the following code snippet:

```
MimeMessage msg=new MimeMessage(session);
String messageBody="Hello from my first E-mail with JavaMail";
DataHandler dh=new DataHandler(messageBody,"text/plain");
msg.setDataHandler(dh);
DataHandler dh=msg.getDataHandler();
if(dh.getContentType().equals("text/plain")) {
  String messageBody=(String)dh.getContent();
}
```

In the preceding code snippet, an object of the DataHandler class (dh) is created by supplying a string value (body of the message) and MIME type to the constructor of the *`DataHandler`* class. The dh object is then passed to the setDataHandler() method to create a message by using the msg object of the MimeMessage class. Similarly, the content stored in the object of the *`DataHandler`* class is retrieved by invoking the *`getContent`*() method on the *`DataHandler`* object.

The Multipart Class

The `javax.mail.Multipart` class is used to create multiple body parts as well as file attachments of a message. The following code snippet shows how to use the Multipart MIME message to manage multiple instances of the `MimeBodyPart` class:

```
MimeMessage msg=new MimeMessage(session);
Multipart mailBody= new MimeMultipart();

//creating first part code
MimeBodyPart mainmessBody=new MimeBodyPart();
mainmessBody.setText("here's the file I promised you.");
mailBody.addBodyPart(mainmessBody);

//creating second part code,attachment
FileDataSource fds=new FileDataSource("c:\temp\photo.jpg");
MimeBodyPart mimeAttach=new MimeBodyPart();
mimeAttach.setDataHandler(new DataHandler(fds));
mimeAttach.setFileName(fds.getName());
mailBody.addBodyPart(mimeAttach);

msg.setContent(mailBody);
```

In the preceding code snippet, we have created a new instance of the `MimeMultipart` class that acts as a container for multiple instances of the Part interface. The path of the file to be attached to a message is stored in the FileDataSource object. In our case, the path of the photo.jpg file is stored in the fds object of the FileDataSource class.

Table 12.5 describes methods of the `Multipart` class:

Table 12.5: Describing the Methods of the Multipart Class

Method	Description
`BodyPart getBodyPart (int index)`	Returns a specified BodyPart from the Multipart object. It returns zero (0), if there is no BodyPart in a message.
`boolean removeBodyPart (BodyPart bp)`	Removes the BodyPart object from the Multipart object, whose reference is same as the bp parameter. If the BodyPart object has been removed successfully, the removeBodyPart method returns true; otherwise, it returns false.
`boolean removeBodyPart (int index)`	Removes the BodyPart object from the list of BodyPart objects in the Multipart object, at the specified position. If the BodyPart object has been removed successfully, this method returns true; otherwise, it returns false.
`int getCount()`	Gets the number of BodyPart objects in the list of BodyPart objects.
`Part getParent()`	Returns the reference of the Part instance that contains the Multipart object. It returns null, if no Part is available.
`String getContentType()`	Returns the MIME type for the Multipart object.

Table 12.5: Describing the Methods of the Multipart Class	
Method	**Description**
`public void addBodyPart (BodyPart bp)`	Adds a BodyPart object to the end of the list of currently used BodyPart objects.
`public void addBodyPart (BodyPart bp,int index)`	Adds the BodyPart object at the specified index in the list of BodyPart objects of the Multipart object.
`public void setParent (Part parent)`	Sets the parent of the MultiPart class as an individual part of a message.

The ContentType Class

The ContentType class defines the nature of data contained in a message. The content type is divided into the following two parts:

- **Primary type**—Signifies the general type of data
- **Sub type**—Signifies a particular format for the primary type

The javax.mail.internet.ContentType class represents the value of content type. This class provides various methods to perform various operations on a ContentType string, such as getting the individual parts of the ContentType value and creating the MIME style ContentType string.

Table 12.6 describes the methods of the ContentType class:

Table 12.6: Describing the Methods of the ContentType Class	
Method	**Description**
`getBaseType()`	Gets the MIME Content Type string, which is a concatenated string of primary type, /, and sub type. For example, the getBaseType() method returns the string text/html for the content type of text/html. The returned string comprises primary type, which is text, forward slash, and the sub type, which is html.
`getPrimaryType()`	Gets the primary type part of the content type.
`getSubType()`	Gets the sub type part of the content type.
`setPrimaryType(String pType)`	Allows you to set the primary type of content type. The pType parameter sets the primary type of the message.
`setSubType(String sType)`	Allows you to set the passed sType argument to the sub type of the message.

The MimeBodyPart Class

The `javax.mail.internet.MimeBodyPart` class specifies MIME body part. The MimeBodyPart class extends the `BodyPart` abstract class and implements the `MimePart` interface.

Table 12.7 describes the methods of the MimeBodyPart class:

Table 12.7: Describing the Methods of the MimeBodyPart Class	
Method	**Description**
`Void attachFile(java.io.File file) throws java.io.IOException, MessagingException`	Attaches the file to be passed as the file parameter with an E-mail
`void saveFile(java.io.File file) throw java.io.IOException,MessagingException`	Saves the content of the current MimeBodyPart object in the file passed as a parameter

The PreencodedMimeBodyPart Class

The `javax.mail.internet.PreencodedMimeBodyPart class` helps to create a message and attaches the encoded data to the message by using the encoding scheme as base64 encoding. Encoding scheme is a

scheme used to convert the data into a format that can be transmitted over the network. You can encode the data by using an E-mail client application, a file, or database. The encoded data is sent to the recipient when the object of the `PreencodedMimeBodyPart class` is created.

Table 12.8 describes the methods of the `PreencodedMimeBodyPart` class:

Table 12.8: Describing the Methods of the PreencodedMimeBodyPart Class

Method	Description
`String getEncoding()`	Returns the content transfer encoding. This encoding information is specified when a PreencodedMimeBodyPart object is created.
`void updateHeaders()`	Updates the headers of messages.
`void writeTo (java.io.OutputStream os)`	Writes to a body of a message by using the OutputStream object.

The MimeUtility Class

The javax.mail.internet.MimeUtility class provides various MIME utilities, such as encoding and decoding the message header. This class provides various methods to encode and decode MIME headers to transfer non-HTML files between E-mail client applications. Mail headers should possess US-ASCII characters for successful transmission over the network. The headers containing non US-ASCII characters are encoded to ensure that they contain US-ASCII characters for transmission of text messages over the network. You can use the BASE64 or QP algorithm, which are encoding schemes, to encode non US-ASCII characters.

Table 12.9 describes the methods of the `MimeUtility` class:

Table 12.9: Describing the Methods of the MimeUtility Class

Method	Description
`java.io.InputStream decode (java.io.InputStream is, String encoding)`	Decodes the given input stream for successful transmission.
`String decodeText (String etext)`	Decodes unstructured headers, that is, it decodes *text token according to RFC 822 into the form which is supported by E-mail standards with the help of algorithm defined by RFC 2047. RFC 822 is a standard format for the Internet text message.
`String decodeWord (String eword)`	Uses the rules specified in RFC 2047 for parsing an encoded data. The decodeWord() method parses the string passed as a parameter to it. RFC 2047 is a standard set of rules for encoded data to be transmitted over the Internet.
`java.io.OutputStream encode (java.io.OutputStream os, String encoding)`	Allows you to wrap an encoder around the output stream passed as an argument to this method.
`static String encodeText(String text)`	Encodes an RFC 822 text token into the mail-safe form according to RFC 2047.
`String encodeWord (String word)`	Encodes an RFC 822 word token into the mail-safe form according to RFC 2047.
`String fold(int used, String s)`	Breaks a string at linear whitespace so that each line can contain only upto 76 characters. If there are more than 76 non-whitespace characters, the string is folded at the first whitespace.
`String unfold (String s)`	Allows you to unfold a folded header.

The InternetHeader Class

The `javax.mail.internet.InternetHeaders.InternetHeader` class extends the Header class and represents an Internet header. An instance of the `InternetHeader` class having a null value is used as a placeholder for headers to maintain the order of headers. A placeholder is an `InternetHeader` object with the : character. The : character indicates a position in the list of headers where new headerS can be added. The getValue() method of the InternetHeader class is used to access the value of a header.

The ParameterList Class

The `javax.mail.internet.ParameterList` class accepts non US-ASCII characters. RFC 2231 provides support to encode non US-ASCII character to MIME headers. JavaMail makes use of two standard properties of the System class, namely `mail.mime.encodeparameters` and `mail.mime.decodeparameters`, whose values are set using the System class `setProperty()` method, to control parameter encoding and decoding. When the `mail.mime.encodeparameters` property is set to true, the non US-ASCII character is encoded. Similarly, when the `mail.mime.decodeparameters` property is set to true, the non US-ASCII character is decoded.

By default, the `encodeparameters` and `decodeparameters` properties are set to false. The `mail.mime.encodeparameters` and `mail.mime.decodeparameters` system properties determines whether or not the encoded parameters are supported in an E-mail. If the value of the `mail.mime.decodeparameters.strict` system property is set to true, the E-mail client application raises the `ParseException` exception that is thrown for errors that are encountered during the decoding of the encoded parameters.

Table 12.10 describes the methods of the `ParameterList` class:

Table 12.10: Describing the Methods of the ParameterList Class

Method	Description
`String get (String name)`	Returns the value of the parameter passed as an argument. Note that parameter names are case insensitive.
`public void set (String name, String value)`	Sets the value of the name parameter. If the name parameter already exists, then the value of the name parameter is replaced by the value parameter.
`public void set (String name, String value, String charset)`	Allows you to set the value of a parameter. If the `mail.mime.encodeparameters` system property is set to true, the parameter value is converted into non US-ASCII character and encoded with the specified charset, as mentioned by RFC 2231, which is a document that specifies the methods, behaviors, innovations that applies to the working of the Internet.
`public void remove (String name)`	Removes the parameter name that is passed as an argument to it from the current ParameterList object.
`public java.util.Enumeration getNames()`	Gets an enumeration object having all the parameter names in the current list of parameters.
`public String toString()`	Changes the ParameterList object in the form of MIME string.

The QuotaAwareStore Interface

The `javax.mail.QuotaAwareStore` interface is used by the `store` objects to support quotas. Quota allows you to limit the mail storage database for a particular user. The `getQuota()` and `setQuota()` methods of the QuotaAwareStore interface provides support to quota, which is defined by the `IMAP QUOTA` extension. The IMAP QUOTA extension restricts the usage of resources (quota) that are to be used by IMAP.

Table 12.11 lists the methods of the `QuotaAwareStore` interface:

Table 12.11: Describing the Methods of the QuotaAwareStore Interface

Method	Description
Quota[] getQuota(String root) throws MessagingException	Allows you to get the quotas for the quota root passed to it as a string argument. Quotas are controlled with the help of a quota root.
void setQuota(Quota quota) `throws MessagingException`	Allows you to set the quotas for the quota root passed to it as an argument.

The Resource Class

The javax.mail.Quota.Resource class represents an individual resource in a quota root. The following code snippet describes the constructors of the Resource class:

```
public Quota.Resource(String name, long usage, long limit)
```

In the preceding code snippet, the constructor of the Quota.Resource class creates an object of the Resource class with the name, usage, and limit argument passed to it.

The Quota Class

The `javax.mail.Quota` class specifies quotas for a specific quota root. Each quota root is modeled by the `Resource` class, which has resources. Each resource has a name (for example, `KOGENT`), and a usage limit provided by the name and limit attributes.

Table 12.12 describes the method of the `Quota` class:

Table 12.12: Describing the Method of the Quota Class

Method	Description
public void setResourceLimit (String name, long limit)	Allows you to set a resource limit value for the Quota object, with which the setResourceLimit() method has been called

The SharedFileInputStream Class

The `javax.mail.util.SharedFileInputStream` class is a subclass of the `BufferedInputStream` class that buffers data from a file. The SharedFileInputStream class permits you to access the file it buffers and also ensures that the file is closed when it is in use.

Table 12.13 describes the methods of the `SharedFileInputStream` class:

Table 12.13: Describing the Methods of the SharedFileInputStream Class

Method	Description
public int available()throws IOException	Retrieves the number of bytes that can be read from a file using the InputStream object. It overrides the `available()` method derived from its `superclass, BufferedInputStream.`
public void reset() throws IOException	Represents the inherited reset() method of its superclass InputStream and reposition the pointer to position in an SharedFileInputStream object that was last marked by mark() method of the SharedFileInputStream object.
public void close() `throws IOException`	Closes an input stream and releases system resources that are used by the input stream. This method overrides the close () method of its superclass, `BufferedInputStream.`
public long getPosition()	Gets the current position of the data stored in a buffer of SharedFileInputStream, with which the method was called.
public InputStream newStream (long start,long end)	Allows you to create a new input stream object that represents a data substring from the input stream object, starting at start (inclusive) up to end (exclusive). Value of start must be positive. If end value is -1,

Table 12.13: Describing the Methods of the SharedFileInputStream Class	
Method	**Description**
	the new input stream object also ends at the same place as the input stream object. The newly created input stream object from this method also implements the `SharedInputStream` interface.
protected void finalize() `throws Throwable`	Forces the `input stream` object to close. This method overrides the `finalize` () method of its superclass, Object.

The SharedByteArrayInputStream Class

The `javax.mail.util.SharedByteArrayInputStream` class extends the `ByteArrayInputStream` class and implements the `SharedInputStream` interface. It provides methods that permit multiple readers to read the data from a common byte array.

Table 12.14 describes the methods of the SharedByteArrayInputStream class:

Table 12.14: Describing the Methods of the SharedByteArrayInputStream Class	
Method	**Description**
`long getPosition()`	Retrieves the current input data position from the current input stream object
InputStream newStream (long start, long end)	Retrieves a new input stream object denoting a subset of the data from the input stream object with which the newStream() method has been called, which starts from the start argument position and ends at the end argument position

The ByteArrayDataSource Class

The `javax.mail.util.ByteArrayDataSource` class is used to initialize a byte array using an InputStream or a string value. It implements the `javax.activation.DataSource` interface.

Table 12.15 describes the methods of the `ByteArrayDataSource class:`

Table 12.15: Describing the Methods of the ByteArrayDataSource Class	
Method	**Description**
`public java.io.InputStream getInputStream()throws java.io.IOException`	Allows you to retrieve an input stream object of a socket, which is a communication channel between the client and the server. A new input stream object is always returned when this method is called.
`public java.io.OutputStream getOutputStream()throws java.io.IOException`	Gets an output stream object of a socket.
`public String getContentType()`	Allows you to get the MIME content type of data in the form of string.
`public String getName()`	Gets the name of an entity, such as class, interface, and array. By default, an empty string:(" ") is returned.
`public void setName(String name)`	Sets the name of the entity, such as class, interface, and array to the string parameter passed to it.

The Address Class

JavaMail API provides the `javax.mail.Address` abstract class that represents E-mail addresses of the sender or the receiver of the E-mail. The Address class contains the following subclasses:

- javax.mail.internet.InternetAddress
- javax.mail.internet.NewsAddress

Let's discuss these classes in detail.

The InternetAddress Class

The `javax.mail.internet.InternetAddress` class provides methods to concatenate and parse E-mail addresses in a field. If you want multiple E-mail addresses in the field, then addresses must be separated with commas. The E-mail addresses are passed as an argument to the setAddress() method of the InternetAddress class.

The following code snippet shows how to set an E-mail address in an E-mail:

```
Internetaddress MyInternetAddress=new InternetAddress();
MyInternetAddress.setAddress("yourEmaiID");
MyInternetAddress.setPersonal("Kogent");
System.out.println("MyAddress="+MyAddress.toString());
```

You can parse an E-mail address by using the parse() method, as shown in the following code snippet:

```
InternetAddress[] parse(String)
```

The following code snippet parses more than one E-mail addresses into an array of `InternetAddress` objects:

```
InternetAddress toField[]=InternetAddress.parse
("Test@Mymail.com,Test1@Mymail.com");

for(int i=0;i<toField.length;i++){
  System.out.println("toField["+i+"].Address="+tofield[i].getAddress());
  System.out.println("toField["+i+"].Address="+tofield[i].getPersonal());
```

The NewsAddress Class

The `javax.mail.internet.NewsAddress` class is used to develop a newsgroup message with a newsgroup name and an optional host name. Newsgroups are organized into different hierarchies, such as `alt` (alternative), `biz` (business), `comp` (computing), `misc` (miscellaneous), and `rec` (recreational).

The Store Class

The `javax.mail.Store` class is used to store and receive messages. In addition, it performs other actions on messages, such as storing similar messages in specific folder objects, which are then stored on a mail server. Table 12.16 describes the methods of the Store class:

Table 12.16: Describing the Methods of the Store Class

Method	Description
`Folder` getFolder(String name)	Gets a folder object corresponding to the name argument passed to it.
`Folder` getFolder(URLName myurl)	Gets a closed folder object corresponding to the myurl argument passed to it.
`Folder[ ]` getPersonalNamespaces()	Gets a set of folder objects denoting the personal namespaces of the current user. A personal namespace contains personal details, such as username and password, of the authenticated user. Typically, only the authenticated users have access to mail folders in their personal namespace. The INBOX folder object is always contained within the personal namespace, reserved for the user. A particular user can have only personal namespace in a typical case.
`Folder[ ] getSharedNamespaces()`	Allows you to get a folder object array, which represents the namespaces shared among multiple users.
`Folder[]` getUserNamespaces(String user)	Allows you to get a folder object array. This array specifies the user namespace, which is passed as a parameter to the getUserNamespaces() method.
`public abstract` Folder `getDefaultFolder()`	Allows you to get a folder object, which specifies the default namespace root.

You need to connect to mail storage to retrieve the session object, which in turn is used to retrieve the store object. The session object is needed to get the instance of POP server's connection and then retrieve the store object by using the getStore() method. The following code snippet shows how to retrieve a store object by connecting to a POP server:

```
//Set up default parameters

Properties myprops=new Properties();
myprops.put("mail.transport.protocol","pop");

//Create the seesion and create a new mail message

Session mymailSession=Session.getInstance(myprops);

//Get the store and connect to the server

Store mymailStore=mymailSession.getStore();
mymailStore.connect("yourpop.server.com",10,"yourname","yourpassword");

//Proceed to manipulate Folder objects
```

In the preceding code snippet, the store object is retrieved by using the getStore() method of the Session class. After retrieving the store object, you need to retrieve the folder object to store a message. You can use the getFolder() method of the Session class to retrieve the folder object. The `exists()` method of the folder object verifies whether or not the folder exists in the store object.

The following code snippet shows how to retrieve the inbox folder object by using the object of the Store class (mymailStore):

```
//Get the Strore and connect to the server
URLName myurlname=new URLName("pop3://<user>;auth=<auth>@<host>:<port>");
Store mymailStore=mymailSession.getStore(myurlname);
mymailStore.connect();

//proceed to manipulate Folder objects
Folder myinbox=mymailStore.getDefaultFolder();
//or
Folder myinbox=mymailStore.getFolder("INBOX");
```

In the preceding code snippet, the `INBOX` keyword is a special name for a folder object in which the users receive their messages.

The Folder Class

The `javax.mail.Folder` class represents a folder containing messages as well as other subfolders in a structure similar to tree hierarchical structure. A folder object is initially in the closed state. You can perform certain operations on the folder object in its closed state, such as renaming the folder object and monitoring it for new messages. To open the folder object, you need to call the `open()` method. You can perform actions, such as retrieving messages and changing notifications on the folder object in the open state.

You can create the folder objects by calling the getFolder() method of the Store class and the Folder class. Alternatively, you can call the list(), list(String), listSubscribed(), and listSubscribed(String) methods of the Folder class to create folder objects.

When a message is deleted from a folder object, the total number of messages is not calculated until expunging occurs on the folder object. The concept of expunging a folder object implies that all the messages which have been marked for deletion are deleted and finally removed from the folder object. When you invoke the `getMessage(msgno)` method, the folder object returns the same message multiple times with the same `message` object, until an expunge is done or the messages marked as deleted are deleted from the folder object.

Table 12.17 describes the fields of the Folder class:

Table 12.17: Describing the Fields of the Folder Class

Field	Description
`HOLDS_FOLDERS`	Contains other folder objects
`HOLDS_MESSAGES`	Contains messages
`READ_ONLY`	Specifies that the folder object is read-only, implying it can cannot be modified
`READ_WRITE`	Specifies that the folder object permits read-write operations
`Store`	Contains the parent object

Table 12.18 describes the methods of the Folder class:

Table 12.18: Describing the Methods of the Folder Class

Method	Description
`boolean hasNewMessages()`	Returns true if any folder object message is flagged with the `Flag.RECENT` flag.
`int getMessageCount()`	Allows you to get the total number of messages of the folder object. The getMessageCount() method can be called on a closed folder object. However, note that for some folder objects implementation, getting the total message count can be an operation that may involve opening of the folder object. In such cases, a provider can select not to support invocation of the getMessageCount() method on the closed folder object.
`int getNewMessageCount()`	Allows you to get number of new messages in a folder object by calling the getMessage(int) method of the folder object to retrieve all its messages, and checking whether or not its RECENT flag is set. If the RECENT flag of a message is set to true then this implies that the message is new and the message is added to the count of new messages, which is to be returned by the getNewMessageCount() method.
`int getUnreadMessageCount()`	Allows you to get number of unread messages from a folder object by calling the getMessage(int) method of the folder object to retrieve all its messages and checking whether or not its SEEN flag is set. If the SEEN flag of a message is set to false then this implies that the message has not been read and the message is added to the count of unread messages, which is to be returned by the getUnreadMessageCount() method.
`Message getMessage(int)`	Allows you to get the message object corresponding to the message number passed as an argument to it. Unlike the folder objects, repeated calls to the getMessage() method with the same message number returns the same message object, as long as no messages in the folder object have been expunged.
`Message[] getMessages()`	Allows you to get all the message objects stored in a folder object. Returns an empty array if the folder object is empty.
`Message[] getMessages(int, int)`	Allows you to get all the message objects corresponding to the message numbers passed as arguments.
`Message[] getMessages(int[])`	Allows you to get the message objects for message numbers given in an array passed to this method.

You can perform the following operations with the Folder class:

- Listing folders
- Opening and closing folders
- Listing messages
- Copying and moving messages
- Searching messages

Let's discuss these in detail.

Listing Folders

If a folder object contains subfolders, you can use the `list()` function of the `Folder` class to list these subfolders. The list() function lists only the top-level folder objects within the folder object hierarchy. The following code snippet shows how to list folder objects:

```
//Set up the default parameters
Properties myprops=new Properties();
```

```
//create the Session anad create a new mail message
Session mymailSession=Sesion.getInstance(myprops);

//Get the store and connect to the server
URLName myurl=new URLName("imap://alan:ceri@mail.microsoft.com");
Store mymailStore=mailSession.getStore(myurl);

mymailStore.connect();
//proceed to list all the Folder objects
Folder thisFolder=mymailStore.getDefaultInstance();

Folder listOfFolders[]=null;
if((thisFolder.getType() & Folder.HOLDS_FOLDERS)) {
  listOfFolders=thisFolder.list();
}
```

In the preceding code snippet, the `getType()` method of the `folder` class returns the status field for a folder object. The `list()` method returns a list of folder objects that match the search string passed as a parameter to it. For example, the following code snippet returns all the folder objects that begin with the string, Client:

```
Folder mylistOfFolders[]=thisFolder.list("Client%");
```

The following code snippet returns all the folder objects, including any subfolder objects, which begin with the letter A:

```
Folder mylistOfFolders[]=thisFolder.list("A*");
```

The following code snippet lists all the folder objects in the folder object hierarchy:

```
//Set up the default parameters
Properties myprops=new Properties();

//create the Session anad create a new mail message
Session mymailSession=Sesion.getInstance(myprops);

//Get the store and connect to the server
URLName myurlname=new URLName("imap://alan:ceri@mail.microsoft.com");
Store mymailStore=mymailSession.getStore(urlname);

mymailStore.connect();
//proceed to list all the Folder objects
Folder thisfolder=mymailStore.getDefaultInstance();

if(thisFolder!=null)
{
  if((thisFolder.getType() & Folder.HOLDS_FOLDERS)!=0)
  {
        Folder[] listOfFolders=thisFolder.list("*");
        for(int i=0;i<listOfFolders.length;i++)
        {
              System.out.println("FolderName="+listOfFolders[i].getName());
        }
  }
}
```

In the preceding code snippet, we have used the list() method to list the folder objects.

Two other methods of the folder objects are used to return a list of folders in the folder. The listSubscribed() method of the folder object returns the list of subscribed folders inside a folder and listSubscribed(String search) method returns the list of folders in the folder namespace that matches the pattern of the string parameter passed to it.

Opening and Closing Folders

You can use the public void open (int mode) method to open a folder object. This method is called on the folder objects that are not empty and are not opened. To check whether the folder object is open or not, you can call the `boolean isOpen()` method. The following code snippet shows how to invoke the getType() method on the folder object:

```
if(thisFolder.getType()==Folder.READ_WRITE)
   System.out.println("This folder was opened with READ_WRITE Access both");
else
   System.out.println("This folder was opened with READ_ONLY Access only");
```

This method helps to determine the permission, read or write, with which the folder should be opened.

To close the folder object, you can use the public void `close(boolean)` method. This method can be called only when the folder object is opened. The Boolean parameter indicates whether the expunge operation should be performed on the folder object or not. If `TRUE` Boolean value is returned, the `expunge()` method is invoked.

Listing Messages

As discussed earlier, the `folder` object is used to store messages. These messages are returned in a list of array. Each message object returns a reference to the actual message and the returned message reference is stored in a list of messages.

The following code snippet shows all the messages within a POP folder and displays the subject field of each retrieved message:

```
inbox.open(Folder.READ_ONLY);

Message[] allTheMessages=inbox.getMessages();
for(int i=0;i<allTheMessages.length;i++)
System.out.println("ID:"+i+"Subject:"+allTheMessages[i].getSubject());

inbox.close();
mailStore.close();
```

The messages retrieved from a folder are references to the actual messages. Such `message` objects can be retrieved by calling the `get()` method. Clients use the `FetchProfile` class to list the `message` attributes to be fetched from the server. The following code snippet shows how to fetch message attributes:

```
void fetch(Message[] msgs,FetchProfile fp)
```

The following code snippet shows how to list messages using the `FetchProfile class`:

```
Message[] mymsgs=thisFolder.getMessages();
FetchProfile myfp=new FetchProfile();
myfp.add("To");
myfp.add("From");
myfp.add("Subject");
thisFolder.fetch(mymsgs,myfp);
for(int i=0;i<thisFolder.getMessageCount();i++)
{
  display(mymsgs[i].getTo());
  display(mymsgs[i].getFrom());
  display(mymsgs[i].getSubject());
}
```

The server-based message access protocols, such as IMAP and SMTP, are used to retrieve attributes of a group of messages sent in a single request. You can fetch group message attributes by using the `javax.mail.FetchProfile.Item` inner class, which is the base class of all group items that can be fetched from the `FetchProfile class`.

Copying and Moving Messages

The `Folder` class provides the `copyMessages()` method that is used to copy and move messages among different folder objects. The `copyMessages()` method appends messages to the destination folder object by invoking `appendMessages(msgs)` on the destination folder object. To append messages to the destination folder object, ensure that the destination folder object is not open.

Searching Messages

JavaMail provides the `javax.mail.search` package to define classes that can be used to search messages. A search expression searches the message based upon a specific criterion from a SearchTerm object. Search expressions are represented by the `SearchTerm` class available in the javax.mail.search package. SearchTerm objects are Serializable, which allows these objects to be stored between sessions. The folder object contains the following two methods to search messages on the basis of the specified search expression of the `SearchTerm object`:

- Message[] search(SearchTerm myterm)
- Message[] search(Searchterm myterm,Message[] mymessageList)

The following code snippet shows the use of the `search()` method:

```
SearchTerm myst=new SearchTerm(new FromStringterm("alan@n-ary.com"),
new FromStringTerm("cormac@n-ary.com"));
Message[] mymessageList=thisFolder.search(myst);
```

The subclasses of the `SearchTerm class` implement the following logical methods on the base class:

- **`AndTerm`**—Allows you to build `AND` expressions. The syntaxes of the overloaded AndTerm method are as follows:
  - `AndTerm(SearchTerm[])`
  - `AndTerm(SearchTerm, SearchTerm)`
- `OrTerm`—Allows you to build `OR` expressions. The syntaxes of the overloaded OrTerm method are as follows:
  - `OrTerm(SearchTerm[])`
  - `OrTerm(SearchTerm, SearchTerm)`
- `NotTerm`—Allows you to build NOT expressions. The syntax of NotTerm is as follows:
  - `NotTerm(SearchTerm)`
- `ComparisonTerm`—Serves as a subclass of the `SearchTerm class`, which is an abstract class. The subclasses of the ComparisonTerm class implements particular functionalities. The ComparisonTerm class has the following constants for numerical comparison types:
  - `ComparisonTerm.EQ`—Corresponds to equal to
  - `ComparisonTerm.GE`—Corresponds to greater than or equal to comparison
  - `ComparisonTerm.GT`—Corresponds to greater than comparison
  - `ComparisonTerm.LE`—Corresponds to less than or equal to comparison
  - `ComparisonTerm.LT`—Corresponds to less than comparison
  - `ComparisonTerm.NE`—Corresponds to not equal to comparison

Table 12.19 describes the classes of the `javax.mail.search` package and their constructor syntaxes to build different expressions:

Table 12.19: Describing the Classes of the javax.mail.search Package

Class	Description	Constructor Syntax
`AddressStringTerm`	Performs string comparisons for message addresses and extends the StringTerm class in a package.	`AddressStringTerm(String pattern)`
`AddressTerm`	Extends the javax.mail.search.SearchTerm class and is used to compare the search expression with message address. The StringTerm class and the AddressTerm class are different in functionality. The StringTerm class performs comparisons on address strings and the AddressTerm class performs comparisons on Address objects.	AddressTerm(address Add)
`BodyTerm`	Implements the MessageBody interface and extends the StringTerm class. This class performs searches on a message body.	BodyTerm(String Pattern)
`DateTerm`	Compares the search expression with dates and extends the ComparisonTerm class.	`DateTerm(int comparison, Date date).`

Table 12.19: Describing the Classes of the javax.mail.search Package

Class	Description	Constructor Syntax
`FlagTerm`	Compares the search expression with message Flag. The FlagTerm class extends javax.mail.search.SearchTerm.	FlagTerm(`Flags flags, boolean set`)
`FromStringTerm`	Extends the javax.mail.search.AddressStringTerm class and compares the search expression with string address.	FromStringTerm(`String pattern `)
`FromTerm`	Allows you to compare the From address header. The FromTerm class extends the javax.mail.search.AddressTerm class. The FromStringTerm class has a major difference from the FromTerm class. This class checks address strings than address objects, as in the case of the FromTerm class. The result address string comparison does not depend on whether the string expression contains lowercase letter or uppercase letter.	FromTerm(`Address add`)
`HeaderTerm`	Compares the search expression with message headers. The result of the comparison does not depend on whether the search expression contains lowercase letter or uppercase letter. The HeaderTerm class extends the javax.mail.search.StringTerm class.	HeaderTerm(String headerName, String pattern).
`IntegerComparisonTerm`	Compares the search expression with integer message number. The IntegerComparisonTerm class extends the ComparisonTerm class.	IntegerComparisonTerm(int mycomparison, int mynumber)
`MessageIDTerm`	Compares the search expression with the MessageId, which is a unique identifier for a message. A message can be searched in a folder object whose ID is same as the messageId argument passed to this method. The MessageIDTerm class extends the javax.mail.search.StringTerm class.	MessageIDTerm(String messageID).
`MessageNumberTerm`	Compares the search expression with message number. The MessageNumberTerm class extends the javax.mail.search.IntegerComparisonTerm class.	MessageNumberTerm(int number).
`ReceivedDateTerm`	Compares the search expression with the date on which the message was received. The ReceivedDateTerm class extends the javax.mail.search.DateTerm class.	ReceivedDateTerm(int comparison, Date date).
`RecipientStringTerm`	Compares the search expression with recipient address headers. The ReceipientStringTerm class extends the javax.mail.search.AddressStringTerm class.	RecipientStringTerm (Message.RecipientType type, String pattern).

Table 12.19: Describing the Classes of the javax.mail.search Package

Class	Description	Constructor Syntax
`RecipientTerm`	Compares recipient address headers with the address object passed to the constructor of the RecipientTerm class during the creation of the RecipientTerm object and extends the javax.mail.search.AddressTerm class. There is a difference between the RecipientStringTerm and RecipientTerm classes. The RecipientStringTerm class performs comparisons on address string objects and the RecipientTerm class performs comparison on address objects. However, string comparisons are case insensitive	RecipientTerm(`Message.RecipientType type, Address` address)
`SentDateTerm`	Compares the date on which message was sent to the recipient with the date passed to the constructor of the class during the creation of the SentDateTerm object. This class extends the javax.mail.search.DateTerm class.	SentDateTerm(int comparison, Date date).
`SizeTerm`	Compares the search expression of the SearchTerm object with the size of the message. The SizeTerm class extends the javax.mail.search.IntegerComparisonTerm class.	SizeTerm (int comparison, int size).
`StringTerm`	Compares a string with the pattern argument passed to its constructor.	StringTerm(`String pattern, Boolean ignoreCase`) StringTerm(`String pattern`)
`SubjectTerm`	Compares the search expression with message subject header. The result of the comparison does not depend on whether the search expression contains lowercase letter or uppercase letter.	SubjectTerm(String pattern)

The Transport Class

The `javax.mail.Transport` class is used to deliver messages to the recipients. This class extends the `Service` class. The `Transport` class provides the following three methods to send messages to the recipient:

- void `send(Message)` – Allows you to send a message
- void `send(Message, Address[])` – Allows you to send a message to the specified addresses
- void `sendMessage(Message, Address[])` – Allows you to send a message to the specified list of addresses

The Transport class throws the `SendFailedException` exception, if a recipient address is found incorrect during the message submission.

The following code snippet shows the use of the `Transport` class and the `send()` method:

```
try
{
  Transport myTransport=session.getTransport("smtp");
  myTransport.connect();
  myTransport.send(msg,msg.getAllreciepents());//msg is a MimeMessage object
  myTransport.close();
  }
  catch(SendFailedException sfe)
```

```
{
        Address[] list=sfe.getInvalidAddresses();
        for(int i=0;i<list.length;i++)
                System.out.println("Invalid Address:"+list[i]);
        list=sfe.getUnsentAddresses();
        for(int i=0;i<list.length;i++)
                System.out.println("Unsent Address:"+list[i]);
        list=sfe.getValidAddresses();
        for(int i=0;i<list.length;i++)
                System.out.println("Valid Address:"+list[i]);
}
```

The following three methods are used to handle errors related to an E-mail address:

- Address[] getInvalidAddresses() – Returns an array of Address objects that are not correct.
- `Address[] getUnsentAddresses()`–Returns an array of Address objects that are not accepted for delivery. It may be due to the reason that a server may not send a message to an outside domain user.
- `Address[] getValidAddresses()`–Returns an array of Address objects that are accepted for delivery.

After having a brief knowledge about the classes and interfaces of JavaMail API, let's now learn how to create an application for sending and receiving mails.

Working with JavaMail

As discussed earlier, JavaMail API provides several classes, such as Address, Authenticator, BodyPart, Flags, and interfaces, such as Part, to develop E-mail client applications.

In this section, let's create a customized E-mail client application, Kogent E-mail client, to understand how JavaMail helps in sending and reading an E-mail from the inbox of the desired E-mail address. Firstly, let's create an E-mail client that sends E-mails through SMTP. To create the E-mail client, create the `SimpleMailSender` class, which sends a message to the desired E-mail address. Next, you learn how to read an E-mail using JavaMail. To read an E-mail, create the SimpleMailReader class that reads the mails from the inbox of the desired E-mail address.

Sending Mails

JavaMail allows you to send an E-mail with or without attachments to one or many recipients. To send the E-mail, first you need to specify the domain name of the SMTP server by setting the value of mail.smtp.host property to the domain name of the SMTP server using the put() method of the property object, which is smtpout.secureserver.net in this application. The TCP/IP protocol is used to connect to the SMTP server to send E-mails on a single or multiple computers. You can develop your customize mailing system for sending an E-mail.

Listing 12.1 shows the `SimpleMailSender.java` file, which is created to send an E-mail through SMTP server (you can find this file on the CD in the `code\JavaEE\Chapter12` folder):

Listing 12.1: Showing the SimpleMailSender.java File

```
import java.security.Security;
import java.util.Properties;

import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;

import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

public class  SimpleMailSender
{
   private static final String SmtpServer = "smtpout.secureserver.net";
   private static final String MsgContent = "Test Message Contents";
   private static final String Subject = "A test Mail";
   private static final String mailFrom= "akanksha.saksena@kogentindia.com";

   private static final String[] sendTo = {"anuj.dixit@kogentindia.com"};
```

```
    public static void main(String args[]) throws Exception
    {

          Security.addProvider(new com.sun.net.ssl.internal.ssl.Provider());
          new SimpleMailSender().sendSSLMessage(sendTo, Subject, MsgContent , mailFrom);
          System.out.println("Sucessfully Sent");
    }

    public void sendSSLMessage(String recipients[], String subject, String message,
    String from) throws MessagingException
    {
          boolean debug = true;
          Properties props = new Properties();
          props.put("mail.smtp.host",  SmtpServer);
          props.put("mail.smtp.auth", "true");
          props.put("mail.debug", "true");
          props.put("mail.smtp.port","3535");
          Session session = Session.getDefaultInstance(props, new
          javax.mail.Authenticator()

          {
                protected PasswordAuthentication getPasswordAuthentication()
                {
                return new PasswordAuthentication("akanksha.saksena@kogentindia.com",
                "akanksha");
                }
          });

          session.setDebug(debug);
          Message msg = new MimeMessage(session);
          InternetAddress addressFrom = new InternetAddress(from);
          msg.setFrom(addressFrom);
          InternetAddress[] addressTo = new InternetAddress[recipients.length];
          for (int i = 0; i < recipients.length; i++)
          {
                addressTo[i] = new InternetAddress(recipients[i]);
          }

          msg.setRecipients(Message.RecipientType.TO, addressTo);
          // Setting the Subject and Content Type
          msg.setSubject(subject);
          msg.setContent(message, "text/plain");
          Transport.send(msg);
    }

}
```

In Listing 12.1, the `send()` method in the `SimpleMailSender` class is used to send messages. Note that a mail session is created by using the `java.mail.Session class` to send the mail. In the SimpleMailSender.java class, the `Session.getDefaultInstance()` method is called to get a shared session, which other applications may reuse. You can also set up an entirely new session by using the `Session.getInstance()` method. To launch a session, you are required to set certain properties, such as the `mail.smtp.host` property.

You can run the `SimpleMailSender class on your computer by changing the SMTP` server name (provided in Listing 12.1) with your SMTP server name.

On the basis of the authentication details of your SMTP server, you also need to modify the value of variables, such as PasswordAuthentication, mailFrom, and sendTo (provided in Listing 12.1). You should ensure that the mail.1.4.2.jar file is mapped to the classpath environment variable, before compiling the SimpleMailSender.java file. Execute the following commands from Command Prompt to compile and run the SimpleMailSender class:

```
javac SimpleMailSender.java
java SimpleMailSender
```

Figure 12.4 shows the output of Listing 12.1:

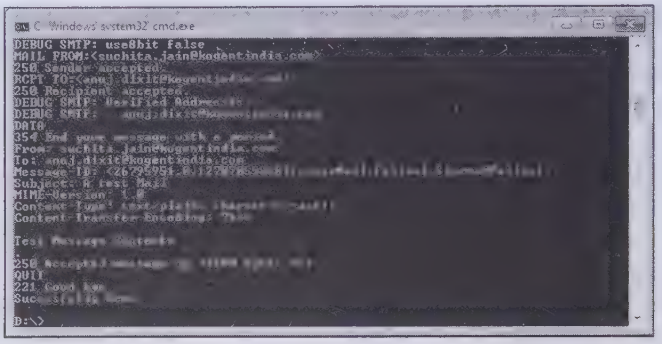


Figure 12.4: Showing the Output of the SimpleMailSender.java File

After creating a session, you need to create a message. The javax.mail.Transport class is used to send the message.

Reading Mails

JavaMail provides various features that allow you to read an E-mail using POP3. You can connect to the message store using the connect() method of the Store class, download messages, and optionally delete them from the server. Let's create the `SimpleMailReader.java` file that is used to read E-mails from the SMTP server, as shown in Listing 12.2 (you can find this file on the CD in the `code\JavaEE\Chapter12` folder):

Listing 12.2: Showing the SimpleMailReader.java File

```
import java.util.Properties;
import javax.mail.Authenticator;
import javax.mail.Folder;
import javax.mail.Message;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;
import javax.mail.Store;

public class SimpleMailReader
{

   public static void main(String[] args) throws Exception
   {

   Properties mailproperties = new Properties();
   String hostmailserver = "pop.secureserver.net";
   mailproperties.setProperty("mail.pop3.port", "110");
   mailproperties.put("mail.pop3.host", hostmailserver);
   mailproperties.put("mail.store.protocol", "pop3");
   Session session = Session.getDefaultInstance(mailproperties);
   Store store = session.getStore();
   store.connect( hostmailserver,110, "akanksha.saksena@kogentindia.com","akanksha");
   Folder inbox = store.getDefaultFolder().getFolder("INBOX");


   if (inbox == null)
   {
         System.out.println("No INBOX");
         System.exit(1);
   }
   inbox.open(Folder.READ_ONLY);
   Message[] messages = inbox.getMessages();
   for (int i = 0; i < messages.length; i++)
   {
         System.out.println("Message " + (i + 1));
         messages[i].writeTo(System.out);
   }
   inbox.close(false);
   store.close();
   }
}
```

In Listing 12.2, we have used the POP3 server of secureserver.net to access E-mails. To compile the SimpleMailReader class, execute the following command from the Command Prompt:

```
javac SimpleMailReader.java
```

To run the `SimpleMailReader class`, execute the following command from Command Prompt:

```
java SimpleMailReader
```

After executing the preceding command, the E-mails of your inbox are displayed on Command Prompt.

Figure 12.5 shows the output of Listing 12.2:

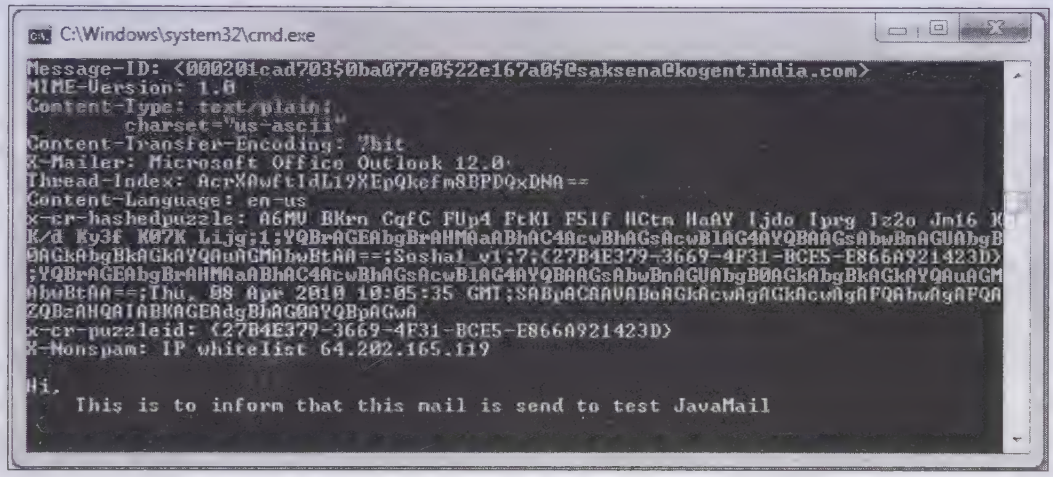


Figure 12.5: Displaying the Output of the SimpleMailReader.java File

This ends up the discussion on JavaMail. Let's now recall what you have learned in the chapter.

Summary

In this chapter, you have learned about the JavaMail and its API. The JavaMail API provides various classes and interfaces used to develop an E-mail client application in Java. You have also learned about various types of mail protocols, such as SMTP, IMAP, and POP3, which are used by JavaMail to send and receive E-mails. In addition, you have learned how to implement the JavaMail API by creating various E-mail applications.

In the next chapter, you learn about EJB 3.1.

Quick Revise

Q1. The BodyPart class belongs to the ..... package.

A. javax.mail.search
B. javax.mail.event
C. javax.mail.internet
D. javax.mail

Ans. D

Q2. The..... is an abstract class that represents a mail message.

A. javax.mail.Folder
B. javax.mail.Message
C. javax.mail.Address
D. javax.mail.Authenticator

Ans. B

Q3. The ..... method returns the total number of messages stored in a `folder` object.

A. getMessage()
B. getMessageCount()
C. getUnreadMessageCount()
D. getNewMessageCount()

Ans. B

Q4. **The `send()` method of the `Transport` class takes an argument of the type…….**

A. **Message** **B.** **Folder**

C. **BodyPart** **D.** **Multipart**

Ans. A

Q5. **The …… class is extended by the `javax.mail.Store` class.**

A. **javax.mail.Session** **B.** **javax.mail.Transport**

C. **javax.mail.Folder** **D.** **javax.mail.Service**

Ans. D

Q6. **What is the significance of JavaMail API?**

Ans. The JavaMail API provides a framework to build complete mailing and messaging applications. This framework is both platform and protocol-independent. All the API classes and interfaces have been organized in the `javax.mail` package and its sub packages. The JavaMail API helps the programmers to develop E-mail client applications in an efficient and manageable manner.

Q7. **What is a mail server?**

Ans. An application receiving and storing mails and messages from E-mail clients is known as a mail server.

Q8. **Define SMTP.**

Ans. SMTP is a standard E-mail protocol. It is an application layer protocol supporting messaging functions. It is used for sending mails.

Q9. **Differentiate between SMTP and POP3.**

Ans. Both SMTP and POP3 are Internet protocols. SMTP is used to deliver mails while POP3 is used to retrieve E-mails from the mail server by using an E-mail client. In other words, POP3 is used to download a message from a mailbox and SMTP routes the message from one server to another or from a client to a server.

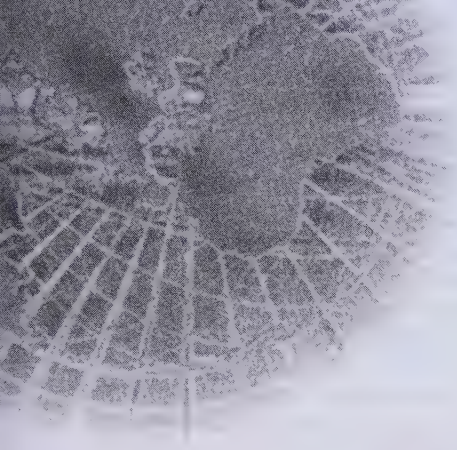
Q10. **What is the importance of `javax.mail.Authenticator` class?**

Ans. When we send or receive a message, we need to authenticate the connection. The `Authenticator` class provides the methods used to check the authenticity of the connection.

Q11. **Which methods of the `Transport` class help in delivering messages?**

Ans. The following methods of the Transport class help in delivering messages:

- void send(Message)
- void send(Message, Address[])
- void sendMessage(Message, Address[])



13

Working with EJB 3.1

An enterprise level application should contain various features, such as distributive, transactional, secure, and portable. Java EE introduced a new technology called server-side component architecture, more commonly known as Enterprise JavaBeans (EJB), which helps in implementing these features in Java enterprise applications. Earlier, a developer had to provide extra code and logic in the application to ensure that an enterprise application is distributive, transactional, secure, and portable. However, with the advent of EJB, these features were managed directly by the EJB container. EJB components and containers used in application servers also reduce the efforts of the developers, as they do not need to understand low level transaction and state management details, connection pooling, multithreading, and other similar complex processes related to application development. The Java EE 6 specification supports EJB 3.1, the latest version of EJB.

In this chapter, while discussing the EJB technology, the main focus is placed on the new concepts introduced by EJB 3, and how it has made developing a Web application easier. Firstly, you learn about the fundamental concepts of EJB 3, followed by the discussion on various types of EJBs, such as session beans and Message-Driven Bean (MDB). Next, the chapter helps you to understand how to manage transactions in Java EE applications. In addition, the classes and interfaces used to implement EJB 3 timer services are described. Moreover, the chapter explores how to work with EJB 3 interceptors and implement life cycle callback methods in the MDB and session beans.

Let's start by learning about the fundamentals of EJB 3.

Understanding EJB 3 Fundamentals

EJB is a standard architecture used for enterprise level applications that are object-oriented, transaction-oriented, and distributed. In the EJB architecture, the enterprise beans are managed by the EJB container. To create an enterprise bean application, you first need to create an interface containing various user-defined enterprise bean methods that encapsulate the business logic of an enterprise application. Next, the enterprise bean class implements the interface and provides the body for the enterprise bean methods. Finally, the EJB clients are created to invoke these methods remotely. The configuration details of the enterprise beans are provided in Deployment Descriptor.

In addition to business logic, you can also provide service information in an enterprise bean by using annotations or Deployment Descriptor. The enterprise beans can use the container for all the services defined in Deployment Descriptor. You need to create a client view to access an enterprise bean and display the output of the bean. The client view refers to creating another enterprise component or Java program, such as an applet or a servlet, to access an enterprise bean and its business methods. The client view of the enterprise bean is independent of the type of container and server in which the enterprise bean is deployed.

NOTE

Please note that throughout the chapter, the container refers to the EJB container.

An enterprise bean can be used to:

- Provide all stateless services, where the client state is not required to be maintained. In addition, you can also use an enterprise bean as a Web service endpoint that provides a stateless service.
- Provide stateless service asynchronously with the arrival of some message.
- Provide stateful services, where the conversational state of a client is maintained.
- Represent a persistent object as an entity, which is managed automatically by the container for its persistence and relation with other entities.

Let's now discuss the need, architecture, and features of EJB 3.

Why EJB 3?

The earlier version of EJB, EJB 2.1, was powerful enough to support all the needs of a distributed enterprise application. However, the development of the distributed application using EJB 2.1 was very complex because a developer has to create a number of interfaces and classes for implementing a single enterprise bean application. In addition, all the interfaces and classes created in an enterprise application need to implement interfaces and

extend classes from the `javax.ejb` package. Moreover, the client had to use Java Naming and Directory Interface (JNDI) look up to access enterprise bean and invoke its business methods.

EJB 3 introduced with the Java EE 5 specification has made the EJB technology simpler and easier to be implemented in the applications. EJB 3 needs fewer interfaces and classes as it supports metadata annotation to support Dependency Injection (DI). In EJB 3, the use of annotations has removed the need of Deployment Descriptor, which was used to configure an enterprise bean. In other words, in EJB 3, you can directly configure an enterprise bean using annotations instead of providing the configuration details in Deployment Descriptor. Moreover, the introduction of EJB 3 has simplified the use of entity persistence and Object Relational Mapping (ORM). As the EJB container provides various services, such as transaction and security, the developer does not need to provide code for these services in an application. This gives the developer enough time to concentrate on the implementation of the business logic. EJB 3 enhances business logic implementation; thereby, speeding up the process of enterprise bean development.

EJB 3 provides the following functionalities to simplify the development of enterprise applications:

- Leverages Java language metadata, which helps simplify all bean types and resource accesses. It reduces the need of Deployment Descriptors; however, it preserves the ability to use an Extensible Markup Language (XML) file as an alternative mechanism to override annotations.
- Defines an enterprise bean in the form of Plain Old Java Objects (POJO) instead of implementing EJB component interfaces.
- Provides all business interfaces as Plain Old Java Interfaces (POJI).
- Eliminates the use of the Home interfaces.
- Allows you to define all callback methods by using annotations.
- Simplifies the persistence model.
- Provides support for reusability of existing components in a new application. The EJB 3 specification helps in migrating from the EJB 2.1 applications to new EJB 3 applications.

EJB 3—Architecture and Concepts

The EJB 3 architecture comprises the EJB server, EJB container, and EJB client. In an enterprise application server, the third party vendors provide the container to process an enterprise bean or EJB file. Let's discuss how an enterprise bean or EJB file is processed.

The first step in the EJB processing model is to process an EJB file to generate deployment artifacts (required interfaces and Deployment Descriptors) according to the EJB 2.1 deployment model. Then, the EJB component is deployed on the EJB server. The deployment artifacts generated in the EJB processing model can be non-standard; however, they are similar to Deployment Descriptors defined in EJB 2.1.

Figure 13.1 shows the EJB 3 architecture:

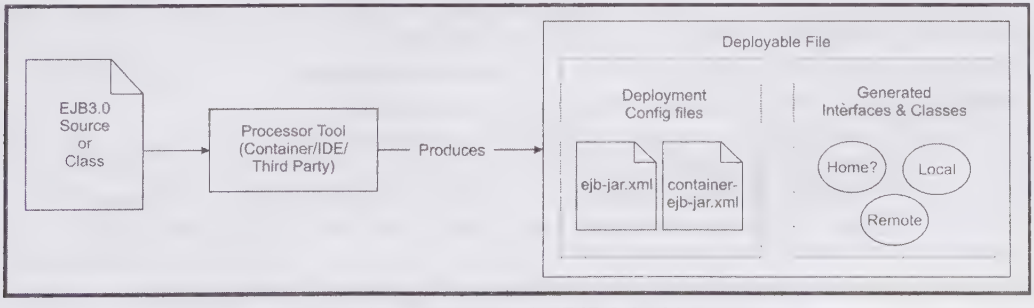


Figure 13.1: Displaying the EJB 3 Architecture

Another EJB 3 processing model is a JavaServer Pages (JSP) drag-and-drop deployment model. This model allows you to add an EJB file into a pre-designated, implementation-defined directory, which can be picked up by the container for processing, deploying, and making it available for use.

Now, let's discuss the EJB architecture in detail by understanding the EJB server, the EJB container, and the EJB client.

The EJB Server

The EJB server provides a runtime environment to execute server applications that use enterprise beans. The EJB server is used to create an infrastructure to deploy server applications also called components. It provides a JNDI-accessible naming service, manages and co-ordinates the allocation of resources to client applications, provides access to system resources, and provides a transaction service.

In the EJB architecture, servers are basically the resource managers. Every server manages the allocation of resources to the containers it controls. It is the resource manager that determines which and how many containers may run within the server.

The EJB server performs the following functions:

- Manages all incoming client requests at the client-level. This also includes providing connections to clients, dispatching client requests to containers, and routing responses back to the clients.
- Manages the processing of the resources that are available to the container at the container-level. This type of management includes allocating available working processes and threads to the running containers.
- Verifies that the container has access to shared services, such as a centralized security manager, a pool of Java Database Connectivity (JDBC) drivers, a transaction service implementation, a global cache manager, and an asynchronous messaging service. This function is done at the server-level.

You can run an EJB application on more than one server. Therefore, the servers must cooperate as a group to allocate resources efficiently across all the containers in a cluster. This cooperation among the servers first provides the status of services to a load balancing process and then assigns the client requests to the least-loaded server to provide the requested services.

The EJB Container

The EJB container provides an environment in which one or more enterprise beans can run. This environment is basically a combination of the available interfaces and classes that the container uses to support enterprise beans throughout their life cycle. The EJB container intercepts all method calls and provides the following services to EJB components:

- **Life cycle management**—Creates and removes enterprise bean instances and handles instance pooling with the activation and passivation of the bean instances
- **Naming services**—Provides JNDI registration of EJB when an enterprise bean is loaded on the Java EE server
- **Security checks**—Performs authentication and access control as well as implements declarative and programmatic security
- **Persistence management**—Helps to store the enterprise beans
- **Transaction coordination**—Defines declarative transaction management
- **Resource pooling**—Provides an indirect access to the bean instance

The EJB container simplifies various complex aspects of a distributed application, such as security, transaction coordination, and data persistence. The EJB infrastructure is implemented by the EJB container as well as service providers and this infrastructure deals with the distribution aspects, transaction management, and security aspects of an application. The Java Application Programming Interfaces (APIs) for the EJB infrastructure are defined by the EJB specification. Therefore, the developers need to focus only on the implementation of the business logic in the application, as the presentation logic can be implemented by the designers.

Figure 13.2 shows the services provided by the EJB container :

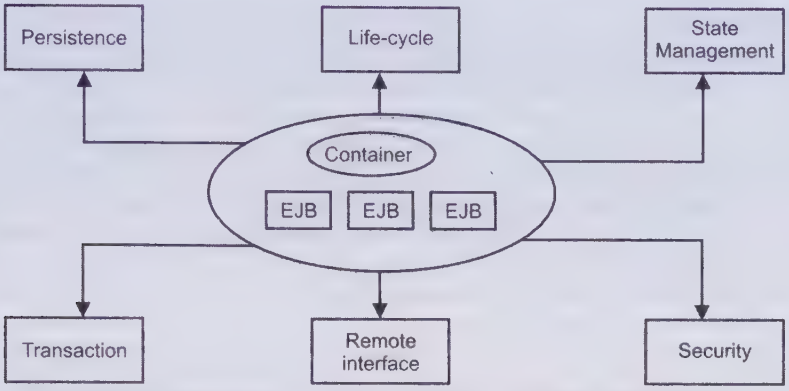


Figure 13.2: Displaying the Services Provided by the EJB Container

The EJB container performs the following tasks for each enterprise bean:

- Registers the object of an enterprise bean on the Java EE server
- Provides a Remote interface for the object
- Creates and destroys object instances
- Checks security for the object
- Manages the active state for the object (activation/passivation)
- Coordinates distributed transactions
- Persists Container-Managed Persistence (CMP) entity beans and saves stateful session beans attributes when passivated

Whenever a client wants to call business methods of an enterprise bean, it needs to connect to the container first. Then, the container verifies that whether or not the call follows the semantics specified in Deployment Descriptor and then dispatches the request to the enterprise bean. This mechanism enables the container to control all the aspects of the execution of an enterprise bean, including the following aspects:

- **Life cycle**—Refers to the invocation of the life cycle methods of an enterprise bean according to the EJB 3 specification. While managing the life cycle of an enterprise bean, the container itself enforces security, transaction, and persistence requirements.
- **Security**—Refers to the permissions set for users to access an enterprise bean. When a client attempts to call a method of EJB, the container looks for the user's permission in an Access Control List (ACL) to verify whether the client has the right to invoke that method. Deployment Descriptor contains the access control declarations about the EJB's methods.
- **Transaction**—Refers to the declarations about the transaction restrictions on the EJB's methods included in the Deployment Descriptor object. When a client attempts to call the method of an EJB, the container verifies whether or not the call obeys the transaction restrictions on that method.
- **Persistence**—Refers to the persistence storage of data of an EJB. There are two different types of persistence–one that is managed by beans and the other that is managed by the container. In the bean managed persistence, the beans are responsible for implementing data access as the code for data access functions need to be provided in the bean class; whereas, in CMP, the implementation of data access functions are done by the container.

The EJB Client

The client code is the code written to execute business logic embedded in an enterprise bean and its methods. An EJB client can be a simple Java program or a Web client. The end-users run these clients to get results from the enterprise bean method invocation. The EJB client can use resource injection or JNDI service to access an enterprise bean. The EJB client can be a simple Java class, applet, servlet, or JSP. An enterprise bean is looked up using resource injection or the lookup() method, irrespective of the different EJB clients.

Features of EJB 3

The EJB 3 specification addresses all the complexities of its previous versions and solves them by implementing new strategies. The new features added in EJB 3 have changed the way the enterprise beans are developed, configured, and deployed. The extensive use of metadata annotations has reduced the use of different interfaces, classes, and Deployment Descriptors. In EJB 3, the container generates the required interfaces and Deployment Descriptor for an application. Due to this, the developers do not need to focus on the development of local or remote home and component interfaces. In addition, in EJB 3, the callback methods can easily be marked in the enterprise bean. The enterprise bean has been simplified to a POJO model and all resources to be used by the components are injected using DI.

In this section, you learn about the following features of EJB 3:

- Annotations
- Elimination of the Home interface
- Elimination of the component interface
- Callback methods
- Simple POJO model
- Java Persistence API
- Dependency injection
- Timer service
- Interceptors

Annotations

The annotation feature is introduced in the Java Platform, Standard Edition 5 (Java SE 5). These annotations are inspected at compile time or runtime by different tools to generate additional constructs, such as Deployment Descriptors, and to customize the component's runtime behavior. You can annotate class fields, methods, and classes. The set of annotations provided by EJB 3 simplifies the development of enterprise beans. You can mark interfaces, classes, fields, and methods created for an EJB implementation with different annotations, such as @Remote, @Stateful, and @Stateless. You should provide annotations only when the default values cannot be used. For example, for an enterprise bean CustomerBean, you do not need to extend `javax.ejb.EnterpriseBean` type of class; instead, you can use @Stateless annotation to declare it as a stateless session bean, as shown in the following code snippet:

```
package com.kogent.ejb;
import javax.ejb.Stateless;
@Stateless
public class CustomerBean
{
  public String sayHello()
  {
        return "Hello from CustomerBean.";
  }
}
```

Elimination of the Home Interface

In EJB 2.1, you need to create a home interface (local or remote) by extending either the `javax.ejb.EJBHome` interface or the `javax.ejb.EJBLocalHome` interface and declaring life cycle methods, such as create() and remove(). In EJB 3, you do not need to create locale or remote home interfaces for an enterprise bean. The EJB 2.1 APIs are still supported in EJB 3 specification and the home interfaces can be used to declare the enterprise bean methods.

Elimination of the Component Interface

The earlier versions of EJB require the local or remote component interfaces to be created for the given enterprise bean. The local component interfaces require extending the javax.ejb.EJBObject class; whereas, the remote component interfaces require extending the javax.ejb.EJBLocalObject class. The component interfaces were used

to declare the business methods that a client could invoke. In EJB 3, you do not need to create component interfaces as they have been replaced by business interfaces that declare all business methods to be provided to the client. The business interface can be local or remote; however, both local and remote home interfaces are simple POJI.

Callback Methods

In EJB 3, you do not need to define all the life cycle callback methods, such as `ejbPassivate()` and `ejbActivate()`, in the enterprise bean class. In the earlier versions, you have to define these callback methods to provide various implementations in an enterprise bean class; however, in EJB 3, they have become optional. In EJB 3, you can mark an arbitrary method as the callback method, which can listen EJB life cycle events. You can use different metadata annotations, such as @PreDestroy, @PostConstruct, @PrePassivate, @PostActivate, and @Remove to mark a method as callback method, as shown in the following code snippet:

```
@Stateful
public class SomeSessionBean
{
   @PreDestroy
   public void someMethod()
   {
        /*Some Logic to be executed before the destroy() method*/
   }
}
```

Simple POJO Model

In EJB 3, an enterprise bean is a simple Java class that does not implement any interface or extend any class. A simple POJO object can be made a powerful component that can handle concurrency, transactions, and security issues. All these functionalities are provided by the container to the simple POJO objects.

The POJO model also helps in creating enterprise bean classes that do not depend on other APIs. The POJO objects help in implementing the unit testing by using frameworks, such as JUnit, without deploying them on a server. The implementation of the POJO model has simplified the development of enterprise beans to a great extent. To create an EJB application, you just need to create simple business interfaces that are POJIs and implement business methods in an enterprise bean class, which is a POJO. All these resources, such as business interfaces and enterprise bean class are made accessible in the component by resource injection using annotations. In addition, all callback methods can be marked with annotations in the enterprise bean class.

Java Persistence API

EJB 3 includes a new Java Persistence API (JPA) to simplify the task of storing the data of an enterprise bean permanently in a database. ORM and all persistence logic are now handled by the container and you just need to provide proper annotations, such as @Id, @Table, @OneToOne, and @OneToMany, in the enterprise bean class. These annotations are used to map entities and their relationships to application's database.

An enterprise application helps to persistently store data of an organization. Initially, JavaBeans was used to handle the organization's data and the developers need to provide code to directly establish connection of the application with the database. In case of large amount of data, the direct communication with database led to the problem of heavy load on the server. As the solution to this problem, JPA was introduced in the Java EE 5 platform to store the data persistently with the help of an enterprise bean. The first release of JPA lacked many features, such as annotations and brought many problems to the developers. For example, the developer needs to provide the code to manage the relationships of the entities or store the foreign key fields of a database in the bean class. The EJB2.1 specification introduced the container-managed entity beans in which the EJB container was responsible to manage the entity relationships. In addition, in the container-managed entity beans, the EJB server was responsible for generating subclasses to manage the persistent data. The EJB2.1 specification also introduced the Enterprise Java Bean Query Language (EJBQL)--a query language designed to create queries for CMP entity beans. This language is similar to SQL and is used to search the persistent attributes of the enterprise beans.

The EJB 2.1 specification, despite of the improved features than its earlier versions, was still overloaded with the major problems and complexities. The problems and complexities were reduced with the introduction of EJB 3,

which provides JPA as a simplified programming model for entity persistence. Now, ORM or persistence approach uses the POJO model instead of the abstract persistence schema model to simplify the complexities of EJB 2.1. ORM maps entities and their relationships to application's database. An EJB 3 entity depicts persistent information stored in a database by using CMP; however, EJB 2.1 entity bean only represents persistent information stored in the database. The optimistic locking technique that was supported only by the TopLink persistence framework is also encapsulated in the EJB 3 specification. This technique allows you to use objects in a disconnected model implying that you can change data and the relationships of objects offline and merge data operations into one transaction. The following are the features and functionalities provided by JPA:

- Requires less number of classes and interfaces
- Introduces a new EntityManager API, similar to Hibernate, which is used to perform various operations, such as creating, removing, and searching entity beans
- Eliminates the use of the lengthy Deployment Descriptors by facilitating the use of annotations
- Provides better ORM
- Eliminates the need for lookup code
- Provides better support for inheritance and polymorphism
- Adds support for named (static) and dynamic queries
- Allows you to perform many database related operations, instead of performing only one operation generating primary key
- Provides a Java Persistence query language—an enhanced EJB QL
- Makes testing the entities without the EJB container easier. Earlier, the developers need to be aware of deployment platform to test EJBs.

The preceding changes in the new JPA are explained in detail in *Chapter 14, Implementing Entities and Java Persistence API 2.0.*

Dependency Injection

The client who wants to use the constructed bean in an application needs to know how to locate and invoke that enterprise bean (constructed bean). The client for an EJB 2.1 session bean gets the reference of the session bean with JNDI. To use an enterprise bean, Calculator, you need to add the following code snippet in the EJB 2.1 client to locate and invoke an enterprise bean method:

```
Context ic = new InitialContext();

Object obj = ic.lookup ("CalculatorJNDI");

CalculatorHome home =

(CalculatorHome)PortableRemoteObject.narrow(obj, CalculatorHome.class);

Calculator calc = home.create();
```

In the preceding code snippet, the JNDI name of the Calculator bean is CalculatorJNDI. The local/remote instance is obtained with the create() method. However, in EJB 3.0, the JNDI lookup and create() method invocation are not required. In EJB 3.0, a reference to a resource is obtained with DI using annotations. EJB, by implementing DI, conveys to the container that a particular resource is dependent on some other resource. A DI comprises the type of resource, the resource properties, and a name to access the resource. The use of annotations in EJB 3 has simplified the task of both the developers and the EJB client. Some examples of DIs are shown in the following code snippet:

```
@EJB (name="sessionBeanName", beanInterface=SessionBeanInterface.class)
@Resource (name="Database", type="javax.sql.DataSource.class")
```

The @EJB annotation used in the preceding code snippet injects stubs of a session bean having the sessionBeanName name in a Java class. The @Resource annotation is used to inject a service object having the `Database` JNDI name. This name may be present in either global or local JNDI tree.

DIs may be associated to a bean class or the member variables and methods of a bean class. The information to be specified in DI depends upon the context and the amount of data to be fetched from that context.

Timer Service

In enterprise applications, sometimes you need to implement time-dependent services. For example, you may need to invoke a particular business method provided by an enterprise bean after a given span of time or repeatedly after certain fixed time intervals. Prior to EJB 2.1, developers had to write code manually for building and deploying time-based workflows. However, with EJB 3, the task of creating such applications has been considerably simplified. Equipped with annotations and DIs, developers can use EJB 3 to build and deploy scheduled applications easily.

Interceptors

The interceptors are used to intercept the invocation of business methods of an enterprise bean to provide some additional functionality. The methods of interceptor can be defined in a separate Interceptor class and invoked before the business method. These Interceptor classes are then used with the session beans and MDBs. For example, you can invoke more than one interceptor on an enterprise bean if you need to validate all passed values to the business methods before executing the actual logic.

Let's use the @Interceptors annotation to import a chain of interceptors associated with the bean. You can define interceptor methods by using the @AroundInvoke annotation. The following code snippet shows how to add interceptor methods to a bean:

```
@Stateful
@Interceptors({MethodProfiler.class})
public class SomeServiceBean
{
   ...
   ...
}
```

After having the basic knowledge of EJB, let's explore different types of EJBs.

Classifying EJBs

There are different scenarios in the enterprise application development where you need to implement different types of enterprise beans. Different enterprise beans provide different functionalities. An entity bean represents an object in a persistent state; whereas; an MDB can be used only when a Java Message Service client is needed, and the user session bean is used to maintain the conversational state of the user. The enterprise beans can be classified into the following three types based on the type of functionality provided by them:

- Session beans
- Message-driven bean
- Entity beans

Now, let's discuss session beans and MDB in detail. The next chapter provides a detailed knowledge of entity beans.

Introducing Session Beans

A session bean is a type of enterprise bean that exists only for the client and server sessions. A session bean is generally created when a client requests a database with the help of a query to retrieve the required data. The bean exists till the client and server session persists. Almost all the services start with session beans, such as LoginVerificationBean and ShoppingCartBean. The session beans hold business processes, such as delivering an order and doing financial calculations for an application. These beans are reusable components, which provide methods that can be accessed by the client. All the services provided by the session bean are in the form of different methods.

As a session bean is never shared among multiple clients, the methods of a particular bean can only be executed by a single client. The length of a session determines the duration for which a session bean remains in use. This implies that a session bean is a short-lived object.

The actions performed using the methods of a session bean may lead to the change of data of an enterprise bean. This results in change in the state of the enterprise bean that may be persistent or transient. A persistent change

is available for the multiple sessions of a client; however, a transient change is available for a single session. There are two types of session beans:

- The stateful session bean
- The stateless session bean

The difference between stateful and stateless session beans is that a stateless session bean can be used for another client also, while a stateful session bean cannot be used for another client. These two types will be discussed in detail later in this chapter.

Prior to the discussion on the stateless and stateful session beans, some of the session bean concepts, such as conversational state as well as state management of a session bean are described in the following sections. It would make the concept of working of the whole session bean architecture clear.

Conversational State

An action performed by a method provided by the session bean may change the state of the session bean instance. The state of a session bean instance is known as its conversational state. In other words, a conversational state can also be defined as the state of all the attributes of session bean instances, state of connections (database, open source), and the reference to some other objects.

If the state of a session bean is stored for one client, then a different bean instance is needed for other clients. This may increase the number of session bean instances on the server. This can be avoided by the EJB container by writing the state of the stateful session bean to a secondary storage device. This process of removing the session bean from the active storage is known as passivation—the reverse process of activation. The EJB container uses serialization at the time of passivation and deserialization at the time of activation. In this way, the state can be preserved and the bean instance can be used to hold the new conversation with the new client.

State Management of a Bean

The word state refers to the state of all the attributes of a session bean class. The state of a bean is managed by the EJB container. The way the state of a session bean is maintained depends upon the type of the session bean.

If the session bean is of type stateless, then it holds the state for a single method invocation. The next invocation of any method on the same instance of the bean does not need to know the past invocation results. In other words, there is no conversational state to be managed. All the clients inside the bean pool are equivalent and any client can be serviced by any other client.

In case of a stateful session bean type, the new state of the bean is decided by the action of the next method invoked and the result of the action performed by the last method call. The state of the bean is maintained across multiple method calls or transactions for a particular client, that is, particular session. A single bean instance has a particular state for a particular client and thereby cannot be reassigned to another client. When the number of active beans allowed by the container exceeds, the container passivates the beans and reactivates them when they are required. In other words, the container passivates those active beans which are not required and activates them whenever they are required.

According to the type of the session bean, the state can be maintained either for a single method call (stateless) or the sequence of method invocations (stateful). All the states of a session bean are managed by the container.

The Stateless Session Beans

Some business processes do not need the state of a bean to be saved through multiple method invocations. The session bean representing such business processes will be a stateless session bean. A stateless session bean does not require its conversational state to be stored. The term stateless explains the fact that this type of session bean does not hold the state of a bean between different method calls.

In other words, a stateless session bean does not have any conversational state as the bean does not hold client details after a method call. The algorithm implemented by an EJB container decides whether the bean is to be destroyed or to be retained for reuse by some other client. The information held by a stateless session is not specific to any client.

After having a brief discussion about a stateless session bean, let's now explore its life cycle.

The life cycle of a stateless session bean can be described with various methods that are invoked throughout its life. A stateless session bean can be in one of the following two states:

- Does Not Exist state
- Ready state

You can change the state of a stateless session bean by invoking different methods, such as `ejbCreate()` and `ejbRemove()`.

Figure 13.3 represents the life cycle of a stateless session bean:

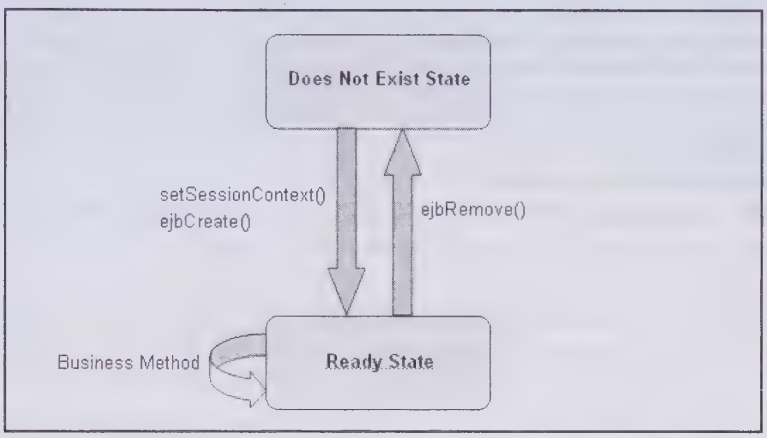


Figure 13.3: Displaying the Life cycle of a Stateless Session Bean

Figure 13.3 shows the two major transitions in the state of a stateless session bean. These transitions are from the Does Not Exist state to the Ready state and from the Ready state to the Does Not Exist state. Let's explain these transitions in brief.

Moving from the Does Not Exist State to the Ready State

There are some callback methods that are used to change the state of a stateless session bean from the Does Not Exist state to the Ready state. These methods are as follows:

- `setSessionContext()` – Contains the reference of the session context. After creating a stateless session bean instance, the EJB container places it into the Ready state. This instance then invokes the setSessionContext() method. The main purpose of calling this method is to associate a bean with a session context because the session context behaves as the gateway to interact with the container.
- `ejbCreate()` – Refers to the method that is invoked only once during the lifetime of the session bean. After the `setSessionContext()` callback method is called, the EJB container calls the `ejbCreate()` callback method to initialize the beans. It is important to remember that the `ejbCreate()` method is always called after the `setSessionContext()` method, as the enterprise bean can be created only after the session context has been created.

When the bean instance is in the Ready state, it can service a client's request and invoke the business method. The EJB container uses the available bean instance to execute the business method. After the execution of the business method is finished, the session bean instance moves from the Ready state to the Does Not Exist state.

Moving from the Ready State to the Does Not Exist State

Suppose the EJB container has a large number of session bean instances and one of these instances needs to access some resources or services of the container. In such a situation, the container needs to reduce the number of session bean instances that are in the ready pool. To reduce the number of session bean instances, the EJB container calls the `ejbRemove()` callback method on the session bean instances that are not in use. Using this method, the EJB container invalidates the reference to the bean instance that is in the ready pool. It does not indicate that the bean has been destroyed; it only sets the status of the bean instance as an inactive one.

The Stateful Session Beans

When a business process needs to maintain the client's state, the stateful session bean should be used. The stateful session beans are basically used for the process in which the state of the bean should be retained to be used by the same client during further method invocations. One of the good examples of a stateful session bean is ShoppingCart. Each method call in a ShoppingCart needs to know the current state of the shopping cart, that is, the number of items in the cart. If the state of the bean changes during the execution of the method, then the changed state should be available to the same client on further method invocations.

The stateful session bean acts on behalf of a particular client, and maintains the client's particular information throughout the session. To understand it more clearly, let's take the example of Online Banking. After a user logs on with the account_id and password, the user can continue with any transaction, such as withdrawal or deposit until either the user logs out or the session expires. All these transactions execute successfully, because during the login period of the user, the particular information of its account_id is being maintained across multiple transactions until the client session expires. In this way, the stateful session bean creates a conversational state across multiple method calls and transactions.

After having a brief discussion about the stateful session bean, let's now explore its life cycle.

The life cycle of the stateful session bean contains the following states:

- **Does Not Exist state** – Refers to the state in which the bean instance does not exist
- **Ready state** – Refers to the state in which the bean instance is tied to a particular client Passive state: Refers to the state in which the bean instance is passivated to optimize the resource utilization

The different types of transitions as well as various methods that are available inside the life cycle of the stateful session bean are shown in Figure 13.4:

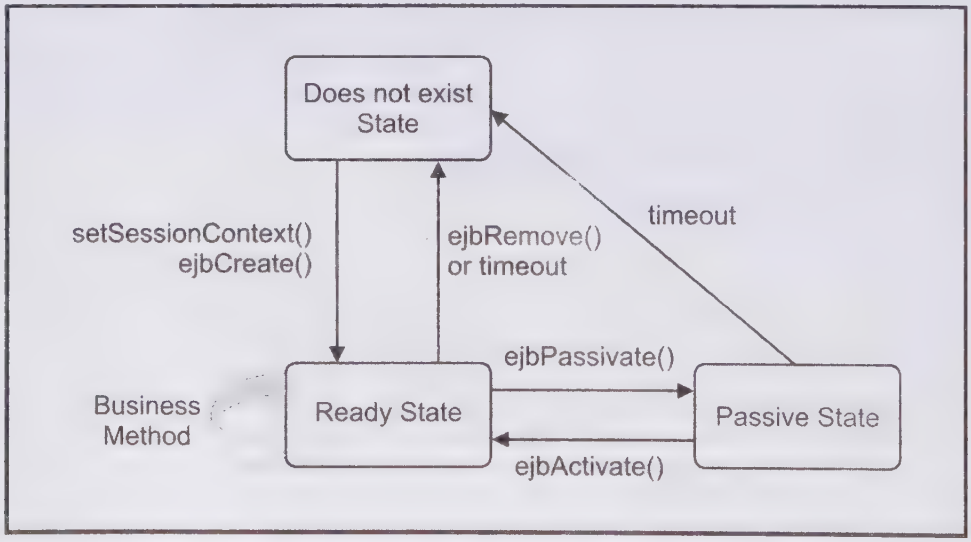


Figure 13.4: Displaying the Life Cycle of a Stateful Session Bean

Let's now discuss different transitions and methods of the life cycle of the stateful session bean.

Moving from the Does Not Exist State to the Ready State

The Ready state of the stateful session bean is similar to the Ready state of the stateless session bean. When there is interaction between the Does Not Exist and Ready states and the state of transition is from the Does Not Exist state to the Ready state, the following methods are used:

- setSessionContext()
- ejbCreate()

The functionality of both the `setSessionContext()` and `ejbCreate()` methods is similar as described in the stateless session bean section in the chapter.

In the Ready state, the session bean instance is associated with a particular client for the duration of the conversation. During this duration, the instance executes the methods that are invoked by the client.

Moving from the Ready State to the Passive State

Suppose a client does not invoke a bean for a long time and lets it remain in memory. As it is not a good practice to keep a bean in memory; therefore, to save memory, the client employs the process of passivation. In the process of passivation, the bean instances that are no longer used are saved into a disk or in some permanent storage device. The container performs this task by serializing the entire bean instance and then moving it into a permanent storage, such as a database or a file.

The EJB container can passivate a session bean by moving it from the Ready state to the Passive state. During passivation, data will be serialized and written back to the disk. To perform a task, the container calls the `ejbPassivate()` callback method. In the Passive state, the bean instances can be categorized as:

- **Not Recently Used (NRU)**—Lists the items that are not used recently
- **Least Recently Used (LRU)**—Lists the item that is used least recently

In the Passive state, if the stateful session bean is set to use the NRU algorithm, the session bean can perform the time-out operation. It means the lifetime of a session bean has expired and it moves to the Does Not Exist state, leading to the invocation of the `ejbRemove()` method by the container. However, if a stateful session bean is set to use the LRU algorithm, the instance of the stateful session bean cannot be expired while it is in the Passive state.

Moving from the Passive State to the Ready State

When a bean instance is needed again by a client, then the instance should move from the Passive state to the Ready state. The process of moving from the Passive state to the Ready state is known as activation. In this process, the container activates the bean instance by retrieving it from the permanent storage, then deserializes the bean instance, and finally sends it back to the memory for future use.

After passivation, if the client wants to continue the conversation by invoking a business method, then it needs to reactivate the session bean instance. The data stored in the disk is used to restore the bean instance state by calling the `ejbActivate()` callback method.

Moving from the Ready State to the Does Not Exist State

When the client completes its task and logs out from the application, the lifetime of the session expires. In this case, the client application invokes a remove method, terminates the conversation, and informs the EJB container to remove the instance. To perform all these tasks, the container calls the `ejbRemove()` method.

Stateless versus Stateful Session Beans

Depending upon the requirement of the business process, you can select whether to use the stateless or the stateful session bean. As implementation of the stateless session bean is easy; therefore, it is preferred over the stateful session bean. However, in cases where the bean instance requires association with the same client for the whole session, the stateful session beans are used.

The stateless session beans are good for the business processes in which you do not need to maintain the state for a single client among the number of methods invoked. The EJB container can use any free stateless session bean from the pool to serve the client request as the stateless session bean is not associated to a particular client. This makes stateless session beans more scalable.

A stateful session bean helps in preserving the bean state that is used for further business method invocations. Therefore, it is recommended to use the stateful session bean for the implementation of such business processes. The stateful session bean is also used whenever a bean requires storing information about a single particular client. The stateful session bean can be used for a single client only.

Implementing Session Beans

The implementation of the stateless and stateful session beans involves development of different components. Therefore, before developing an application using a stateless session bean, you should get familiar with the following components:

- Business interface
- Bean class

Let's discuss these components, one by one.

Exploring Business Interface

An interface through which a client is able to access a bean is known as a business interface. Generally, this interface contains bean methods that are required for developing applications related to beans. The bean methods declared in business interface are known as business methods. This interface is necessary for the stateless session bean. The business interface can be of two types—local interface and remote interface. You can define the type of a business interface by using annotations, such as `@Remote` and `@Local`. If you do not provide any annotation, then this interface becomes local business interface.

Exploring Bean Class

The stateless session bean is represented by a class known as the bean class, which implements business interfaces to provide implementation to a business method. The bean class can implement multiple business interfaces. A stateless session bean should be defined as stateless in Deployment Descriptor or it must be annotated with the @Stateless annotation. Due to the use of the @Stateless annotation, you need not to implement the `javax.ejb.SessionBean` interface in a bean class, as it was required in earlier versions of EJB.

NOTE

javax.ejb.SessionBean contains methods, such as setSessionContext(), ejbremove(), ejbActivate(), and ejbPassivate(). The stateless annotation is denoted as @Stateless and it indicates that the bean is a stateless session bean.

After understanding about the bean interface and bean class, let's develop an application using a stateless session bean.

Working with a Stateless Session Bean

Now, let's develop a simple application, Hello, to demonstrate how to work with a stateless session bean. Perform the following steps to develop the Hello application:

- Create a business interface
- Create a bean class
- Create a client
- Create the directory structure of the Hello application
- Package the application
- Deploy the application
- Run the application

Let's perform these steps, one by one.

Creating a Business Interface

A business interface is designed to perform the following tasks:

- Defines the client view of the bean
- Declares all the business methods

The business interface created in this application is HelloRemote and the only business method declared is hello().

Listing 13.1 provides the code for the HelloRemote.java file (you can find this file on the CD in the code\JavaEE\Chapter13\Hello\Hello-ejb\src\com\kogent\ejb\ folder):

Listing 13.1: Showing the Code of the `HelloRemote.java` File

```
package com.kogent.ejb;
public interface HelloRemote
{
   public String hello(String h);
}
```

The `HelloRemote.java` file is stored under the com.kogent.ejb package. Save HelloRemote.java under the src directory of the Hello-ejb directory.

NOTE

It is a good practice to write the programs inside the package while developing any application. It avoids the name conflicts among files.

Creating a Bean Class

After creating the business interface, the HelloBean class is required to implement the HelloRemote interface. The HelloBean class is a plain Java class, which implements business methods defined in business interface. This bean class is also known as the implementation class of the stateless session bean class as it provides implementation to all business methods. In our case, the bean class is `HelloBean`, which implements the `HelloRemote` business interface.

Listing 13.2 shows the code to create `HelloBean` class (you can find this file on the CD in the code\JavaEE\Chapter13\Hello\Hello-ejb\src\com\kogent\ejb\ folder):

Listing 13.2: Showing the Code of the `HelloBean.java` File

```
package com.kogent.ejb;
import javax.ejb.*;
@Stateless

public class HelloBean implements HelloRemote
{
   public String hello(String h)
   {
         return "Hello " + h + "!";
   }
}
```

Listing 13.2 shows that the `HelloBean` class is created in the `com.kogent.ejb` package and the @Stateless annotation is used to define the `HelloBean` class as a stateless session bean. Instead of using the @Stateless annotation, you can configure the bean in Deployment Descriptor (web.xml). In EJB 3, annotations are introduced to reduce the efforts required to create and configure different components of EJB.

Creating a Client

To access an enterprise bean, a client code uses JNDI, which is a directory service that provides a mechanism to find and lookup data and objects by using a naming service. When the client code is executed completely, connection of the naming directory with a bean's container is established. The code used to obtain the JNDI context depends upon the application server being used.

Listing 13.3 shows the code of the JSP client, hello.jsp, of the Hello application (you can find the hello.jsp file on the CD in the \code\JavaEE\Chapter13\Hello\Hello-war \ folder):

Listing 13.3: Showing the Code of the `hello.jsp` File

```
<%@ page import="com.kogent.ejb.*, javax.naming.*, java.text.*"%>
<%!
private HelloRemote hel = null;
public void jspInit ()
{
   try
   {
         InitialContext ctx = new InitialContext();
```

```
            hel = (HelloRemote) ctx.lookup("java:comp/env/ejb/HelloBean");
        }
        catch (Exception e)
        {
            e.printStackTrace ();
        }
    }
    %>
    <%
    String result = null;
    String name = null;
    try
    {
        name=request.getParameter("name");
        if(name!=null)
            result = hel.hello(name);
    }
    catch (Exception e)
    {
        result = "Not valid";
    }
    %>
    <html>
    <head>
        <title>Example of Stateless session bean</title>
        <link rel="stylesheet" href="mystyle.css" type="text/css"/>
    </head>
    <body>
        <h1>Using Stateless session bean</h1>
        <br><br>
        <form action="hello.jsp" method="POST">
            Enter Your Name : <input type="text" name="name">
            <br><br><input type="submit" value="Submit"><br><br>
        </form>
        <%
        if(result!=null)
        out.println("<b>"+result+"</b>");
        %>
    </body>
    </html>
```

Save the code shown in Listing 13.3 as `hello.jsp` in the Hello-war directory under the root directory, named Hello. After the connection is established and the context is obtained from the `InitialContext()` method, the context can be used to look up the EJB's business interface by using the `lookup()` method. The `hello` JSP page looks up for the business interface, `HelloRemote`, by using the `lookup()` method. The `hello.jsp` file shows a text field and a submit button. Now, when you enter a name in the text field and click the submit button, the `String hello(String)` business method returns a string value.

NOTE

JNDI offers:

- *A mechanism to bind an object to a name*
- *A directory lookup interface*
- *An event interface that define when to modify the directory entries*

Creating the Directory Structure and Configuring the Hello Application

To create the directory structure of the Hello application, you first need to create a folder named Hello, depicting the name of the application that you are creating. The Hello folder contains three subfolders, `Hello-war`, `Hello-ejb`, and `META-INF`. The `Hello-ejb` folder and the Hello-war folder are also known as the bean module and Web module, respectively. The `Hello-ejb` folder contains all the Java files, such as interfaces and

bean files that are inside your application. The `Hello-war` folder stores all the JSP files, Hypertext Markup Language (HTML) files, and WEB-INF folder. The META-INF folder contains the `MANIFEST.MF` file, by default.

The directory structure of the Hello application is shown in Figure 13.5:

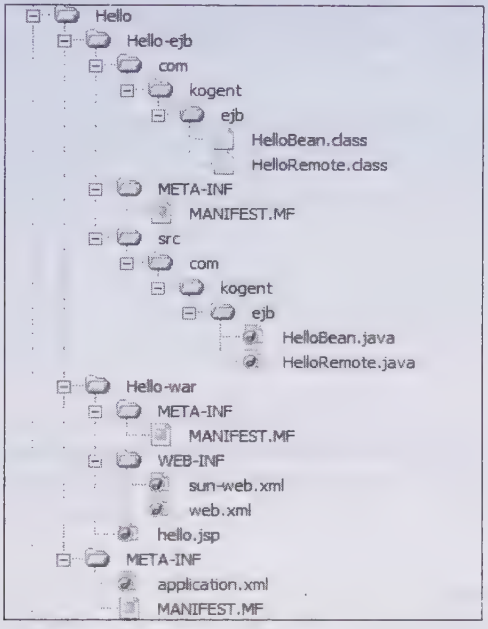


Figure 13.5: Showing the Directory Structure of the Hello Application

In Figure 13.5, the .class files of the bean and remote interface of the Hello application are placed within the Hello-ejb folder, as it is the EJB module of the application where the bean would be looked up.

The only additional file you have to create is `application.xml`. This file is necessary for the EJB 3 applications to provide the application details, such as the name of the EJB and Web modules. Listing 13.4 provides the code for the application.xml file (you can find this file on the CD in the code\JavaEE\Chapter13\Hello\META-INF\ folder):

Listing 13.4: Showing the Code for the `application.xml File`

```
    <?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>Hello</display-name>
      <module>
            <ejb>Hello-ejb.jar</ejb>
      </module>
      <module>
            <web>
            <web-uri> Hello-war.war</web-uri>
            <context-root>/Hello-war</context-root>
            </web>
      </module>
    </application>
```

The `application.xml` file configures two modules used in the `Hello` application. These modules are Hello-war and Hello-ejb.

The code for `web.xml` file of `WEB-INF` is given in Listing 13.5 (you can find this file on the CD in the code\JavaEE\Chapter13\Hello\Hello-war\WEB-INF folder):

Listing 13.5: Showing the `web.xml File`

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0"
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_3_0.xsd">

  <session-config>
        <session-timeout>30</session-timeout>
  </session-config>
  <welcome-file-list>
        <welcome-file>hello.jsp</welcome-file>
  </welcome-file-list>
  <ejb-ref>
        <ejb-ref-name>ejb/HelloBean</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <remote>com.kogent.ejb.HelloRemote</remote>
  </ejb-ref>
</web-app>
```

After creating the required files of the Hello application, you should ensure that all the files are stored at the right location according to the directory structure shown in Figure 13.5. In addition, compile all the Java source files and place them along with their package directory under the appropriate folder.

After developing all the components, let's discuss how to deploy this application on the Glassfish V3 application server.

Packaging the Application

As you know that before deploying an application, you first need to package it in an archive file. Therefore, in this section, you learn how to package the Hello application into the Hello.ear file. To create the Enterprise ARchive (EAR) file, you the first need to package the `Hello-ejb` module into **Hello-ejb.jar** and the `Hello-war` module into the **Hello-war.war** archive file. You can package the EJB module through Command Prompt by using the following command from the Hello-ejb folder:

```
jar -cvf Hello-ejb.jar *.*
```

Similarly, the Web module can be packaged by executing the following command from the Hello-war folder:

```
jar -cvf Hello-war.war *.*
```

You need to package the Hello-ejb.jar, Hello-war.war, and META-INF files in the EAR file of the Hello application. Now, let's create an `EAR` file through Command Prompt by executing the following command from the location of the Hello folder:

```
jar -cvf Hello.ear *.*
```

In the Hello application, the EAR file named Hello.ear needs to be deployed on the Glassfish V3 application server to run the Hello application.

Deploying the Application

After packaging the Hello application, let's deploy it on the Glassfish V3 application server. Prior to the deployment of the Hello application, ensure that the server is running. Perform the following steps to deploy the Hello application:

1. Browse the `http://localhost:4848/` URL to open the Admin console. The index page appears in which you need to enter the username and password to login to the Admin console.
2. Enter username and password to login to the Admin console. In our case, we have entered admin as username and adminadmin a password.
3. Expand the Applications node from the left panel.

   Figure 13.6 shows the Deployment panel for enterprise applications:

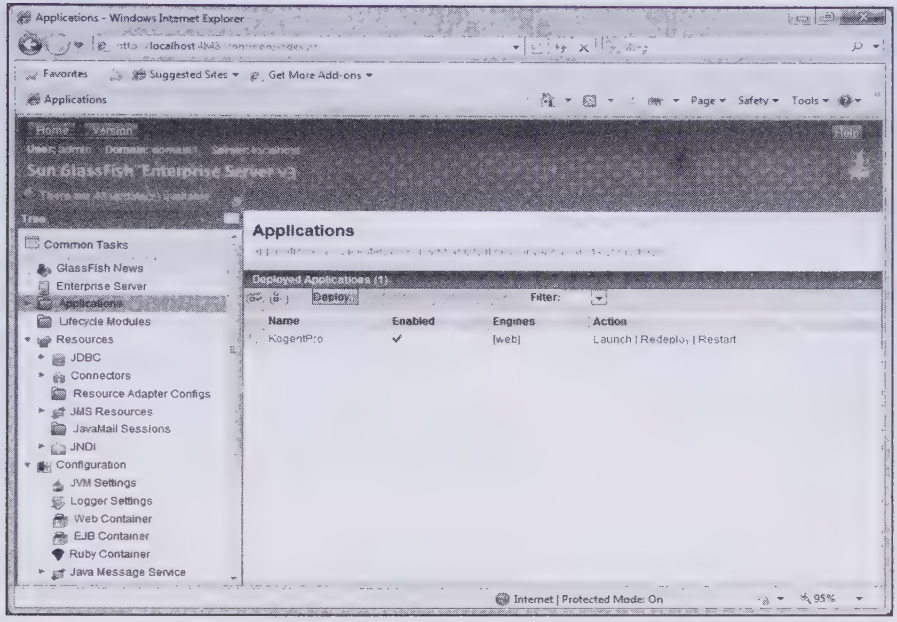


Figure 13.6: Deploying the Enterprise Applications

4. Click the Deploy button to deploy the enterprise application. The Deploy Applications or Modules pane appears (Figure 13.7).
5. Click the Browse button to browse the Hello.ear file saved on your computer. After browsing the file, the information required for the Hello application gets automatically filled in the relevant fields, as shown in Figure 13.7:

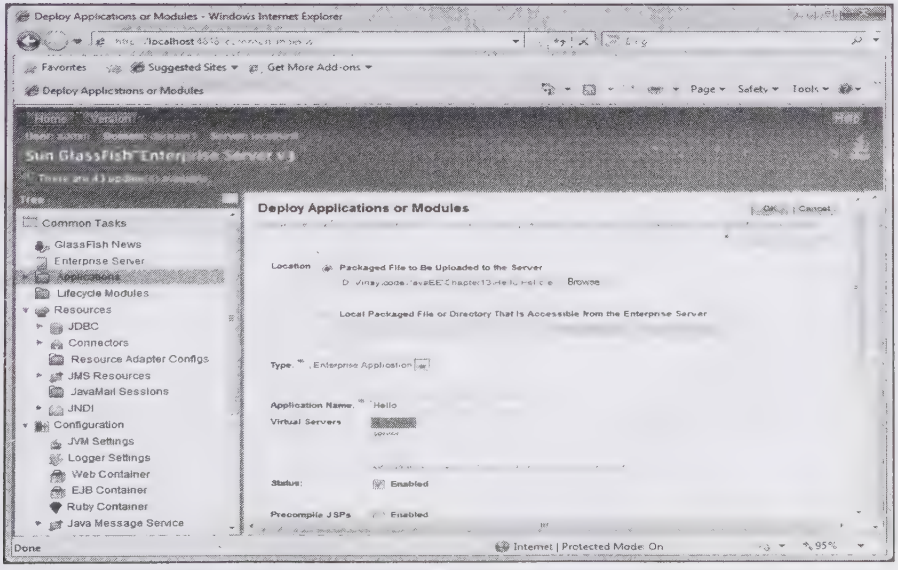


Figure 13.7: Uploading the Packaged Archive File

6. Click the OK button to deploy the Hello application. Figure 13.8 displays the list of the deployed applications, including the Hello application:

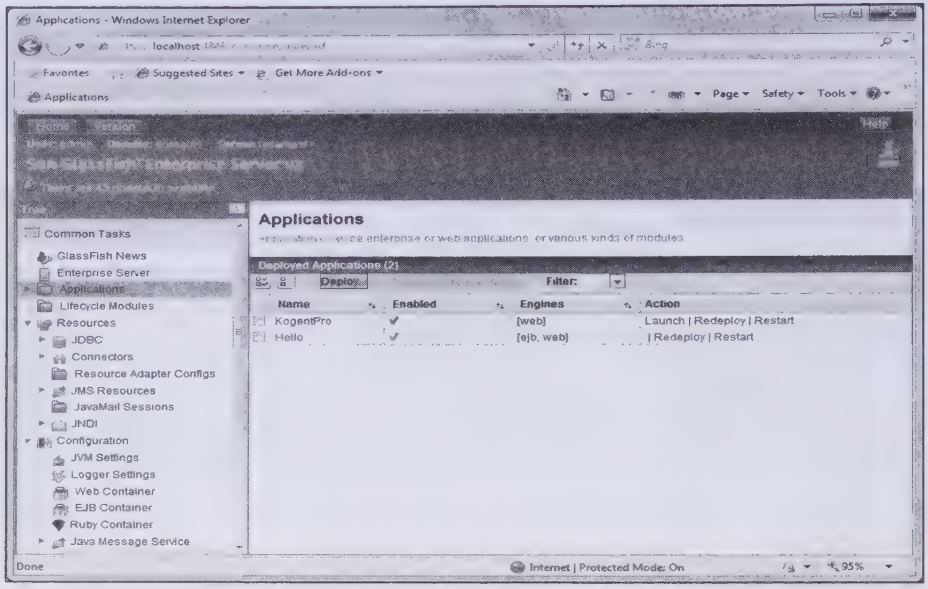


Figure 13.8: Displaying the Deployed Applications

In the Hello application, the Enabled column shows the true value, implying that you can execute the application (Figure 13.8). However, if this column shows false, then select the checkbox beside the Hello application, which activates the Enable button. Finally, click the Enable button to enable the application for execution.

Now, let's run the application to see its output.

Running the Application

After deploying the application, you can run it by browsing the `http://localhost:8080/Hello-war/hello.jsp` URL. The hello.jsp page appears, as shown in Figure 13.9:

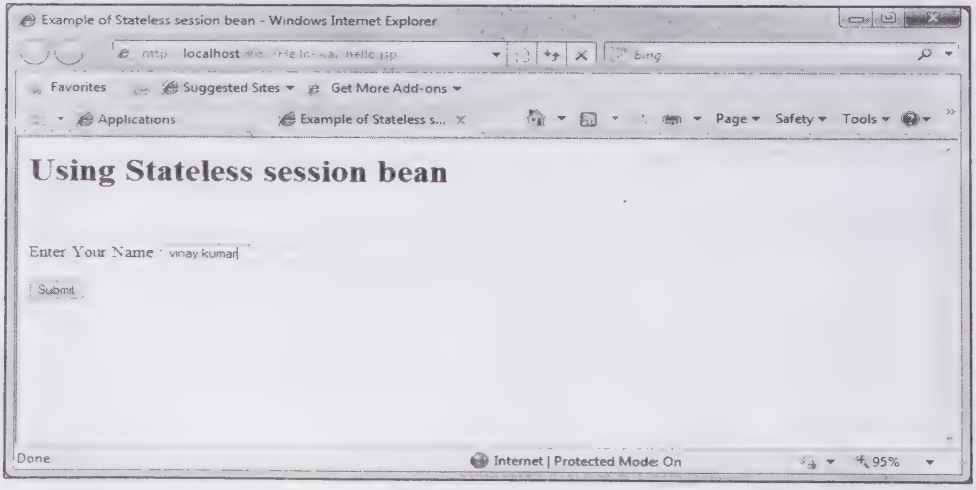


Figure 13.9: Displaying the hello.jsp Page

Enter your name in the textbox shown on the hello.jsp page and click the Submit button (Figure 13.9).

The name entered by you in the text box is displayed, as shown in Figure 13.10:

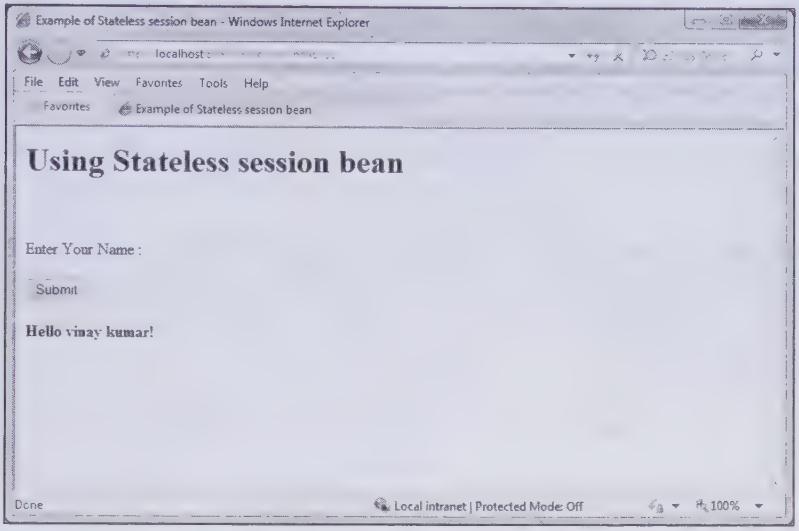


Figure 13.10: Displaying the Output of Hello Application

After developing the Hello application to understand the working of a stateless session bean, let's learn how to work with a stateful session bean.

Working with a Stateful Session Bean

Let's create an application named Cart, which uses a stateful session bean to enable users to insert items into the cart. Perform the following steps to create the Cart application:

- Create a business interface
- Create a bean class
- Create a JSP client
- Package, deploy, and run the application

Let's discuss these steps one by one.

Creating a Business Interface

In EJB 3, you can create a business interface as a remote interface by using the `@Remote` annotation. In the Cart application, we create the CartRemote interface, which declares three business methods—`addItem()`, `removeItem()`, and `getItems()`.

Listing 13.6 provides the code for the CartRemote.java file (you can find this file on the CD in the code\JavaEE\Chapter13\Cart\Cart-ejb\src\com\kogent\ejb\ folder):

Listing 13.6: Showing the Code for the `CartRemote.java` File

```
package com.kogent.ejb;
import java.util.Collection;
import javax.ejb.Remote;
@Remote
public interface CartRemote
{
   public void addItem(String item);
   public void removeItem(String item);
   public Collection getItems();
}
```

Save the code of Listing 13.6 as `CartRemote.java` inside the src\com\kogent\ejb directory of the EJB module. Compile the `CartRemote.java` file, the com.kogent.ejb package structure is created containing the `CartRemote` class. The directory structure of the Cart application is similar to the Hello application created in the *Developing a Stateless Session Bean* section.

Creating a Bean Class

A bean class implements the business interface, `CartRemote`, to provide implementations to all the business methods of the business interface. Similar to the stateless session bean, you can use the @Stateful annotation to create a stateful session bean, `CartBean`.

Listing 13.7 provides the code of the `CartBean` stateful session bean (you can find the CartBean.java file on the CD in the code\JavaEE\Chapter13\Cart\Cart-ejb\src\com\kogent\ejb\ folder):

Listing 13.7: Showing the Code for the `CartBean.java File`

```
package com.kogent.ejb;
import java.util.ArrayList;
import java.util.Collection;
import javax.annotation.PostConstruct;
import javax.ejb.Stateless;
@Stateless
public class CartBean implements CartRemote
{
   private ArrayList items;
   @PostConstruct
   public void initialize()
   {
         items = new ArrayList();
   }
   @SuppressWarnings("unchecked")
   public void addItem(String item)
   {
         items.add(item);
   }
   public void removeItem(String item)
   {
         items.remove(item);
   }
   public Collection getItems()
   {
         return items;
   }
}
```

In Listing 13.7, the `initialize()` method that has been annotated with the `@PostConstruct` annotation is invoked just after the bean instance is created to initialize the `items` ArrayList.

Creating a Client

To access a stateful session bean and invoke the `addItem()` or `getItems()` method over it, you need to create a JSP client. Let's create the index.jsp page as a client that invokes the business methods of the CartBean class. Listing 13.8 provides the code for the `index.jsp file` (you can find this file on the CD in the code\JavaEE\Chapter13\Cart\Cart-war\ folder):

Listing 13.8: Showing the Code for the `index.jsp File`

```
<%@ page import="com.kogent.ejb.*, javax.ejb.*,javax.naming.*
,java.util.logging.*,java.util.*" %>
<%!
private CartRemote cart ;
public void jspInit() {
   try {
         Context c = new InitialContext();
         cart=(CartRemote) c.lookup("java:comp/env/ejb/CartBean");
   }
   catch(NamingException ne) {
         Logger.getLogger(getClass().getName()).log(Level.SEVERE,"exception
         caught" ,ne);
         throw new RuntimeException(ne);
   }
}
public void jspDestroy() {
```

```
        cart = null;
    }
    %>
    <html>
    <head>
      <title>A Stateful session bean Implementation.</title>
      <link rel="stylesheet" href="mystyle.css" type="text/css"/>
    </head>
    <body bgcolor="white">
      <h1>Shopping Cart</h1>
      <hr>
      <h3>Adding Items to Shopping Cart</h3>
      <form method="get">
            Enter Item Name : <input type="text" name="item" size="25">
            <br>
            <input type="submit" value="Add Item">
      </form>
    <%
      String item = request.getParameter("item");
      if(item!=null)
      cart.addItem(item);
      Collection items = cart.getItems();
    %>
    <p>
    <%= items %>
    </body>
    </html>
```

This JSP file works as a client code for the Cart application. Save the `index.jsp` file inside the Web module of the Cart folder. After creating the client, you need to configure various details, such as context root, ejb-ref of the application in the application.xml and web.xml files, respectively.

The `application.xml` file for the `Cart` application is similar to the file given in Listing 13.4 (you can find this file on the CD in the `code\JavaEE\Chapter13\Cart\META-INF\`folder). The only difference is that the context-root name is Cart-war, as shown in the following code snippet:

```
<display-name>Cart</display-name>
  <module>
        <ejb>Cart-ejb.jar</ejb>
  </module>
  <module>
        <web>
        <web-uri>Cart-war.war</web-uri>
        <context-root>/Cart-war</context-root>
        </web>
  </module>
```

Listing 13.9 provides the code for the web.xml file of the Cart application (you can find this file on the CD in the code\JavaEE\Chapter13\Cart\Cart-war\WEB-INF\ folder):

Listing 13.9: Showing the Code for the `web.xml` File

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

<web-app version="3.0" 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_3_0.xsd">

  <session-config>
        <session-timeout>
            30
        </session-timeout>
  </session-config>
  <welcome-file-list>
  <welcome-file>
        index.jsp
```

```
    </welcome-file>
    </welcome-file-list>
    <ejb-ref>

        <ejb-ref-name>ejb/CartBean</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <remote>com.kogent.ejb.CartRemote</remote>

    </ejb-ref>

</web-app>
```

Let's now learn to package, deploy, and run the Cart application.

Packaging, Deploying, and Running the Application

To package, deploy, and run the Cart application, you first need to ensure that the Web (Cart-war) and EJB (Cart-ejb) modules of the `Cart` application are arranged according to the directory structure of the `Hello` application (Figure 13.5).

To create the `Cart.ear` file, you first need to create the `Cart-war.war` and `Cart-ejb.jar` files. Let's create an EAR file named `Cart.ear` by using the `jar -cvf Cart.ear *.*` command, as described in the Hello application. Execute this command through Command Prompt from the location at which the `Cart` folder is stored. Now, deploy your corresponding EAR file that is `Cart.ear`. The Glassfish server is used for deploying this application. After successfully deploying the `Cart.ear` file, open the browser with the corresponding address bar and browse the `http://localhost:8080/Cart-war` URL.

Figure 13.11 shows the output of the Cart application:

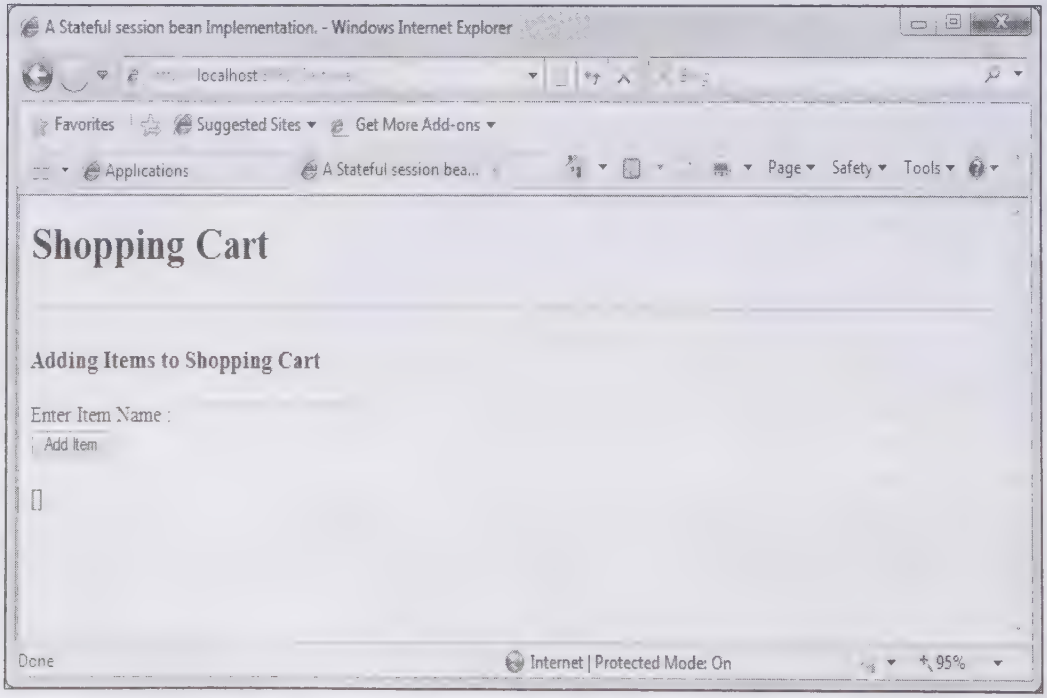


Figure 13.11: Displaying the index JSP Page of the Cart Application

Figure 13.11 displays the client, index.jsp. You can enter an item of your choice in the `Enter Item Name` text box to add it to your shopping cart. Enter the item name in the text field and click the **Add Item** button.

Figure 13.12 shows the list of items added in the shopping cart:

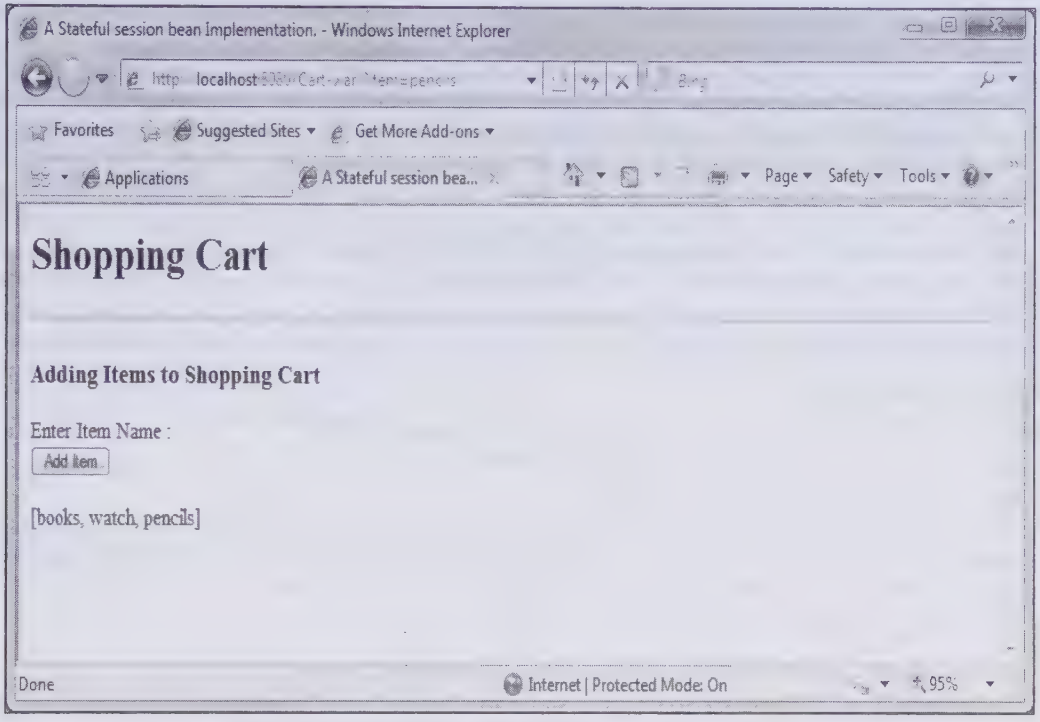


Figure 13.12: Displaying the Cart after Adding the Items

In the `Cart` application, the previous state of application is stored. In our case, you can see that the items, such as pencils and watch, added earlier by us, are displayed on the browser even after the new item books is added. This application helps to maintain the client's state.

Introducing the MDB

The term MDB itself suggests that it is associated with some sort of messaging. To communicate among software components or applications, messaging is used. Messaging is a facility by using which a client can send/receive messages to/from other clients. In case of MDBs, each client connects to a message agent that facilitates creating, sending, receiving, and reading messages.

Java Messaging Service (JMS) is a core service provided by Java EE application servers. JMS allows asynchronous invocation of different services via messages. JMS clients send messages to the server maintained message queues. To monitor message queues, a special kind of EJB is needed, called MDB.

Characteristics of the MDB

Some of the characteristics of the MDB are as follows:

- MDB does not have a remote or local business interface.
- You cannot call an MDB by using an object-oriented remote method invocation interface. The MDB processes the messages coming from any messaging client.
- MDBs support generic listener methods for message delivery.
- MDB's listener methods do not have any return values. The EJB specification does not have any restriction over MDB listener method to return a value to the client; however, certain type of messaging is not suitable for this purpose. For example, let's consider a listener interface of a messaging type, which supports asynchronous messaging, such as JMS. In this case, the message producers do not wait for the MDB to respond as the interaction between the message producers and consumers is asynchronous.

- The exceptions might not be sent back to the clients by the MDBs. EJB does not restrict the MDB listener interface methods from throwing application exceptions; however, some of the messaging types in MDB listener interface might not be able to throw these exceptions to clients. In case the listener interface of messaging type does not support asynchronous messaging, such as JMS, the message sender would not wait for the MDB to send the response and directly sends the message. This is because of the asynchronous nature of the interaction for which the client cannot receive any exceptions.
- MDBs are stateless in nature. Similar to the stateless session bean, MDBs do not hold any conversational state for a specific client. The MDBs do not have any client-visible identity. In such a case, the container can treat each message-driven instance as equivalent to other instances. Therefore, multiple instances of the bean can process multiple messages from a JMS destination. This is the reason why it is stateless in nature.
- MDBs are thread safe. A single MDB can process only one message at a time. The container is responsible for serializing messages to a single MDB, so there is no need for synchronizing code in the bean class.

Structure of the MDB

The structure of MDB can be summarized through the following statements:

- Home and remote interfaces are not required for MDB
- Bean class is the main focusing part of MDB

MDB is very much similar to the stateless session bean. The interaction between MDB and client is similar to the interaction of MDB with the JMS application or JMS server. The instances of a particular MDB are same because they are not visible to clients directly and do not maintain a conversational state.

Life Cycle of the MDB

Life cycle of an MDB is exhibited through the following two states:

- Does Not Exist state
- Method-Ready Pool state

The state of MDB depends upon the container as the container decides when to insert a new instance to its pool. The container creates a new instance by calling the `setMessageDrivenContext()` and `ejbCreate()` methods.

The setMessageDrivenContext() method provides access to the runtime message-driven context that the container provides for a message-driven enterprise bean instance. It is called by the container after creating the bean instance and it passes a reference of the MessageDrivenContext object to the bean. This means that the container passes the MessageDrivenContext reference to an instance after the instance has been created. An MDB can also acquire a reference to the MessageDrivenContext object by using DI by simply using the @Resource annotation. The message-driven context remains associated with the corresponding instance as long as the instance is alive, that is throughout its lifetime. After creating the instance and getting the MessageDrivenContext reference, the MDB is ready to receive messages. If any annotation, such as `@PostConstruct`, has been used to define a life cycle callback method, then the relevant method is invoked. According to the configuration specified in Deployment Descriptor, the application server generates an initial pool of beans at the startup time. The size of this pool can be expanded as the size of messages increases. When the bean instance is processing a JMS message, the `onMessage()` method is invoked on the bean instance to perform a task. The container can destroy an instance by removing it from the pool while shutting down the system. This can also be done to decrease the size of the pool in the container to conserve memory. To decrease the size of the pool, it needs to call the `ejbRemove()` method, before the instance is ready for the garbage collection.

To take an instance out of the bean pool, the container calls the `@PreDestroy()` method. The diagrammatical representation of the life cycle of the MDB is shown in Figure 13.13:

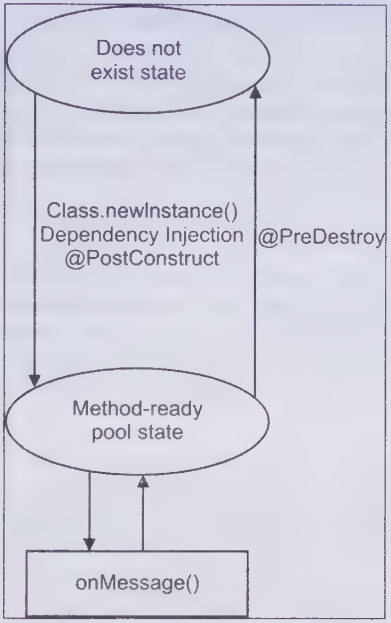


Figure 13.13: Displaying the Life cycle of an MDB

You can observe that the life cycle of the stateless session bean described in the previous section is quite similar to that of the MDB. Let's discuss the two states of the MDB:

- **Does Not Exist state**—Refers to the stage in which an MDB instance is not considered as an instance in the system memory. It means that it has not been instantiated yet.
- **Method-Ready Pool state**—Refers to the stage in which an MDB instance is needed by the container to handle incoming messages. As soon as the EJB server starts, it may create a number of instances and move the instances into the method-ready pool. When the MDB instances are not enough to handle the incoming messages, then more instances of MDB are created and inserted into the pool.

Let's now discuss the transition from one state to another.

Transitioning to the Method-Ready Pool

When an instance moves from the Does Not Exist state to the method-ready pool state, the following three tasks are performed by the container:

- Instantiates a bean instance, when the newInstance() method is invoked on an MDB class
- Injects the required resource with the help of annotation or XML Deployment Descriptor
- Invokes the method annotated by `the @PostConstruct` annotation

The @javax.ejb.PostConstruct annotation may or may not be used with any of the methods of the bean class. However, in case it is present, this annotated method will be called by the container only after the bean is instantiated. The return type of the `@PostConstruct` annotated method should always be void and the method should not have any parameter.

The following code snippet shows how to use the `@PostConstruct` annotation:

```
@MessageDriven
public class MyBean implements MessageListener
{
  @PostConstruct
  public void myInit( )
  { }
}
```

Exploring the Life Time of an MDB in the Method-Ready Pool

As soon as an instance reaches to the method-ready pool state, it is ready to handle incoming messages. While a message is being forwarded to MDB, it is delegated to any available instance in the method-ready pool state. When an instance in MDB is executing a request, it is unable to execute or process other messages. In such scenarios, the MDB delegates the responsibility to different MDB instances to handle messages. When an instance has finished the processing of the message, then immediately this instance becomes available to handle another new message in the MDB.

Destroying the MDB Instance

When the server does not need a bean instance, its state is changed from the Method-Ready Pool state to the Does Not Exist state. This type of situation generally occurs when the server decides to decrease the total size of the method-ready pool by releasing one or more instances from memory. At this time, the bean instance can perform any cleanup operation, such as closing open resources by using the `@PreDestroy` annotation. It is important to note that the callback method annotated by the `@PreDestroy` annotation can only be invoked once during the life cycle of an MDB instance.

The following code snippet shows how to call the `cleanup()` callback method annotated with @PreDestroy:

```
@MessageDriven
public class MyBean implements MessageListener
{
  @PreDestroy
  public void cleanup( )
  {
        ...
  }
```

Implementing the MDB

Prior writing an MDB, a developer should take the following tasks into consideration:

- Implementing the `javax.ejb.MessageDrivenBean` and `javax.jms.MessageListener` interfaces in the MDB class.
- Implementing the business logic in the `onMessage()` method.

Now, let's discuss them one by one.

Implementing the MessageDrivenBean and MessageListener Interfaces

JMS MDBs are the classes that implement `javax.jms.MessageListener` and `javax.ejb.MessageDrivenBean` interfaces to define a Java class as an MDB. The `javax.ejb.MessageDrivenBean` interface is optional and must provide a constructor with no argument. In case of the earlier versions of the EJB, it was compulsory to use the MessageDrivenBean interface; however, this is not required in the new version of the EJB, EJB 3.

Implementing Business Logic inside the onMessage() Method

The `onMessage()` method is the only method in the JMS message listener interface `javax.jms.MessageListener`. This accepts JMS messages that can represent a byte message, object message, text message, and map message. The main goal of the `onMessage()` method is to type-cast the incoming message of any type to a text message and to display the text. It acts like a business interface as all the business logic is exhibited inside the `onMessage()` method of the MDB.

Let's now create a sample MDB application and then package, deploy, and run the application.

Creating a Sample MDB Application

Let's create an application, MDB, which displays a message while the investment is calculated by using the investment calculator. While the server is calculating your investment, it will give a message, Please wait while I

am checking whether the message has arrived, to let you know that you have to wait for the required result. To create the MDB application, you need to perform the following steps:

- Create the CalculationRecord class
- Create the RecordManager class
- Create the CalculatorBean class
- Create the calculator.jsp file
- Create the check.jsp file

Let's perform these steps, one by one.

Creating the CalculationRecord Class

Let's first create the CalculationRecord class, which has the CalculationRecord (Timestamp, Timestamp, double) constructor, setting three different fields—sent, processed, and result. Listing 13.10 shows the code of the CalculationRecord class (you can find this file on the CD in the code\JavaEE\Chapter13\MDB\MDB-ejb\src\jms\ folder):

Listing 13.10: Showing the CalculationRecord.java File

```
package jms;
import java.sql.Timestamp;
public class CalculationRecord
{
  public CalculationRecord(Timestamp sent,
  Timestamp processed, double result)
  {
        this.sent = sent;
        this.processed = processed;
        this.result = result;
  }
  public Timestamp sent;
  public Timestamp processed;
  public double result;
}
```

In Listing 13.10, the constructor of the CalculationRecord class is defined.

Creating the RecordManager Class

The RecordManager class provides different methods, such as `addRecord()`, and `getRecord()` to manipulate records. Listing 13.11 shows the code for the RecordManager class (you can find the RecordManager.java file on the CD in the code\JavaEE\Chapter13\MDB\MDB-ejb\src\jms\ folder):

Listing 13.11: Showing the Code for the RecordManager.java File

```
package jms;
import java.sql.Timestamp;
import java.util.ArrayList;
public class RecordManager
{
  public RecordManager()
  {
  }
  public static void addRecord(Timestamp sent, double result)
  {
  if(crs.size() > maxSize)
        crs.remove(0);
        Timestamp processed = new Timestamp(System.currentTimeMillis());
        crs.add(new CalculationRecord(sent, processed, result));
  }
  public static CalculationRecord getRecord(long sent)
  {
        for(int i = 0; i < crs.size(); i++)
        {
              CalculationRecord cr = crs.get(i);
              if(cr.sent.equals(new Timestamp(sent)))
```

```
                return cr;
            }
            return null;
        }
        private static ArrayList<CalculationRecord> crs =
        new ArrayList<CalculationRecord>();
        private static int maxSize = 100;
    }
```

In Listing 13.11, the addRecord() method is defined to add the time records in the crs ArrayList of the CalculationRecord type. In addition, the getRecord() method is defined to retrieve all the records from the ArrayList.

Creating the CalculatorBean Class

The CalculatorBean class is the MDB class. This class has been annotated with the @MessageDriven annotation to ensure that the container considers it as an MDB. Listing 13.12 provides the code of the CalculatorBean class (you can find the CalculatorBean.java file on the CD in the code\JavaEE\Chapter13\MDB\MDB-ejb\src\jms\ folder):

Listing 13.12: Showing the Code for the CalculatorBean.java File

```
package jms;
import java.sql.Timestamp;
import java.util.StringTokenizer;
import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;
@MessageDriven(
   mappedName = "jms/Queue", activationConfig = {
   @ActivationConfigProperty(propertyName = "acknowledgeMode",
   propertyValue = "Auto-acknowledge"),
   @ActivationConfigProperty(propertyName = "destinationType",
   propertyValue = "javax.jms.Queue") })
public class CalculatorBean implements MessageListener {
   public void onMessage (Message msg)
   {
         try
         {
                TextMessage tmsg = (TextMessage) msg;
                Timestamp sent =
                new Timestamp(tmsg.getLongProperty("sent"));
                StringTokenizer st = new StringTokenizer(tmsg.getText(),",");
                int start = Integer.parseInt(st.nextToken());
                int end = Integer.parseInt(st.nextToken());
                double growthrate = Double.parseDouble(st.nextToken());
                double saving = Double.parseDouble(st.nextToken());
                // Pause to simulate a long running task
                Thread.sleep(1000);
                double result = calculate (start, end, growthrate, saving);
                RecordManager.addRecord (sent, result);
                System.out.println ("The onMessage() is called");
         }
         catch (Exception e)
         {
                e.printStackTrace ();
         }
   }
   private double calculate (int start, int end, double growthrate, double
   saving)
   {
   double tmp = Math.pow(1. + growthrate / 12., 12. * (end - start) +    1);
   return saving * 12. * (tmp - 1) / growthrate;
   }
}
```

In Listing 13.12, we have created a bean class which implements the `javax.jms.MessageListener` interface and is annotated using the `@MessageDriven` annotation that specify EJB 3 characteristics.

Compile the `CalculationRecord.java`, `RecordManager.java` and `CalculatorBean.java` files and the class files are generated in a package named jms, which is present in the MDB-ejb module. The directory structure of the files of this MDB application is similar to that of the session bean.

Creating the calculator.jsp File

Let's create the calculator.jsp file that looks up the required connection with the jms/Queue Queue by using the lookup methods. Listing 13.13 shows the code for the calculator.jsp file (you can find this file on the CD in the code\JavaEE\Chapter13\MDB\MDB-war\ folder):

Listing 13.13: Showing the Code for the calculator.jsp File

```
<%@ page import="jms.*,
    javax.naming.*,javax.jms.Queue,javax.jms.*,java.text.*,java.sql.Timestamp"%>

<%
  if ("send".equals(request.getParameter ("action"))) {

    QueueConnection cnn = null;
    QueueSender sender = null;
    QueueSession sess = null;
    Queue queue = null;

    try {
      InitialContext ctx = new InitialContext();
      queue = (Queue) ctx.lookup("jms/Queue");
      QueueConnectionFactory factory =
          (QueueConnectionFactory) ctx.lookup("jms/ConnectionFactory");
      cnn = factory.createQueueConnection();
      sess = cnn.createQueueSession(false,
                  QueueSession.AUTO_ACKNOWLEDGE);

    } catch (Exception e) {
      e.printStackTrace ();
    }

    TextMessage msg = sess.createTextMessage(
        request.getParameter ("start") + "," +
        request.getParameter ("end") + "," +
        request.getParameter ("growthrate") + "," +
        request.getParameter ("saving")
    );
    // The sent timestamp acts as the message's ID
    long sent = System.currentTimeMillis();
    msg.setLongProperty("sent", sent);

    sender = sess.createSender(queue);
    sender.send(msg);
    // sess.commit ();
    sess.close ();
%>

<html>
<head><meta http-equiv="REFRESH" content="3;URL=check.jsp?sent=<%=sent%>">
<link rel="stylesheet" href="mystyle.css" type="text/css"/>
</head>
  <body>
    Please wait while I am checking whether the message has arrived.<br/>
    <a href="calculator.jsp">Go back to Calculator</a>
  </body>
</html>

<%
```

```
        return;

      } else {

        int start = 25;
        int end = 65;
        double growthrate = 0.08;
        double saving = 300.0;
    %>

    <html>
    <body>
    <p>Investment calculator<br/>
    <form action="calculator.jsp" method="POST">
      <input type="hidden" name="action" value="send">
      Start age = <input type="text" name="start" value="<%=start%>"><br/>
      End age   = <input type="text" name="end" value="<%=end%>"><br/>
      Annual Growth Rate = <input type="text" name="growthrate" value="<%=growthrate%>"><br/>
      Monthly Saving = <input type="text" name="saving" value="<%=saving%>"><br/>
      <input type="submit" value="Calculate">
      <INPUT type="button" value="Close Window" onClick="window.close()">
    </form>
    </p>
    </body>
    </html>

    <%
        return;
      }
    %>
```

After creating the calculator.jsp file, let's create the check.jsp file.

Creating the check.jsp File

The code for the `check.jsp` file is given in Listing 13.14 (you can find this file on the CD in the code\JavaEE\Chapter13\MDB\MDB-war\ folder):

Listing 13.14: Showing the Code for the check.jsp File

```
<%@ page import="jms.*, java.text.NumberFormat"%>
<%
long sent = Long.parseLong(request.getParameter ("sent"));
CalculationRecord rc = RecordManager.getRecord(sent);
if (rc == null)
{
  %>
  <html>
        <head><meta http-equiv="REFRESH" content="3;
                URL=check.jsp?sent=<%=sent%>"></head>
        <body>
                Please wait while I am checking whether the message has
                arrived.<br/>
                <a href="calculator.jsp">Go back to Calculator</a>
        </body>
  </html>
  <% return;
}
else
{
  NumberFormat nf = NumberFormat.getInstance();
  nf.setMaximumFractionDigits(2); %>
  <html>
  <body>
        The message was sent at<br/>
        <b><%=rc.sent%></b>.<br/><br/>
        The message was processed at<br/>
        <b><%=rc.processed%></b>.<br/><br/>
```

```
        The calculation result (total investment) is
        <b><%=nf.format(rc.result)%></b>.<br/>
        <a href="calculator.jsp">Go back to Calculator</a>
    </body>
    </html>
<% return; } %>
```

After creating the required files, you need to configure the calculator.jsp file as the welcome page to be displayed at the starting of the MDB application. You can configure this file in the web.xml file.

The code for the `web.xml` file is given in Listing 13.15 (you can find this file on the CD in the code\JavaEE\Chapter13\MDB\MDB-war\WEB-INF\ folder):

Listing 13.15: Showing the web.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" 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_3_0.xsd">
  <session-config>
        <session-timeout>
        30
        </session-timeout>
  </session-config>
  <welcome-file-list>
        <welcome-file>calculator.jsp</welcome-file>
  </welcome-file-list>
  </web-app>
```

You should save the `calculator.jsp and check.jsp` files in the Web module (MDB-war) of the MDB application. The Web module created for the MDB application is similar to the Hello application.

Now, you might be thinking why MDB is implemented when the same result could be achieved by using a session bean. This is because MDB allows you to implement the concept of messaging, which is essential for performing different business tasks, such as messaging and calculating. These business tasks are complex and require more time to complete, because they require implementation of entity bean. Now, if you are using the session bean to perform a task, this task would have to wait until the session bean is completed and returns the control to the application. However, in case of MDB, a waiting message is displayed to users while the request sent by the user is processed.

Packaging, Deploying, and Running the Application

To deploy the MDB application on the application server, the Glassfish server, you need to perform the following tasks:

- Create an EAR file
- Configure JMS resources
- Configure JMS connection factories
- Configure destination resources
- Run the application

Now, let's perform these steps, one by one.

Creating the EAR file

After completing the coding part, you should package the MDB application by creating an EAR file that would be deployed on the Glassfish V3 application server. You can create the EAR file through Command Prompt by using the `jar -cvf MDB.ear *.*` command after changing your current directory to MDB. The execution of this command adds the beans module, Web module, and META-INF in the MDB.ear file.

Configuring JMS Resources

After packaging the MDB application, start the Glassfish V3 application server and then open the `http://localhost:4848/` URL. Next, login to the Admin console by entering a username and password. In our case, we have entered admin as the user name and adminadmin as the password. The Admin console

appears. Now, expand the `Resources` node in the Admin console and select the JMS Resources option. The right panel of Admin console displays two options: Connection Factories and Destination Resources, as shown in Figure 13.14:

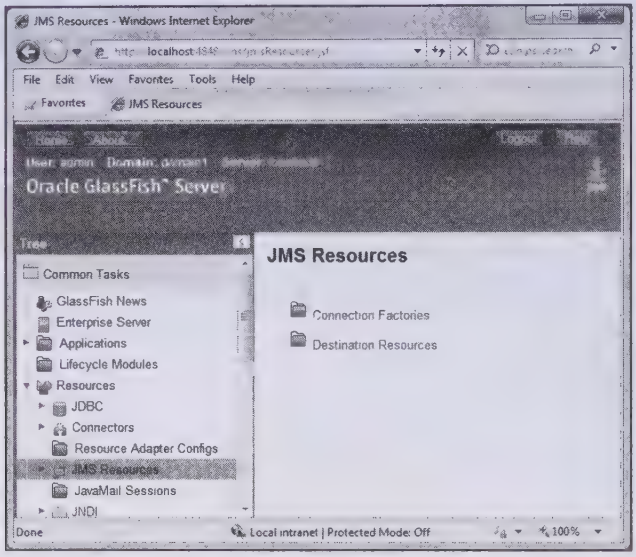


Figure 13.14: Displaying the JMS Resources

Now, select the Connection Factories option to create a JMS connection factory. The following section describes how to create the JMS connection factory for the MDB application created earlier.

Configuring JMS Connection Factories

When you select the Connection Factories option in the JMS Resource pane (Figure 13.14), it displays the existing JMS Connection Factories, as shown in Figure 13.15:

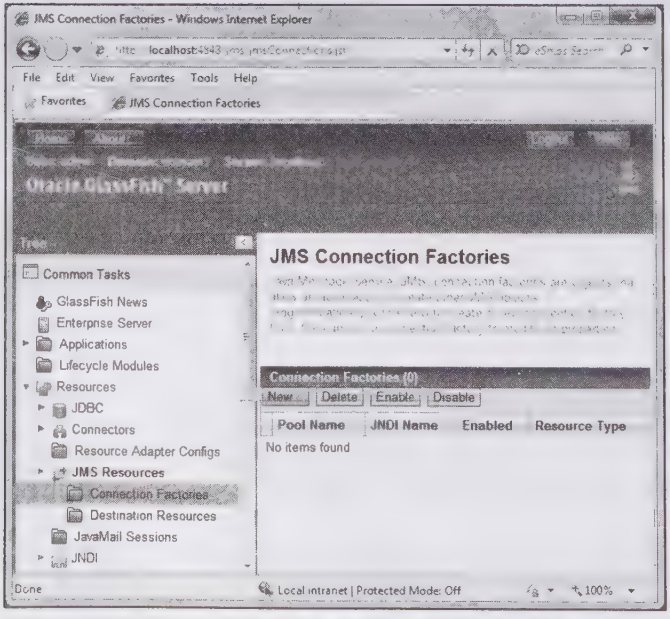


Figure 13.15: Displaying the Existing JMS Connection Factories

To create a new JMS connection factory, click the New button and configure the new JMS connection factory, as shown in Figure 13.16:

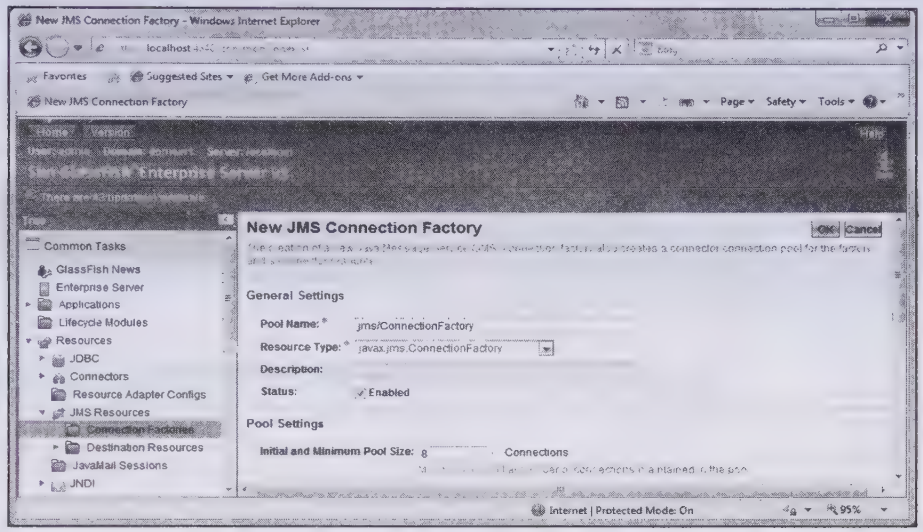


Figure 13.16: Configuring a New JMS Connection Factory

In Figure 13.16, you can see that pool name (JNDI name) for the new JMS connection factory is jms/ConnectionFactory and the Resource Type is javax.jms.ConnectionFactory. After configuring the new JMS connection factory, click the OK button. The jms/ConnectionFactory pool is added to the existing JMS connection factories list and is displayed under the JNDI Name column, as shown in Figure 13.17:

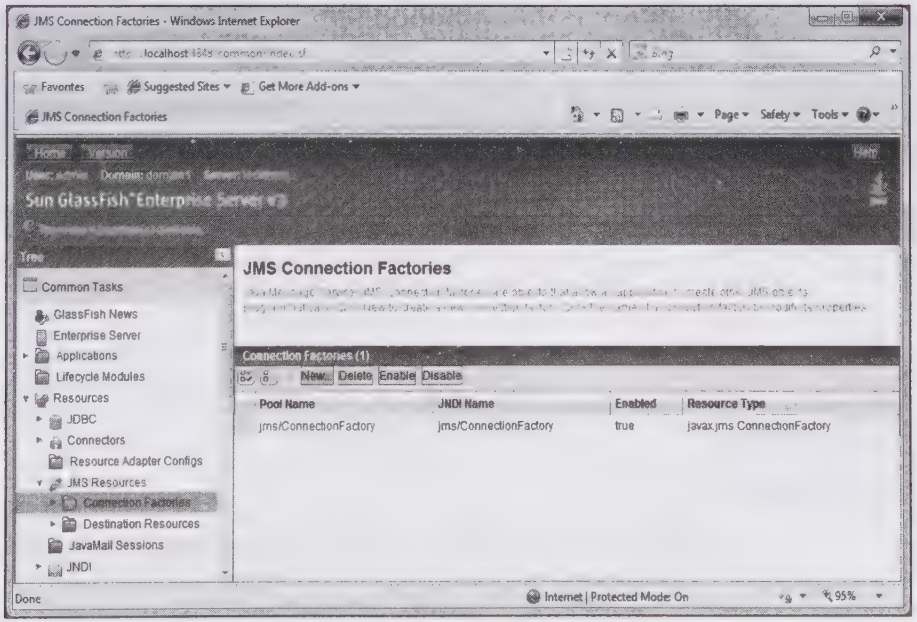


Figure 13.17: Displaying the JMS Connection Factories

After creating a new JMS connection factory for the MDB application, you need to configure new destination resources. The following section describes how to configure new JMS destination resources for the MDB application.

Configuring Destination Resources

JMS destination resources serve as a repository for JMS messages. To create a new JMS destination resource, expand the JMS Resources node and select the Destination Resources option. A list of the existing JMS Destination Resources appears, as shown in Figure 13.18:

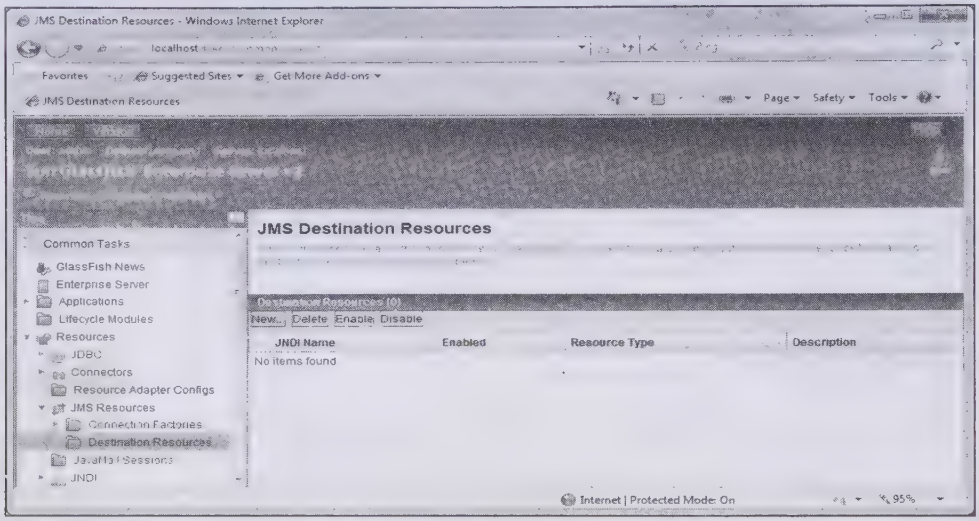


Figure 13.18: Displaying the Existing Destination Resources

Now, click the New button to open the New JMS Destination Resource pane. In the New JMS Destination Resource pane, you need to provide all configuration settings, as shown in Figure 13.19:

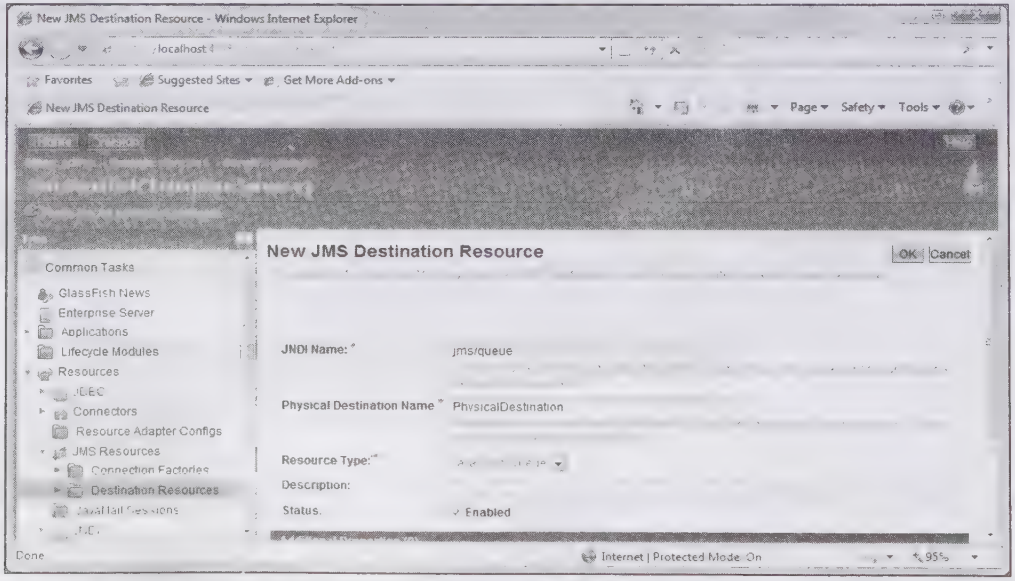


Figure 13.19: Displaying a New JMS Destination Resource

In Figure 13.19, you can see that the JNDI name for the new JMS destination resource for the MDB application is jms/queue. In our case, we enter `PhysicalDestination` in the `Physical Destination Name` textbox and `javax.jms.Queue` in the `Resource Type` textbox. Now, click the `OK` button to add the new JMS destination resource, jms/queue, in the existing JMS destination resource list, as shown in Figure 13.20:

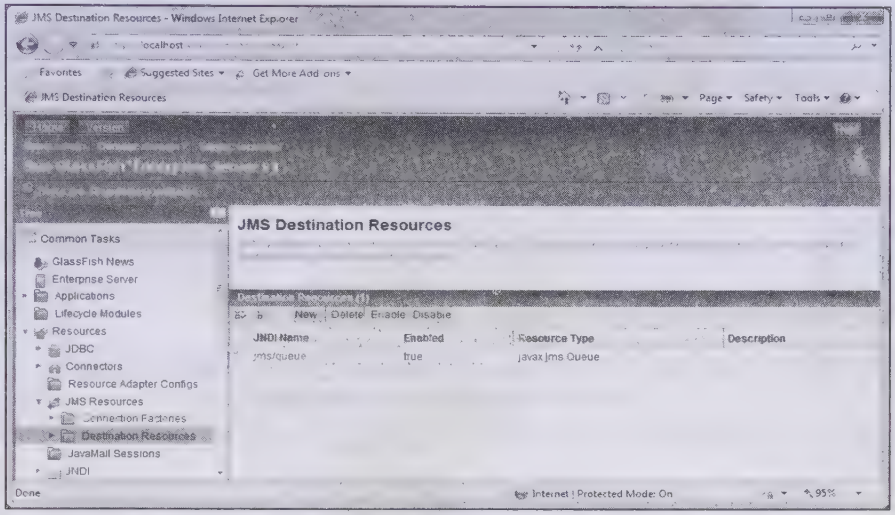


Figure 13.20: Displaying the jms/Queue Destination Resource

After configuring the JMS connection factory and JMS destination resource for the given application, deploy this application in the same way as done for the Hello and Cart applications. Upload the MDB.ear file on the Glassfish V3 application server and deploy the MDB application. Now, let's run the application.

Running the Application

To run the MDB application, open Internet Explorer and browse the http://localhost:8080/MDB-war/calculator.jsp URL. A calculator is displayed, as shown in Figure 13.21:

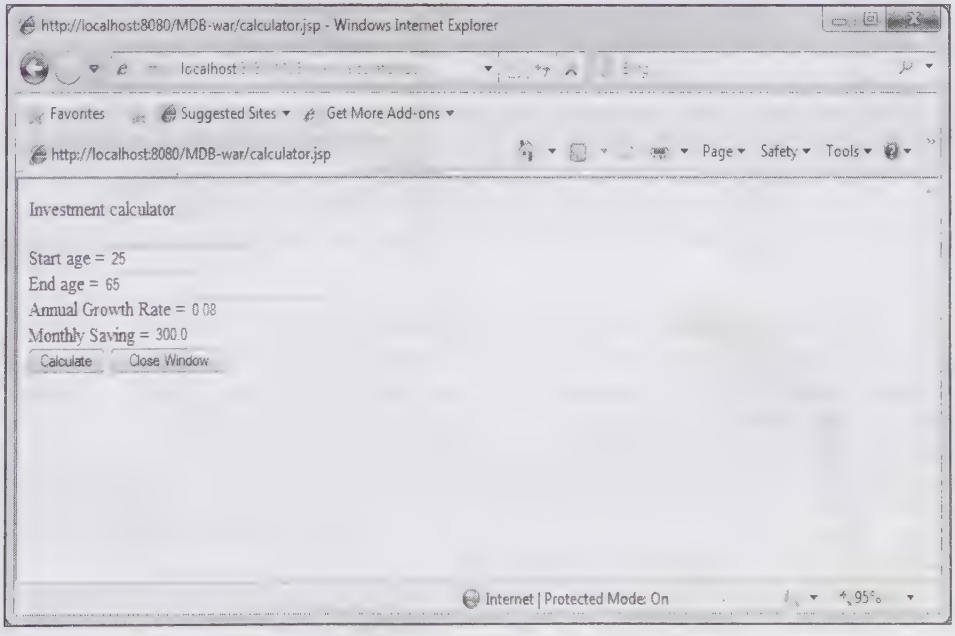


Figure 13.21: Displaying the calculator.jsp Page

As shown in Figure 13.21, enter the details and click the Calculate button. The `Please wait while I am checking whether the message has arrived` message appears, as shown in Figure 13.22:

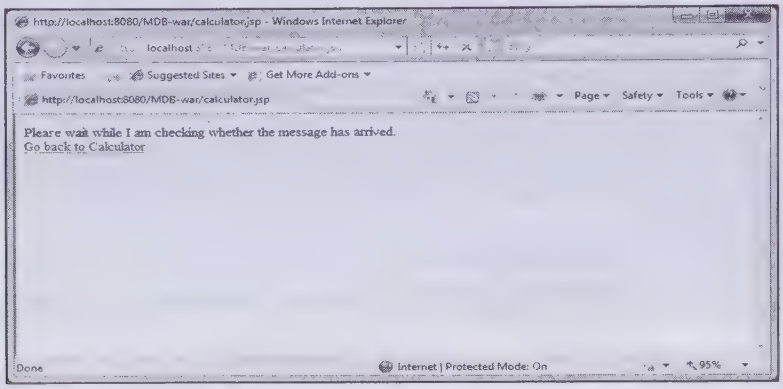


Figure 13.22: Displaying a Message

After a few seconds, the browser window changes and displays the time at which the message was sent, as shown in Figure 13.23:

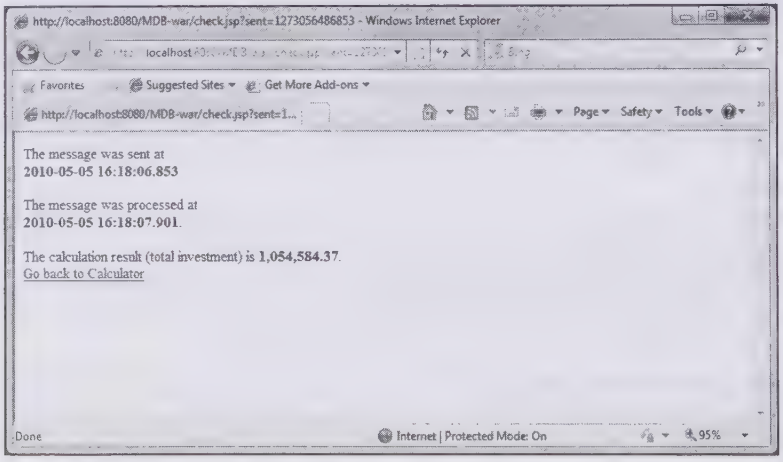


Figure 13.23: Displaying the Time at Which Message was Sent

Now, after understanding how to create and run an MDB application, let's discuss how to manage transactions in Java EE applications.

Managing Transactions in Java EE Applications

A transaction is defined as a group of operations that should be performed as a single unit. These operations can be synchronous or asynchronous, and can involve persisting data objects, sending mails, validating credit cards, and so on. A common example of a transaction involving operations is an online transfer of funds. In this transaction, one operation debits amount from one account and updates the record in a database. The other operation credits the same amount to another account, which updates another row in the same or a different database. This whole process involves two different operations— updating an account by debiting the amount and updating another account by crediting the same amount. This process of money transfer cannot be considered complete if any of these two operations fails.

The operations in a transaction are performed in a sequential or concurrent manner. The transaction is not considered complete and committed until all the operations are not performed successfully. If any of these operations fail due to any error or invalid condition, then the transaction is considered incomplete and all changes done by various operations roll back, that is, they are undone.

Let's discuss the transaction properties in detail.

Exploring Transaction Properties

Any type of transactions whether it is simple or complex, small or large contain some features in common, known as ACID components. ACID is defined as four characteristics for a reliable transaction:

- Atomicity
- Consistency
- Isolation
- Durability

A transaction must comply with these four properties to be considered a transaction.

Let's take a look at each of these four ACID properties.

Atomicity

A transaction consists of one or more operations that are performed as a group, known as a unit of work. As per the atomicity property, a transaction must always be treated as a single unit similar to an atom. A transaction is always a set of different processes and all these processes must perform successfully to make a transaction complete. A transaction fails if any of these processes fails. In other words, either all processes of a transaction affect the operations or none of them affects the operations at all.

Let's consider a scenario where you need to draw a specific amount from an account and deposit in another account. This transaction involves a number of steps, such as entering details of the check and updating two accounts for the check amount. However, the transaction is stopped if the account does not have sufficient funds. Therefore, any changes that were made by the transaction till now must be undone.

A transaction indicates the beginning and ending steps of a logical grouping of a task. The changes are effected if all of these steps complete successfully; otherwise; the transaction is rolled back. This is the atomicity property and it ensures that either all the operations in a transaction are performed successfully or none of the operations are performed. The rule of atomicity is violated if some of the operations are executed successfully.

Consistency

All transactions interact with a database to manipulate information. The data is said to be in consistent state if it is in agreement with all the constraints or rules defined for the database.

According to the consistency property of transactions, the computer system must remain in the consistent state before and after the transaction is performed, regardless of whether the transaction succeeds and is committed, or fails and is rolled back.

The consistency is assured both by transaction manager and developer. Transaction manager ensures that ACID properties of a transaction are intact and the developer ensures its consistency by specifying different constraints.

To maintain consistency of the database when a transaction is committed, the consistency of transaction is validated by Database Management System (DBMS) against all the constraints defined before the changes made by transactions are reflected in the database. If the results do not satisfy the requirements, the transaction is rolled back.

Isolation

The isolation property of a transaction ensures that the data being interacted by the transaction is not accessible by another transaction. In other words, no other transaction is allowed to access the data used by the running transaction until its execution completes.

The isolation property is important to support concurrent access to data. The probability of getting errors and corruption of data is high in a concurrent system, if the operations are not controlled properly.

The transaction manager takes the responsibility of ensuring isolation and managing concurrent execution of transactions.

Transaction isolation specifies that the intermediate state of a transaction is not exposed at all. Outside programs viewing the data objects involved in a transaction must not see the modified data objects until the transaction has been committed.

Durability

After a transaction is committed, the changes that are affected by the transaction should become visible to other applications. The durability property of transactions defines that all the changes made by the transaction must be recorded in some permanent storage. These changes can be recorded in some log files known as transaction log. A transaction log is a list that shows all the changes made by a transaction to the database. This helps in recovering the data to a previous valid state in case of the loss of data due to some error or system failure.

Exploring Transaction Model

A transaction model is a generalized framework that describes a class of transactions. Based on the transaction usage and structure, following are the transactional models:

- Flat transactions
- Nested transactions
- Chained transactions
- Saga transactions

Let's discuss each of them in detail.

Flat Transactions

A flat transaction is the simplest type of transaction. It is a single task comprising a number of steps. If any of the steps of this transaction fails, the transaction is rolled back. Flat transactions are standard for database operations and EJBs.

Nested Transactions

Nested transactions, as the name suggests, are the transactions that are further nested inside another transaction. In the nested transactions model, a transaction A can have another transaction B, as transaction A comprises multiple processes. In other words, a nested transaction has several sub-transactions. A sub-transaction can be either a flat transaction or another nested transaction.

Figure 13.24 shows an example of a nested transaction:

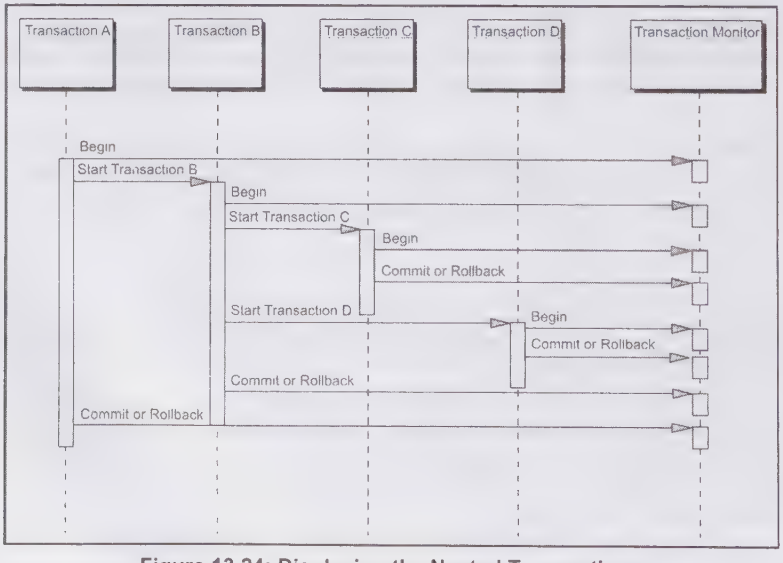


Figure 13.24: Displaying the Nested Transactions

In Figure 13.24, the Transaction A begins and starts the Transaction B. The Transaction B, in turn, first invokes the Transaction C and then invokes the Transaction D.

Another example of nested transactions is an airline reservation system. Let's say that you want to book a flight from New Delhi to Mumbai. Your first transaction (T) entails your attempt to book a direct flight. If there is no direct flight, your transaction fails. This is a flat transaction. Then you start a new transaction (U) and try to find a flight that is available between New Delhi and Pune. You want to hold this reservation, so you keep the transaction open. You then begin a new transaction (V), with your current one that will attempt to book a flight from Pune to Mumbai. There are no direct flights between the two cities, so your transaction fails and rolls back. However, since the transaction is localized at Pune, you will not lose the New Delhi booking. Next, another transaction (W) is started to book a flight between Pune and Hyderabad. You are successful in finding a reservation, so you hold transaction W. Finally, you initiate a new transaction (X) hoping that you will find a flight from Hyderabad to Mumbai. If you are successful, then you commit transaction X. Transaction W checks the status of transaction X and seeing that it was successful, W commits. This continues with W's parent transaction, that is U, and then T. With all of the transactions committed (T, U, W, and X), you have booked your flight. If you decide to scrap the whole affair thinking it a complex affair, X would roll back. W would see that X has failed and, as part of a business rule, would roll back. Finally, the entire tree of transactions (T, U, W, and X) would be aborted canceling the entire transaction.

NOTE

Nested transactions do not roll back the transaction in which they are contained.

Chained Transactions

A series of transactions is referred as serial transactions or chained transactions. These transactions are contiguous and related to each other. The execution of the next transaction in a series of transactions is based on the result of the previous transaction. The set of transactions is submitted to the transaction manager and the transaction manager executes each transaction in a series one by one.

In a chained transaction, all resources being used by a transaction in the series are retained to make them available for the next transaction in the series. This is achieved by using the `commit()` and `beginTransaction()` methods in a single step. The resources which are not required by the next transaction in the series are released. Any transaction other than the one included in the series of transactions being executed cannot access the data being used by these chained transactions. If a transaction in the series fails, only the concurrent transactions are rolled back.

Figure 13.25 gives you an idea of the general progression through a chain of transactions:

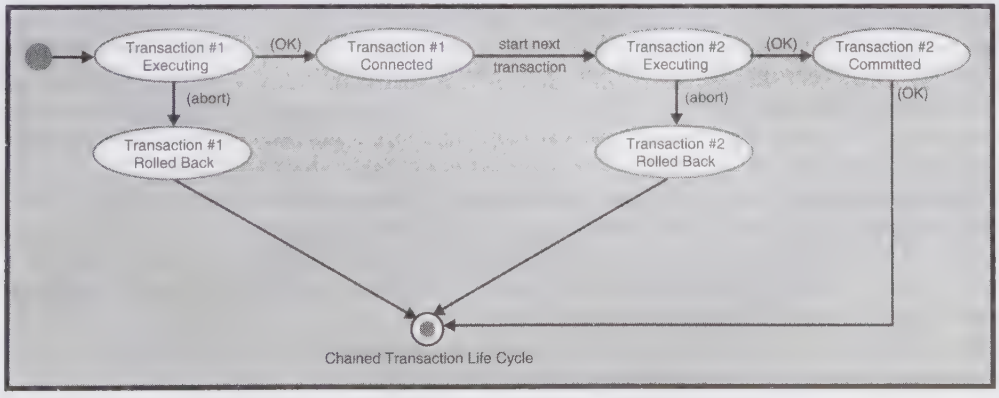


Figure 13.25: Displaying the Chain of Transactions

The disadvantage of chained transactions is that they can block a large set of valuable resources. Therefore, you should consider this disadvantage while deciding the set of transactions in a chained transaction.

Saga Transactions

Saga transactions, similar to the chained transactions, are long-lived transactions. These transactions also consist of multiple transactions. However, each saga transaction has a corresponding compensating transaction. When any of the transactions fail, the compensating transactions for all the successfully executed transactions are invoked automatically by the transaction manager. Therefore, before the execution of the saga transaction, the relationship between the successfully executed transactions and their compensate transactions should be identified by the transaction manager.

Figure 13.26 illustrates a saga transaction that consists of three transactions:

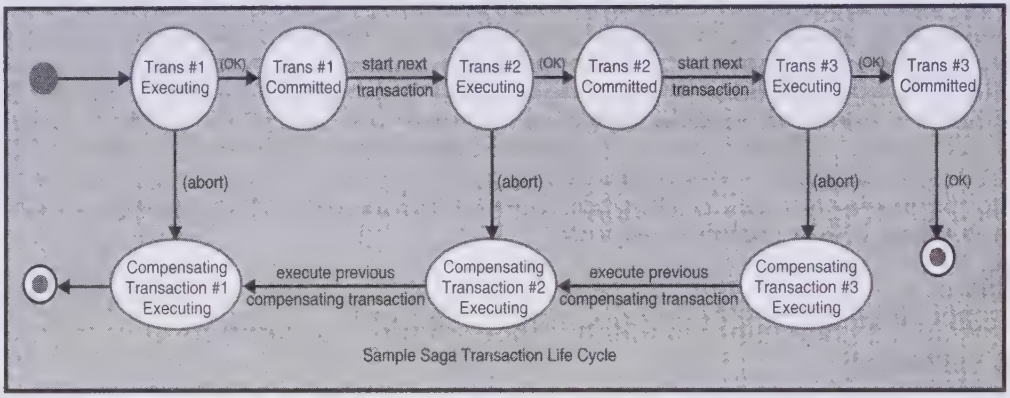


Figure 13.26: Displaying Various Transactions in a Saga Transaction

Explaining Distributed Transactions

A transaction may span more than one resource. This means that the context of some transactions can be propagated or shared by more than one component. Such types of transactions are known as distributed transactions. In other words, a distributed transaction is said to have taken place when the operations in a transaction are performed across database or other resources that reside on separate computers or processes. The distributed transactions support scenarios, such as when a component is required to communicate with multiple resources within the same atomic operation or when multiple components need to operate within the same atomic operation.

A distributed transaction often spans multiple resource managers. Each resource manager may be hosted on a heterogeneous processing node, may manage its own threads of control, and may have a different resource adapter. A distributed transaction model is more complex than the local one because it has more participants involved in it. The participants of a distributed transaction are as follows:

- **Transaction Originator**—Initiates a transaction. The transaction originator can be a Java application in the client tier, a servlet in the Web tier, or a session bean in the EJB tier. In the Web tier or EJB tier, the transaction originator can also be Java Messaging Service (JMS) producer/consumer.
- **Transaction Manager**—Manages transactions on behalf of the originator. It is responsible for enforcing the ACID properties as it coordinates access among all participating resource managers. Whenever resource manager fails to commit the transaction, the transaction manager decides to commit or rollback the pending transactions. In the J2EE architecture, Java Transaction API (JTA) is responsible for implementing the transaction manager.
- **Recoverable Resource**—Provides persistent storage for transaction data to ensure durability of the transaction. In most cases, it is a database or a flat file resource.
- **Resource Manager**—Manages a DBMS, a JMS provider, or a Java Connector Architecture (JCA) resource. It is one of the previously mentioned transaction-aware types.

Figure 13.27 summarizes the protocols and interactions among all participants of a distributed transaction:

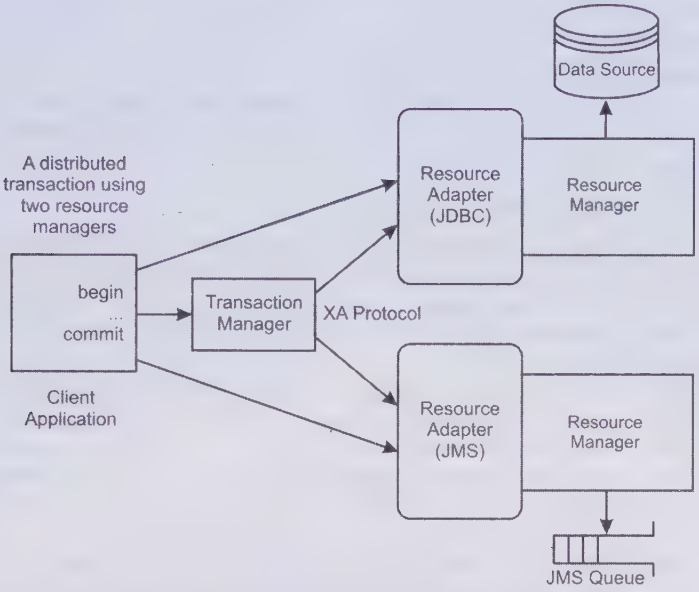


Figure 13.27: Displaying Protocols and Interactions in a Distributed Transaction

To perform a distributed transaction, the transaction manager coordinates the transaction execution across multiple resource managers. As all the participant resource managers are not aware of each other, an algorithm has been established as a standard protocol to control the interactions of all participants. This standard protocol is known as the Two-Phase Commit protocol.

The Two-Phase Commit (2PC) protocol enforces the ACID properties and is implemented into two phases:

- **Phase 1**—Refers to the preparation phase. The transaction manager or coordinator asks each resource manager to be prepared to commit a transaction (also called vote to commit). This process is mainly concerned with providing locks to shared resources without persisting data on permanent storage.
- **Phase 2**—Refers to a phase in which the transaction manager requests all the resource managers to commit their changes as a result of successful transaction execution. Otherwise, the transaction is rolled back, indicating the transaction failure to the application. The transaction succeeds, if and only if all the resource managers commit successfully.

Implementing Transaction Management in EJB 3

EJB 3 handles transaction management through JTA. In EJB 3, you can implement transactions in the following two ways:

- Using Container-Managed Transactions (CMT), which can be implemented by using annotations or Deployment Descriptor.
- Using Bean-Managed Transactions (BMT), which is used to manage transactions programmatically. In this case, all the transactions are not managed by the container; instead, the developer needs to provide code to manage transaction explicitly.

Let's now learn how transactions are supported by the EJB platform. To provide transactional capabilities needed by an EJB application, the EJB container is required to support the high-level interface defined by the JTA for transaction demarcation. The key specifications on transaction processing help you to understand JTA and its role in EJB.

The JTA defines a set of Java interfaces. It specifies how a transaction manager communicates with an application, a resource manager, and the application server in the J2EE architecture. JTA consists of the following primary interfaces:

- The javax.transaction.UserInterface interface
- The javax.transaction.xa.XAResource interface
- The javax.transaction.TransactionManager interface

In context of using JTA in EJB, you need to have the knowledge of the `javax.transaction.UserTransaction` interface. This is because the container takes care of most transaction management details behind the scenes.

Explaining Bean-Managed Transactions

In some cases, the CMTs do not provide the solutions which some enterprise beans require. Let's take an example of a client who wants to call several methods on a session bean even if none of the method commits its work. The EJB handles isolation of transactions in such a way that it terminates this problem. To do this, first turn off the entire container managed services by simply specifying the value of the TransactionManagementType field as BEAN by using the `@TransactionManagement(TransactionManagementType.BEAN)` annotation or by assigning equivalent metadata to the session bean in the ejb-jar.xml file. With BMT isolation, the EJB container provides the transaction support to the bean. The primary difference between CMT and BMT is that the enterprise beans in BMT calls begin, commit, and rollback transactions explicitly. Any given enterprise bean must select either CMT or BMT for the bean methods. Both bean types BMT and CMT can interact with each other within the same transaction context.

BMT is implemented in those enterprise beans that manage their own transactions. BMT can be used only for a session bean and not for an entity bean. The methods of a BMT do not contain transaction attributes. Let's see how a session bean manages transaction explicitly. To isolate transactions, an enterprise bean implements the `javax.transaction.UserTransaction` interface. This interface provides *`begin()`*, *`commit()`*, and *`rollback()`* transaction isolation methods to the bean, so that the EJB manages the transaction isolation explicitly.

A session bean initiates a transaction in one method by using BMT at the isolation level. The bean is then propagated to subsequent method calls until the transaction is finally committed in a separate method. This can be done by specifying the value for the `TransactionManagerType` field as `BEAN`, as shown in the following code snippet:

```
import javax.ejb.* ;
import javax.annotation.* ;
import javax.transaction.UserTransaction;
@Stateless
@TransactionManagement(TransactionManagerType.BEAN)
public class DepartmentManagerBean implements DepartmentManager
{
...}
```

Enterprise beans need to obtain the UserTransaction object to manage their own transactions. The object can be obtained either by using the @Resource annotation or by accepting a reference of the object provided by EJBContext to an enterprise bean, as shown in the following code snippet:

```
import javax.ejb.* ;
import javax.annotation.* ;
import javax.transaction.UserTransaction;
@Stateless
@TransactionManagement(TransactionManagerType.BEAN)
public class DepartmentManagerBean implements
DepartmentManager {
  @Resource SessionContext ejbContext;
  public void someMethod( )
  {
        try
        {
                UserTransaction ut = ejbContext.getUserTransaction( );
                ut.begin( );
                // Do some work.
                ut.commit( );
```

```
        }
        catch(IllegalStateException ise)
        {
                ...
        }
        catch(SystemException se)
        {
                ...
        }
        catch(TransactionRolledbackException tre)
        {
                ...
        }
        catch(Exception e)
        {
                throw new EJBException(e);
        }
. . . .
. . . .
```

Alternatively, the UserTransaction object can be directly injected into the bean, as shown in the following snippet (see user of `@Resource`):

```
import javax.ejb.* ;
import javax.annotation.* ;
import javax.transaction.UserTransaction;
@Stateless
@TransactionManagement(TransactionManagerType.BEAN)
public class DepartmentManagerBean implements DepartmentManager
{
  @Resource UserTransaction ut;
  ...
}
```

An enterprise bean can also access the `UserTransaction` object from JNDI ENC. The enterprise bean performs the lookup by using the `java:comp/env/UserTransaction` context, as shown in the following code snippet:

```
InitialContext jndiCntx = new InitialContext( );
UserTransaction tran = (UserTransaction)
  jndiCntx.lookup("java:comp/env/UserTransaction");
```

In Java EE, a client application can get the UserTransaction object by using JNDI. The following code snippet shows how a client obtains the `UserTransaction` object if the EJB container is a part of a Java EE system:

```
Context jndiCntx = new InitialContext( );
UserTransaction ut = (UserTransaction)
  jndiCntx.lookup("java:comp/UserTransaction");
ut.begin( );
...
ut.commit( );
```

Explaining Container-Managed Transactions

In case of CMT, the EJB container is responsible for managing the transaction demarcation for every method of the bean. Transaction behavior is described in the bean's Deployment Descriptor or in the annotated class. Transaction attributes are used to define handling of transactions by the EJB container on the invocation of bean's methods. These methods can be associated with any number of transactions at a time. A transaction begins with the execution of the method and commits just before the completion of the execution of the method. Not all the methods of the bean are associated with transactions; however, only those methods are supported that are specified by transaction attributes in the bean's Deployment Descriptor. The following section introduces the transaction attributes that are vital in configuring the bean's methods participating in a CMT.

Enterprise JavaBeans support six types of transactional attributes:

- REQUIRED
- REQUIRES_NEW

- MANDATORY
- SUPPORTS
- NOT_SUPPORTED
- NEVER

Let's discuss these in detail.

REQUIRED

The REQUIRED attribute defines that the specified method always executes within a transaction. In case, the method is invoked within the context of some existing transaction, the same transaction is used to perform a task; otherwise, a new transaction is created by the container.

The new transaction starts with the method invocation and commits just after the execution ends. If the transaction does not commit, then the changes made by the transaction are not reflected and the RollBackException exception is thrown.

The REQUIRED attribute is used when a method is involved to make some serious data change that needs to be protected by a transaction.

REQUIRES_NEW

The `REQUIRES_NEW` transaction attribute assures that a method is always executed within a new transaction. A new transaction is created before the method is invoked. It means that the method is going to be executed into its own transaction and is helpful in implementing local rollback and local commit. The local rollback and local commit mean that a transaction does not affect outcome of other transactions outside the method.

If the method is invoked in context of some existing transaction, this transaction is suspended and the new transaction is started. The suspended transaction is reinstated after the completion of new transaction.

MANDATORY

The MANDATORY attribute assures that a method is going to be executed in the context of some pre-existing transaction only. Therefore, it becomes mandatory that a client should have a transaction running and a method would be invoked in the context of the running transaction. In case there is no transaction context, the container throws the `TransactionRequiredException` or `TransactionRequiredLocalException` exception.

You should use the MANDATORY attribute when the method needs to verify that a component was invoked within the context of a transaction managed by a client.

SUPPORTS

The SUPPORTS attribute assures that a method can be invoked with an existing transaction context; however, if there is no transaction context, the method can be invoked without a transaction.

In case, when a transaction does not need to incur processing overhead or suspend and resume an existing transaction, then you can use the `SUPPORTS` attribute. You must also ensure that your method does not cause any exception which signals a failure within the context of a transaction.

NOT_SUPPORTED

The `NOT_SUPPORTED` transaction attribute informs the container that a method should not be invoked within a transaction. If the method is invoked by using some existing transaction context, the transaction is suspended before the method is invoked. Any exception caused by the method does not affect this suspended transaction.

NEVER

The NEVER transaction attribute specifies that a method should never be invoked in context of another transaction. You should use this attribute when you need to check that the method is not invoked within a transaction managed by a client and the container is not going to attempt to provide a transaction for it.

This attribute is used when the given method is not capable enough to participate in a transaction. If the method is invoked and a transaction context exists, the container throws the `RemoteException` or `EJBException` exception for remote and local clients, respectively.

Explaining EJB 3 Timer Services

The EJB applications are meant for large-scale enterprises that operate on the principle of job scheduling or task scheduling. In simple words, job scheduling refers to the scheduling or managing tasks and activities of an organization within time constraints. These scheduled tasks may be either date-based or time-based, or recurring—that is, performed on a routine basis. In other words, large organizations would perform these tasks at a particular point of time or within a period of time. This would become much clearer with the help of the following scenarios:

- In a customer care center, all complaints are addressed and processed according to the assigned batches. A batch can be the number of complaints received in one hour, a day, or at any fixed period. The complaints are grouped in batches so that necessary action can be promptly taken to resolve these complaints. All the complaints made in one single day are settled together instead of dealing with them separately. This work is usually scheduled in the evening to reduce the load of processing on the system.
- In large-scale enterprises, managers need to run specific reports on a daily basis. A scheduling system can help by running these reports automatically according to the scheduled time and deliver them as e-mail to the managers.
- In a system dealing with mortgages, several different tasks need to be completed before a mortgage can be closed. The operations may concern appraisals, closing appointments, and so on. The processing of this job is time-bound and is, therefore, carried on schedule.
- A company with multiple job shifts needs to run the attendance report after the completion of the job shift. This is done on a regular basis to maintain a daily record of the employees' attendance.

The tasks specified in the preceding scenarios require task scheduling. To implement task scheduling in `Java EE` applications, `EJB 3` has introduced the Timer service facility. Timer services are used for the purpose of building J2EE applications based on time-based services. The time-based services are basically used in scheduling applications. In technical terms, scheduling applications are called workflows, and they define the sequence or batch of tasks to be performed at a particular point of time.

Prior to EJB 2.1, building and deploying time-based workflows need to be performed manually. However, with `EJB 3`, the task of creating such applications has been considerably simplified. Equipped with annotations and DIs, developers can use `EJB 3` to build and deploy scheduled applications easily. The EJB container provides Timed Event API for the Timer service facility. The Timed Event API schedules the timer for a specific date, period, or interval. As soon as the Timed Event API schedules the timer, the timer becomes associated with the enterprise bean responsible for setting it. The Timed Event API calls the `ejbTimeout()` method of the enterprise bean as soon as the `Timer` object expires. Let's now learn about the different types of timers and understand their functions.

Different Types of Timers

EJB supports the following two forms of timer objects:

- The Single Action Timer
- The Interval Timer

The Single Action Timer

The single action timer expires only once, which means that there is no subsequent instantiation and expiration of this timer. It is also known as a single interval timer. EJB provides two ways for constructing a single action timer—one is to create a timer which expires at a specific point of time, such as a specified date, and the other is to create a timer which expires after a certain period of time, such as 6 hours or 3 days. The enterprise bean receives the notification, when the specified time expires. In other words, once the specified time expires, the container calls the `ejbTimeout()` method or the method associated with the `@Timeout` annotation to destroy the timer.

The Interval Timer

The interval timer recurs at multiple intervals of time and expires multiple times at regular intervals. As with the single action timer, there are two ways of constructing an interval timer. The first way is to create a timer having an initial expiration at a particular point of time, and the subsequent expirations happening at specified intervals. The other way is to create the timer having the initial expiration after a certain period of time and subsequent expirations happening after specified intervals.

Strengths and Limitations of EJB Timer Services

The Timer service facility provided by Java EE has eased the work scheduling in Java EE applications. The strengths of Timer service are as follows:

- The timers are persistence objects. It implies that even if the server shuts down, the timers will still be available and become active again as soon as the server restarts. This is possible because the Timer service persists the `Timer` object. The `Timer` object is stored in the database, and becomes active again once the server starts normal functioning.
- The EJB Timer service facility is used to build robust scheduling applications.

However, there are certain flaws as pointed out by Java developers, leaving a lot of scope of improvement as far as the EJB Timer service is concerned. Some limitations of the Timer service facility are as follows:

- Very little information can be retrieved about the `Timer` object itself. It is very difficult to know whether the timer is a single action timer or an interval timer.
- There is no mechanism in an interval timer to predict that timer has executed its first expiration or not.

Timer Service API

The Timer Service API provides an interface to implement the Timer service facility in Java EE applications. The Timer Service API schedules the timers for the specified date or for recurring intervals. To use the Timer service facility, an enterprise bean must implement the `javax.ejb.TimedObject` interface. This interface defines the callback method, `ejbTimeout()`.

Since `EJB 3` focuses on the use of annotations, the `@javax.ejb.Timeout` annotation can be applied to a method. However, the argument passed to this method must be void and should have the `javax.ejb.Timer` object as one of its parameters. The four interfaces provided by the Timer Service API are as follows:

- The `TimerService` interface
- The `Timer` interface
- The `TimedObject` interface
- The `TimerHandle` interface

Let's discuss these interfaces in detail.

The TimerService Interface

The `TimerService` interface is used to create timers. It also helps in retrieving existing timers from the EJB container. Now that you are familiar with the use of the `TimerService` interface, let's study the following in detail:

- Creating Timers
- Listing Existing Timers

Creating Timers

The TimerService interface creates a timer with the help of the createTimer method. Timers can be either single action or interval, depending on their expiry notifications.

The TimerService interface provides the following two methods for creating single action timers:

```
Timer createTimer (long duration, Serializable info)
Timer createTimer (Date expiration, Serializable info)
```

While working with the different types of timers, it was observed that the single action timer lapses after the specified duration of time or after the specified point of time. The first method creates a single action timer that

expires after the specified duration of time, while the second method creates a single action timer that expires at a specified date. In both the methods, the first parameter specifies the expiry notification and the second parameter specifies the application information to be passed to the `Timer` object.

NOTE

If there is no application-specific information to be passed, then the second parameter can be ignored by passing a null value. In addition, the duration time must be in milliseconds.

The TimerService interface provides the following two methods to create an interval timer:

```
Timer createTimer (Date initialExpiration, long intervalDuartion, Serializable info)
Timer createTimer (long initialDuration, long intervalDuartion, Serializable info)
```

While creating interval timers, the createTimer method takes three arguments. The first argument, `initialExpiration`, is of the Date type and accepts the date when the expiration of a task would begin. The second argument, `intervalDuration`, specifies the subsequent expiration in milliseconds. Moreover, just as in single action timers, interval timers also provide application-specific information to the Timer object at the time of their creation.

Listing Existing Timers

You can use the getTimers() method to retrieve a list of existing timers that have been already created by the other enterprise beans. The getTimers() method returns the java.util.Collections class, which is essentially an unordered collection of existing javax.ejb.Timer objects. Each Timer object represents a separate timed event for the enterprise bean. This method is also helpful in managing Timer objects. With the help of the getTimers() method, the enterprise bean can cancel any timer which is no longer in use or it can reschedule the timer according to the requirement.

The following code snippet defines the getTimers() methods of the TimerService interface:

```
package javax.ejb;
import java.util.Date;
import java.io.Serializable;
public interface TimerService
{
// retrieves all the active timers associated with this bean
public java.util.Collection getTimers( ) throws
IllegalStateException,EJBException;
}
```

Now, let's discuss the next interface of the Timer Service API called the `Timer` interface.

The Timer Interface

The `Timer` interface holds information of the created `Timer` object throughout the EJB life cycle. This interface is available in the javax.ejb package. After the `Timer` objects are returned by the `TimerService.createTimer()` and `TimerService.getTimers()` methods, the information in these objects is updated. The following code snippet shows the implementation of the `Timer` interface:

```
package javax.ejb;
public interface Timer
{
  public void cancel( )
  throws IllegalStateException,NoSuchObjectLocalException,EJBException;

  public java.io.Serializable getInfo( )
  throws IllegalStateException,NoSuchObjectLocalException,EJBException;

  public java.util.Date getNextTimeout( )
  throws IllegalStateException,NoSuchObjectLocalException,EJBException;

  public long getTimeRemaining( )
  throws IllegalStateException,NoSuchObjectLocalException,EJBException;

  public TimerHandle getHandle( )
  throws IllegalStateException,NoSuchObjectLocalException,EJBException;
}
```

The `Timer` interface provides the following methods:

- `public void cancel()` – Cancels the `Timer` object. When the Timer object is cancelled, all notifications associated with the expiration of the timer are also cancelled. Then, the enterprise bean does not receive timed events. Cancelling a timer is useful when you want to remove the timer or reschedule it. You have to cancel an old timer and create a new one whenever you want to reschedule a timed event.
- `public java.io.Serializable getInfo()` – Returns the application-specific information passed to the `getInfo()` method at the time of creating `Timer` objects. This method must implement the Serializable interface to write the state of the `Timer` object.
- `public java.util.Date getNextTimeout( )` – Returns the time when the next expiration of the `Timer` object is due, as a Date object.
- `public long getTimeRemaining( )` – Returns the time, in milliseconds, left for the next expiration of the `Timer` object. Similar to the `getNextTimeout()` method, this method also returns the time when the next expiration will occur; however, the former returns the date as an object while the latter returns the milliseconds as an object.
- `public TimerHandle getHandle( )` – Provides the reference of the `TimerHandle` interface which can be used later for reconstructing or rescheduling the `Timer` object.

The TimedObject Interface

The TimedObject interface provides callback methods to deliver timer expiration notifications. After the timer expires, the `ejbTimeout()` method is invoked by the EJB container. If the timer is a single action timer, then the `ejbTimeout()` method will be invoked only once; however, if it is an interval timer, then the `ejbTimeout()` method is invoked multiple times. The TimedObject interface is implemented by the enterprise bean to receive time-based notification. When the `Timer` object expires, the EJB container calls the `void ejbTimeout(java.ejb.Timer timer)` method.

The following are the two ways to implement the callback method:

- Implementing the enterprise bean with the `TimedObject` interface. After implementing the enterprise bean, you should implement the `void ejbTimeout(java.ejb.Timer timer)` method, as shown in the following code snippet:

```
@Stateless

@Remote

class FirstBeanImpl implements FirstBean,TimedObject

{
   public void ejbTimeout(Timer timer)
   {
         // code to implement ejbTimeout method
   }
}
```

- Annotating a method with the `@Timeout` annotation. In this way, instead of implementing the Bean class with the `TimedObject` interface, the method defined within the Bean class is annotated. The return type of the method must be void and should accept only one argument of `Timer` type, as shown in the following code snippet:

```
@Stateless
@Remote
class FirstBeanImpl implements FirstBean
{
   @Timeout
   public void TimeOutMethod(Timer timer)
   {
         // code to implement ejbTimeout method
   }
}
```

Let's now learn about the `TimerHandle` interface.

The TimerHandle Interface

The `TimerHandle` interface is used by the container so that the `Timer` object can be rescheduled or reconstructed by the enterprise bean. This interface provides the `Timer.getHandle()` method that returns the `TimerHandle` object. This `TimerHandle` object can then be saved either in a file or at some other resource to regain access to the timer.

The following code snippet shows the implementation of the TimerHandle interface:

```
package javax.ejb;
public interface TimerHandle extends java.io.Serializable {
   Public Timer getTimer() throws NoSuchObjectLocalException, EJBException;
}
```

The TimerHandle objects cannot be used outside the container because these objects are generated by the container and therefore are considered as local objects. The local enterprise beans are located within the same container; thereby, making it possible for the `TimerHandle` objects to pass to the local enterprise bean by using local interfaces. The `TimerHandle` interface is valid until the timer exits. In other words, if the timer expires, then the `TimerHandle` interface becomes invalid. In this case, if you call the `getTimer()` method, then the `javax.ejb.NoSuchObjectException` exception is thrown. Even if the Cancel method is invoked for a timer's explicit expiration before the timer's expiration date, then also the `TimerHandle` becomes invalid.

Now, let's create an application implementing the EJB 3 Timer service.

Implementing EJB 3 Timer Service

The enterprise bean creates an instance of the `Timer` object by using the Timer service facility The Timer service lasts permanently by storing the `Timer` object in the database. Even if the server shuts down, the timer still remains available and becomes active as soon as the server resumes normal functioning.

Figure 13.28 shows the relationship between the enterprise bean and the `Timer` interface:

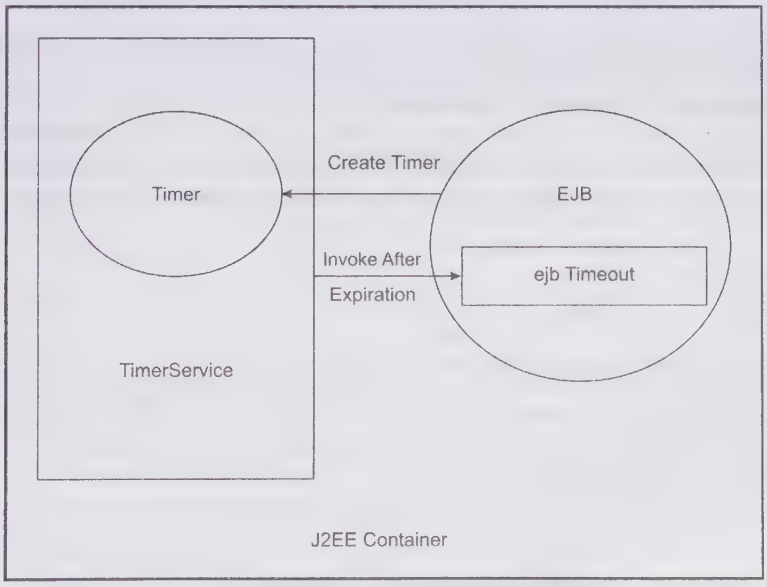


Figure 13.28 Displaying the Relationship between Enterprise Bean and Timer

As shown in Figure 13.28, the enterprise bean creates the `Timer` object with the help of the `TimerService` interface. The `TimerService` interface provides the `createTimer()` method to create the `Timer` object as per the requirement. As soon as the expiry date of the `Timer` object arrives, the `TimerService` interface invokes the `ejbTimeout()` callback method to destroy the `Timer` object (Figure 13.28).

The APIs of the Timer service which help to create and handle Timer objects have been already discussed. Let's now learn how the enterprise bean invokes the various methods of the Timer Service API for creating or cancelling `Timer` objects. To implement the Timer service facility, you need to provide the code to:

- Create the `Timer` object
- Cancel the `Timer` object
- Expire the `Timer` object

Now, let's discuss each of these in detail.

Creating Timer Objects

You can use the createTimer() method of the TimerService interface to create the Timer object. However, you learn how the enterprise bean uses the `createTimer()` method to create the `Timer` object. Stateless beans and MDBs create `Timer` objects. The enterprise bean creates a Timer object by performing the following three tasks:

- Obtaining the reference of the EJBContext object
- Obtaining the reference of the `TimerService` interface using the context object
- Creating the `Timer` object using the `createTimer` method of the `TimerService` object

Let's now look at each task individually.

Obtaining Reference of EJBContext

`EJB 3`, equipped with new features of annotation and DI, has simplified the task of getting the reference to the `EJBContext` object. This class has two subclasses, `SessionContext` and `MessageDrivenContext`. These subclasses have various methods to manipulate runtime environment of session and MDBs, respectively. You have to create only the references of these classes to obtain the `EJBContext` instance.

The following code snippet shows how to obtain the reference for the stateless session bean and the MDB, respectively:

```
@Resource
Private SessionContext sctx;
@Resource
Private MessageDrivenContext msgdrvnctx;
```

In the preceding code snippet, when the container creates session and MDB instances, it implicitly passes both context references, SessionContext and MessageDrivenContext, to the created beans instances, respectively.

After getting the reference to the EJBContext object, let's learn how to obtain the reference to the `TimerService` object.

Obtaining Reference to TimerService

The `getTimerService()` method provides the reference to the `TimerService` interface. The following code snippet shows how to obtain the reference to the `TimerService` object:

```
TimerService timersrvc = sctx.getTimerService();
TimerService timersrvc =msgdrvnctx.getTimerService();
```

In the preceding code snippet, timersrvc is the `TimerService` reference obtained with the help of the reference of `EJBContext`. The stateless session `EJBContext` reference, sctx, invokes the `getTimerService()` method to obtain the reference to `TimerService`.

After getting the `TimerService` reference, the `Timer` object is created by invoking the `createTimer()` method.

Creating the Timer Object Using the createTimer () Method

As discussed earlier, timers are of two types, single action timer and interval timer. You can create a `Timer` object according to your requirement.

Let's first discuss how to create a single action timer. The following code snippet shows how to create a `Timer` object that will expire after 50 seconds from the current time:

```
Calendar today =Calendar.getInstance();
Timer timer =timersrvc.createTimer( today.getTimeInMillis() + (50 * 1000), null);
```

In the preceding code snippet, a `Timer` object (single action) is created by calling the `createTimer()` method through the `TimerService` reference.

The following code snippet shows how to create a single action timer which will expire on a specified date (18th November 2008):

```
Calendar date = new GregorianCalendar(2008, Calendar.NOVEMBER, 18);
Timer timer = timersrvc.createTimer(date, null);
```

Apart from the single action timer, you can also create an interval timer, as shown in the following code snippet:

```
long fiveweeks = 5* 7 * 24 * 60 * 60 * 1000;
Timer timer=timersrvc.createTimer(fiveweeks, (4 * 7 * 24 * 60 * 60 * 1000), null);
```

In the preceding code snippet, the `Timer` object expires after 5 weeks and the subsequent expirations will occur after every 4 weeks.

The following code snippet shows how to create an interval timer, which will elapse on 18th November 2008 for the first time and then the subsequent expirations will take place after every two weeks:

```
Calendar date = new GregorianCalendar (2008, Calendar.NOVEMBER, 18);
long twoweeks = (2 * 7 * 24 * 60 * 60 * 1000);
Timer timer = timersrvc.createTimer( date, twoweeks, null);
```

This is how the enterprise bean creates and uses the `Timer` object. The `Timer` object can also be explicitly cancelled by invoking the `Timer.cancel()` method. Let's now learn how to cancel a Timer object.

Canceling a Timer Object

The container cancels a `Timer` object that is no longer used by the `cancel()` method. After the `cancel()` method of the `Timer` object is invoked explicitly, the enterprise bean will not receive the callback methods, such as `ejbTimeout()`, from the container. In other words, after the cancellation of the `Timer` object, the enterprise bean will not receive any notification from the container. The following code snippet shows how to call the `cancel()` method:

```
Timer timer =timersrvc.createTimer(.....);
.....
timer.cancel();
```

In the preceding code snippet, the cancel() method is invoked on the timer object to cancel the timer.

Expiring the Timer Object

Expiration only happens in case of single action timers, and interval timers never expire. Instead, interval timers have to be stopped by invoking the `cancel()` method explicitly. A single action timer expires when its specified date or period of time has elapsed. After the timer expires, the container gives the notification to the enterprise bean by calling the annotated method or the `ejbTimeout()` method.

When you create a single `Timer` object, you have to specify the date after which the container expires the `Timer` object. Similarly, if you specify the period of time (in milliseconds) for a single action timer, the timer will expire after the specified time period.

Exploring EJB 3 Interceptors

Interceptors are useful in situations where you want to add some common features, such as logging, to all EJBs in your application. Now, it is very complex and time consuming task to add code for logging to all EJBs in each EJB class. It requires you to modify many EJB classes. EJB 3 interceptors can solve this problem easily. For example, in this case, you can simply create an EJB interceptor which performs logging and then you can make this interceptor as a default interceptor for your application. The default interceptor will be executed when any bean method is executed. If your requirement further changes, then you need to only change the logging interceptor.

Interceptors are objects that are automatically invoked when an EJB method is invoked. Generally, interceptors are used in session and MDBs. They can be defined within a bean class or in an external class and you can use any number of constructors within these classes. Interceptor encapsulates common code that you can reuse; however, it is not required to include it within the business logic in an application. In case of logging, it is not desirable to include the logging code into the EJB classes. Therefore, the use of interceptors in EJB 3 specification

makes applications simpler and easier to use. Interceptors are one of the parts of the changes made in EJB 3 to modularize the application and to extend the EJB container.

Specifying Interceptors

Interceptors are the methods that are used to implement business logic in a business method or in a life cycle callback method. The two types of interceptors available in the EJB 3 specification are as follows:

- Interceptors that intercept the business methods
- Interceptors that intercept the life cycle callback methods

The interceptors can be specified for both the session and MDBs. Interceptors can be specified within a bean class as well as in an external class. You can also use annotations to simplify the use of interceptors in your application. Annotations are used to specify whether an external class or a bean class can be used as an interceptor. The @Interceptor and @Interceptors annotations are used to determine whether an external class is used as an interceptor in an application. The @AroundInvoke annotation is used to identify a method as an interceptor. The method can be either in a bean class or in an external class. The interceptor methods can access the methods that are used to trigger them in an application. The interceptor methods can also be used to stop the processing of business methods. Initially, they check for some security information on the methods; if it is not found, it would halt the processing of the business methods.

The lifetime of an interceptor instance is the same as that of the life cycle of the bean instance with which the interceptors are associated. These interceptors are invoked for JNDI, JDBC, JMS, and other data sources. All interceptors are configured within a single package named javax.interceptor. The following procedure must be followed to specify interceptors for an application:

- Decide whether the interceptor methods are specified within a bean class or within an external class.
- Implement the interceptors within an external class by considering the following instructions:
  - Using the @javax.interceptor.Interceptors annotation in the bean class to associate the interceptor class to the bean class
  - Annotating the business method in the interceptor class with the @javax.interceptor.AroundInvoke annotation.
  - Specifying any number of interceptors within a bean class. Interceptors are executed according to the order specified in the annotation.
  - Specifying an interceptor either at class or method level. If you specify interceptor at the class level, then the interceptor methods can be applied to all appropriate bean class methods. On the other hand, if you specify interceptor at the method level, then the interceptor methods can only be applied to the annotated methods.
- If you decide to implement interceptors within a bean class, then you must consider the following instructions:
  - The methods and classes are annotated with the @javax.interceptor.ExcludeClassInterceptors annotation in the bean class.
  - The classes and methods of a default interceptor are annotated by using the @javax.interceptor.ExcludeDefaultInterceptor annotation. In other words, you can set one or more interceptors as default that you want to define for every EJB.
  - Default interceptors are configured within the ejb-jar.XML Deployment Descriptor and can be applied to all EJBs in the JAR file.

Exploring the Life Cycle of Interceptors

The life cycle of an interceptor is the same as that of the EJBs it intercepts. Interceptor classes are created, activated, and passivated according to the bean instances. The interceptor class also has the limitations of the bean classes with which they are associated. You cannot inject an extended persistent context into an interceptor class if that interceptor class does not intercept a stateful session bean.

The interceptor life cycle events are generated by the EJB container. The life cycle events have internal states that are held by the EJB container. The internal states are useful whenever you want the interceptor class to obtain an open connection to a remote system and to close the connection at the destroy time. Therefore, an internal state is maintained from the time of the creation of a bean instance to the destruction of the instance in which the interceptor class is intercepted. You can hold the internal state of the bean instance within the interceptor class and do the clean up when the interceptor and the bean instances are destroyed.

Working with the Interceptor Class

Interceptor classes are the plain Java classes that include the interceptor annotations to specify which methods intercept business methods and the life cycle callback methods. All the classes of interceptors must be annotated with the @javax.interceptor.AroundInvoke annotation. The methods of an interceptor must be either within an external class or within a bean class. The annotation used by the methods has the signature, as shown in the following code snippet:

```
// Signature for @javax.interceptor.InvocationContext //
@AroundInvoke
Object <any-method-name>(javax.interceptor.InvocationContext
invocation)
throws Exception;
```

The @AroundInvoke annotation is the primary annotation used by all interceptor classes and methods. As the name implies, the @AroundInvoke annotation wraps the program calls to the business methods that are available in the interceptor. The calls to the business methods are made by the bean instances in the same transaction. The javax.interceptor.InvocationContext interface represents the business method invoked by the client. You can access information of a target bean by using the InvocationContext parameter. A reference to the java.lang.reflect.Method class is made to determine the actual invoked method. The javax.interceptor.InvocationContext interface is used to start the invocation. You should provide the implementation of the business and life cycle callback methods in an interceptor class.

The javax.interceptor.InvocationContext interface is also used to invoke several methods from a bean class. All the methods, as shown in the following code snippet, belong to the javax.interceptor.InvocationContext interface:

```
package javax.interceptor;
public interface InvocationContext
{
   public Object getTarget( );
   public Method getMethod( );
   public Object[] getParameters( );
   public void setParameters(Object[] newArgs);
   public java.util.Map<String, Object> getContextData( );
   public Object proceed( ) throws Exception;
}
```

In the preceding code snippet, the getTarget() method returns a reference to the current or the target bean instance. The getMethod () method is used to invoke the name of the method being called. The getParameter() method is used to display the name of the parameters associated with the bean instance and the setParameter() method is used to set values for the parameters in a bean class.

Let's explore more about interceptors by understanding how to:

- Apply interceptors through XML
- Disable interceptors
- Use the business methods
- Use the life cycle callback methods
- Specify default interceptor methods

Let's discuss these one by one.

Applying **Interceptors** through XML

The `@Interceptor` annotation is used to apply the `interceptor` easily to your `bean` class. However, each time when the modifications are done in a bean class, you need to modify and recompile the classes. As `interceptors` are a part of the business logic, they may create complexities in modifying the class again and again as the business logic is changed from time to time. Therefore, the use of XML is a good practice for annotating code in a `bean` class. In EJB 3, source code annotations are used in place of XML Deployment Descriptors and due to this reason, EJB 3 supports partial XML deployment. Applying `interceptors` through XML is better and easier because it is easy to change XML file as the business method changes.

The following code snippet shows the use of `interceptors` in an XML file:

```
<ejb-jar
  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/ejb-jar_3_0.xsd"
  version="3.0">
  <assembly-descriptor>
  <interceptor-binding>
  <ejb-name> </ejb-name>
  <interceptor-class> </interceptor-class>
  <method-name> </method-name>
  <method-params>
  <method-param> </method-param>
  <method-param> </method-param>
  </method-params>
  </interceptor-binding>
  </assembly-descriptor>
</ejb-jar>
```

If you want to apply the `interceptor` to all the business methods of a particular `EJB`, then you need to modify the XML file (preceding code snippet), as shown in the following code snippet:

```
<ejb-jar
  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/ejb-jar_3_0.xsd"
  version="3.0">
  <assembly-descriptor>
  <interceptor-binding>
  <ejb-name> </ejb-name>
  <interceptor-class> </interceptor-class>
  </interceptor-binding>
  </assembly-descriptor>
</ejb-jar>
```

In the preceding code snippet, you can see that `<method-name>` and `<method-param>` have been omitted to make the `XML` file available to all the business methods in a `bean` class.

Disabling **Interceptors**

Whenever you use default or class-level `interceptors`, you may need to disable them for a particular `EJB` or a particular method in an `EJB`. You can disable the `interceptors` either by using the XML file or by using the `annotations`. The following code snippet shows how to disable `interceptors` using an XML file:

```
<ejb-jar
  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/ejb-jar_3_0.xsd"
  version="3.0">
  <assembly-descriptor>
  <interceptor-binding>
  <ejb-name>*</ejb-name>
  <interceptor-class>
```

```
    </interceptor-class>
    </interceptor-binding>
    </assembly-descriptor>
</ejb-jar>
```

In the preceding code snippet, the XML file is deployed and configured in the JAR file. The preceding code snippet disables the default interceptor and the interceptor is made applicable to method level only.

To exclude the default interceptor from a bean class, the @javax.interceptor.ExcludeDefaultInterceptor annotation is used. The following code snippet shows how to disable a default interceptor in a bean class:

```
@Stateful
@ExcludeDefaultInterceptors
@Interceptors
(com.titan.SomeOtherInterceptor.class)
public class ExampleInterceptor implements MyInterceptor
{
   ....
}
```

Using the Business Method Interceptors

Business method interceptors can be specified by using the @AroundInvoke annotation. The @AroundInvoke annotation can be represented only once throughout the program structure. It can be associated with the bean class or with the interceptor class. You can execute multiple interceptor classes for a business method by specifying multiple interceptor classes within a bean file. The interceptor methods are executed according to the order they are specified within the interceptor classes. The @AroundInvoke annotation cannot be used with a business method. The following code snippet shows the signature of the @AroundInvoke annotation for a business method:

```
Object<METHOD>(InvocationContext) throws Exception
```

The invocationContext.proceed() method should be called by the method that is annotated with the @AroundInvoke annotation. No other business method or any subsequent @AroundInvoke methods are invoked. The business method invocation occurs within the same transaction in which the @AroundInvoke method is invoked. The business method interceptor may throw the runtime or the application exception. These exceptions are allowed in the throws clause of the business method.

Using the Life Cycle Callback Methods

The life cycle callback methods are obtained from the life cycle events generated by the EJB life cycle. The life cycle events are generated by the EJB container. The creation, passivation, and destruction of the bean instance in an EJB container are the life cycle events of an EJB. Some annotations are used to specify that the method is a life cycle callback interceptor method. These annotations are given as follows:

- @javax.ejb.PrePassivate—Refers to the annotation used for the method to be notified by the EJB container, when it is about to passivate a stateful session bean.
- @javax.ejb.PostActivate—Refers to the annotation used for the method that is to be notified by the EJB container after re-activation of the stateful session bean.
- @javax.annotation.PostConstruct—Refers to the annotation used for the method that is notified by the EJB container before the business method is invoked and after the DI is applied to a resource. The annotation is in the javax.annotation package instead of the javax.ejb package.
- @javax.annotation.PreDestroy—Refers to the annotation used for the method that is notified by the EJB container before destruction of the bean instance. This annotation is applied to the method used to release the resources held by the bean class. The annotation is in the javax.annotation package instead of the javax.ejb package.

You can annotate the life cycle callback methods either in a bean class or in an external class. These methods can be annotated by more than one annotation. To specify multiple interceptor classes to execute in a given life cycle callback event, you need to associate multiple interceptor classes within the same bean file. The interceptor methods are executed in the order they are listed in the @Interceptor annotation. The signatures of the annotated methods depend on the associated method, as described in the following cases:

- If the life cycle method is defined within a bean class, then the method signature is as shown in the following code snippet:

```
void <METHOD>()
```

- If the life cycle method is defined in the interceptor class, then the method signature is as shown in the following code snippet:

```
void <METHOD>(InvocationContext)
```

The following code snippet shows the use of the @AroundInvoke annotation along with the InvocationContext interface by using both, the business methods and the life cycle callback methods:

```
package examples;
import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;
import javax.ejb.PostActivate;
import javax.ejb.PrePassivate;
public class ExampleInterceptor
{
   public ExampleInterceptor() {}
   @AroundInvoke
   public Object example(InvocationContext ctx) throws Exception
   {
         System.out.println("Invoking method: "+ ctx.getMethod());
         return ctx.proceed();
   }
   @PostActivate
   public void postActivate(InvocationContext ctx)
   {
         System.out.println("Invoking method: "+ ctx.getMethod());
   }
   @PrePassivate
   public void prePassivate(InvocationContext ctx)
   {
         System.out.println("Invoking method: "+ ctx.getMethod());
   }
}
```

The preceding code snippet initially imports all the metadata annotations used in the ExampleInterceptor class. The interceptor class named ExampleInterceptor is the plain Java class and is followed by a no argument constructor, ExampleInterceptor(). The method-level @AroundInvoke annotation specifies the interceptor method. In the interceptor class, this annotation can be used only once. The @PreActivate annotation defines the methods that are called by the EJB container after reactivating the bean instance; whereas, after passivation, the EJB container should invoke the methods defined by the @PrePassivate annotation. The life cycle callback interceptor methods can only be applied to the stateful session beans.

Specifying Default Interceptor Methods

The business and life cycle callback methods are specified at the method level by using default interceptors. The default interceptor methods can be applied to all the components of a particular EJB JAR file. Therefore, these methods can be configured in the ejb-jar.xml Deployment Descriptor file and not with the metadata annotation. The default interceptors are invoked by the EJB container before all other defined interceptors. To prevent invocation of the default interceptor in a particular EJB, you need to specify the class level by using the @javax.interceptor.ExcludeDefaultInterceptor annotation in the bean class file. The default interceptor can be also specified by using the <interceptor-binding> as parent and <assembly-descriptor> as child element in the ejb-jar.xml file. Then, by setting the <ejb-name> child element to *, which indicates that the corresponding interceptor class is applicable to all EJBs; and the <interceptor-class> child element is applicable to the name of the interceptor class.

The following code snippet shows how to map the default interceptor to the org.mycompany.DefaultIC class:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.0"
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/ejb-jar_3_0.xsd">
...
<assembly-descriptor>
...
<Interceptor-binding>
<ejb-name>*</ejb-name>
<interceptor-class>org.mycompany.DefaultIC</interceptor-class>
</interceptors>
</assembly-descriptor>
</ejb-jar>
```

Figure 13.29 shows the order in which business method `interceptors` are invoked:

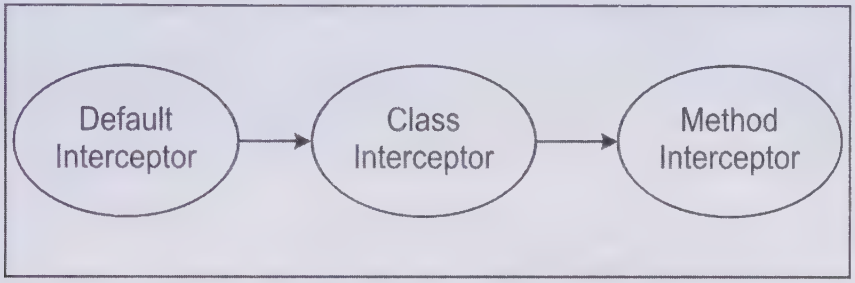


Figure 13.29: Displaying the Order for Invoking Business Method Interceptors

The `interceptors` are invoked in the order they are represented in the bean file. Default `interceptors` are invoked first, then the class-level `interceptors`, and in the end, the method-level `interceptors` are invoked. The default `interceptor`, shown in Figure 13.29, is applicable to all the methods of all EJBs in the `ejb-jar` package; the class-level `interceptors` are applicable to all the methods specified within that particular class; and the method-level `interceptors` are applicable to only one method in a particular class.

Exploring the Life Cycle Callback Methods in an Interceptor Class

The `annotations` named `@PreDestroy`, `@PostConstruct`, `@PostActivate`, and `@PrePassivate` are used to receive life cycle callback methods. These `annotations` work in the same way as the life cycle methods of `interceptors`. The life cycle callbacks defined in an `interceptor` is known as life cycle callback `interceptors` or life cycle callback listeners. The life cycle callback methods can be applied to both session and MDBs. Generally, four `annotations` are used along with the beans. These `annotations` are as follows:

- @PreDestroy
- @PostConstruct
- @PostActivate
- @PrePassivate

Let's discuss these annotations, one by one.

The @PreDestroy Annotation

The @PreDestroy `annotation` is specified by the `javax.annotation.PreDestroy` class. This `annotation` specifies the life cycle callback method which notifies the bean class that the bean instance is going to be destroyed by the EJB container. `Annotation` is specifically applied to the methods that release resources held by the bean class. Only a single method can be annotated by using the `@PreDestroy annotation`. This `annotation` can be applied to both session and MDBs. The method on which the `@PreDestroy annotation` is to be applied must match the following criteria:

- The return type of the method must be void
- The method must not throw a checked exception
- The method must be public, protected, or private

- The method must not be static and final
- The method may or may not contain any attribute

The @PostConstruct Annotation

The @PostConstruct annotation is specified by the javax.annotation.PostConstruct class. It specifies the life cycle callback method that the EJB container should execute before invoking the business method and after the DI is implemented to perform any initialization. This annotation can be applied to any bean class that includes the DI. The @PostConstruct annotation can be applied to only one method in a bean class. It can be applied to both session and MDBs. The method on which the @PostConstruct annotation is to be applied should match the criteria as defined for the @PreDestroy annotation.

The @PostActivate Annotation

The @PostActive annotation is specified by the javax.ejb.PostActivate class. This annotation is used to notify the life cycle callback method that the EJB container has been activated in the bean instance. This annotation does not have any attribute and can only be applied to stateful beans because the state of a stateful bean instance is maintained even when the bean is in the Passivate state. Only a single method of the bean class can be annotated by using the @PostActivate annotation. If more than one method is annotated by using this annotation, the EJB will not deploy. The method annotated with the @PostActivate annotation must match the following criteria:

- The return type of the method must be void
- The method must not throw a checked exception
- The method must be public, private, or protected
- The method must not be static and final

The @PrePassivate Annotation

The @PrePassivate annotation is specified by the javax.ejb.PrePassivate class. The @PrePassivate annotation does not contain any attribute. This annotation is used to notify the life cycle callback method that the EJB container is about to passivate the bean instance. It is also applicable to the stateful beans because the EJB container automatically maintains its state of the bean instance. It is not necessary to specify this annotation for the stateful bean instance; however, it can be specified only along with the @PostActivate annotation. If more than one method is annotated by using this annotation, then the EJB will not deploy. The method on which the @PrePassivate annotation is to be applied should match the criteria as defined for the @PostActivate annotation.

The life cycle callback methods are mainly applied to the session and MDBs. The @PreDestroy, @PostConstruct, @PostActivate, and @PrePassivate annotations are used for the stateful session bean. The @PreDestroy and @PostConstruct annotations are used for MDBs, and the @PrePassivate and @PostActivate annotations cannot be applied to MDBs as the activate and passivate states are not available for the life cycle of the MDB. The following section shows the use of life cycle callback methods in MDBs.

Exploring the Life Cycle Callback Interceptor Methods in an MDB

You can use the life cycle callback methods in the interceptor class by using annotations life cycle. To configure life cycle callback methods in an MDB, you need to perform the following tasks:

- Create an interceptor class that can be any POJO class. The methods of the interceptor class are invoked according to the response to business method invocation and the life cycle events on the bean. The following code snippet shows how to specify the AroundInvoke interceptor and the life cycle callback interceptors in a bean class for an MDB:

```
public class ExampleInterceptor
{
   ...
   @AroundInvoke
```

```
    protected Object exampleInterceptor(InvocationContext invctx)
throws Exception
    {
          Principal p = invctx.getEJBContext().getCallerPrincipal();
          if (!userIsValid(p))
          {
                throw new SecurityException( "Caller: '" + p.getName() +
"' does not have permissions for method " + invctx.getMethod()
                );
          }
          return invctx.proceed();
    }
    @PreDestroy
    public void examplePreDestroyMethod (InvocationContext ctx)
    {
          ...
          invctx.proceed();
          ...
    }
}
```

In the preceding code snippet, the Principal object is used to authenticate the user. If the authentication fails, it throws a SecurityException exception.

- Associate the interceptor class with an MDB by using the `@Interceptor` annotation. The association of the ExampleInterceptor interceptor class with an MDB is shown in the following code snippet:

```
@MessageDriven
@Interceptors(ExampleInterceptor.class)
public class MessageLogger implements MessageListener
{
  @Resource javax.ejb.MessageDrivenContext mc;
  public void onMessage(Message message) {
        ....
  }
  @PostConstruct
  public void initialize()
  {
        items = new ArrayList();
  }
  ...
}
```

In the preceding code snippet, when the `onMessage()` method is called, the methods annotated with the `@AroundInvoke` annotation and the methods of the `ExampleInterceptor` interceptor class are automatically invoked. The life cycle callback interceptor methods are invoked according to the occurrence of appropriate life cycle events.

While creating your interceptor class for an MDB, you must consider the following points:

- The interceptor class that is created must have a public constructor with no argument.
- The life cycle callback `interceptor` method must be implemented in the `bean` class. The callback methods defined in the bean class have the following signature:

```
Object <METHOD> (InvocationContext)
```

- A life cycle event is to be associated with the callback `interceptor` method. The life cycle callback event can be associated with one callback `interceptor`; however, a life cycle callback `interceptor` method can be used to associate multiple callback events.

The `interceptor` class methods can be specified as the life cycle callback methods by using the `@PostConstruct` and `@PreDestroy` annotations. The following code snippet shows the use of these annotations in a bean class:

```
public class ExampleInterceptor
{
  ...
  public void InterceptorMethod (InvocationContext invctx) {
        ...
```

```
        invctx.proceed ();
        ...
    }
    @PostConstruct
    public void PostContructMethod (InvocationContext invctx)
    {
        ...
        invctx.proceed();
        ...
    }
    @PreDestroy
    public void PreDestroyMethod (InvocationContext invctx)
    {
        ...
        invctx.proceed();
        ...
    }
}
```

The preceding code snippet shows the use of the `@PostConstruct` and `@PreDestroy` annotations in a bean class. The `postConstructMethod()` and `preDestroyMethod ()` methods are called whenever an appropriate event occurs.

Exploring the Life Cycle Callback Interceptor Methods in a Session Bean

An `interceptor` method can be associated with an `interceptor` class as a life cycle callback `interceptor` method. To configure the life cycle callback `interceptor` methods in an `interceptor` class, you must perform the following steps:

- Create an `interceptor` class that can be any POJO class. The methods of the `interceptor` class are invoked according to the response to business method invocation and the life cycle events on the bean class. The `@AroundInvoke` annotation and the `exampleInterceptor() method` is invoked each time a business method is called. The following code snippet shows how to specify an `AroundInvoke interceptor` and the life cycle callback `interceptors` in a bean class for a session bean:

```
public class ExampleInterceptor
{
  ...
  @AroundInvoke
protected Object exampleInterceptor(InvocationContext invctx)
throws Exception {
        Principal p = invctx.getEJBContext().getCallerPrincipal;
        if (!userIsValid(p))
        {
                throw new SecurityException( "Caller: '" + p.getName() +
"' does not have permissions for method " + invctx.getMethod()
                );
        }
        return invctx.proceed();
  }
  @PreDestroy
  public void myPreDestroyMethod (InvocationContext invctx)
  {
        ...
        invctx.proceed();
        ...
  }

}
```

- Associate an `interceptor` class with a session bean by using the `@Interceptors` annotation. The association of an `interceptor` with a Stateful session bean is shown in the following code snippet:

```
@Stateful
@Interceptors(ExampleInterceptor.class)
public class CartBean implements Cart
```

```
{
   private ArrayList items;
   @PostConstruct
   public void initialize() {
         items = new ArrayList();
   }
   ...

}
```

While creating an interceptor class for a session bean, you must consider the following points:

The interceptor class must have a public constructor without any argument.

- Implement the life cycle callback method in the bean class. The callback method defined in the bean class has the following signature:

```
Object <METHOD> (InvocationContext)
```

The life cycle callback event should be associated with the callback interceptor method. The life cycle event can only be associated with one callback interceptor method; however, a life cycle callback interceptor method may be used to associate with multiple callback events.

You can specify the life cycle interceptor method in a bean class as an EJB 3 session bean life cycle method by using any of the following annotations:

- @PostConstruct
- @PreDestroy
- @PrePassivate
- @PostActivate

The following code snippet shows the use of annotations for a stateful session bean:

```
public class ExampleStatefulSessionBeanInterceptor {
   ...
   protected void exampleInterceptorMethod (InvocationContext invctx)    {
         ...
         invctx.proceed();
         ...
   }
   @PostConstruct
   @PostActivate
protected void examplePostConstructInterceptorMethod
   (InvocationContext invctx) {
         ...
         invctx.proceed();
         ...
   }

   @PrePassivate
protected void examplePrePassivateInterceptorMethod
   (InvocationContext invctx) {
         ...
         invctx.proceed();
         ...
   }
```

The preceding code snippet shows an interceptor class for a stateful session bean. It provides the implementation of the examplePrePassivateInterceptorMethod method as the life cycle callback interceptor method by using the @PrePassivate annotation. It also implements examplePostConstructInterceptorMethod as the life cycle callback method by using the @PostActivate and @PostConstruct annotations.

Summary

The chapter introduced EJB that can be used in distributed, highly secure, and scalable enterprise applications. All characteristics and new features introduced by EJB 3 have been discussed in the chapter. The chapter further explained different types of EJBs, the functionality provided by these EJBs, and their corresponding life cycles.

You also learned to implement the different types of enterprise beans with the help of enterprise applications. Next, the chapter discussed the new concepts introduced in EJB 3, such as Interceptors and Timer service.

The next chapter discusses the entity beans and the role of JPA in creating entity beans.

Quick Revise

Q1. What is the use of EJB component?

Ans. The EJB component helps in easy and fast development of distributed, transactional, secure, and portable Java EE applications.

Q2. What is JPA?

Ans. JPA is a framework introduced with the release of EJB 3.0 and simplifies the programming model for entity persistence. It implies that the JPA provides an interface to store the data for persistent use with the help of enterprise beans of an enterprise application.

Q3. What is a session bean? How many types of session beans are there?

Ans. A session bean is a reusable component that consists of business logic of an application. Session beans are of two types:

- The stateless session bean
- The stateful session bean

Q4. How is stateful session bean different from stateless session bean?

Ans. A stateful session bean maintains a client session during the lifetime of an application; whereas, a stateless session bean does not save the client's state.

Q5. Describe the activation and passivation states of a stateful session bean.

Ans. When a bean instance is not used by a client, the EJB container removes the instance from memory and stores it in the secondary storage so that the memory can be reused. This process is known as passivation of stateful session bean. Now, when the client invoked the bean instance again, the EJB container uses the passivated bean instance from secondary storage. Further, the EJB container stores the stateful bean instance in memory to serve the client request. This process is called the activation of a bean instance.

Q6. Define an MDB.

Ans. An MDB is an enterprise bean that allows Java EE applications to process messages asynchronously. It acts as a JMS message listener that receives messages sent by any Java EE component or an application client.

Q7. List different types of transactional models.

Ans. The different types of transactional models are as follows:

- Flat transactions
- Nested transactions
- Chained transactions
- Saga transactions

Q8. Which two forms of Timer objects are supported by EJB 3?

Ans. EJB 3 supports the following two forms of Timer objects:

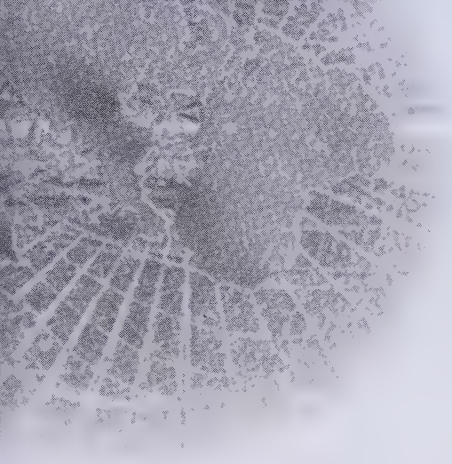
- Single Action Timer
- Interval Timer

Q9. What are interceptors in EJB 3?

Ans. Interceptors are methods that are automatically invoked when an EJB method is invoked. Interceptors provide a logging feature to all EJBs in an application.

Q10. Discuss the role of an EJB container.

Ans. The EJB container provides an environment in which one or more enterprise beans can run. This environment is a combination of available interfaces and classes that the container uses to support enterprise beans throughout their life cycle.



14

Implementing Entities and Java Persistence API 2.0

Java Persistence API (JPA) is the combination of several classes and interfaces used to store the data of the applications persistently in a relational database. JPA also allows you to retrieve the data from a database. The data to be stored or retrieved is managed with the help of the entity classes of an application. An entity class is a simple Java class used to set and retrieve the properties of a bean. To store the data of an entity in a database, you need to create the Java object of the entity class and map the object to the table. This process of mapping the Java objects to the database tables is known as Object Relational Mapping (ORM). To map the Java objects, various details, such as name of the table, row, and column need to be specified in an enterprise application in the form of configuration metadata. This data allows an application to determine the particular rows and columns where the data held by the objects needs to be stored. Configuration metadata refers to the mapping details related to ORM provided in the Deployment Descriptor of an application.

For example, in an online library application, the details of books, such as ISBN number, book name, author name, and name of the publisher are maintained in a database. To store all these details, you need to create an entity class, say Book in our case. The fields of the Book entity class are mapped with the rows and columns of the table in which the data of the books need to be stored. To create the Book entity class and map its details with the table, you need to use the classes and interfaces provided by JPA. The details of mapping the Java object of the Book entity class to the table is provided in the form of configuration metadata in the application.

Initially, Java Database Connectivity (JDBC) was used to store the data of an application in a database. In this process, developers used to write a large amount of code and SQL queries to manage various database operations, such as inserting, updating, and deleting records. The introduction of ORM eliminated these problems by providing a persistent provider that is used to generate the optimized code required for performing database operations. In Enterprise JavaBeans (EJB) 3, JPA was introduced as the ORM mechanism in which the entity beans are mapped to rows and columns of a table to store the data persistently in a database. EJB 3 has facilitated the mapping of an entity by using annotations, such as @Table, @Column, @Enumerated, @Lob, @Temporal, and @Embeddable.

The chapter begins with the discussion of Java persistence and EntityManager. Next, you learn about entity beans and how they are different from session beans. You also learn how entity fields are mapped to the database columns using JPA. In addition, the chapter explores about various relationships that can exist between two entities, such as one-to-one, one-to-many, and many-to-many. The chapter further discusses about Java Persistence Query Language (JPQL). Finally, the Customer application is developed to demonstrate the use of JPQL to query the data from a database.

Understanding Java Persistence and EntityManager API

JPA provides Plain Old Java Objects (POJO) standard and ORM for maintaining persistent data among the applications. The term persistent refers to the data of an application that needs to be stored permanently in a database. In JPA, the persistent data refers to the entities that serve as a logical collection of the data that needs to be stored or retrieved. For example, in an online book purchasing application, the details of the book to be purchased and the person buying the book need to be maintained in the database. Therefore, in this example, book and person are treated as entities. The personal details of person, such as name, credit card details, and address are logically grouped together to represent the Person entity. Similarly, International Standard Book Number (ISBN), book name, and author name may be grouped together for representing the Book entity. In an enterprise application, you can define an entity by creating a POJO by using annotations.

An entity includes the persistence, transaction ability, and identity as its unique properties. The persistence property refers to storing and retrieving data held by entities in the relational databases. The identity property specifies the unique identification of one entity from other entities. For example, the entity named Person will be uniquely identified by the PersonID. It can also be said that the PersonID will be the primary key of the Person table in a database. Therefore, each person would be assigned a unique ID, whose value cannot be null and would not be similar to the value assigned to other person. In this example, the PersonID serves as the identity for the Person entity or the primary key for the Person table in a database.

After having an overview about the entities in JPA, let's now discuss the EntityManager API that is used to implement the persistence property in the EJB 3 applications. API stands for Application Programming Interface.

The EntityManager API is an interface used to access entities in an application's persistent context. As discussed earlier, all the entities available in the persistent context must be unique and can be identified by the primary key associated with a table. The EntityManager API manages the life cycle of the entities and serves as a connecting bridge between the object-oriented data and relational database, as shown in Figure 14.1. The EntityManager translates an entity into a new database record. When the client requests to update the entity, the EntityManager looks at the relational database corresponding to the entity and updates it. Similarly, when the request is to delete an entity, the EntityManager deletes the entity from the persistent context. When the request is to save an entity in the persistent context, the EntityManager creates an entity object, maps the object to the relational database, and returns the entity object to the application.

Figure 14.1 shows the communication between the object and the relational database with the help of the EntityManager:

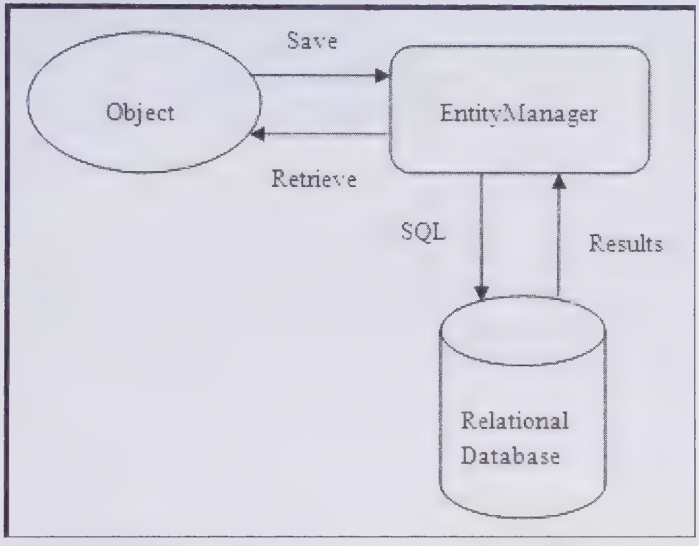


Figure 14.1: Showing the Access Pattern in the EntityManager API

The EntityManager provides the following two types of interfaces:

- **Container-managed EntityManager**—Specifies the EntityManager for an application, which is provided by the Java EE runtime container. The EntityManager is defined by using the @PersistenceContext annotation and the lookup() method is used to obtain the EntityManager from the persistence context.
- **Application-managed EntityManager**—Creates the EntityManager by using the code provided in the application.

Let's now discuss about entities that are used to store persistent data.

Introducing Entities

Entities are POJOs that are used to store data in a database. Entities hold the data of an application in the form of fields and the methods associated with the details of an entity. All the information regarding an entity is always available in ORM. Entities support relational and object-oriented capabilities, such as entities, inheritance, and polymorphism. A database-driven application is created to collect data from the entities, which are stored in a database. To create a database-driven application, the developer needs to manage entities within the application, as the data is maintained by these entities.

Entity beans are designed to bridge the gap between the object and the relational representation of the application data. Figure 14.2 shows how a client can manipulate the data with the help of entity beans. The client can be the Java application or the Web browser, which does not communicate directly with the relational database. As shown in Figure 14.2, the Web browser accesses the JavaServer Pages (JSP) page available in the

Web container. Then, the JSP page calls the appropriate entity bean and Java objects of the entity bean class that holds the data provided by the client in the JSP page. These Java objects, with the help of the EJB container, store the data in the mapped tables of a relational database.

Figure 14.2 shows how entity beans communicate with each other, according to the client's request:

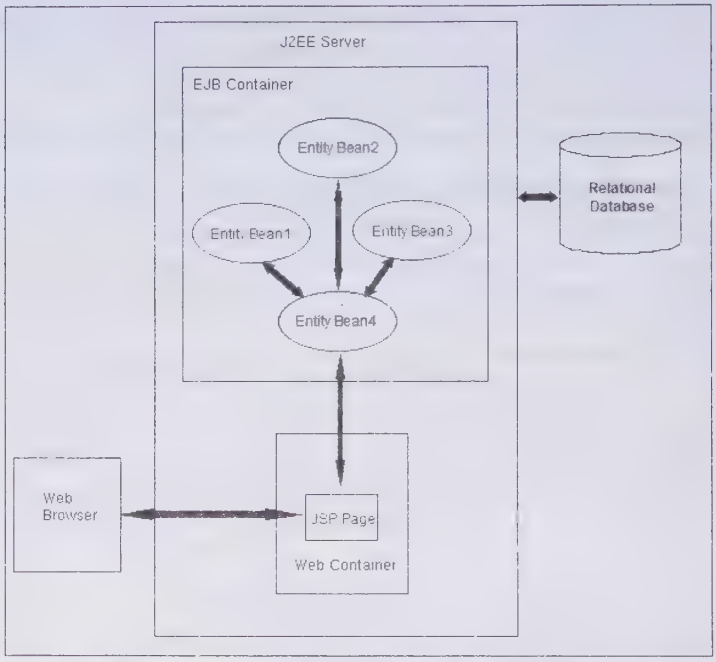


Figure 14.2: Showing the Entities Communication Based on Client's Request

In EJB 3, entity beans are POJOs that are defined by using annotations. These beans are also used to specify how Java objects are stored in the database for persistence usage. The EJB 3 container automatically maps these Java objects to the relational database tables; therefore, the developers need not worry about the details of the data in the relational database.

EJB 3 has made programming simpler for developers as the container handles the data held by entity beans. In EJB 2.x, there were two types of entity beans—container-managed and bean-managed. In the bean-managed entity beans, the code needs to be written by the developer to make the database calls to access the data from a database. In case of the container-managed entity beans, the required database calls are automatically invoked by the EJB container.

In EJB 3.0, the JPA has updated and renewed the concept of the entity beans. To understand how entity beans are created in EJB 3.0, let's first learn to introduce a plain Java class and then convert it into an entity with the help of annotations. The advanced features, such as inheritance, polymorphism, and complex relations, which were not available in EJB 2.x, are supported in JPA.

The code for the plain Java class, Customer.java that contains the getter and setter properties to retrieve and set the data, respectively is shown in Listing 14.1:

Listing 14.1: Showing the Code of the Customer.java Files

```
package com.example.entity;

import java.util.List;
import javax.persistence.*;

public class Customer implements java.io.Serializable {
```

```
private Integer customerId;
private String firstName;
private String lastName;
private String company;
private String address1;
private String address2;
private String city;
private String state;
private String zip;
private String emailAddress;
private String phoneNumber;

public Customer() {
      firstName = "";
      lastName = "";
      company = "";
      address1 = "";
      address2 = "";
      city = "";
      state = "";
      zip = "";
      emailAddress = "";
      phoneNumber = "";
}

public Integer getCustomerId() {
      return customerId;
}

public void setCustomerId(Integer customerId) {
      this.customerId = customerId;
}

public String getFirstName() {
      return firstName;
}

public void setFirstName(String firstName) {
      this.firstName = firstName;
}

public String getLastName() {
      return lastName;
}
public void setLastName(String lastName) {
      this.lastName = lastName;
}

public String getCompany() {
      return company;
}

public void setCompany(String company) {
      this.company = company;
}

public String getAddress1() {
      return address1;
}

public void setAddress1(String address1) {
      this.address1 = address1;
}

public String getAddress2() {
```

```
        return address2;
    }

    public void setAddress2(String address2) {
        this.address2 = address2;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
    }

    public String getZip() {
        return zip;
    }

    public void setZip(String zip) {
        this.zip = zip;
    }

    public String getEmailAddress() {
        return emailAddress;
    }

    public void setEmailAddress(String emailAddress) {
        this.emailAddress = emailAddress;
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }

    public void setPhoneNumber(String phoneNumber) {
        this.phoneNumber = phoneNumber;
    }

}
```

Listing 14.1 defines the Customer class as the Plain Old Java class having 11 variables, namely customerID, firstName, lastName, company, address1, address2, city, state, zip, emailAddress, and phoneNumber. The getter and setter properties are set for all these variables. The getter property returns the value of the variable and the setter property sets the data for the variable. To make the Plain Old Java Class an entity, annotations, such as `@Entity`, `@Table`, `@Column`, and `@ID` are used. Apart from these annotations, `JPQL` is also used to query the database. Moreover, the instance of the `EntityManager` is created to manage the entity.

NOTE

The detailed study of JPQL and EntityManager is provided later in this chapter.

Listing 14.2 provides the code of the Customer entity class containing the required annotations (you can find the Customer.java file on CD in the `code\JavaEE\Chapter14\Customer\Customer-ejb\src\com\kogent\entity` folder):

Listing 14.2: Showing the Code of the `Customer.java` File

```
package com.kogent.entity;

import java.util.List;
import javax.persistence.*;
@Entity
public class Customer implements java.io.Serializable {

   private Integer customerId;
   private String firstName;
   private String lastName;
   private String company;
   private String address1;
   private String address2;
   private String city;
   private String state;
   private String zip;
   private String emailAddress;
   private String phoneNumber;

   public Customer() {
         firstName = "";
         lastName = "";
         company = "";
         address1 = "";
         address2 = "";
         city = "";
         state = "";
         zip = "";
         emailAddress = "";
         phoneNumber = "";
   }

   @Id
   @Column(name="custId", insertable=false, updatable=false)
   public Integer getCustomerId() {
         return customerId;
   }
   public void setCustomerId(Integer customerId) {
         this.customerId = customerId;
   }
   public String getFirstName() {
         return firstName;
   }

   public void setFirstName(String firstName) {
         this.firstName = firstName;
   }
   public String getLastName() {
         return lastName;
   }
   public void setLastName(String lastName) {
         this.lastName = lastName;
   }
   public String getCompany() {
         return company;
   }
   public void setCompany(String company) {
         this.company = company;
   }
   @Column(name="address_1")
   public String getAddress1() {
         return address1;
   }
   public void setAddress1(String address1) {
         this.address1 = address1;
```

```
    }
    @Column(name="address_2")
    public String getAddress2() {
          return address2;
    }
    public void setAddress2(String address2) {
          this.address2 = address2;
    }
    public String getCity() {
          return city;
    }
    public void setCity(String city) {
          this.city = city;
    }

    public String getState() {
          return state;
    }
    public void setState(String state) {
          this.state = state;
    }
    public String getZip() {
          return zip;
    }
    public void setZip(String zip) {
          this.zip = zip;
    }
    public String getEmailAddress() {
          return emailAddress;
    }
    public void setEmailAddress(String emailAddress) {
          this.emailAddress = emailAddress;
    }
    public String getPhoneNumber() {
          return phoneNumber;
    }
    public void setPhoneNumber(String phoneNumber) {
          this.phoneNumber = phoneNumber;
    }

    public static List<Customer> findAllCustomers(EntityManager em) {
          Query query = em.createQuery(
                  "SELECT OBJECT(cust) FROM Customer cust");
          List<Customer> customers = query.getResultList();
    return customers;
    }

    public static Customer findCustomerById(EntityManager em, int custId) {

    Query query = em.createQuery(
          "SELECT OBJECT(cust) FROM Customer cust "
                  + "WHERE cust.customerId = :custId");
          System.out.println("customerid in customer"+custId);
          query.setParameter("custId",custId);
          Customer customer = (Customer)query.getSingleResult();
          return customer;
    }

}
```

In Listing 14.2, the Customer entity class is defined in the `com.kogent.entity` package.

Let's now discuss about various annotations, such as `@ENTITY`, `@ID`, and `@Column` that can be used in an entity class.

Specifying the @ENTITY Annotation

The use of the `@Entity` annotation before the declaration of a class specifies that the Java class is an entity bean. The `Customer` entity (shown in Listing 14.2) has various fields, such as `customerID`, `firstName`, `lastName`, and `company` to represent the details of each customer. The following code snippet shows the use of the `@Entity` annotation before the declaration of the `Customer` entity bean class:

```
@Entity
public class Customer implements java.io.Serializable {

   private Integer customerId;
   private String firstName;
   private String lastName;
   private String company;
   private String address1;
   private String address2;
   private String city;
   private String state;
   private String zip;
   private String emailAddress;
   private String phoneNumber;
...
...
}
```

Specifying the @Table Annotation

The `@Table` annotation is used to provide the name of the table containing the columns to which an entity is to be mapped. By default, all the persistent data for the entity is mapped to the table with the help of the `name` parameter of the `@Table` annotation. The use of the `@Table` annotation is optional. If it is omitted, then the entity is mapped to the default schema with the same name as the entity class. If you do not provide the name of the table as the value of the `name` parameter, the name of the entity would be taken as the table name. The persistent provider provides a feature of automatic schema generation and creates database objects for entities when the objects do not exist in the database. The following code snippet shows the use of the `@Table` annotation to map an entity:

```
@Entity
@Table(name="Customer", schema=" ")
public class Customer
```

The preceding code snippet shows the use of the `@Table` annotation. The `name` parameter specifies the name of the table (in our case, Customer) and the `schema` parameter denotes the schema associated with the database. If you do not provide a value for the schema parameter, the automatic schema generation process is used to create objects for the entities in the database. The `@Table` annotation also contains some other parameters, such as catalog and uniqueConstraints, as shown in the following code snippet:

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface Table {
  String name() default "";
  String catalog() default "";
  String schema() default "";
  UniqueConstraint[] uniqueConstraints() default {};
}
```

Now, let's discuss how the @Columns annotation is used in an entity class.

Specifying the @Column Annotation

The `@Column` annotation is used to map a persisted field to a table column and provides various parameters, such as `name`, `unique`, `nullable`, `insertable`, `updatable`, and `columnDefinition`. The following code snippet shows the type of value accepted by the parameters of the `@Column` annotation:

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @Column {
```

```
    String name() default "";
    boolean unique() default false;
    boolean nullable() default true;
    boolean insertable() default true;
    boolean updatable() default true;
    String columnDefinition() default "";
    String table() default "";
    int length() default 255;
    int precision() default 0;
    int scale() default 0;
}
```

In the preceding code snippet:

- The `name` parameter—Defines the column name, which is optional. If the column name is not specified in the `name` parameter, then by default, the `name` parameter is set to the property name.
- The `nullable` parameter—Defines whether the column supports null values.
- The `unique` parameter—Defines whether or not the column supports any unique constraints.
- The `insertable` and `updatable` parameters—Control the behavior of the persistent provider. If the `insertable` parameter is set to false, then the specified field or property is not included in the insert statement. If the `updatable` parameter is set to false, then the field or property of an entity cannot be updated in the database. These properties are useful for the read-only data, such as the generation of the primary key.
- The `length` parameter—Specifies the size of the database column.
- The `precision` parameter—Specifies the precision of the decimal field.
- The `scale` parameter—Specifies the scale of the decimal column.

The following code snippet shows the use of the `@Column` annotation in an entity class:

```
@Column(name="custId", insertable=false, updatable=false)
public Integer getCustomerId()
{
  return custId;
}
```

In the preceding code snippet, the column named `custId` does not allow the inserting and updating of the data in the database, as the value of the `insertable` and `updatable` parameters are set to false. This also indicates that the `custId` column is the primary key in the `Customer` table.

Specifying the @Enumerated Annotation

The use of the `@Enumerated` annotation indicates that the field's persistent property should be stored in the form of an Enumeration. The `@Enumerated` annotation can also be used in conjunction with the `@Basic` annotation. The possible values of EnumType can be `ORDINAL` or `STRING`.

The following code snippet shows the use of the `@Enumerated` annotation with the ORDINAL EnumType:

```
@Enumerated(EnumType.ORDINAL)
...
protected firstName fName;
```

The preceding code snippet shows that if the value of the field is set to `firstName.fName`, then the ORDINAL value of `fName` is stored in the database. You can also specify the enumeration value as strings, as shown in the following code snippet:

```
@Enumerated(EnumType.STRING)
...
protected firstName fName;
```

If the value of the field provided in the preceding code snippet is set to `firstName.ADMIN`, then the string value, ADMIN is saved in the database. By default, the ORDINAL EnumType is used for the @Enumerated annotation.

Specifying the @Lob Annotation

One of the important features of relational database is its ability to store large objects. These objects are categorized in either Binary Large Object (BLOB) or Character Large Object (CLOB). The categorization depends on the type of available data. These two types correspond to JDBC `java.sql.Blob` and `java.sql.Clob` objects. If the data is of the `char []` or string type, then the persistent provider maps the data to a `CLOB` column; otherwise, the data is mapped to the `BLOB` column. This annotation can be used within the `@Basic` annotation, which is an auxiliary annotation. If the application is accessing the @Lob object for the first time, the `@Basic` annotation causes the `@Lob` object to be loaded from the database.

Specifying the @Temporal Annotation

The @Temporal annotation is used to specify the persistent property or field as a temporal type. You should note that the relational databases support temporal data types. The DATE, TIME, and TIMESTAMP types can be used to map a temporal type. The `java.util.Date` and `java.util.Calender` packages are used to access the temporal data types. The following code snippet shows the mapping of temporal types to the database:

```
@Temporal(TemporalType.DATE)
protected Date creationDate;
```

The preceding code snippet defines the `creationDate` variable as the DATE type, which is specified as the temporal type for the `@Temporal` annotation. If no value is specified for the `TemporalType` parameter of the `@Temporal` annotation, the TIMESTAMP is selected as the default value.

Now after understanding how to create an entity using various annotations, let's discuss how entity beans are different from the session beans.

Exploring Entity and Session Beans

Enterprise Java beans are classified into different categories, such as session beans and entity beans. Session beans are further divided into two types—the stateless session bean and the stateful session bean. As their names suggest, stateful session beans maintain their state across multiple client requests along with the bean instance; whereas, stateless session beans do not maintain their state across multiple client requests. Session beans are used to model business processes, whereas entity beans are used to model business data. The session beans are used to perform various actions, such as accessing the database through its instance or calling the legacy system. However, the Entity beans are the Java objects used to access the database. Session beans along with entity beans are used to accomplish business transactions.

The session bean exists till the client or user runs the application; whereas, the entity bean exists even after the client closes the application, as the data of the entity beans is stored in the database. In other words, the session bean exists till the client or the user exists. On the other hand, entity beans persist till the data is in the database. Entity beans have a persistent state; whereas, session beans have conversational state. In each entity bean, there is a unique object identifier known as a primary key that helps a client to locate a particular entity bean. However, in session beans the fields identifying the client's identity allows a client to create its conversational state with the bean. One instance of session bean is for a single client; whereas, in case of entity beans, a single instance is shared by multiple clients.

Now, let's discuss the conditions where entity beans can be used.

Describing When To Use Entity Beans

Entity beans are used under the following conditions:

- When the bean's state needs to be kept persistent. Even when a user exits an application, the entity bean's state is stored persistently in a database.
- When a bean represents a business entity. Consider an example where a customer makes some purchases online using his credit card. In this case, credit card is an entity containing unique details, such as credit card number, name, and the billing address of the customer. While shopping online, a customer needs to log in with an id and a password. After the customer had logged in, a session is being maintained till the customer logs out or the session expires. During the session, the customer adds the required items in the shopping cart. Next, the customer pays the total amount of the items added in the shopping cart with the

help of the credit card. After making the payment, the customer logs out and the session of online shopping expires. The details related to this transaction persist in the database. In other words, the details, such as credit card number used to make payment, number of items purchased, and the billing address of the customer is stored in the database.

In this example, the session of the online shopping by a customer is maintained with the help of the session bean and the shopping details have been saved persistently by using the entity bean. To implement this example in an application, you can create two beans, CreditCardEJB and CreditCardVerifierEJB. The CreditCardEJB bean is used to represent the details of a credit card; therefore, this bean would be an entity bean. The CreditCardVerifierEJB bean is used to validate the credit card number; therefore, this bean would be a session bean.

Describing an Entity Class

An entity class is a simple Java class that contains a metadata annotation or an Extensible Markup Language (XML) Deployment Descriptor. Entity classes have the following features in the persistent API:

- Refers to the Java classes that do not extend any framework classes or interfaces. In addition, the entity classes do not implement the java.io.Serializable interface. If they implement the Serializable interface, then the entity object can be used as a simple data object and can be transferred as an argument in the remote invocation. However, the entity itself does not provide direct remote invocation.
- Maps to a data definition in a relational database. During runtime, the entity instance of the class is mapped to a particular field in a relational database. The Java Persistent API maps entity classes to relational database tables and allows the user to control the mapping by annotations or XML Deployment Descriptors.
- Allows you to declare a primary key in the relational database. This key is useful to differentiate between entities in the relational database. In case of complex relationships, the primary key represents the entire object. JPA provides flexibility to identifiers by including a primary key class with an existing entity. This class must be public, serializable, and should contain a public constructor.
- Allows you to access the persistent state of the entity class by directly accessing its related fields. These fields can be accessed by using the get and set methods similar to JavaBean. The persistent provider determines which methods are to be applied for accessing the fields related with the entity and also determines the use of annotations.
- Provides some business methods. The entity class must provide some callback methods and listeners to apply the business methods. The persistent provider calls these methods to explicitly manage the entities.

After understanding the entity class, let's now discuss the life cycle of an entity.

Describing the Life Cycle of Entity

The life cycle of an entity is managed by the `EntityManager` API, which provides various methods, such as new(), remove(), find(), merge(), and persist() (Figure 14.3). The entity life cycle is based on two main aspects, its relationship to a specific persistence context and the synchronization of its state with the database. The entity life cycle consists of the following four different stages:

- **New**—Refers to the state in which a new instance of an entity is created, which is not associated with any persistent identity or persistent context. Any changes made to the entity at this stage are not synchronized in the database.
- **Managed**—Refers to the state in which an entity is associated with the persistent context. The entity has a persistent identity in the database and is in the managed state after the invocation of the `persist()` method. Changes made to the entity are synchronized in the database after the transactions are committed or the `remove ()` method is called.
- **Detached**—Refers to detaching the association of the entity with persistent context. In this stage, the entity also has its persistent identity.
- **Removed**—Removes the entity data from the database. In this stage, the entity also has an association with the persistent context.

Figure 14.3 demonstrates the entity life cycle with the specified methods:

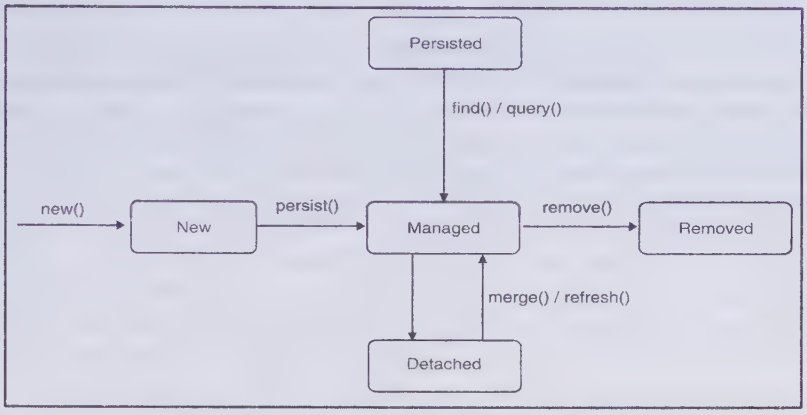


Figure 14.3: Showing the Entity Life Cycle

To destroy the entity data from the database, you need to invoke the `remove()` method. You should note that the invocation of the `remove()` method does not remove the entity from the database; instead, it schedules the entity that is to be removed. The remove operation does not work with new and removed entities. It only works with the managed entities. The `remove()` method cannot be called on the detached entities. Calling the method on a detached entity throws the `IllegalArgumentException` exception. Deletion of a particular entity takes place when transaction is committed or when the `remove()` method is called on the entity. The `merge()` method is used to bring the detached entity back to the persistent context. Entities are detached completely when their persistent context ends. The `merge()` method returns a managed entity.

Entity Listeners and Callbacks

An application reacts with respect to certain events that occur during the persistence mechanism. This process allows implementation of some definite type of generic functionalities and a reference to some built-in functionalities. The EJB 3 specification provides the following two mechanisms for this purpose:

- **Using the `EntityListener` class**—Refers to the mechanism in which you need to define an `EntityListener` class that is used to handle the life cycle events of an entity.
- **Using the callback method**—Refers to the mechanism in which the business method of an entity can be defined as the callback method with the help of the `@Callback` annotation. This callback method receives the notifications about the life cycle events of a particular entity.

Let's now discuss each of these mechanisms in detail.

Entity Listeners

The `EntityListener` class is a stateless bean class containing a non-argument constructor. A class can be made as the EntityListener class by using the `@EntityListener` annotation. The entity listeners are applied to entity classes. You can define several entity listeners for a single entity within different levels of hierarchy. However, for the same event within an entity, you cannot define two listeners. The occurrence of an event leads to the execution of the entity listeners according to inheritance hierarchy. Entity listeners for an entity can be defined by using the following annotations:

- **@EntityListeners**—Specifies the callback listener classes that should be used for an entity.
- **@ExcludeSuperclassListeners**—Specifies the invocation of the superclass listeners that are to be excluded for the entity and the subclass.
- **@ExcludeDefaultListeners**—Specifies the invocation of the default listeners that are to be excluded for the entity class. The listener is to be excluded for both the super and the sub classes.

Now, let's see how callback methods are used to receive the notification of entity life cycle events.

Callbacks

Callbacks are the methods of entities that are used to receive notifications about a specific entity. These methods are represented by the callback annotations. Life cycle callbacks are used for receiving callbacks, validating data, auditing, sending notifications regarding the changes made in a database, and generating data after an entity is loaded. The life cycle callback methods are not invoked by the EJB container; rather, they are invoked by the persistence provider. The Java persistent API specification defines the following life cycle events for an entity:

- **PrePersist** – Refers to the event that is represented by the @PrePersist annotation. It is executed before the execution of the EntityManager's persist operation.
- **PostPersist** – Refers to the event that is represented by the @PostPersist annotation. It is executed after the database persist operation is executed. It is called after the database INSERT operation is executed.
- **PreRemove** – Refers to the event that is represented by the @PreRemove annotation. It is synchronous with the database remove operation and executed before the execution of the EntityManager's remove method.
- **PostRemove** – Refers to the event that is represented by the @PostRemove annotation. It is executed after the execution of the EntityManager's remove method. It is synchronous with the remove operation.
- **PreUpdate** – Refers to the event that is represented by the @PreUpdate annotation. It is executed prior to the database UPDATE operation.
- **PostUpdate** – Refers to the event that is represented by the @PostUpdate annotation. It is executed after the database UPDATE operation.
- **PostLoad** – Refers to the event that is represented by the @PostLoad annotation. It is executed after an entity is loaded into the persistent context or the entity is being refreshed.

Packaging a Persistence Unit

The entities of a Java EE application are packaged in a persistence unit. In other words, the entities are not limited to the EJB modules; instead, they are packaged in a Web module, Enterprise ARchive (EAR) module, or the standard jar file. These entities are packaged in a Deployment Descriptor called `persistence.xml`.

The sample code for the `persistence.xml` is given in Listing 14.3:

Listing 14.3: Showing the Code of the persistence.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0">

  <persistence-unit name="manager1" transaction-type="JTA">
        <provider> </provider>
        <jta-data-source> </jta-data-source>
        <mapping-file> </mapping-file>
        <jar-file> </jar-file>
        <class> </class>
  </persistence-unit>
</persistence>
```

Let's now discuss the elements and attributes specified in the persistence.xml file under Listing 14.3.

The name Attribute

The `name` attribute of the `persistent-unit` element, listed in Listing 14.3, specifies the name of the entity manager.

The transaction-type Attribute

The value for the `transaction-type` attribute can be `Java Transaction API (JTA)` or `RESOURCE_LOCAL`. If the `jta-data-source` element is used, then the transaction type is `JTA`. However, if the `non-jta-data-source` element is used, then the `RESOURCE_LOCAL` value is used.

The provider Element

The `provider` element is used to specify the class name of the EJB persistence provider. If you are not working with the multiple EJB 3 implementations, then the provider may not be specified.

The jta-data-source and non jta-data-source Elements

The `jta-data-source` element is used to specify the Java Naming and Directory Interface (JNDI) name of the JDBC data source. The name of the specified data source is automatically enlisted in JTA transactions and is used in global transaction. In the `non-jta-data-source` element, the JNDI name of JDBC data source is specified that is not enlisted in JTA transactions.

The mapping-file Element

The `class` element specifies an EJB3 compliant XML mapping file to be mapped, say Customer class. The name of the EJB3 Deployment Descriptor is provided in the mapping-file element, defined in the persistent.xml file in Listing 14.3. The other elements of the persistence.xml file are:

- **Jar file** – Defines the `jar-file` element that specifies the name of the jar file required in an application
- **Class** – Defines the `class` element that specifies the class name that has to be mapped to the entity

Now, let's continue with the sample example in which a simple `Customer` entity class is created in Listing 14.2. Create a persistence.`xml` file for the Customer entity class, as shown in Listing 14.4 (you can find this file on CD in the `code\JavaEE\Chapter14\Customer\Customer-ejb\META-INF` folder):

Listing 14.4: Showing the Code of the persistence.xml File for the Customer Entity

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="Customer-ejbPU" transaction-type="JTA">
        <provider>oracle.toplink.essentials.PersistenceProvider</provider>
        <jta-data-source>customer</jta-data-source>
        <class>com.kogent.entity.Customer</class>
        <exclude-unlisted-classes>true</exclude-unlisted-classes>
        <properties>
        <property name="toplink.ddl-generation" value="create-tables"/>
        </properties>
  </persistence-unit>
</persistence>
```

In Listing 14.4, the persistence-unit name is `Customer-ejbPU` and the transaction-type is `JTA`. The `Customer-ejbPU` persistence unit packages all the entities. You have already created a single entity named Customer in Listing 14.2 and the JNDI name for the `Customer-ejbPU` persistence unit is customer, as specified in Listing 14.3. The `persistence.xml` file helps in locating the required entity as it packages the `Customer` entity in the persistence unit named `Customer-ejbPU`. You can save the `persistence.xml` file in the META-INF folder of the application (Figure 14.19).

After the entity bean is created and packaged in a persistence unit, you need to obtain an EntityManager to create an interaction between the client session and the entity instances.

Obtaining an EntityManager

The EntityManager provides the services offered by the EJB 3 persistence framework for the client. The client sessions must have an EntityManager instance for interacting with the entities. The EntityManager helps in querying, updating, removing, and refreshing the entity instances.

The collection of the instances within the transaction context called persistence context is also maintained by the EntityManager. The entity instance can be used either inside or outside of the EJB container. A client can obtain an EntityManager instance with the help of any of the following approaches:

- Using the container injection
- Using the `EntityManagerFactory` interface

- Looking up the `EntityManager` through JNDI

Let's explore each of these approaches in detail.

Using the Container Injection

The session bean uses the container injection to obtain an `EntityManager` instance that is bound to a persistence unit named as `persistence`. The following code snippet shows how to obtain the EntityManager instance, `em` using the container injection:

```
@Stateless
public class Customer {
  @PersistenceContext("Customer-ejbPU")
  private EntityManager em;
  public void createCustomer() {
       final Customer cust = new Customer();
       cust.setName("abc");
       em.persist(cust); }
}
```

In the preceding code snippet, the `em` instance is bound to the persistence unit named, `Customer-ejbPU`. This `EntityManager` is used to persist a new `Customer` instance.

Using the EntityManagerFactory Interface

Sometimes, an application needs to have more control over the life cycle of the EntityManager. In such situations, the client has a better option for obtaining the instance of an EntityManager by using the `EntityManagerFactory` interface. In the following code snippet, the `EntityManager` instance is obtained by using `EntityManagerFactory` factory:

```
public static void main(String[] args) {
  final EntityManagerFactory emf =
  Persistence.createEntityManagerFactory("Customer-ejbPU");
  final EntityManager em = emf.createEntityManager();
  final Customer cust = new Customer();
  cust.setName("abc");
  em.persist(cust);
}
```

In the preceding code snippet, the emf instance of the `EntityManagerFactory` interface is bound to the Customer-ejbPU persistence-unit, which includes the customer entity. Next, the `em` instance of the EntityManager class is created from the emf instance, which is used to manage the data of a new Customer instance, `cust`.

Looking up the `EntityManager` Using JNDI

The `lookup()` method of JNDI is used either by EntityManagerFactory or EntityManager for finding the reference of the entity. The following code snippet shows how the `EntityManager` instance is obtained by using the JNDI:

```
EntityManager em = (EntityManager)ctx.lookup("Customer-ejbPU");
```

In the preceding code snippet, you can see that the em is the instance of the `EntityManager` that is used to look for `Customer-ejbPU` persistence unit.

Interacting with an EntityManager

The process of persisting an entity is similar to the act of injecting an entity in the database. If the persisted entity is not created in the database, then it is automatically created. After the creation of the entity, the properties of the entity are set by using the setXXX methods. Next, the relationship of the entity is created with other Java objects by using EntityManager.

The EntityManager interface is used to interact with the persistence context. The EntityManager API provides the methods to insert and remove entities from the database. To interact with the `Customer-ejbPU` persistence unit using EntityManager, you first need to obtain the instance of the EntityManager interface, as discussed in the previous section, *Obtaining an EntityManager*. Then, the SessionFacade beans are created to interact with the Customer entity using the EntityManager API.

Listing 14.5 provides the code for the CustomerFacade session bean (you can find this file on CD in the code\JavaEE\Chapter14\Customer\Customer-ejb\src\com\kogent\session folder):

Listing 14.5: Showing the Code of the CustomerFacade.java File

```
package com.kogent.session;
import com.kogent.entity.Customer;
import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateless
public class CustomerFacade implements CustomerFacadeLocal {
   @PersistenceContext
   private EntityManager em;

   public void create(Customer customer) {
         em.persist(customer);
   }

   public void edit(Customer customer) {
         em.merge(customer);
   }

   public void remove(Customer customer) {
         em.remove(em.merge(customer));
   }

   public Customer find(Object id) {
         return em.find(com.kogent.entity.Customer.class, id);
   }

   public List<Customer> findAll() {
return em.createQuery("select object(o) from Customer as o").getResultList();
   }
}
```

In Listing 14.5, the persist(), merge(), remove(), and find() methods of the EntityManager interface are used. The find() method of EntityManager interface helps in finding the Customer entity. The find() method accepts the name of an entity class as a parameter and the instance of entity's primary key. The find() method returns null if the entity is not found in the database. In Listing 14.5, the Customer entity's class is located. The create(), edit(), remove(), find(), and findAll() methods are declared in the CustomerFacadeLocal interface.

Listing 14.6 provides the code for the CustomerFacadeLocal interface (you can find this file on CD in the code\JavaEE\Chapter14\Customer\Customer-ejb\src\com\kogent\session folder):

Listing 14.6: Showing the Code of the CustomerFacadeLocal.java File

```
package com.kogent.session;

import com.kogent.entity.Customer;
import java.util.List;
import javax.ejb.Local;

@Local
public interface CustomerFacadeLocal {

   void create(Customer customer);

   void edit(Customer customer);

   void remove(Customer customer);

   Customer find(Object id);
```

```
    List<Customer> findAll();
}
```

In Listing 14.6, the `create()`, `edit()`, `remove()`, `find()`, and `findAll()` methods have been declared. The `CustomerFacade` session bean and the `CustomerFacadeLocal` interface are defined under the `com.kogent.session` package. Listing 14.5 interacts with the `Customer` entity by using the em instance of the EntityManager interface. The em instance invokes the persist() method, as shown in Listing 14.6, to create a new `Customer` instance. The `remove()` method is invoked to remove the `Customer` instance.

After packaging the entity into a persistence unit (persistence.xml) and understanding various ways to obtain the packaged persistence unit, let's now discuss the various types of entity relationships.

Understanding Entity Relationship Types

Entities can have a single reference or collections of references to other entities. A relationship is always implemented with the help of a relationship field on either one or both entities involved in a relationship. An entity relationship can be unidirectional or bidirectional. In contrast to EJB 2.1, the EJB container of EJB 3.0 maintains bidirectional relationships. In a unidirectional relationship between entity A and entity B, updating an instance of entity A leads to updation of some related instance of entity B. However, the same is not entertained by the container in reverse direction, i.e. updating an instance of entity B does not affect any instance of entity A. While using container-managed relationships, you do not need to declare the actual database foreign key constraints for a field of a table. The entities can have any of the following types of relationships:

- One-to-one
- One-to-many
- Many-to-one
- Many-to-many

The use of annotations has further improved the way in which entity relationships are defined in EJB 3.0. The new functionalities introduced in EJB 3.0, such as using annotations, using JPA, and entities being POJOs have greatly simplified EJB programming. In addition to the annotations used in EJB 3 for different Object Relational (O/R) mapping, there are several annotations available for defining different types of relationships.

Table 14.1 lists the annotations used to define different types of entity relationships:

Table 14.1: Showing the Types of Entities and their Corresponding Annotations

Type of Relationship	Annotation
One-to-one	@OneToOne
One-to-many	@OneToMany
Many-to-one	@ManyToOne
Many-to-many	@ManyToMany

Let's now discuss each entity relationship in detail with their real world examples and their implementation in EJB 3.

The One-to-One Relationship

One-to-one relation specifies that an instance of an entity is associated to only a single instance of another entity. For example, there are two entities, Division and Manager. As you know that one division is managed by one manager only; therefore, the Division entity is said to have a one-to-one relationship with the Manager entity. This relationship is unidirectional. However, if it is given that a manager can manage a single division only, this relationship changes to one-to-one bidirectional entity relationship.

Figure 14.4 shows the one-to-one unidirectional and bidirectional relationships between the Division and Manager entities:

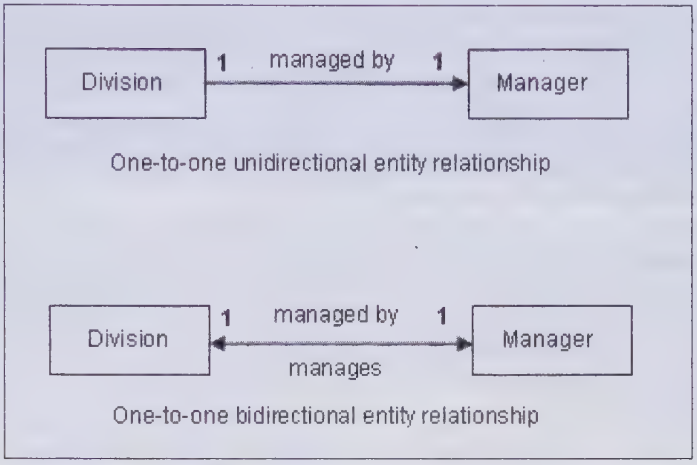


Figure 14.4: Showing One-To-One Unidirectional and Bidirectional Relationship

After understanding the concept of the one-to-one relationship, let's learn how to implement this relationship in an entity class. Consider another example that defines two entities Employee and Department. The Employee entity instance represents a table containing the record of employees working for different departments. The Department entity represents the department details for which different employees are working. There is a one-to-one relationship between the two entities, Employee and Department.

Figure 14.5 shows the two tables, EMPLOYEE and DEPARTMENT, which represents Employee and Department entities:

EMPLOYEE

ID	NAME
A1	VINAY
A2	ASHUTOSH
A3	MANOJ

DEPARTMENT

ID	DEPARTMENT	MANAGER
A1	ACCOUNTS	R BATRA
A2	HRD	P SINHA
A3	ADMIN	N SARKAR

Figure 14.5: Showing the EMPLOYEE and DEPARTMENT Table Representing Entities

Figure 14.5 shows that an employee works only in a single department. For example, an employee with ID A1 has only a single department record in the DEPARTMENT table. Therefore, the Employee entity has a one-to-one unidirectional relationship with the Department entity, as shown in Figure 14.6:

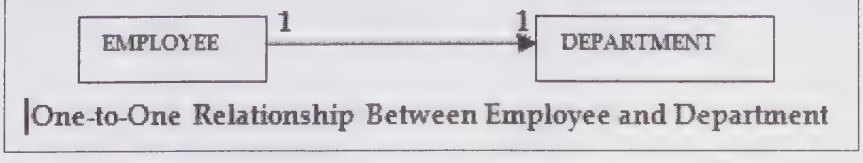


Figure 14.6: Showing a Unidirectional Relationship between Employee and Department Entities

Let's now implement the relationship between Employee and Department entities by creating the entity bean classes for the two entities (Employee and Department), a session bean class, and a class implementing the client code. The client code accesses the session bean and this session bean interacts with the entity beans on behalf of the client. Let's begin with the first entity bean, which is Employee.

The Employee and Department entities have a one-to-one relationship, as shown in Figure 14.6. This can be implemented by providing a single value entity reference at one or both ends of the relationship. Listing 14.7 shows the code of the Employee entity bean class:

Listing 14.7: Showing the Code of Employee.java File Representing One to One Relationship

```
package com.kogent.one_to_one;

import java.io.Serializable;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.OneToOne;


@Entity(name="EmployeeOnetoOne")
public class Employee implements Serializable {
   private String id;
   private String employeeName;
   private Department department;

   public Employee() {
         id = "SPM" + java.lang.Math.random();
   }

   @Id
   public String getId() {
         return id;
   }

   public void setId(String id) {
         this.id = id;
   }

   public String getEmployeeName() {
         return employeeName;
   }

   public void setOrderName(String employeeName) {
         this.employeeName = employeeName;
   }

   @OneToOne(cascade={CascadeType.PERSIST})
   public Department getShipment() {
         return department;
   }

   public void setDepartment(Department department) {
         this.department = department;
   }
}
```

In Listing 14.7, the @OneToOne annotation is used to define a single-valued association. You do not need to define the associated target entity as it is specified from the type of the object (in this case, the Department entity is used) that is being referenced.

You can use a set of optional elements with the `@OneToOne` annotation, such as cascade and fetch.

Table 14.2 lists the available elements for the `@OneToOne` annotation:

Table 14.2: Describing the Optional Elements of the @OneToOne Annotation

Element	Description
CascadeType[] cascade	Helps to define operations that must be cascaded to the associated target entity
FetchType fetch	Helps to define whether the association should be slowly loaded or must be quickly fetched
String mappedBy	Helps to define the fields from where the relationship exists
boolean optional	Sets whether or not the association is optional
Class targetEntity	Defines the target entity class

Prior to implementing the Department entity bean class, let's have a brief discussion over elements used with the `@OneToOne` annotation. The elements used with the @OneToOne annotation are as follows:

- **Cascade**—Defines operations that must be cascaded to the target entity in the association. By default, no operation is cascaded.

Table 14.3 provides various possible constant values that can be used for the CascadeType[] cascade element:

Table 14.3: Describing the CascadeType Constants

Constants	Description
CascadeType.ALL	Cascades all operations
CascadeType.MERGE	Cascades the merge operation
CascadeType.PERSIST	Cascades the persist operation
CascadeType.REFRESH	Cascades the refresh operation
CascadeType.REMOVE	Cascades the remove operation

- **Fetch**—Sets with FetchType constants, which decide whether the association is lazily loaded or eagerly fetched. The possible two values are FetchType.EAGER and FetchType.LAZY. By default, for performance reasons, all the fields' values are fetched lazily.
- **mappedBy**—Defines the field which owns the relationship. This element is specified on the inverse (non-owning) side of the association.
- **Optional**—Sets to either true or false. False defines a not null relationship; whereas, true defines a null relationship.
- **targetEntity**—Sets the target entity in the relationship.

The other entity in the relationship, which represents the DEPARTMENT table, is the Department entity. Let's now define the Department class that does not contain any annotation to define the relationship between the Employee and Department entity.

Listing 14.8 shows the code of the Department entity class:

Listing 14.8: Showing the Code of the Department.java File

```
package com.kogent.one_to_one;
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.Id;


@Entity(name="DepartmentOnetoOne")
public class Department implements Serializable {
   private String id;
   private String departmentName;
```

```
    private String manager;

    public Department() {
          id = "SPM" + java.lang.Math.random();
    }

    @Id
    public String getId() {
          return id;
    }

    public void setId(String id) {
          this.id = id;
    }

    public String getDepartmentName() {
          return departmentName;
    }
    public void setDepartmentName (String departmentName) {
          this. departmentName = departmentName;
    }

    public String getManager() {
          return manager;
    }

    public void setManager(String manager) {
          this.manager = manager;
    }
}
```

Let's create a session bean class to work as an interface between the client code and entity classes. Therefore, creation of a remote interface is required to implement the session bean class. Listing 14.9 provides the code for the `EmployeeDepartment` remote interface:

Listing 14.9: Showing the Code of the EmployeeDepartment.java File

```
package com.kogent.one_to_one.interfaces;

import java.util.List;
import javax.ejb.Remote;


@Remote
public interface EmployeeDepartment {
   public void setQueries();

   public List getEmployees();
}
```

The two methods defined in the `EmployeeDepartment` interface are `setQueries()` and `getEmployees()`. The session bean class, `EmployeeDepartmentBean`, implements the `EmployeeDepartment` interface. An instance of the `EntityManager` interface is acquired through container injection, which is achieved by using the `@PersistenceContext` annotation. The code for the `EmployeeDepartmentBean` class is shown in Listing 14.10:

Listing 14.10: Showing the Code of the EmployeeDepartmentBean.java File

```
package com.kogent.one_to_one;

import java.util.List;

import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;

import com.kogent.one_to_one.interfaces.EmployeeDepartment;
```

```
@Stateless
public class EmployeeDepartmentBean implements EmployeeDepartment {
  @PersistenceContext
  EntityManager em;

  public void setQueries() {
        Departent d = new Department();
        d.setDepartmentName("Accounts");
        d.setManager("Ashutosh Verma");

        Employee e = new Employee();
        e.setEmployeeName("Vinay");
        e.setDepartment(d);

        em.persist(e);
  }

  public List getEmployees() {
        Query q = em.createQuery("SELECT e FROM EmployeeOto e");
        return q.getResultList();
  }
}
```

In Listing 14.10, the `setQueries()` method creates instances of the Department and Employee entities and sets the department field of the Employee entity with the Department object. Finally, the `persist()` method is called and the Employee entity instance is added to a persistence context as a managed instance. In other words, the record of an employee is inserted in the EMPLOYEE table and the associated DEPARTMENT table.

Let's now create the client class, `EmployeeDepartmentClient` to access this session bean, as shown in Listing 14.11:

Listing 14.11: Showing the Code of the EmployeeDepartmentClient.java File

```
package com.kogent.one_to_one.client;
//import goes here
public class EmployeeDepartmentClient {
  public static void main(String[] args) {
        try {
              InitialContext ic = new InitialContext();
              EmployeeDepartment ed =
             (EmployeeDepartment)ic.lookup(EmployeeDepartment.class.getEmployeeName());

              ed.setQueries();

              System.out.println("Unidirectional One-To-One client\n");

              for (Object o : ed.getEmployees()) {
                    Employee emp = (Employee)o;
                    System.out.println("Employee "+emp.getId()+":
                    "+emp.getEmployeeName());
                    System.out.println("\tDepartment:
                    "+emp.getDepartment().getDepartmentName()+"
                    "+emp.getDepartment().getManager());
              }
        }
        catch (NamingException e) {
              e.printStackTrace();
        }
  }
}
```

Now, you can compile all the Java source files provided in Listing 14.7 to 14.11 and create an EAR file to deploy the application on the application server. You can also run the `EmployeeDepartmentClient` class to see the output.

Let's now discuss the one-to-many relationship.

The One-to-Many Relationship

A one-to-many entity relationship shows the association of an instance of an entity with multiple instances of another entity, for example, many orders can be placed for a single customer. Therefore, if there are two entities named, Customer and Order, there is a one-to-many relationship between them.

Figure 14.7 displays the one-to-many relationship between the Customer and Order entities:

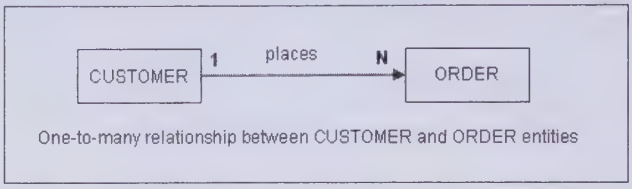


Figure 14.7: Showing the Unidirectional Relationship between CUSTOMER and ORDER Entities

After understanding the concept of the one-to-many relationship, let's learn how to implement this relationship in an entity class. Consider another example that defines two entities, Company and Employee. A company has many employees working for it. The data of these entities is represented by the two tables, COMPANY and EMPLOYEE.

Figure 14.8 shows the two tables, named COMPANY and EMPLOYEE:

COMPANY

COMPANY_ID	COMPANY_NAME
C101	Kogent
C102	SBP
C103	Wiley

EMPLOYEE

COMPANY_ID	EMPLOYEE_NAME
C101	Santosh
C101	Prakash
C101	Ishita
C102	Suman
C102	Jyotsna

Figure 14.8: Showing the COMPANY and EMPLOYEE Table

In Figure 14.8, records of the COMPANY table and the EMPLOYEE table. You should note that the Company_ID field of the EMPLOYEE table contains the records of the COMPANY_ID field of the COMPANY table. You can see the one-to-many relationship between the two entities in Figure 14.9:

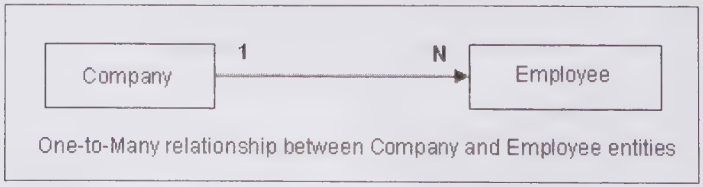


Figure 14.9: Showing the Relationships between Company and Employee Entities

Similar to the one-to-one relationship, let's create the two entity bean classes (Company and Employee), a remote interface, a session bean, and a client class to demonstrate the one-to-many relationship. The `@OneToMany` relationship annotation is used to define the one-to-many relationship. Listing 14.12 provides the code of the Company entity bean class:

Listing 14.12: Showing the Code of the Company.java File

```
package com.kogent.one_to_many;
import java.io.Serializable;
import java.util.Collection;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
@Entity(name="CompanyOMUni")
public class Company implements Serializable {
  private int id;
  private String name;
  private Collection<Employee> employees;

  public Company() {
        id = (int)System.nanoTime();
  }

  @Id
  public int getId() {
        return id;
  }

  public void setId(int id) {
        this.id = id;
  }

  public String getName() {
        return name;
  }

  public void setName(String name) {
        this.name = name;
  }

  @OneToMany(cascade={CascadeType.ALL},fetch=FetchType.EAGER)
  public Collection<Employee> getEmployees() {
        return employees;
  }

  public void setEmployees(Collection<Employee> employees) {
        this.employees = employees;
  }
}
```

The use of generic collection type in Listing 14.12 (Collection<Employee>) enables persistence framework to determine the entity type at the other end of the relationship. Different elements of the `@OneToMany` annotation are the same as those used with the `@OneToOne` annotation except for the optional element, which is not used. The `mappedBy` element is needed only in case when the relationship is unidirectional.

Let's now create the Employee entity bean class, as shown in Listing 14.13:

Listing 14.13: Showing the Code of the Employee.java File Representing One to Many Relationship

```
package com.kogent.one_to_many;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity(name="EmployeeOMUni")
public class Employee implements Serializable {
  private int id;
  private String name;
  private char sex;
```

```
    public Employee() {
          id = (int)System.nanoTime();
    }

    @Id
    public int getId() {
          return id;
    }

    public void setId(int id) {
          this.id = id;
    }

    public String getName() {
          return name;
    }

    public void setName(String name) {
          this.name = name;
    }

    public char getSex() {
          return sex;
    }
    public void setSex(char sex) {
          this.sex = sex;
    }
}
```

In Listing 14.13, a session bean class is used to interact with different entities on behalf of the client code. Let's now create the CompanyEmployeeOM remote interface, as shown in Listing 14.14:

Listing 14.14: Showing the Code of the CompanyEmployeeOM.java File

```
package com.kogent.one_to_many.interfaces;
import java.util.List;
import javax.ejb.Remote;
@Remote
public interface CompanyEmployeeOM {
  public void doSomeStuff();

  public List getCompanies();

  public List getCompanies2(String query);

  public void deleteCompanies();
}
```

In Listing 14.15, the `CompanyEmployeeOM` remote interface defines four different methods--`doSomeStuff()`, `getCompanies()`, `getCompanies2()`, and `deleteCompanies()`. Now, you need to create the session bean class, `CompanyEmployeeOMUniBean`, which implements the `CompanyEmployeeOM` interface. You can see the code of the `CompanyEmployeeOMUniBean` class in Listing 14.15:

Listing 14.15: Showing the Code of the CompanyEmployeeOMUniBean.java File

```
package com.kogent.one_to_many;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;

import com.kogent.one_to_many.interfaces.CompanyEmployeeOM;

@Stateless
```

```
public class CompanyEmployeeOMUniBean implements CompanyEmployeeOM {
   @PersistenceContext
   EntityManager em;

   public void doSomeStuff() {
         Company c = new Company();
         c.setName("M*Power Internet Services, Inc.");

         Collection<Employee> employees = new ArrayList<Employee>();
         Employee e = new Employee();
         e.setName("Vinay Kumar");
         e.setSex('M');
         employees.add(e);

         e = new Employee();
         e.setName("Prakash Kumar");
         e.setSex('M');
         employees.add(e);

         c.setEmployees(employees);
         em.persist(c);

         c = new Company();
         c.setName("Kogent Solutions Inc.");

         employees = new ArrayList<Employee>();
         e = new Employee();
         e.setName("Shilpa Sharma");
         e.setSex('F');
         employees.add(e);

         e = new Employee();
         e.setName("Vikas");
         e.setSex('M');
         employees.add(e);

         c.setEmployees(employees);
         em.persist(c);

         c = new Company();
         c.setName("ABC Pvt. Ltd.");
         em.persist(c);
   }

   public List getCompanies() {
         Query q = em.createQuery("SELECT c FROM CompanyOMUni c");
         return q.getResultList();
   }
   public List getCompanies2(String query) {
         Query q = em.createQuery(query);
         return q.getResultList();
   }

   public void deleteCompanies() {
         Query q = em.createQuery("DELETE FROM CompanyOMUni");
         q.executeUpdate();
   }
}
```

The different methods of the `CompanyEmployeeOMUniBean` class either create instances of the entity or invoke the `persist()` method. In addition, the session bean class executes various queries, such as select and delete, to provide the desired results.

Finally, let's create the client class, `CompanyEmployeeClient`, as shown in Listing 14.16:

Listing 14.16: Showing the Code of the CompanyEmployeeClient.java File

```
package com.kogent.one_to_many.client;

//import goes here

public class CompanyEmployeeClient {
  public static void main(String[] args) {
        try {
                InitialContext ic = new InitialContext();
                CompanyEmployeeOM ceom =
                (CompanyEmployeeOM)ic.lookup
                (CompanyEmployeeOM.class.getName());

                ceom.deleteCompanies();

                ceom.doSomeStuff();

                System.out.println("All Companies:");
                for (Object o : ceom.getCompanies()) {
                        Company c = (Company)o;
                        System.out.println("Here are the employees for
                        company: "+c.getName());
                        for (Employee e : c.getEmployees()) {
                                System.out.println("\tName: "+e.getName()+",
                                Sex: "+e.getSex());
                        }
                        System.out.println();
                }
        }
        catch (NamingException e) {
                e.printStackTrace();
        }
  }
}
```

Now, compile all the Java source code files created in Listing 14.12 to 14.16 and bundle them into an EAR file to be deployed on the Glassfish V3 application server. You can view the output by executing the CompanyEmployeeClient class.

The Many-to-One Relationship

A many-to-one entity relationship shows the association of multiple instances of an entity with a single instance of another entity. For example, there are two entities, ORDER and ORDER_ITEM. There is always a collection of order items related with a single order placed by any customer. In other words, for a single Order entity instance, there are multiple Order_Item instances. You can also say that multiple Order_Item instances can be associated with a single Order instance.

Figure 14.10 shows the many-to-one entity relationship:

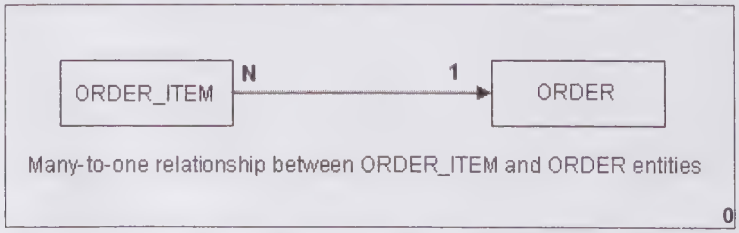


Figure 14.10: Showing Many-To-One Relationship between ORDER_ITEM and ORDER Entities

The two tables representing the Order_Item and Order entities have been shown in Figure 14.11:

ORDER_ITEM

ID	ITEM_NAME	Quantity
1001	Monitor	1
1001	Keyboard	2
1001	Processor	1
1002	Monitor	3
1002	Pen Drive	2

ORDER

ID	CUSTOMER_NAME	PAYEMENT_MODE
1001	John	Cash
1002	Danish	Credit Card

Figure 14.11: Showing the ORDER_ITEM and ORDER Tables

In Figure 14.11, you can see that there are multiple entries in the ORDER_ITEM table, which are associated with a single record in the ORDER table.

After understanding the concept of the many-to-one relationship, let's learn how to implement this relationship in an entity class. Consider another example that defines two entities Employee and BusinessAddress. There can be multiple employees working at a single working place, which is known as their business address. Therefore, you can say that the Employee entity has a many-to-one relationship with the BusinessAddress entity, as shown in Figure 14.12:

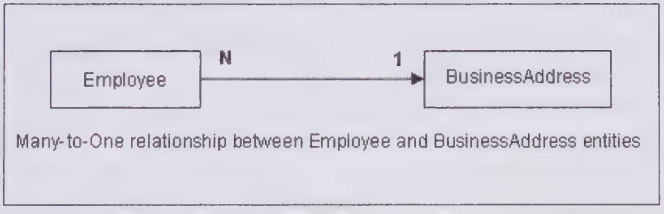


Figure 14.12: Showing Many-To-One Relationships between Employee and BusinessAddress Entities

Let's now implement the relationship between Employee and BusinessAddress entities by creating the entity bean classes for the two entities (Employee and BusinessAddress), a session bean class, and a class implementing the client code. The `@ManyToOne` annotation is used to define the many-to-one relationship between `Employee` and `BusinessAddress` entities.

The code of the Employee entity is shown in Listing 14.17:

Listing 14.17: Showing the Code of the Employee.java File Representing Many to One Relationship

```
package com.kogent.many_to_one;

import java.io.Serializable;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.ManyToOne;

@Entity
public class Employee implements Serializable {
  private int id;
  private String name;
  private BusinessAddress address;

  public Employee() {
      id = (int)System.nanoTime();
  }

  @Id
```

```
    public int getId() {
          return id;
    }

    public void setId(int id) {
          this.id = id;
    }

    public String getName() {
          return name;
    }


    public void setName(String name) {
          this.name = name;
    }

    @ManyToOne(cascade={CascadeType.ALL})
    public BusinessAddress getAddress() {
          return address;
    }

    public void setAddress(BusinessAddress address) {
          this.address = address;
    }
}
```

The `@ManyToOne` annotation can be used with different optional elements, such as `cascade`, `fetch`, `optional`, and `targetEntity`.

The other entity in the many-to-one relationship is BusinessAddress in which you need to define the primary key by using the `@Id` annotation with the id field.

Listing 14.18 shows the code of the BusinessAddress entity class:

Listing 14.18: Showing the Code of the BusinessAddress.java File

```
package com.kogent.many_to_one;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class BusinessAddress implements Serializable {
  private int id;
  private String city;
  private String zipcode;

  public BusinessAddress() {
        id = (int)System.nanoTime();
  }

  @Id
  public int getId() {
        return id;
  }

  public void setId(int id) {
        this.id = id;
  }

  public String getCity() {
        return city;
  }

  public void setCity(String city) {
        this.city = city;
```

```
    }
    public String getZipcode() {
          return zipcode;
    }

    public void setZipcode(String zipcode) {
          this.zipcode = zipcode;
    }
}
```

The session bean created in this example is `EmployeeAddressMOUniBean` class that implements a remote interface, `EmployeeAddressMOUni`. The code of the `EmployeeAddressMOUni` interface is provided in Listing 14.19:

Listing 14.19: Showing the Code of the EmployeeAddressMOUni.java File

```
package com.kogent.many_to_one.interfaces;

import java.util.List;
import javax.ejb.Remote;
@Remote
public interface EmployeeAddressMOUni {
  public void doSomeStuff();

  public List getEmployees();
}
```

The `EmployeeAddressMOUni` interface declares two methods, `doSomeStuff()` and `getEmployees()`. Now, you need to create the `EmployeeAddressMOUniBean` bean class that implements the `EmployeeAddressMOUni` interface and provides the body to the `doSomeStuff()` and `getEmployees()` methods.

The code of the `EmployeeAddressMOUniBean` entity bean class is shown in Listing 14.20:

Listing 14.20: Showing the Code of the EmployeeAddressMOUniBean.java File

```
package com.kogent.many_to_one;

import java.util.List;

import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;

import com.kogent.many_to_one.interfaces.EmployeeAddressMOUni;

@Stateless
public class EmployeeAddressMOUniBean implements EmployeeAddressMOUni {
  @PersistenceContext
  EntityManager em;

  public void doSomeStuff() {
        BusinessAddress a = new BusinessAddress();
        a.setCity("Mumbai");
        a.setZipcode("400001");

        Employee e = new Employee();
        e.setName("Ishita");
        e.setAddress(a);
        em.persist(e);

        e = new Employee();
        e.setName("Prakash");
        e.setAddress(a);
        em.persist(e);

        e = new Employee();
```

```
            e.setName("Santosh");
            e.setAddress(a);
            em.persist(e);
      }

      public List getEmployees() {
            Query q = em.createQuery("SELECT e FROM Employee e");
            return q.getResultList();
      }
}
```

You can see the doSomeStuff() method in Listing 14.20 where the `BusinessAddress` entity instance is being associated with multiple instances of the `Employee` entity before calling the persist() method. Finally, you need to create the client code in the form of the EmployeeAddressClient class, as shown in Listing 14.21:

Listing 14.21: Showing the Code of the EmployeeAddressClient.java File

```
package com.kogent.many_to_one.client;

//import goes here

public class EmployeeAddressClient {
   public static void main(String[] args) {
         try {
               InitialContext ic = new InitialContext();
EmployeeAddressMOUni ea =
   (EmployeeAddressMOUni)ic.lookup(EmployeeAddressMOUni.class.getName());

               ea.doSomeStuff();

               for (Object o : ea.getEmployees()) {
                     Employee e = (Employee)o;
System.out.println("Name: "+e.getName()+", Business Address: "+
   e.getAddress().getCity()+", "+e.getAddress().getZipcode());
               }
         }
         catch (NamingException e) {
               e.printStackTrace();
         }
   }
}
```

Now, compile all the Java source code files created in Listing 14.17 to 14.21 and bundle them into an EAR file to be deployed on the Glassfish V3 application server. You can view the output by executing the EmployeeAddressClient class.

The Many-to-Many Relationship

A many-to-many relationship is defined as the association between two entities where one instance of an entity is associated with multiple instances of another entity and vice-versa. For example, there are two entities Student and Course. An instance of the Student entity can be associated with multiple instances of the Course entity and a single instance of the Course entity can be associated with multiple instance of the Student entity.

Figure 14.13 shows the many-to-many relationship between the Student and Course entities:

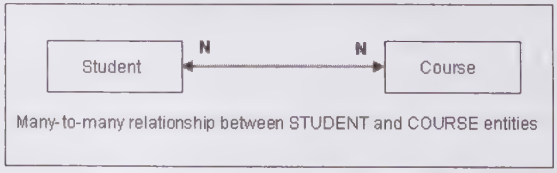


Figure 14.13: Displaying Many-to-Many Relationship between Student and Course Entities

A many-to-many relationship always has two sides called an owning side and a non-owning side. The join operation of a table is defined on the owning side. If the relationship is bidirectional, then either of the side can be designated as the owning side.

To maintain the data of the Student and Course entities, you need to create various tables in the database, such as STUDENT, COURSE, and STU_COURSE. The STU_COURSE table is created by using the join operation between the STUDENT and COURSE tables and the STU_COURSE table specifies many-to-many relationship.

Figure 14.14 shows the STUDENT, COURSE, and STU_COURSE tables:

STUDENT

ST_ID	NAME
101	SUJEET VERMA
102	AMIT KUMAR
103	PUNEET SAXENA

COURSE

CO_ID	CO_NAME
C01	JAVA EE6
C02	VB.NET
C03	PHP

STU_COURSE

ST_ID	CO_ID
101	C01
101	C03
102	C02
103	C01
103	C02
103	C03

Figure 14.14: Showing the STUDENT, COURSE, and STU_COURSE Tables

You can see in Figure 14.14 that a student, say 101, may be enrolled in multiple courses (C01, C03). Similarly, a course can be studied by several students, such as C01 and C03 are studied by students with ID 101 and 103.

After understanding the concept of the many-to-many relationship, let's learn how to implement this relationship in an entity class. Consider another example that defines two entities, Author and Book.

An author can work on various books and a book can be written by various authors. Therefore, the many-to-many relationship exists between the Author and Book entities, as shown in Figure 14.15:

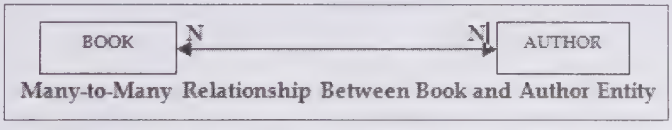


Figure 14.15: Showing the Many to Many Relationship between Book and Author

Let's now implement the relationship between Author and Book entities by creating the entity bean classes for the two entities (Book and Author), a session bean class, and a class implementing the client code. The `@ManyToMany` annotation is used to define many-to-many associations as well as multiplicity of the associations. Whenever you search for the associated entity instance, you always get a collection of objects. You do not need to specify the target entity class if the collection is defined using generics. The code of the Book entity class is shown in Listing 14.22:

Listing 14.22: Showing the Code of the Book.java File

```
package com.kogent.many_to_many;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Collection;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.Id;

import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;
@Entity(name="BookUni")
public class Book implements Serializable {
```

```
    private int id;
    private String bookName;
    private Collection<Author> authors = new ArrayList<Author>();
    public Book() {
          id = (int)System.nanoTime();
    }

    @Id
    public int getId() {
          return id;
    }

    public void setId(int id) {
          this.id = id;
    }

    public String getBookName() {
          return bookName;
    }

    public void setBookName(String bookName) {
          this.bookName = bookName;
    }

    @ManyToMany(cascade={CascadeType.ALL},fetch=FetchType.EAGER)
    @JoinTable(name="BOOKUNI_AUTHORUNI")
    public Collection<Author> getAuthors() {
          return authors;
    }

    public void setAuthors(Collection<Author> authors) {
          this.authors = authors;
    }

}
```

In Listing 14.22, the `@ManyToMany` annotation defines cascade and fetch elements of the `@OneToMany` annotation. The `@JoinTable` annotation is used to specify the joining of table. If the `@JoinTable` annotation is not used, the default values of the annotation elements are applied. The join table name can be created by concatenating the names of the two tables and is separated by an underscore. The optional elements of the `@JoinTable` annotation are listed in Table 14.4:

Table 14.4: Showing the Optional Elements of the @JoinTable

Elements	Description
`String catalog`	Sets the catalog of the table
`JoinColumn[] inverseJoinColumns`	Specifies the foreign key columns of the join table with reference to the primary table of the entity that does not own the association
`JoinColumn[] joinColumns`	Specifies the foreign key columns of the join table with reference to the primary table of the entity owning the association
`String name`	Sets the name of the join table
`String schema`	Sets the schema of the table
`UniqueConstraint[] uniqueConstraints`	Sets the unique constraints that are to be placed on the table

The following code snippet shows the implementation of the @JoinTable annotation:

```
@JoinTable(
  name="Book_Author",

  joinColumns=
  @JoinColumn(name="B_ID", referencedColumnName="ID"),
```

```
    inverseJoinColumns=
    @JoinColumn(name="A_ID", referencedColumnName="ID")
)
```

The entity on the non-owning side of the relationship is Author. The code of the Author entity class is shown in Listing 14.23:

Listing 14.23: Showing the Code of the Author.java File

```
package com.kogent.many_to_many;
import com.kogent.many_to_many.Book;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.Collection;

import javax.persistence.Entity;
import javax.persistence.Id;
@Entity(name="AuthorUni")
public class Author implements Serializable {
   private int id;
   private String authorName;
   private Collection<Book> bookss = new ArrayList<Book>();

   public Author() {
         id = (int)System.nanoTime();
   }

   @Id
   public int getId() {
         return id;
   }

   public void setId(int id) {
         this.id = id;
   }

   public String getAuthorName() {
         return authorName;
   }

   public void setAuthorName(String authorName) {
         this.authorName = authorName;
   }

   public Collection<Book> getBooks() {
         return books;
   }

   public void setBooks(Collection<Book> books) {
         this.books = books;
   }
}
```

As shown in Listing 14.23, the Author entity has the `getBooks()` method, which returns a collection of the associated `Book` instances. Similarly, the `Book` entity has the `getAuthors()` method, which returns a collection of the `Author` objects that are associated with an instance of the `Book` entity.

To interact with the Book and Author entities, you need to create a session bean class. In our case, create the session bean class named as `BookAuthorUniBean`, which implements the remote interface `BookAuthor`.

Listing 14.24 shows the code for the `BookAuthor` interface:

Listing 14.24: Showing the Code of the BookAuthor.java File

```
package com.kogent.many_to_many.interfaces;
import java.util.List;
import javax.ejb.Remote;
import com.kogent.many_to_many.Book;
@Remote
```

```
public interface BookAuthor {
  public void doSomeStuff();

  public List<Book> getAllBooks();
}
```

The two methods declared in the `BookAuthor` interface are `doSomeStuff()` and `getAllBooks()`. These two business methods are defined by the `BookAuthorUniBean` class, which implements the `BookAuthor` interface.

The code for the `BookAuthorUniBean` entity bean class is shown in Listing 14.25:

Listing 14.25: Showing the Code of the BookAuthorUniBean.java File

```
import com.kogent.many_to_many.*;
import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;
import com.kogent.many_to_many.interfaces.BookAuthor;
@Stateless
public class BookAuthorUniBean implements BookAuthor {
  @PersistenceContext
  EntityManager em;
  public void doSomeStuff() {
        Author a1 = new Author();
        a1.setAuthorName("Vinay");

        Author a2 = new Author();
        a2.setAuthorName("Ashutosh");

        Book b1 = new Book();
        b1.setBookName("Java EE6");

        b1.getAuthors().add(a1);

        b1.getBooks().add(b1);

        Book b2 = new Book();
        b2.setBookName("Asp 4.0");

        b2.getAuthors().add(a1);
        b2.getAuthors().add(a2);

        a1.getBooks().add(b2);
        a2.getBooks().add(b2);
        em.persist(a1);
        em.persist(a2);
  }
  public List<Book> getAllBooks() {
        Query q = em.createQuery("SELECT b FROM BookUni b");
        return q.getResultList();
  }
}
```

In Listing 14.25, the `doSomeStuff()` method creates two instances each of the `Book` entity and the `Author` entity, respectively. Now, you need to create a client class to access the session bean.

The client code in the `BookAuthorClient` entity bean class is shown in Listing 14.26:

Listing 14.26: Showing the Code of the BookAuthorClient.java File

```
package com.kogent.many_to_many.client;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import com.kogent.many_to_many.Book;
import com.kogent.many_to_many.Author;
import com.kogent.many_to_many.interfaces.BookAuthor;
public class BookAuthorClient {
  public static void main(String[] args) {
```

```
        try {
                InitialContext ic = new InitialContext();
                BookAuthor ba =
                (BookAuthor)ic.lookup(BookAuthor.class.getName());
                ba.doSomeStuff();
                for (Book b : ba.getAllBooks()) {
                        System.out.println("Book: "+b.getBookName());
                        for (Author a : b.getAuthors()) {
                                System.out.println("\tAuthor:
                                "+a.getAuthorName());
                        }
                }
        }
        catch (NamingException e) {
                e.printStackTrace();
        }
    }
}
```

Now, compile the Java source files created in Listing 14.22 to 14.26 and package them in the EAR file that needs to be deployed on the Glassfish V3 application server. You can view the desired output by executing the `BookAuthorClient` class.

This discussion concludes different types of entity relationships and their implementations in EJB 3. Let's now learn how to map collection-based relationships.

Mapping Collection-Based Relationships

The Java persistence specification also allows us to represent a relationship with the `java.util.List` or `java.util.Map` interface. Therefore, the collection-based relationship can now be expressed by the `java.util.List` interface. The List collection type returns the collection of related entity instances based on a specific set of criteria. This List collection requires the additional metadata provided by the `@javax.persistence.OrderBy` annotation. In other words, you can use the `@OrderBy` annotation to specify the sequence of elements of the collection valued association. This ordering is done when the collection object is retrieved from the persistent storage using some method. You can define the ordering element and the required order, that is `ASC` for ascending and `DESC` for descending order, with the `@OrderBy` annotation. If no order is specified, `ASC` is assumed by default and if no ordering element is defined, the collection is ordered on the basis of the primary key defined for the entity.

Let's take the example of the Booking/Customer relationship, which is a many-to-many bidirectional relationship. The `customer's` attribute of the `Booking` entity represents a list of customers that is stored alphabetically by the `Customer` entity's last name. The implementation of the `getCustomers()` method is provided in the following code snippet:

```
. . . . .

@Entity
public class Booking implements Serializable {

  // define some variables relating to booking
  private List<Customer> customers =
  new ArrayList<Customer>( );
  . . . . . .

  @ManyToMany
  @OrderBy ("lastName ASC")
  @JoinTable(name="BOOKING_CUSTOMER"),
  joinColumns={@JoinColumn(name="BOOKING_ID")},
  inverseJoinColumns={@JoinColumn(name="CUSTOMER_ID")})
  public List<Customer> getCustomers( ) {
        return customers;
  }

  public void setCustomers(Set customers) {
```

```
        this.customers = customers;
    }
}
```

After understanding how to map a List type, let's explore the entity inheritance.

Understanding Entity Inheritance

Inheritance is a basic feature of Object Oriented Programming that allows a subclass to derive the state and behavior of a superclass. A superclass refers to the main class from which the subclass is derived and a subclass refers to the derived class. When a subclass is inherited from a superclass, the subclass derives or inherits all the methods and variables of the superclass. This inheritance feature was absent in the earlier version of EJB. In other words, you could not derive/inherit an entity from another entity in the EJB 1.x or EJB 2.x persistence model.

Now, the Java persistence specification supports entity inheritance for enhanced O/R mapping. Entity inheritance is used as a bridge between object-oriented technology (Java) and relational database technology (RDBMS). The strategies for supporting the object-oriented concept of inheritance in relational database are as follows:

- Single table per class hierarchy
- Separate table per subclass
- Single table per concrete entity class

These strategies are described later in this chapter. Figure 14.16 shows the model that is used for each of these strategies. In this model, the base class is Employee. Two classes are inherited by this class--PartTime and FullTime. Moreover, the Admin and NonAdmin classes are inherited from the FullTime class.

Figure 14.16 shows the class hierarchy:

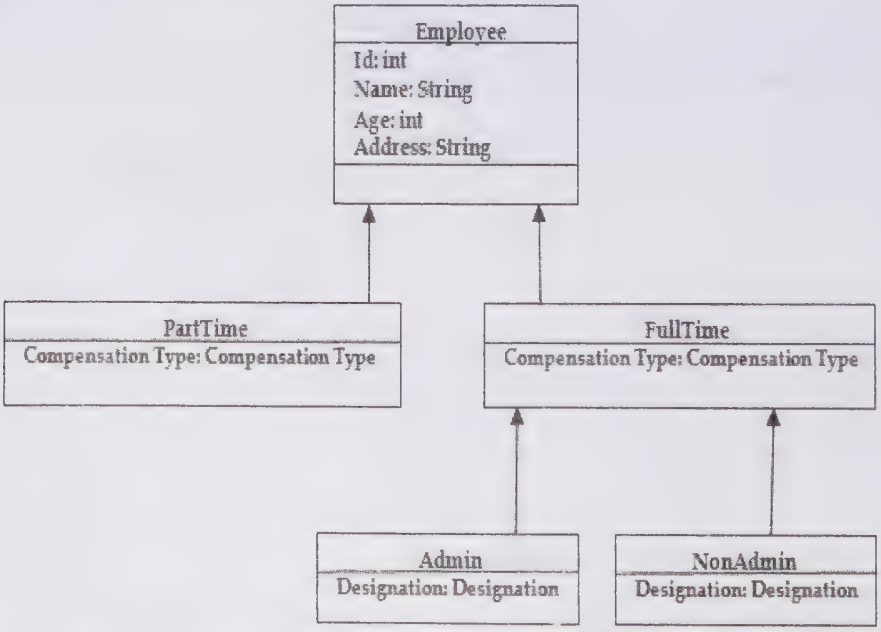


Figure 14.16: Showing the Hierarchical Structure of the Employee Class

Now, create the entity class hierarchy which has been shown in Figure 14.16. Listing 14.27 shows the code implementation of the `Employee` class:

Listing 14.27: Showing the Code of the Employee.java File

```
public class Employee {
public enum Compensation {WAGES, SALARY};
  protected int id;
  protected int age;
  protected String name;
     protected String address;
  public String getName() {
        return name;
  }

  public void setAddress(String address) {
        this.address = address;
  }
  public String getAddress() {
        return address;
  }

  public void setName(String name) {
        this.name = name;
  }

  public int getId() {
        return id;
  }

  public void setId(int id) {
        this.id = id;
  }
  public int getAge() {
        return age;
  }
  public void setAge(int age) {
        this.age = age;
  }

  public String toString(){
        return "ID="+ id +",Age ="+
               age;
  }
}
```

In Listing 14.27, the Employee entity is the root class or the superclass used to derive the other entity classes. The code for creating the `PartTime` class is shown in Listing 14.28:

Listing 14.28: Showing the Code of the PartTime.java File

```
public class PartTime extends Employee {
public final Compensation compensation = Compensation.WAGES;
  public PartTime() {
        id = 0101170101;
        age = 25;
  }

  public String toString(){
        return "PartTime :"+super.toString();
  }
}
```

The `PartTime` entity, created in Listing 14.28, is derived from the `Employee` entity. As shown in Figure 14.16, the `FullTime` entity is also derived from the `Employee` entity. Listing 14.29 provides the code for the `FullTime` class:

Listing 14.29: Showing the Code of the FullTime.java File

```
public class FullTime extends Employee {
public final Compensation compensation = Compensation.SALARY;
  public FullTime() {
        id = 0101190101;
        age = 25;
  }

  public String toString(){
        return "FullTime :"+super.toString();
  }
}
```

In Listing 14.29, the `FullTime` entity is derived from the `Employee` entity class. Listing 14.30 creates the `Admin` entity derived from the `FullTime` entity:

Listing 14.30: Showing the Code of the Admin.java File

```
public class Admin extends FullTime {
public enum Designation {MANAGER, CEO, ACCOUNTANT};
private Designation designation;
public Admin() {
  id = 010280101;
  age = 25;
}
Public Designation getDesignation() {
     Return designation;
}
Public void setDesignation(Designation designation) {
     This.designation = designation;
}
public String toString(){
  return "Admin :"+super.toString();
}
}
```

The code for the `NonAdmin` entity bean class that is derived from the `FullTime` entity bean class, is shown in Listing 14.31:

Listing 14.31: Showing the Code of the NonAdmin.java File

```
public class NonAdmin extends FullTime {
public enum Role {QA, TECHNICAL WRITER, CLEANER};
private Role role;
public NonAdmin() {
  id = 010180101;
  age = 25;
}
Public Role getRole() {
     return role;
}
Public void setRole(Role role) {
     This.role = role;
}
public String toString() {
  return "NonAdmin :"+super.toString();
}
```

Listings 14.27 to 14.31 show the implementation of the entity inheritance. The `Employee` class is the root entity or the superclass from which the `FullTime` and the `PartTime` entities are derived. The `FullTime` and the `PartTime` entities use the `extends` keyword to show that these entities are extended from the root entity, `Employee`. The other two entities, named `Admin.java` and `NonAdmin.java`, are extended from the entity `FullTime`.

Let's now look at the three strategies: single table per class hierarchy, separate table per subclass, and single table per concrete entity class that support the object-oriented concept of inheritance in relational database.

Single Table Per Class Hierarchy

In single table per class hierarchy, the superclass and the subclasses are mapped to a single table. In other words, the mapping of each class is done to a single table. Listing 14.32 shows the implementation of the single table per class strategy:

Listing 14.32: Showing the Code of Employee with the @Entity Annotations

```
//imports go here
@Entity ( name = "EmployeeSingle")
@Inheritance ( strategy = InheritanceType.SINGLE_TABLE )
@DiscriminatorColumn ( name ="DISC", discriminatorType =
DiscriminatorType.STRING)
@DiscriminatorValue ( "EMPLOYEE")

public class  Employee implements  Serializable{
public enum Compensation {WAGES, SALARY};
@Id
protected int id;
protected int age;
protected String name;
protected String address;
public Employee(){
      id = java.lang.Math.random();
}
public String getName() {
   return name;
}
        public void setAddress(String address) {
   this.address = address;
}
public String getAddress() {
   return address;
}
public void setName(String name) {
   this.name = name;
}
public int getId() {
   return id;
}
public void setId(int id) {
   this.id = id;
}
public int getAge() {
   return age;
}
public void setAge(int age) {
   this.age = age;
}
}
```

In Listing 14.32, various new annotations, such as `@Inheritance`, `@DiscriminatorColumn`, and `@DiscriminatorValue`, are used. These annotations are inspected by the Java EE container at the deployment time. These annotations are also known as hint annotations as they give hint to the application server that a hierarchy is being set using the single table strategy. Let's see the use of these annotations in the `FullTime`, `PartTime`, `Admin`, and `NonAdmin` entity classes. The code for the `PartTime` class with entity annotations is provided in the following code snippet:

```
//imports go here

@Entity
@DiscriminatorValue ("PARTTIME")
public class PartTime extends Employee implements Serializable{
public final Compensation compensation = Compensation.WAGES;
public PartTime() {
   super();
```

```
        name = "Arjun";
        address = "NewDelhi";
    }
    }
```

The following code snippet shows the code for the `FullTime` class with entity annotations:

```
    // imports go here

    @Entity
    @DiscriminatorValue ("FULLTIME")
    Public class FullTime extends Employee implements Serializable {
    public final Compensation salary = Compensation.SALARY;
    public FullTime() {
      super();
        name = "Sujeet";
        address = "NewDelhi";
    }
    }
```

The following code snippet shows the code for the `Admin` class with entity annotations:

```
    // imports go here

    @Entity
    @DiscriminatorValue (" ADMIN")
    public class Admin extends FullTime implements Serializable {
    public enum Designation {MANAGER, CEO, ACCOUNTANT};
    private Designation designation;
    public Admin() {
          super();
      name = "Vinay";
    }
    Public Designation getDesignation() {
          Return designation;
    }
    Public void setDesignation(Designation designation) {
          this.designation = designation;
    }
    }
```

The following code snippet shows code for the `NonAdmin` class with entity annotations:

```
    // imports go here

    @Entity
    @DiscriminatorValue (" NONADMIN")
    public class NonAdmin extends FullTime implements Serializable {
    public enum Role {QA, TECHNICAL WRITER, CLEANER};
    private Role role;
    public NonAdmin() {
      super();
      name = "Ashutosh";
    }
    Public Role getRole() {
          return role;
    }
    Public void setRole(Role role) {
          This.role = role;
    }
    }
```

When you deploy the code provided in preceding code snippets on a server, a table is created according to the rules specified by the annotations. The following code snippet shows the structure of the EMPLOYEE table:

```
    CREATE TABLE EMPLOYEE {
      ID INTEGER NOT NULL,
      NAME VARCHAR(31),
      AGE INTEGER,
      COMPENSATION INTEGER,
      ADDRESS VARCHAR(255),
```

```
    ROLE VARCHAR(20),
    DESIGNATION VARCHAR(20),
        DISC VARCHAR(32)
    };
```

The preceding code snippet shows an extra column called DISC, which is a discriminator column defined by using the `@DiscriminatorColumn` annotation in the Employee class. This field in the database has different values, depending on the type of object being persisted to the database.

The following code snippet shows how to persist the previously defined entities, such as `FullTime`, `PartTime`, `Admin`, and `NonAdmin` in the database:

```
. . . . . .
@PersistenceContext
EntityManager emgr;
. . . . . .
Admin fa = new Admin();
fa.setName ( " Vinay");
fa.setDesignation (Designation.MANAGER);
emgr.persist (fa);

NonAdmin fn = new NonAdmin();
fn.setName ("Ashutosh");
fn.setRole (Role.QA);
emgr.persist(fn);

PartTime pt = new PartTime();
emgr.persist(pt);
. . . . . .
```

When the persist() method is invoked, as shown in preceding code snippet, the data inserted into the database. Table 14.5 describes the structure of the inserted data:

Table 14.5: Showing the Persisted Data

ID	DISC	NAME	DESIGNATION	ROLE
010170101	PARTTIME	Ajay	NULL	NULL
010180101	ADMIN	Ashutosh	QA	NULL
010190101	NONADMIN	Vinay	NULL	TECHNICAL WRITER

Let's now discuss how separate tables are created for each subclass in the hierarchy.

Separate Table Per Subclass

In the separate table per subclass strategy, separate tables are created for each subclass in the hierarchy. The layout of the table shows only those properties that are defined in the subclass and are separate from parent classes in the hierarchy. The code for the `Employee` class in this strategy is shown in the following code snippet:

```
@Entity (name = "EmployeeJoined")
@Table (name = " EMPLOYEEJOINED")
@Inheritance (strategy = InheritanceType.JOINED)
public class Employee
{
   . . . . . .
}
```

In the preceding code snippet, the InheritanceType.JOINED strategy is specified by using the @Inheritance annotation. This specifies that the separate table per subclass strategy is implemented in the Employee class. In this strategy, a join between tables must be performed to resolve all the properties of subclasses. The `@Table` annotation is used to specify a different table name from the class name. The structure of the EMPLOYEEJOINED table is shown in Table 14.6:

Table 14.6: Showing the Data from EMPLOYEEJOINED Table				
ID	DTYPE	NAME	DESIGNATION	ROLE
010170101	PARTTIME	Ajay	NULL	NULL
010180101	ADMIN	Ashutosh	QA	NULL
010190101	NONADMIN	Vinay	NULL	TECHNICAL WRITER

Table 14.7 shows the structure of the PARTTIME table:

Table 14.7: Showing the Data from the PARTTIME Table	
ID	WAGES
010170101	600

Table 14.8 shows the structure of the FULLTIME table:

Table 14.8: Showing the Data from the FULLTIME Table	
ID	SALARY
010160101	0
010160103	0

Table 14.9 shows the structure of the ADMIN table:

Table 14.9: Showing the Data from the ADMIN Table	
ID	DESIGNATION
010180101	MANAGER

Table 14.10 shows the structure of the NONADMIN table:

Table 14.10: Showing the Data from the NONADMIN Table	
ID	DESIGNATION
010190101	TECHNICAL WRITER

Let's now discuss single table per concrete entity class strategy.

Single Table Per Concrete Entity Class

In the single table per concrete entity class strategy, each concrete class has its own table. Each table has all the properties found in the inheritance chain up to the parent class.

Table 14.11 shows the database layout for the Admin class:

Table 14.11: Showing the Database Table Layout Mapped for Admin.java File					
ID	NAME	AGE	ADDRESS	DESIGNATION	ROLE

Table 14.12 shows the database layout for the NonAdmin class:

Table 14.12: Showing the Database Table Layout Mapped for NonAdmin.java File					
ID	NAME	AGE	ADDRESS	DESIGNATION	ROLE

Table 14.13 shows the database layout for the PartTime class

Table 14.13: Showing the Database Table Layout Mapped for PartTime.java File				
ID	NAME	AGE	ADDRESS	WAGES

Now, let's proceed to discuss a Java Persistence query language (JPQL) used to operate Java classes and objects (entities).

Understanding JPQL

JPQL is the extended version of the EJB Query Language (EJB QL). Although referred as a query language, JPQL is different from SQL. JPQL operates on classes and objects (entities) available in the Java workspace, while SQL operates on table properties, such as rows and columns in the database space. Using SQL, the columns of a table are selected; however, by using JPQL, the entity fields are selected. JPQL statements can be executed only after they are converted into simple SQL statements, as the SQL statements operate on the table properties and JPQL works on Java classes and objects. The JPQL query parser helps to convert JPQL statements into simple SQL statements. The JPQL query parser is used as an intermediate between the JPQL query and the SQL query, as shown in Figure 14.17. JPQL is a collection of JPQL statements, functions, and operators, which are discussed later in this chapter.

Figure 14.17 demonstrates the conversion of a JPQL query into a simple executable SQL query:

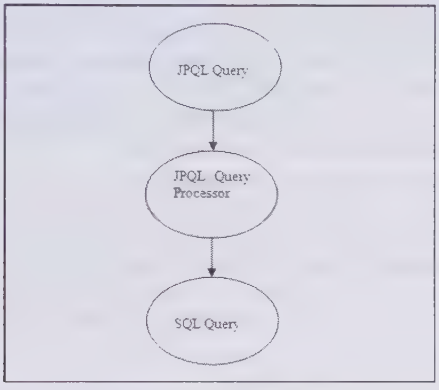


Figure 14.17: Showing the Conversion of JPQL to SQL

Figure 14.17 shows the conversion of a JPQL query into an SQL query with the help of the JPQL Query Processor. The JPQL Query Processor serves as a parser to convert JPQL statements into the SQL statements. To learn how to create a JPQL query, it is essential to know about various statements, functions, and clauses of JPQL.

The following code snippet shows a simple JPQL query, which returns all the employee names from the Employee entity:

```
SELECT e.empname from Employee e
```

In the preceding code snippet, SELECT is a JPQL statement used to retrieve the employee names. Apart from SELECT, you can also use various other JPQL statements, such as UPDATE and DELETE, which are discussed later in the chapter. In the preceding JPQL query, the following elements are used:

- e.empname—Refers to an expression resulting in the string type, which means that it represents the result in a sequence of characters.
- from—Refers to the FROM clause of SQL
- Employee—Refers to the Employee entity
- e—Represents an identifier variable for the Employee entity

Let's now discuss each JPQL function in detail.

JPQL Functions

JPQL provides some built-in functions that are to be used by JPQL statements to perform arithmetic and string operations. These functions can be used with the WHERE or the HAVING clauses of the JPQL statements. The functions available in JPQL are as follows:

- The String functions
- The Arithmetic functions

- The Temporal functions

The String Functions

String functions, used in the SELECT clause in JPQL, filter the results returned by a query. They also perform string manipulation on the data. The following is the description of the string functions available in JPQL:

- **The CONCAT function**— Concatenates (or add) two strings or literals (or constants) together. It is used along with the WHERE clause and takes the string parameters, as shown in the following syntax:
  ```
  CONCAT (String, String)
  ```
  The return type of this function is a string.
- **The SUBSTRING function**—Returns a substring of a specified length from the parent string. It accepts a string value, position from where the characters need to be read, and the length of the string as parameters, as shown in the following syntax:
  ```
  SUBSTRING (String, position, length)
  ```
  The SUBSTRING function returns a string, which is a substring from the first argument and contains the characters from position specified in second argument and contains length-1 characters. The `position` argument defines the starting position of the substring in the parent string (provided as first argument) and the `length` argument defines the length of the substring.
- **The TRIM function**—Trims the whitespaces or a specified character from a string. However, if no character is specified, then this function removes the blank spaces from the string. The TRIM function accepts a character to be trimmed as a parameter, as shown in the following syntax:
  ```
  TRIM ([[LEADING|TRAILING|BOTH] char) FROM] (String)
  ```
- **The LOWER function**—Converts the specified string into its lower case and returns the converted string. The LOWER function accepts a string as its parameter, as shown in the following syntax:
  ```
  LOWER (String)
  ```
  The return type of the LOWER function is a string.
- **The UPPER function**—Converts the specified string into upper case and returns the converted string. The UPPER function accepts a string as its parameter, as shown in the following syntax:
  ```
  UPPER (String)
  ```
  The return type of the `UPPER` function is a string.
- **The LENGTH function**—Returns the length of the specified string. The LENGTH function accepts a string as its parameter, as shown in the following syntax:
  ```
  LENGTH (String)
  ```
  The return type of this function is an integer.
- **The LOCATE function**—Searches the position of one string within another. The search starts at position 1, if `initialPosition` is not specified. The LOCATE function accepts the `initialPosition`, `searchString`, and `stringToBeSearched` as parameters, as shown in the following syntax:
  ```
  LOCATE (searchString, stringToBeSearched [initialPosition])
  ```
  The return type of the LOCATE function is an integer.

Now after understanding String functions, let's discuss the arithmetic functions.

The Arithmetic Functions

Arithmetic functions are used to manipulate data for generating analysis reports. These functions can be used with the `WHERE` clause or the `HAVING` clause. The following is the description of the arithmetic functions available in JPQL:

- **The ABS function**—Returns the absolute value of the expression passed to the function. The syntax of the ABS function is as follows:
  ```
  ABS (number)
  ```
  The return type of this function can be an integer, double, or float.
- **The SQRT function**—Returns the square root of the expression passed to the function. The square root is in the form of a double value. The syntax of the SQRT function is as follows:
  ```
  SQRT (double)
  ```

The return type of the SQRT function is double.

- **The MOD function**—Returns the modulus of the operation for the specified number in the function. The syntax of the MOD function is as follows:

```
MOD (int, int)
```

The preceding function returns the output in the integer format and returns the remainder as a result when the first argument is divided by the second.

- **The SIZE function**—Returns the number of items in a collection. The syntax of the SIZE function is as follows:

```
SIZE (Collection)
```

The SIZE function returns the number of elements in a given collection in the integer format.

The Temporal Functions

JPQL provides certain time-related functions, known as temporal functions, which are used to obtain the current time, date, or timestamp. These functions are translated into database-specific SQL functions and the requested values can be retrieved from the database as desired. This translation is required as the values of current time, date, or timestamp obtained from JPQL Temporal functions vary from the SQL values retrieved from the database. The following are the temporal functions provided by JPQL:

- **CURRENT_TIME ()**—Returns the current time
- **CURRENT_DATE ()**—Returns the current date
- **CURRENT_TIMESTAMP ()**—Returns the current timestamp

After discussing the various built-in JPQL functions implemented by the JPQL statements, let's now discuss the SELECT, UPDATE, and DELETE statements.

JPQL Statements

JPQL includes SQL statements to represent the query language. Three types of statements are available in JPQL—SELECT, UPDATE, and DELETE. These statements are called queries in JPQL. Let's now look at these statements in detail.

The SELECT Statement

The SELECT statement is used to access entity-related data from the Java workspace. An entity is defined as the collection of data that can be stored or retrieved. Entity can also be referred as a class or an object. The SELECT statement can use some clauses, such as from, where, and group by, as shown in the following syntax:

```
Select_statement ::= select_clause from_clause [where_clause] [groupby_clause]
[having_clause] [orderby_clause]
```

The clauses used in the preceding syntax are described as follows:

- **Select**—Determines the type of object or value that is to be retrieved or selected
- **From**—Specifies the entity set from where the selection is to be made
- **Where**—Helps to retrieve conditional data
- **Group By**—Aggregates the data retrieved by using the SELECT statement
- **Having**—Filters the aggregated data retrieved by the SELECT statement
- **Order By**—Helps to order the data that are returned by the query

The SELECT and FROM clauses are mandatory for representation of the SELECT statement, while all the other clauses, such as WHERE or ORDER BY, are optional.

The following code snippet shows the use of the SELECT statement with the WHERE and ORDER BY clauses:

```
SELECT b
FROM Book e
WHERE b.bookName LIKE :bookName
ORDER BY b.id
```

In the preceding code snippet, the SELECT statement arranges the returned value according to the id of the BOOK table.

The UPDATE Statement

The UPDATE statement is used to update the records of a table. The use of the UPDATE statement available in JPQL is very similar to the UPDATE statement in SQL. You should note that at a time, only one entity can be updated in JPQL. The WHERE clause is used to restrict the number of entities affected by the UPDATE statements. The syntax of the UPDATE statement is as follows:

```
UPDATE entityName indentifierVariable
SET single_value_path_expression1 = value1, ...
single_value_path_expressionN = valueN
WHERE where_clause
```

The following code snippet shows the UPDATE query that sets the status of the employees to inactive if their last working day is less than the date specified in the `inactiveThresholdDate` field:

```
UPDATE EMPLOYEE e
SET e.status = 'inactive'
WHERE e.lastworked < :inactiveThresholdDate
```

In the preceding code snippet, the UPDATE statement is used along with the WHERE clause in which EMPLOYEE is the entity name and e is the identifier variable for the entity.

The DELETE Statement

The use of the DELETE statement in JPQL is similar to the DELETE statement of SQL. The DELETE statement is used to delete an entity. You should note that at a time, only one entity can be deleted by using the DELETE statement. The WHERE clause can be used to restrict the number of entities that are affected by the DELETE statement. The syntax for the DELETE statement is as follows:

```
DELETE entityName indentifierVariable
WHERE where_clause
```

The following example shows how to delete all the employees whose status is set to inactive, i.e. they are no longer in the company:

```
DELETE FROM EMPLOYEE e
WHERE e.status = 'inactive'
AND e.company IS EMPTY
```

In the preceding code snippet, the EMPLOYEE is the entity and e is used as the identifier variable

Now, you must be aware that JPQL statements are a combination or collection of JPQL clauses, such as SELECT, WHERE, FROM, and ORDER BY. Now, let's study these clauses in detail.

The SELECT Clause

In JPQL, the SELECT clause is used to retrieve the result of a query. The SELECT clause can have more than one identifier, single valued path expressions, or aggregate functions. The following code snippet shows the syntax of using a SELECT clause:

```
SELECT [DISTINCT] exp1, exp2, exp3.....expN
```

The SELECT clause can have multiple expressions to be retrieved from the database. Therefore, the SELECT clause may contain duplicate values. To avoid retrieving duplicate values, the DISTINCT keyword is used.

Let's now learn how to use the SELECT clause to retrieve the records of all employees from a table in a database. While using the SELECT clause, an identification variable is followed by a navigational operator (.) and a state field or association field (column name) is called a path expression. Remember that while using the SELECT clause, a single valued path expression must be used, and not a collection value path expression. However, multiple path expressions can be used in a clause, which are separated by commas. The following example shows the use of path expressions in a clause:

```
SELECT e.empName1, e.empName2 FROM Employee e
```

In the preceding example, e is the identification variable, followed by the navigational operator (.), forming e.empName1 and e.empName2 as the path expressions.

The SELECT clause can also use the aggregate functions, which are used to group the results retrieved by the SELECT clause. These functions are described as follows:

- **The AVG function**—Calculates the average value of the fields to which it is applied. The following code snippet shows the use of the AVG function along with the SELECT clause:

```
select AVG(e.empsal) from Employee e
```

The preceding code snippet calculates the average of the salary field of the entity Employee.

- **The COUNT function**—Determines the number of results returned by the SELECT clause. The following code snippet shows the use of the COUNT function along with the SELECT clause:

```
select COUNT(e) from Employee e
```

The preceding code snippet counts the number of entities found.

- **The MAX function**—Determines the maximum value of the fields among the results. The following code snippet shows the use of the MAX function along with the SELECT clause:

```
Select MAX(e.empsal) from Employee e
```

The preceding code snippet provides the maximum value of the employee's salary and empsal is passed as an argument to the MAX function.

- **The MIN function**—Determines the minimum value of the fields among the results. The following code snippet shows the use of the MIN function along with the SELECT clause:

```
Select MIN(e.empsal) from Employee e
```

The preceding code snippet provides the minimum value of the employee's salary and empsal is passed as an argument to the MIN function.

- **The SUM function**—Returns the sum of the values of the fields to which it is applied. The following code snippet shows the use of the SUM function along with the SELECT clause:

```
Select SUM(e.empsal) from Employee e
```

The preceding code snippet calculates the sum of the employee's salary and empsal is passed as an argument to the SUM function.

The FROM Clause

The FROM clause defines the domain for identifying the variables or expressions used in the SELECT clause. The domain of the expressions can be constrained by using conditional expressions. Multiple identification variables, separated by commas, can be used in the FROM clause. The following is the syntax of the FROM clause within the SELECT statement:

```
SELECT e FROM domainName e
```

In the preceding syntax, domainName can be any domain from where the specific data needs to be retrieved and e is used as the identification variable. The identification variables are used in JPQL according to the FROM clause and must match the entity specified for the particular query. You can specify multiple identification variables in the FROM clause. The following code snippet shows the use of the FROM clause:

```
From Employee e
```

In the preceding code snippet, Employee is the domainName to be queried and e is the identifier of the type Employee.

JPQL imposes the following restrictions while creating the identifiers:

- Must ensure that the name of the identifier does not belong to any of the reserved words available in JPQL
- Must ensure that the name of the identifier does not belong to the stated categories as statements and clauses (SELECT, FROM, UPDATE, DELETE, and WHERE), joins (inner, outer, left, and fetch), conditions, and operators (AND, OR, NOT, true, false, LIKE, IN, and AS), and functions (AVG, MAX, MIN, COUNT, MOD, UPPER, and TRIM)

You can use multiple identification variables in the FROM clause. These variables are case sensitive and cannot be used with any other clauses.

The WHERE Clause

The output of the SELECT clause occasionally has no limit. Therefore, the WHERE clause is used to filter or limit the result of the SELECT clause. This clause is also used to restrict or limit the UPDATE and DELETE clauses as well. The following code snippet shows the use of the WHERE clause in the SELECT statement:

```
select e FROM Employee e
```

The preceding code snippet results in retrieving all the instances for the Employee entity and does not specify any conditions. Now, look at the following code snippet that specifies a condition by using the WHERE clause:

```
SELECT e FROM Employee e WHERE e.empID > 100
```

In the preceding code snippet, a condition is specified along with the `WHERE` clause. Therefore, only the instances where the `empID` is greater than 100 are displayed in the result. The `WHERE` clause supports most of the Java literals, such as Boolean, float, string, int, and enum. However, it does not support numeric type literals (octal and hexadecimal) and array type (byte [] and char []) values.

The ORDER BY Clause

The `ORDER BY` clause is used to order the values of the data and objects retrieved by the `SELECT` clause. The following is the syntax of the `ORDER BY` clause:

```
ORDER BY path_exp1 [ASC | DESC]. . . . path_expN [ASC | DESC]
```

In the preceding syntax, `ASC` and `DESC` are used to order the objects in either ascending or descending order, respectively. The use of `ASC` and `DESC` are optional. Even if no such declaration is provided, then the `ASC` value is taken by default. If you are using single valued path expressions instead of an identification variable, then the `SELECT` clause must contain the path expressions that is used in the `ORDER BY` clause. You can include more than one `ORDER BY` clause in a single statement. These `ORDER BY` clauses act according to precedence, with the left most `ORDER BY` clause having the highest precedence among the clauses in the statement. If a JPQL query contains both the `WHERE` and the `ORDER BY` clauses, the result is first filtered by the `WHERE` clause and then ordered by the `ORDER BY` clause.

Conditional Expressions

Conditional expressions are used in the `Where` and `Having` clauses of the JPQL query consisting of any one of the `Select`, `Update`, or `Delete` JPQL statements. These expressions include other conditional expressions, comparison operators, and path expressions that evaluate Boolean values and Boolean literals. A conditional expression can be any arithmetic expression, which returns the Boolean value. JPQL provides various operators used by the conditional expressions, as described in Table 14.14:

Table 14.14: Describing the Operators used in Conditional Expressions

Operator Type	Use	Example
Navigational Operator	Helps to form the path expression used in the query.	.
Unary Operators	Helps to denote the sign of an expression.	+, -
Arithmetic Operators	Helps to perform the arithmetic operations required for conditional expressions.	*, /, +, -
Relational Operators	Returns boolean values. The relational operators are used for checking the conditions.	=,>,<,>=,<=, [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL, IS [NOT] EMPTY, [NOT] MEMBER [OF]
Logical Operators	Helps to control the program flow by joining two or more conditions.	NOT,AND,OR

The navigational operator (.) is used in the path expressions for navigation purpose. The unary and arithmetic operators are used to perform the arithmetic operations. The relational operators, such as `BETWEEN`, `EMPTY`, and `IS NULL` are used for checking the conditions, such as <, >, =, or >=. The logical operators are used for controlling the flow of a program.

Exploring the Path Expressions

The navigational operator (.) is used in the path expression. Consider the following code snippet to understand the use of the navigational operator:

```
Select e.empname from Employees e
```

In the preceding code snippet, `e.empname` is the path expression, `e` is the identification variable for the Employee entity, and `empname` is the state-field. The navigation operator (.) is used after the identification variable and before the state-field.

NOTE

The state-field includes the fields or properties of an entity, which are independent and not interrelated. The association-field, on the other hand, includes those fields of an entity having a relationship.

Let's now discuss relational operators next.

Using the BETWEEN Operator

The `BETWEEN` operator is used in an arithmetic expression to compare a range of values. A certain range is specified in the expression and the `BETWEEN` operator compares the values within this range. The following code snippet represents the use of the `BETWEEN` operator in a query:

```
path_expression [NOT] BETWEEN lowerRange and upperRange
```

In the preceding code snippet, `lowerRange` refers to the minimum value from where the comparison needs to begin, while `upperRange` refers to the maximum value to which the comparison can be done.

Let's suppose you need to display the id of the employees between the specified ranges. To do this, `upperRange` and `lowerRange` must be specified in the query. The following code snippet shows the use of the `BETWEEN` operator by specifying the highest and lowest ranges:

```
Select e.empId from Employee e WHERE e.empID BETWEEN :200 AND :300
//this will display the employee ID in between the specified range.
```

In the preceding code snippet, 200 is the `lowerRange` and 300 is the `upperRange` for the `bETWEEN` operator.

Using the IN Operator

The `IN` operator is used to create conditional expressions within a list of values, which are separated by commas. The following is the syntax of the `IN` operator in JPQL:

```
Path_expression [NOT] IN (List_of_values)
```

In the preceding syntax, the use of the `NOT` keyword is optional. The following code snippet shows the implementation of the IN operator:

```
SELECT e FROM Employee e WHERE e.empId IN (1, 2)
```

The preceding code snippet is used to select all the fields of the Employee entity where `empId` is 1 or 2. If `Not` is used in this case, the output of the query changes and the query selects all the fields of the Employee entity where `empId` is neither 1 nor 2. The following code snippet shows the implementation of Not in the IN operator:

```
SELECT e FROM Employee e WHERE e.empId Not IN (1, 2)
```

Using the LIKE Operator

The `LIKE` operator is used to determine whether a single-value path expression matches with a specified string pattern. The syntax of the `LIKE` operator is:

```
String_value_path_expression [NOT] LIKE pattern_value
```

The `pattern_value` specified in the syntax is an input character, which may contain an underscore (_) or a percentage (%). The underscore (_) in a `pattern_value` is used to represent a single character. The following code snippet shows the implementation of the LIKE operator:

```
WHERE e.empName LIKE '_am'
```

The preceding code snippet evaluates the names of the employees whose names end with the pattern am.

The percentage (%) symbol is used to represent any number of characters. It can be represented as follows:

```
WHERE e.empName LIKE 'ana%'
```

The preceding code snippet evaluates the names of those employees whose names start with ana and end with any number of characters following this pattern.

Using the NULL Operator

The NULL operator is used to test whether a single valued expression contains a null value or not. The IS NULL returns true if a single valued path expression contains a null value. The following code snippet shows the use of the NULL operator:

```
Select e from Employee e WHERE e.empName IS NOT NULL
```

The preceding code snippet results in selecting all the fields of the Employee entity, where empname is not null. If you use the IS NULL value instead of NOT NULL, the query provides all the fields of the Employee entity where empname is NULL. The following code snippet shows the use of IS NULL:

```
Select e from Employee e WHERE e.empName IS NULL
```

Using the EMPTY Comparison Expression

To understand the EMPTY expression, let's consider an example of the Item and Category entities having a many-to-many relationship between them. The following code snippet is used to query all objects in the Category entity that have associated items:

```
SELECT distinct c FROM Category c
WHERE c.items is NOT EMPTY
```

In the preceding code snippet, c.items represents many-to-many association. Such association fields are called collection types and the path expressions that contain collection types are known as collection-value path expressions.

You should note that the NULL comparison operator is not applicable to the path expressions of collection type. Instead, the IS [NOT] EMPTY operator checks whether or not the collection valued path expression is empty.

Using the JPQL Collection Member Expression

The collection member expression tests whether or not a value is a member of the collection specified by the collection_valued expression. If the collection valued path expression defines an empty collection, then the value of the MEMBER OF expression returns false. In this case, the value of the NOT MEMBER OF expression returns true. If the value of the collection_valued path expression is unknown, then the collection member expression is also unknown. The syntax of a MEMBER OF expression in a query is given as follows:

```
Entity_expression [NOT] MEMBER [OF] collection_value_path_expression
```

Using the EXISTS Expression

The EXISTS expression is used to test whether or not a query contains any result set. It returns true if the result set contains one or more values; otherwise, it returns false. The EXISTS expression is mostly used in a subquery. The syntax of the EXISTS expression is given as follows:

```
Exists_expression [NOT] EXISTS subquery
```

Now, consider the following code snippet:

```
Select Distinct emp from Employee emp where EXISTS (Select sal from Employee
emp where sal=5000)
```

In the preceding code snippet, the result of the given query consists of all the employees having the salary equal to 5000.

Using the ALL, ANY, and SOME JPQL Expressions

The ALL conditional expression returns a true value, if the comparison operation is true for all the values in the result of the subquery, or if the result of the subquery is empty. If a single instance of the query returns false, then the ALL conditional expression returns false. The ANY conditional operator is true if the comparison operation is true for some values in the result of the subquery. The ANY conditional operator returns false, if the result of the subquery is empty or if the comparison operation is false for every value of the result of the subquery. The SOME operator is synonymous with the ANY operator. The comparison operators used with the ALL and ANY conditional expressions are =, <, >, <=,>=, and <>.

After learning about the operators used in JPQL statements, let's discuss another important concept, known as Query API.

Query API

The Query API feature of EJB allows you to create custom queries to access a single or a collection of entities from the database. All the query definitions, parameter binding, execution, and pagination are performed by the Query interface. JPA Query API uses JPQL or SQL to create queries. SQL deals with queries used to access database records; whereas, JPQL deals with entities or Java objects and classes. The Query API consists of the EntityManager methods, the Query interface methods, and JPQL. The `EntityManager` methods are used to create queries and the Query interface methods are used to execute them.

Figure 14.18 shows the structure of the Query API:

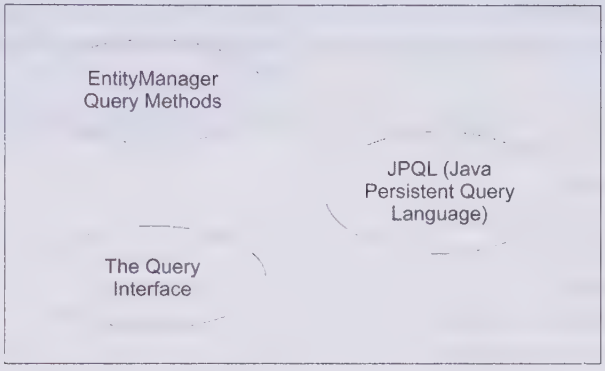


Figure 14.18: Showing Structure of the Query API

The Query API available in JPQL supports two types of queries, called named queries and dynamic queries. The main difference between the named query and the dynamic query is that the dynamic query uses the `createQuery()` method while the named query uses the `createNamedQuery()` method while creating queries.

Defining Dynamic Queries

Dynamic queries are used to create dynamic query statements based on the user input or the conditions of the application logic. The following code snippet shows the use of the dynamic query by retrieving all the results:

```
@PersistenceContext em;
...
public List findAllEmployees() {
  Query query = em.createQuery("SELECT e FROM Employee e"); ...
  return query.getResultList(); ............
}
```

The preceding code snippet returns all the results retrieved by the `select` clause by using the dynamic query. The preceding code snippet creates an instance of the `EntityManager` with the help of the `@PersistentContext` annotation. The EntityManager then creates an instance of the Query object to get the result using the `EntityManager.createQuery()` method. The `createQuery()` method takes the query string as an object. The final step of the query is to return the result by using the `getResultList()` method. The JPQL and SQL can also be used for defining a dynamic query.

Defining Named Queries

Named queries, also known as static queries, are defined by the entities using metadata annotations or the `XML` file defining the O/R mapping. These queries can be accessed by their name when the EntityManager creates their instance. Named queries are created and saved in the entity specified by the `EntityManager` and can be accessed by the `EntityManager.createNamedQuery()` method. The following are the benefits of the named queries:

- Enhances the performance of execution of code
- Improves the maintenance and reusability of code

Named queries are created and stored by using metadata annotation. The following code snippet shows the use of a named query:

```
@Entity
@NamedQuery(
name = "findAllEmployees",
query = "SELECT e FROM Employee e WHERE e.employeeName
LIKE :employeeName ")
public class Employee implements Serializable {
..
}
```

The preceding code snippet shows a named query using the `@javax.persistence.NamedQuery` annotation. Named queries are scoped with the persistence unit, as they are static queries and cannot be changed.

Executing Queries

Queries (dynamic or named) are executed in order to extract the desired result from a database. The various steps or stages required to execute queries are:

- Creating a Query instance
- Creating the Named Query instances
- Creating dynamic query instances
- Describing query interface
- Establishing parameter setting in query
- Retrieving a single entity in a query
- Retrieving a Collection of entities
- Paginating a result list
- Controlling the flush mode
- Specifying Query hints

Creating a Query Instance

As already discussed, you can create two types of queries in JPQL, called named and dynamic. To execute these queries, you need to create query instances. The `EntityManager` interface provides the following methods to create query instances and SQL queries:

- **createQuery(String sqlString)**—Helps to create dynamic queries using a JPQL statement.
- **createNamedQuery(String name)**—Creates a query instance by using a named query. This method can be used for both JPQL and SQL queries.
- **createNativeQuery(String sqlString)**—Helps to create a dynamic query by using a native SQL statement with the UPDATE or DELETE clauses.
- **createNativeQuery(String sqlString, Class result-class)**—Helps to create a dynamic query by using a native SQL statement that receives a single entity type.
- **createNativeQuery(String sqlString, String result-setMapping)**—Helps to create a dynamic query by using a native SQL statement that retrieves a result set with multiple entity types.

Creating the Named Query Instances

Named queries have global scope because they are static queries and cannot be changed. A named query instance can be created from any instance that has access to the persistence unit. The `EntityManager` interface is needed to create an instance of a named query instance. Named queries are created and saved in the entity specified by the `EntityManager`. You can access a named query by the `EntityManager.createNamedQuery()` method. The following code snippet shows the creation of named query instances by using the stored named query:

```
Query query = em.createNamedQuery("findAllEmployees");
```

The preceding code snippet shows the creation of the named query instance by using the instance of the EntityManager. In this case, findAllEmployees is a named query stored in the entity and name of the query is

passed as a parameter to the method. The `EntityManager` instance, em, is used to access the named query and returns the reference to the query object.

Creating Dynamic Query Instances

To create dynamic queries, an `EntityManager` is used. The `EntityManager` includes session beans, message driven beans, Web applications, and some outside containers. Dynamic queries are not supported by previous versions of EJB. The `EntityManager.CreateQuery()` method is used to create dynamic queries in JPQL. This method takes a valid JPQL statement as a parameter and returns the result according to the query. The following code snippet shows how to create a dynamic query by using the `EntityManager.CreateQuery()` method:

```
Query query = em.createQuery ("select e FROM Employee e");
```

In the preceding code snippet, query is the object that stores the result returned from the `em.createQuery()` method. The em is an instance of the `EntityManager` interface. This instance is used to access the method in the `EntityManager` interface. The method evaluates the JPQL query and the result of the selected query is stored in the query object.

Describing the Query Interface

The Query interface contains some methods to execute the query. It also contains methods to set parameters in a query, such as specifying the pagination properties for the query and controlling the flush mode. The flush mode determines whether all the changes in the transaction have been written out. The Query interface works for both JPQL and SQL queries. The `javax.persistence.Query` interface is used to set the parameters for the methods, and control the flush mode. The following are the methods of the Query interface that are used to retrieve the results:

- **getResultList()** – Retrieves a result set for a query
- **getStringResult()** – Returns a single result or object for a query
- **executeUpdate()** – Executes the JPQL UPDATE or DELETE statement
- **setMaxResults()** – Sets the maximum number of results that is to be retrieved
- **setFirstResult()** – Specifies the initial position for the first element or result that is to be retrieved
- **setHint()** – Sets the vendor specific hint for the query
- **setParameter(String name, object value)** – Sets the value for the named parameter
- **setParameter(String name, Date value, TemporalType temporalType)** – Sets the value for the named parameter when the parameter is of the Date type
- **setParameter(String name, Calendar value, TemporalType temporalType)** – Sets the value for Calendar type named parameter
- **setParameter(int position, object value)** – Sets the value for a positional parameter
- **setParameter(int position, Calendar value, TemporalType temporalType)** – Sets the value for Calendar type positional parameter
- **setFlushMode(FlushModeType flushMode)** – Helps to set the flush mode

Let's now consider the following code snippet to demonstrate the use of dynamic queries along with the methods of the `EntityManager` interface:

```
query = em.createNamedQuery("findEmployeeByName");
query.setParameter("employeeName", employeeName);
query.setMaxResults(10);
query.setFirstResult(3);
List categories = query.getResultList();
```

In the preceding code snippet, the em instance of the `EntityManager` is used to create the queries. The `query.setParameter()` method is used to set the parameters for the method that is to be executed. The `setMaxResult()` method is used to set the maximum result for the query and the `setFirstResult()` is used to set the initial position of the first element or result to be retrieved.

Establishing Parameter Setting in Query

The number of entities retrieved by a query can be limited by using the `WHERE` clause. You can specify the parameters in a string in two ways: by names or by numbers. If the parameter specified in the method takes an integer value, the parameter is called a positional parameter or a numbered parameter. Positional parameters are common in query languages. EJB 2 does not support positional parameters. You have to specify the parameters before executing a query in JPQL.

When you specify a name for a parameter, that parameter becomes a named parameter. You should note that a named parameter starts with a colon (:), followed by the name of the parameter. These parameters increase the readability of the code. The major difference between a positional and named parameter is their structure. A positional parameter starts with a Question mark (?) and is followed by the position of parameter. A named parameter, on the other hand, starts with a colon (:), followed by the name of the parameter.

Retrieving a Single Entity in a Query

A single query instance can be retrieved by using the methods available in the `Query` interface. The following code snippet shows the implementation of the `query.getSingleResult()` method:

```
query.setParameter(1, "Jhon Smith");
Employee emp = (Employee)query.getSingleResult();
```

The `getSingleResult()` method returns a single query instance, if there is a unique result available for the entity. In case of a repeated entity, the method throws the `NonUniqueResultException` exception. If the query does not retrieve any result, the `getSingleResult()` method throws the `NoResultException` exception. To avoid these exceptions, you can write the query within the try catch statements, as shown in the following code snippet:

```
try {
   ...
   query.setParameter (1, "Jhon Smith");
   Employee emp= (Employee) query.getSingleResult ();
   ...
   }
   catch (NonUniqueResultException ex) {
         handleException (ex);
   }
   catch (NoResultException ex) {
         handleException(ex);
}
```

Retrieving a single result from a query needs an active transaction. In other words, the query should not contain any `UPDATE` or `DELETE` statement.

Retrieving a Collection of Entities

You can retrieve a collection or set of entities in a result set or in a result list. The `getResultList()` method is used to retrieve all the results of a query. If the `getResultList()` method does not return any value for a query, then it returns an empty list and no exceptions are thrown. Retrieving all results from a query does not need any active transactions, as the `getResultList()` method does not throw any exception.

Paginating a Result List

Pagination is the property to deal with a large number of results retrieved by the query. This allows you to iterate through the results of a query, when you need to display only a part of a large number of results on a page or in a report and other results on the next page or report. The following code snippet shows the use of the iterator in a result list:

```
Iterator l =names.iterator();
while (l.hasNext()) {
   Item name = (Item)l.next();
   System.out.println("Id:" + name.getNameId() +
   " Initial Name:"+name.getInitialName());
}
```

In the preceding code snippet, an iterator is specified to navigate the result list obtained by the query. JPA provides the ability to paginate through the result list of a query. The following code snippet is used to set the pagination property in a result list:

```
query.setMaxResults (10);
query.setFirstResult (10);
List names = query.getResultList ();
```

The preceding code snippet uses two methods: the `setMaxResult()` method and the `setFirstResult()` method. The `setMaxResult()` method is used to set the maximum number of entities to be displayed to the user and the `setFirstResult()` method is used to indicate the location from where the pagination takes place. The values passed to these functions are the indicators to the result lists of the query. In this case, 10 entities are to be displayed to the user in a single page and the iteration starts from the 10th entry in the result list.

Controlling the Flush Mode

The flush mode determines the working of the EntityManager. The result of a query depends on the setting of the flush mode, whose function is to check whether or not all changes in a transactions are written on the database. The `Query.setFlushMode()` method is used to set the flush mode for a query. The default flush mode is AUTO. In the default mode, the persistent provider is responsible for the updates made to the entities in the persistent context. If the flush mode is set to `FlushModeType.COMMIT`, then the updates made to the entities are not defined by the persistent provider.

Specifying Query Hints

The persistent provider provides some extensions during the execution of the queries. These are the performance optimizations and are called query hints. In other words, a query hint is a tip that is provided by the persistent provider while executing or retrieving an entity. The hints specified by the persistent provider are implementation-specific. You can specify the hint for a query by using the `Query.setHint()` method. The following are some specified links available to provide hints for queries:

- **toplink.jdbc.fetch-size**—Provides information about the number of rows fetched by the JDBC driver
- **toplink.cache-usage**—Specifies the usage of cache
- **toplink.refresh**—Specifies whether or not the cache is refreshed from the database
- **toplink.jdbc.timeout**—Specifies the timeout for a query

After understanding how entities are created using JPA, let's create an enterprise application using the Customer entity (Listing 14.2) and CustomerFacade (Listing 14.5) session bean classes created earlier in the chapter.

Developing Sample Application

Let's consider a scenario of the ABC company where the employees need to view all the customers of their company. The employees also want to look up the details of each customer on the basis of the customer id. To do this, you need to create Customer enterprise application having the Customer entity. As seen in Listing 14.2, `Customer.java` can be considered as the Customer entity under this scenario. In the Customer application, some JSP pages are designed to view the customer details. In addition, some Java classes, such as FormConstants and UrlUtils, are also designed in the following sections.

Let's first understand the directory structure for Customer application, which helps in understanding the locations for JSP pages and servlets.

Exploring the Directory Structure

In the Customer application, two modules are created namely, Customer-war and Customer-ejb. The Customer-war module is a Web module containing the JSP pages and servlets. The Customer-ejb module comprises the entity class (Customer) and the session façade bean (CustomerFacade).

Figure 14.19 displays the complete directory structure for Customer enterprise application:

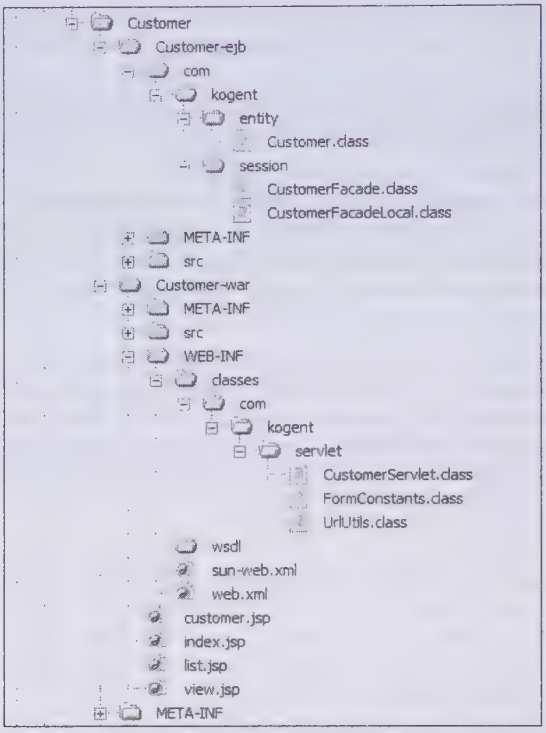


Figure 14.19: Displaying Directory Structure of Customer Application

Figure 14.19 shows the directory structure of the Customer application, in which various Java classes, such as CustomerServlet (a servlet class), FormConstants, and UrlUtils, which are located under the classes subfolder (along with the package directory) of the WEB-INF folder of the Customer-war directory. The Customer, CustomerFacade, and CustomerFacadeLocal Java classes are located under the classes subfolder of the Customer-ejb directory. The web.xml is a configuration file and a persistence-unit for the application is packaged in the persistence.xml file. The JSP pages for the application are index.jsp, list.jsp, view.jsp, and customer.jsp. In the Customer application, oracle is used as the data source. The database name is XE and the table name is Customer.

The table structure of the Customer table is provided in Table 14.15:

Table 14.15: Showing Structure of the Customer Table

Field Name	Data Type
custId	NUMBER(10)
firstName	VARCHAR2(20)
lastName	VARCHAR2(20)
Company	VARCHAR2(20)
address_1	VARCHAR2(30)
address_2	VARCHAR2(30)
City	VARCHAR2(10)
State	VARCHAR2(10)
Zip	VARCHAR2(8)
emailAddress	VARCHAR2(20)
PhoneNumber	VARCHAR2(12)

NOTE

The table name must be customer, as it maps the customer entity; whereas, the database name can be any name of your choice

Now, let's create the servlet classes and JSP pages for the Customer-war module.

Creating the Web Module

The Web module for the Customer enterprise application is Customer-war. The Customer-war module comprises FormConstants and UrlUtils Java classes. It also contains a servlet class named as CustomerServlet. Apart from servlets, the Web module also contains various JSP pages, such as list.jsp, view.jsp, and customer.jsp. The FormConstants Java source file is used to declare the constants that are used throughout the Customer application. This source file also declares and initializes the form fields, request attributes, form actions, and the controller servlet for the Customer-war module.

Listing 14.33 provides the code for the FormConstants class (you can find the FormConstants.java file on CD in the `code\JavaEE\Chapter14\Customer\Customer-war\src\com\kogent\servlet` folder):

Listing 14.33: Showing the Code of the FormConstants.java File

```
package com.kogent.servlet;

/**
 * Some constants used in servlets and jsps for Customer application
 */
public class FormConstants {

  // Form Actions
  public static final String ACTION_QUERY_FORM = "queryForm";
  public static final String ACTION_VIEW_FORM = "viewForm";
  public static final String ACTION_QUERY = "query";
  public static final String ACTION_EDIT_FORM = "edit";

  // Form Fields
  public static final String FIELD_ACTION = "action";
  public static final String FIELD_COMPANY = "company";
  public static final String FIELD_FNAME = "fname";
  public static final String FIELD_LNAME= "lname";
  public static final String FIELD_CUSTOMER_ID = "customer_id";
  public static final String FIELD_EMAIL = "email";
  public static final String FIELD_PHONE = "phone";
  public static final String FIELD_ADDRESS1 = "address1";
  public static final String FIELD_ADDRESS2 = "address2";
  public static final String FIELD_CITY = "city";
  public static final String FIELD_STATE = "state";
  public static final String FIELD_ZIP = "zip";

    // Request attributes
    public static final String ATTRIBUTE_CUSTOMER = "customer";
    public static final String ATTRIBUTE_CUSTOMER_LIST = "customer_list";

    // Controller Servlet
    public static final String CONTROLLER = "CustomerServlet";
}
```

The FormConstants class is defined under the com.kogent.servlet package and is saved at the location shown in the directory structure (Figure 14.19). The other Java class, UrlUtils, is used to manipulate the Uniform Resource Locators (URLs) depending upon the request made to the server.

Listing 14.34 provides the code for the UrlUtils Java class (you can find the UrlUtils.java file on CD in the `code\JavaEE\Chapter14\Customer\Customer-war\src\com\kogent\servlet` folder):

Listing 14.34: Showing the Code of the UrlUtils.java File

```
package com.kogent.servlet;

import static com.kogent.servlet.FormConstants.*;
```

```
/**
* Utility class to generate URLs for Customer application
*/
public class UrlUtils {

   private static String makeActionUrl(String action) {
         return CONTROLLER + "?" + FIELD_ACTION + "=" + action;
   }

   public static String makeCustomerViewUrl(String customerId) {
         return makeActionUrl(ACTION_VIEW_FORM) + "&" + FIELD_CUSTOMER_ID
         + "=" + customerId;
   }
}
```

In Listing 14.34, the UrlUtils class is created to generate URLs for Customer application. Now, let's create the CustomerServlet is class, which the Controller servlet class, which is used to handle the type of request sent by the various JSP pages.

Listing 14.35 provides the code for the CustomerServlet servlet class (you can find the CustomerServlet.java file on CD in the `code\JavaEE\Chapter14\Customer\Customer-war\src\com\kogent\servlet` folder):

Listing 14.35: Showing the Code of the CustomerServlet.java File

```
package com.kogent.servlet;

import com.kogent.entity.Customer;
import com.kogent.session.CustomerFacadeLocal;
import java.io.*;
import java.net.*;
import java.util.*;
import javax.ejb.EJB;
import javax.persistence.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class CustomerServlet extends HttpServlet {
   @EJB
   private CustomerFacadeLocal customerFacade;

protected void processRequest(HttpServletRequest request, HttpServletResponse response)
   throws ServletException, IOException
   {

String action = request.getParameter(com.kogent.servlet.FormConstants.FIELD_ACTION);
if (com.kogent.servlet.FormConstants.
ACTION_QUERY_FORM.equals(action)) {
                displayQueryForm(request, response);
} else if (com.kogent.servlet.FormConstants.ACTION_VIEW_FORM.equals(action)) {
                displayViewForm(request, response);
} else if (com.kogent.servlet.FormConstants.ACTION_QUERY.equals(action)) {
                doQueryAll(request, response);
        }
        else {
                displayQueryForm(request, response);
        }
   }
   private void displayQueryForm(HttpServletRequest request,
   HttpServletResponse response) throws ServletException, IOException
   {

getServletConfig().getServletContext().
getRequestDispatcher("/index.jsp").forward(request, response);
   }
   @Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
   throws ServletException, IOException {
```

```
            processRequest(request, response);
    }

    @Override
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
            processRequest(request, response);
    }

    @Override
    public String getServletInfo() {
            return "Short description";
    }

private void displayViewForm(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
    {
int custId = Integer.parseInt(request.getParameter(com.kogent.
servlet.FormConstants.FIELD_CUSTOMER_ID));
            Customer customer = customerFacade.find(custId);
request.setAttribute(com.kogent.servlet.FormConstants.
ATTRIBUTE_CUSTOMER, customer);
getServletConfig().getServletContext().getRequestDispatcher
 ("/view.jsp").forward(request, response);

    }

private void doQueryAll(HttpServletRequest request, HttpServletResponse response) throws
IOException, ServletException
    {
            List<Customer> cust = customerFacade.findAll();
request.setAttribute(com.kogent.servlet.FormConstants.
ATTRIBUTE_CUSTOMER_LIST, cust);
getServletConfig().getServletContext().getRequestDispatcher
("/list.jsp").forward(request, response);
    }
}
```

When the `CustomerServlet` class is called, the request is being forwarded to the `index` page, by default. Depending upon the action, the respective method is called in the Customer application. For example, if the action form is `ACTION_QUERY_FORM`, the `displayQueryForm()` method is called and the request is forwarded to the `index` page. If the action is `ACTION_VIEW_FORM`, the `displayViewForm()` method is called, which uses `CustomerFacade`, an instance of the `CustomerFacadeLocal` interface to find the customer details on the basis of `custId`. The `custId`, an integer variable, contains the customer id that is retrieved from the FIELD_CUSTOMER_ID field defined in the FormConstants class. Next, the request is forwarded to the list page. If the form action is ACTION_QUERY_FORM, the doQueryAll method is invoked, which further calls the findAll() method of the session façade bean and stores the data in the List form. This data is set to the ATTRIBUTE_CUSTOMER_LIST attribute defined in the FormConstants class.

After creating and understanding the servlet required for the Customer-war module, let's now create the JSP pages. In this module, four JSP pages are designed namely: index, list, view, and customer.

Listing 14.36 provides the code for the index.jsp file (you can find this file on the CD in the `code\JavaEE\Chapter14\Customer\Customer-war` folder):

Listing 14.36: Showing the Code of the index.jsp File

```
<%@page import="com.kogent.servlet.*"%>
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Customer Database</title>
```

```
    </head>
    <body>
        <center><h1>Sample Customer Database</h1></center>

<p>This application highlights Java Persistence API and its usage from a Java Servlet to
  build a sample customer database. </p>

        <p>You can start the application and go to the list of customers by
        clicking the "Go" button below. </p>
        <hr/>
<form action="<jsp:expression>FormConstants.CONTROLLER
</jsp:expression>" method="post">
<input name="<jsp:expression>FormConstants.FIELD_ACTION
</jsp:expression>" type="HIDDEN"
        value="<jsp:expression>FormConstants.ACTION_QUERY</jsp:expression>">
        </input>
        <input name="Submit" type="submit" value="Go"/>
        </form>
        <hr/>
  </body>

</html>
```

In Listing 14.36, the index JSP page is designed to forward the request to the CustomerServlet class. The list JSP page is used to display the list of the customers along with their company names.

Listing 14.37 provides the code for the list.jsp file (you can find this file on CD in the `code\JavaEE\Chapter14\Customer\Customer-war` folder):

Listing 14.37: Showing the Code of the list.jsp File

```
<%@page import="java.util.List"%>
<%@page import="com.kogent.entity.Customer"%>
<%@page import="com.kogent.servlet.*"%>
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Customer List</title>
  </head>
  <body>
        <center><h1>Sample Customer List</h1></center>
   <p>Here is the list of customers. You can view more details on a customer by clicking
  the customer name.
        <hr/>
        <table>
                <tr bgcolor="#99CCCC">
                        <td><strong>Company Name</strong></td>
                        <td><strong>Customer Name</strong></td>
                </tr>
                <%
                List<Customer> customers =
(List<Customer>)request.getAttribute(FormConstants.
 ATTRIBUTE_CUSTOMER_LIST);
                if (customers == null || customers.size() == 0) {
                %>
                <tr>
                        <td colspan="2"><i>No customers</i></td>
                </tr>
                <%
                        } else {
                        for (Customer cust: customers) {
                        String actionUrl = UrlUtils.makeCustomerViewUrl(
                        cust.getCustomerId().toString());
```

```
                    %>
                    <tr>
                    <td><jsp:expression>cust.getCompany()</jsp:expression></td>
                    <td><a href="<jsp:expression>actionUrl</jsp:expression>">
<jsp:expression>cust.getFirstName() +  " " +  cust.getLastName()</jsp:expression>
                    </a>
                    </td>
                    </tr>
                    <%
                    }
                    }
                    %>
            </table>
            <hr/>
    </body>
</html>
```

In Listing 14.37, the list JSP page is designed. The required packages are imported and the `<jsp:expression>` tag is used to display customer name and company name in a table format. The `getCompany()` and `getFirstName()` methods are used to retrieve customer's name and the company's name.

Figure 14.20 displays the list JSP page:

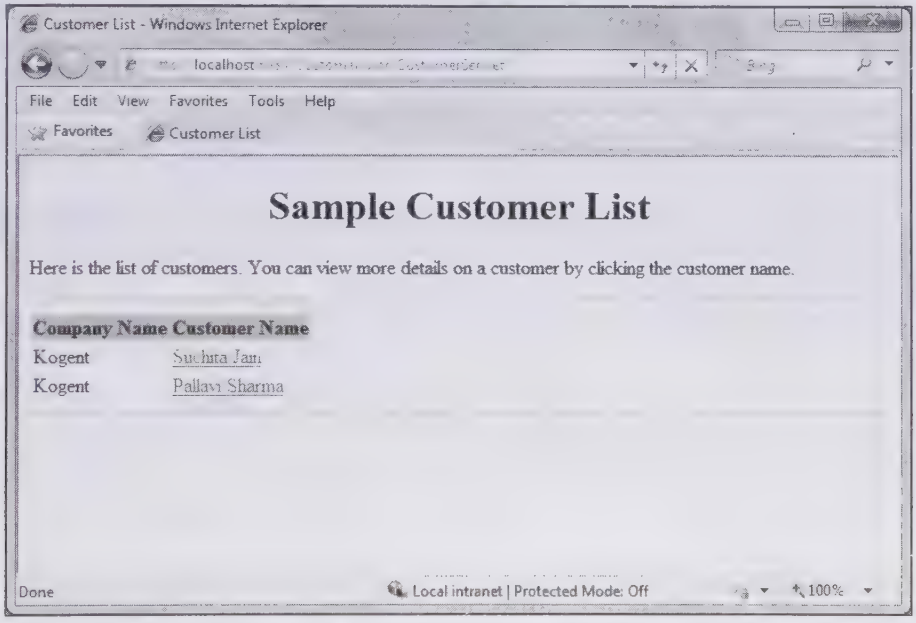


Figure 14.20: Showing the List of Customers

When the user clicks the name of a customer, the view page displays the details of the customer, such as address, city, and state of the customer.

Listing 14.38 provides the code for the view.jsp file (you can find this file on CD in the `code\JavaEE\Chapter14\Customer\Customer-war` folder):

Listing 14.38: Showing the Code of the view.jsp File

```
<%@page import="com.kogent.entity.Customer"%>
<%@page import="com.kogent.servlet.*"%>
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
```

```
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>View Customer</title>
    </head>
    <body>
        <center><h1>View Customer</h1></center>
<p>Here is the detailed information on the customer you selected.</p>
        <hr/>
        <%
    Customer customer = (Customer)request.getAttribute(FormConstants.ATTRIBUTE_CUSTOMER);
        %>
        <form method="post" action=
        "<jsp:expression>FormConstants.CONTROLLER</jsp:expression>">
        <%@include file="customer.jsp"%>
<input type="hidden" name="<jsp:expression>FormConstants.FIELD_ACTION</jsp:expression>"
value="<jsp:expression>FormConstants.
ACTION_QUERY</jsp:expression>"/>
        <input type="submit" name="Submit" value="Back"/>
        </form>
        <hr/>
<p><a href="<jsp:expression>FormConstants.CONTROLLER
</jsp:expression>">Return to the application top page</a></p>
        <hr/>
    </body>
</html>
```

When a customer name is clicked in the `list` JSP page, then the request is forwarded to the CustomerServlet class. The `CustomerServlet` class then forwards the request to the `view` JSP page. In Listing 14.38, the com.kogent.entity.Customer and com.kogent.servlet packages are imported. The `view` JSP page includes the `customer` JSP page, which retrieves the details of a customer from the `Customer` entity class. Figure 14.21 displays the details of the customer in a table form:

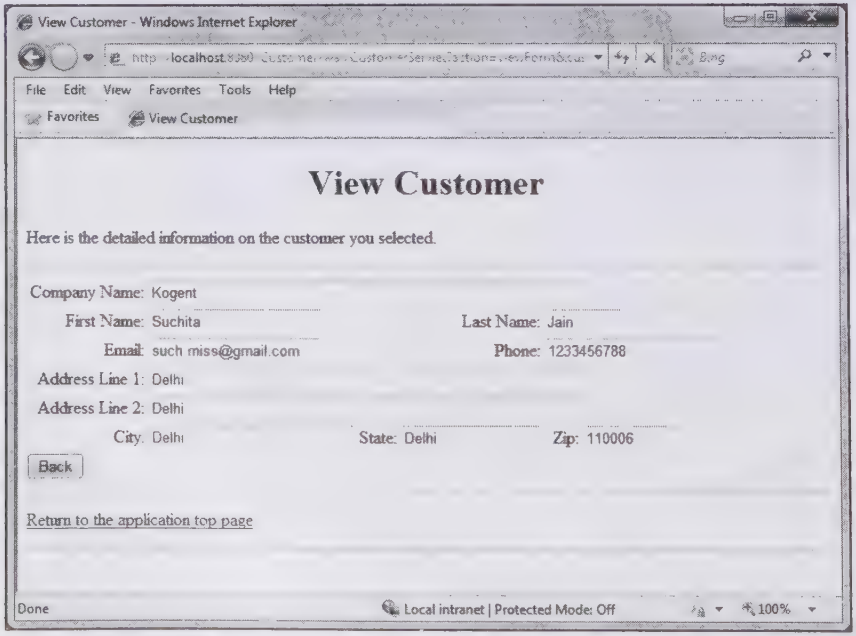


Figure 14.21: Showing the Customer Details

In Figure 14.21, the customer details, such as email address, customer address, city, and state are displayed. Listing 14.39 provides the code for the customer.jsp file (you can find this file on CD in the `code\JavaEE\Chapter14\Customer\Customer-war` folder):

Listing 14.39: Showing the Code of the customer.jsp File

```
<table>
  <tr>
        <td colspan="1" align="right">Company Name: </td>
        <td colspan="5">
<input type="text" name="<jsp:expression>FormConstants.FIELD_COMPANY</jsp:expression>"
        value="<jsp:expression>customer.getCompany()</jsp:expression>"
        size="60"/>
  </td>
  </tr>
  <tr>
        <td align="right">First Name: </td>
        <td colspan="2">
<input type="text" name="<jsp:expression>FormConstants.FIELD_FNAME</jsp:expression>"
        value="<jsp:expression>customer.getFirstName()</jsp:expression>"
        size="20"/>
        </td>
        <td align="right">Last Name: </td>
        <td colspan="2">
<input type="text" name="<jsp:expression>FormConstants.FIELD_LNAME</jsp:expression>"
        value="<jsp:expression>customer.getLastName()</jsp:expression>"
        size="20"/>
        </td>
  </tr>
  <tr>
        <td align="right">Email: </td>
        <td colspan="2">
<input type="text" name="<jsp:expression>FormConstants.FIELD_EMAIL</jsp:expression>"
        value="<jsp:expression>customer.getEmailAddress()</jsp:expression>"
        size="20"/>
        </td>
        <td align="right">Phone: </td>
        <td colspan="2">
<input type="text" name="<jsp:expression>FormConstants.FIELD_PHONE</jsp:expression>"
        value="<jsp:expression>customer.getPhoneNumber()</jsp:expression>"
        size="20"/>
        </td>
  </tr>
  <tr>
        <td colspan="1" align="right">Address Line 1: </td>
        <td colspan="5">
<input type="text" name="<jsp:expression>FormConstants.FIELD_ADDRESS1</jsp:expression>"
        value="<jsp:expression>customer.getAddress1()</jsp:expression>"
        size="60"/>
        </td>
  </tr>
  <tr>
        <td colspan="1" align="right">Address Line 2: </td>
        <td colspan="5">
<input type="text" name="<jsp:expression>FormConstants.FIELD_ADDRESS2</jsp:expression>"
        value="<jsp:expression>customer.getAddress2()</jsp:expression>"
        size="60"/>
        </td>
  </tr>
  <tr>
        <td align="right">City: </td>
        <td>
<input type="text" name="<jsp:expression>FormConstants.FIELD_CITY</jsp:expression>"
        value="<jsp:expression>customer.getCity()</jsp:expression>"
        size="25"/>
        </td>
        <td align="right">State: </td>
        <td>
<input type="text" name="<jsp:expression>FormConstants.FIELD_STATE</jsp:expression>"
        value="<jsp:expression>customer.getState()</jsp:expression>"
```

```
            size="15"/>
            </td>
            <td align="right">Zip:</td>
            <td>
    <input type="text" name="<jsp:expression>FormConstants.FIELD_ZIP</jsp:expression>"
            value="<jsp:expression>customer.getZip()</jsp:expression>"
            size="10"/>
            </td>
      </tr>
    </table>
```

In Listing 14.39, various fields and methods are used to display the details of the customer, such as customer address, city, state, zip, phone number, and email address. The servlet-mapping is done in the web.xml file and the code is provided as Listing 14.40 (you can find this file on CD in the `code/JavaEE/Chapter14/Customer/Customer-war/WEB-INF` folder):

Listing 14.40: Showing the Configuration File of the Customer Application

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" 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_3_0.xsd">
  <servlet>
        <servlet-name>CustomerServlet</servlet-name>
        <servlet-class>com.kogent.servlet.CustomerServlet</servlet-class>

  </servlet>

  <servlet-mapping>
        <servlet-name>CustomerServlet</servlet-name>
        <url-pattern>/CustomerServlet</url-pattern>
  </servlet-mapping>
  <session-config>
        <session-timeout>
              30
        </session-timeout>

  </session-config>

  <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

</web-app>
```

In the web.xml file, the CustomerServlet servlet is mapped and the index page is defined as the welcome page. The customer details are displayed if the JDBC resource is configured on the server.

Let's now configure the connection pool and the JDBC resource customer to run the application.

Configuring Connection Pool and JDBC Resource

You need to configure the JDBC resource, customer, for the `Customer` application. To store, organize, and retrieve data, most applications use relational database. A relational database can be accessed by the Java EE applications with the help of the JDBC API.

Let's create a new connection for the Customer application to access a database. To create a new connection pool, start the Glassfish V3 application server and open the Admin console by browsing http://localhost:4848 URL. Next, expand the Resources node from the left pane of the Admin console. Then select the JDBC node, which displays a hierarchy of nodes. Finally, select the Connection Pools node.

Figure 14.22 displays the list of existing connection pools:

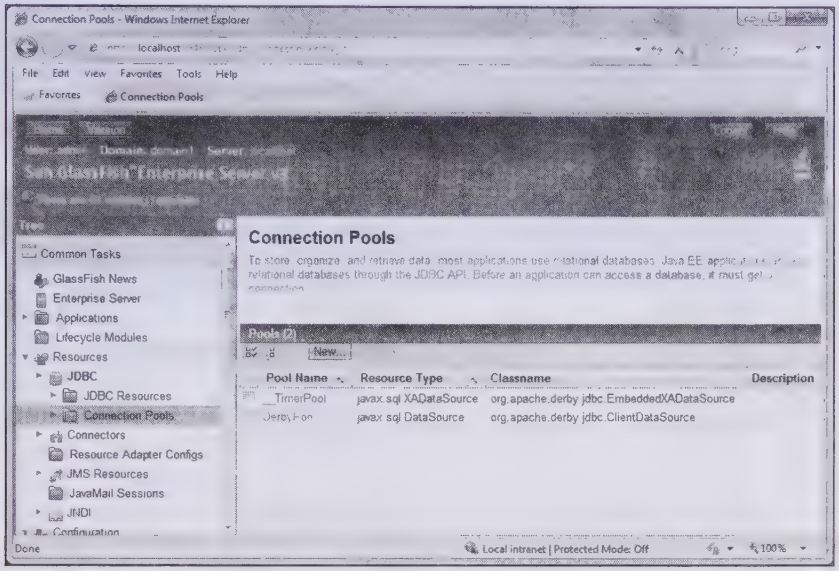


Figure 14.22: Showing the JDBC Connection Pools

To create a new connection pool, click the `New` button. As a result, the right pane displays general settings for configuring a new connection, as shown in Figure 14.23:

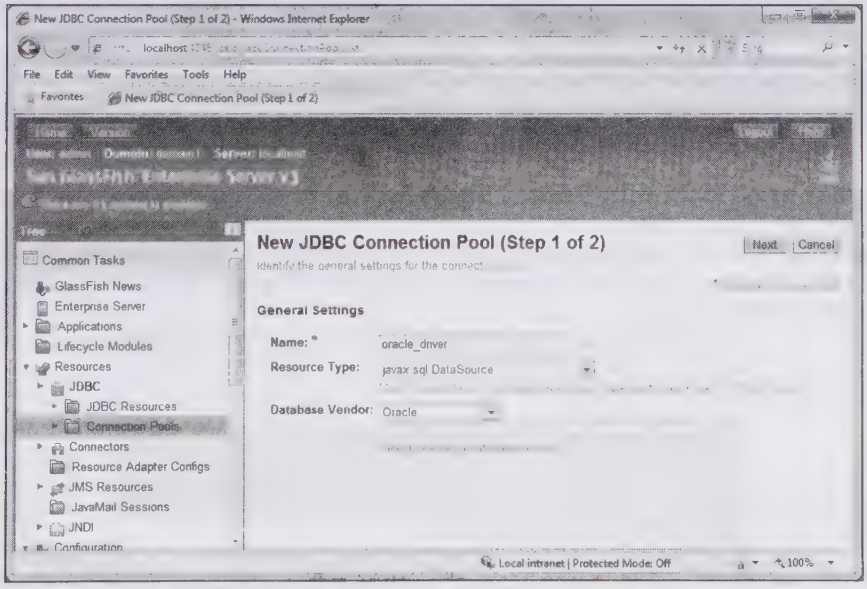


Figure 14.23: Showing the General Settings for New JDBC Connection Pool

In Figure 14.23, specify the name for the new connection pool; in our case, the name specified is `oracle_driver`. Then, select the `javax.sql.DataSource` option in the Resource Type field for the connection pool and Oracle as the Database Vendor, as shown in Figure 14.23. Now, click the `Next` button to provide further settings for this connection pool, as shown in Figure 14.24:

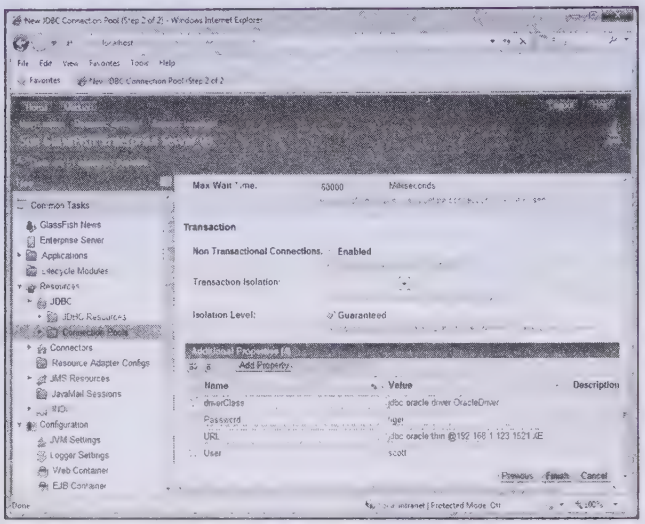


Figure 14.24: Showing the Values for Various Properties of New JDBC Connection Pool

Table 14.16 describes the properties to configure a new connection pool:

Table 14.16: Showing the Properties for JDBC Connection Pool

Property Name	Description	Value
`User`	Specifies the user name used to connect the database server	scott
`driverClass`	Specifies the driver class for the JDBC connection	jdbc.oracle.driver.OracleDriver
`URL`	Specifies the URL to access the JDBC connection	jdbc:oracle:thin:@192.168.1.123:1521:XE
`Password`	Specifies the password for the database connection	tiger

Now, click the `Finish` button (Figure 14.24) to create a new connection pool. After establishing a new connection pool, you need to create a new JDBC resource means to connect an application with a database. The JDBC resource used to connect the Customer application with the oracle data source is `customer`. To create the `customer` JDBC resource, expand the Resources node and click the JDBC Resources sub node. Figure 14.25 displays the list of JDBC resources:

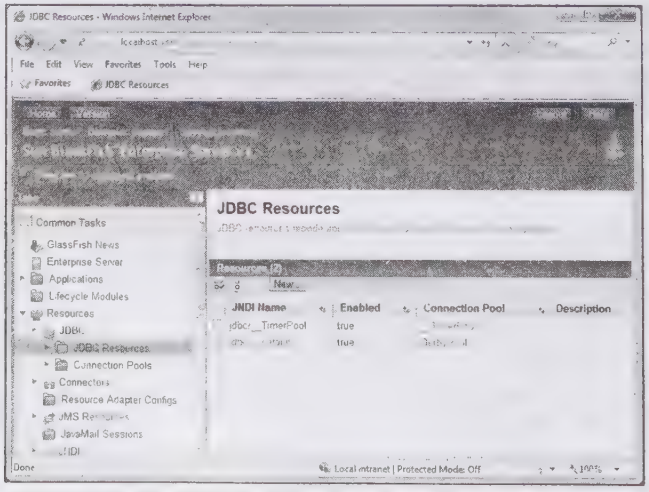


Figure 14.25: Showing the List of Existing JDBC Resources

In Figure 14.25, the existing JDBC resources are provided. Click the New button to create a new JDBC resource. The JNDI Name and the name of the Connection Pool are specified for a new JDBC resource, as shown in Figure 14.26:

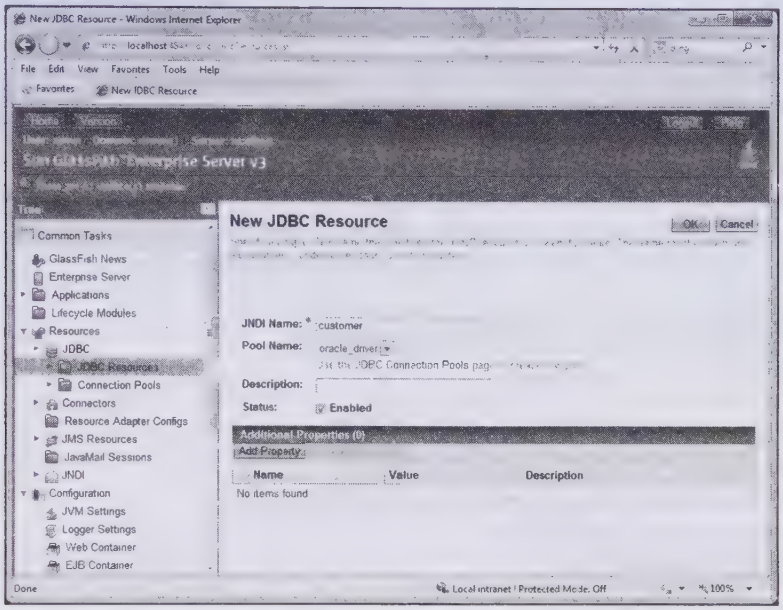


Figure 14.26: Configuring New JDBC Resource

In Figure 14.26, the JNDI name specified is `customer` and the pool name is the `oracle_driver`. Now, click the OK button to add the `customer` JDBC resource in list. After configuring the connection pool and JDBC resource for the `Customer` application, deploy and run the application by opening the http://localhost:8080/Customer/Customer-war URL.

Summary

This chapter introduced you to the concepts of ORM, JPA, and EntityManager. The chapter has also explained how to package and obtain a persistence unit in an application. You have also learned how to interact with a persistence unit using EntityManager. Next, has the chapter explored various types of entity relationships, such as one-to-one, one-to-many, and many-to-many. The concept of JPQL has also been explored by describing about JPQL statements, functions, and clauses. Further, the mapping of collection-based relationships has been discussed in the chapter. Towards the end, you learned to create the `Customer` enterprise application by configuring new connection pools and JDBC resources.

The next chapter discusses about the implementation of Java persistence using Hibernate 3.5.

Quick Revise

Q1. What is ORM?

Ans. The mapping of Java objects into a relational database is termed as ORM. The Java objects are saved in the tables available in the databases.

Q2. What is JPA?

Ans. JPA stands for Java Persistence API, which provides POJO standard and ORM for the maintaining persistent data among the applications.

Q3. What is the main difference between entity and session beans?

Ans. The session bean exists till the client or user runs the application; whereas, the entity bean exists even after the client or the user has closed the application, as the data is stored in the database.

Q4. List the stages of entity life cycle.

Ans. The following are the different stages of an entity life cycle:

- New
- Managed
- Detached
- Removed

Q5. What are entity listeners?

Ans. An entity listener is a stateless bean class that has non-argument constructor and the @EntityListeners annotation is used to specify it. The entity listeners are applied to entity classes.

Q6. Specify the elements and attributes of the persistence.xml file.

Ans. The following are the elements and attributes of the persistence.xml file:

- name
- transaction-type
- provider
- jta-data-source
- non jta-data-source
- mapping-file

Q7. What are the different ways of obtaining an EntityManager?

Ans. An EntityManager can be obtained by:

- Using Container Injection
- Using EntityManagerFactory
- Using JNDI

Q8. What are the different types of entity relationship?

Ans. The entities can have any of the following types of relationships:

- One-to-one
- One-to-many
- Many-to-one
- Many-to-many

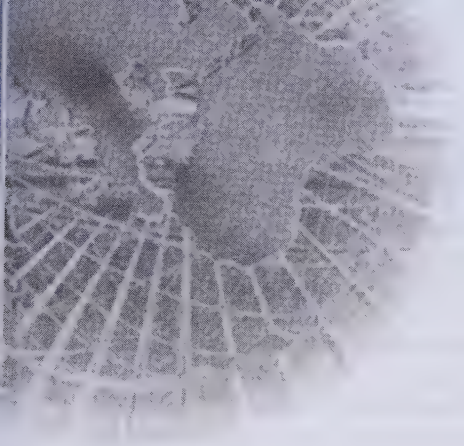
Q9. What are the different strategies for supporting the object-oriented concept of inheritance in relational database?

Ans. The different strategies for supporting the object-oriented concept of inheritance in relational database are as follows:

- Single table per class hierarchy
- Separate table per subclass
- Single table per concrete entity class

Q10. What is JPQL?

Ans. The full form of JPQL is `Java Persistence Query Language`, which is the extended version of the EJB-QL. JPQL operates on classes and objects (entities) available in the Java workspace.



15

Implementing Java Persistence Using Hibernate 3.5

Hibernate is an Object Relational Mapping (ORM) framework that is used to map an object-oriented domain model to a relational database. Initially, the developers used Structured Query Language (SQL) to query and retrieve data from a database. In such a scenario, the developers need to provide additional Java code in the application to handle the resultsets containing the data retrieved from the database.

With the introduction of the Hibernate framework, the JavaBean classes can now directly be mapped to the database tables, eliminating the need to provide the additional Java code in the applications to handle query results. The Hibernate framework automatically generates the SQL calls and ensures that a Web application is portable with all SQL supported databases. This framework also provides the persistence feature for Plain Old Java Objects (POJOs). You can use the Hibernate framework not only in a Java Platform, Standard Edition (J2SE) application but also in the Java EE applications using servlets or Enterprise JavaBeans (EJB) session beans.

The chapter discusses features and architecture of Hibernate. You also learn Hibernate Query Language (HQL), which is a very powerful and simple query language to retrieve data from databases. Next, you learn to create a simple Hibernate Web application by using create, read, update, and delete (CRUD) operations by using HQL instead of the Java DataBase Connectivity (JDBC) resultset.

Introducing Hibernate

The Hibernate framework is a lightweight ORM, which is a technique for mapping an object model to a relational model. This framework handles mapping from Java classes to database tables and provides Application Programming Interface (API) for querying and retrieving data from a database.

Earlier, JDBC and SQL were used to retrieve data from a database; however, writing queries using SQL to insert, retrieve, update, or delete data from a database was a time consuming process. This was simplified with the introduction of Hibernate that provides HQL. As compared to SQL, in HQL, you do not need to write code to perform common data persistence related programming tasks.

Hibernate uses a transparent programming model for mapping Java objects to the relational databases. To provide mapping for Java objects, you need to write a simple POJO class and create an Extensible Markup Language (XML) mapping file. This XML file describes the relationship of the database with the class attributes and calls the Hibernate APIs to load/store the persistent objects. Hibernate establishes the relationship between object-oriented systems and relational databases. It allows transparent persistence that enables the applications to use any Relational DataBase Management System (RDBMS) software.

Hibernate works efficiently with applications that use swings, servlets, and EJB session beans. The developer does not need to implement interfaces or extend classes for persistence, as Hibernate provides extensive support of Java collections API (Map, Set, List, SortedMap, SortedSet, and Collection).

After being familiar with the Hibernate framework, let's now try to understand the need Hibernate framework to develop Web application in the next section.

Why Hibernate?

The Hibernate framework provides an extensive approach to manage the object relational mapping and persistence management. The following are the reasons due to which Hibernate in a Web application can be used to query or retrieve data from a database:

- Supports object-oriented programming models, such as inheritance, polymorphism, composition, abstraction, and the Java collections framework.
- Provides developers with persistence feature and a code generation library (CGLIB) that helps in extending Java classes and implementing Java interfaces at runtime environment. The changes that are made to objects associated with a transaction are automatically addressed in the database. This saves the time spent in extra coding for bytecode processing.
- Provides object-oriented query language called HQL, which is similar to SQL. HQL is an ORM query language defined in EJB 3.0. It helps in writing multi-criteria search and dynamic queries.
- Provides Object/Relational mapping for bridging the gap between object-oriented systems and relational databases.

- Enables the developer to build a Hibernate Web application very efficiently in MyEclipse by using Hibernate eclipse plug-ins that provide mapping editor, interactive query prototyping, and schema reverse engineering tool.
- Reduces the development time, as it supports object-oriented programming, such as inheritance, polymorphism, composition, and Java Collection framework.
- Provides ID generator classes to generate surrogate keys also called unique keys/identifiers. Some of the classes are:
  - Native
  - Identity
  - Sequence
  - Increment
  - Hilo and seqhilo (HI/LO) algorithm
  - Universally Unique Identifier (UUID) algorithm
- Provides a Multi Level Cache Architecture that ensures thread safety and continuous data access.
- Provides features, such as lazy initialization and eager or batch fetching to improve the performance of an application.
- Provides integration with Java EE.
- Operates transactions in a managed or non-managed environment. It can run outside an application server container, within standalone applications. This facilitates the following tasks:
  - SessionFactory can be easily bound to Java Naming and Directory Interface (JNDI)
  - Stateful session bean or a HttpSession servlet with loadbalancing to handle a session
  - JDBC connections may be provided by application data source
  - Hibernate Transaction system integrates with Java EE applications through Java Transaction API (JTA)
  - Automatic binding of the Hibernate session to the global JTA transaction scope

Let's now see how the information flows in the Hibernate framework by understanding its architecture.

What is New in Hibernate 3.5?

Hibernate 3.5 edition introduces some new features that help to develop more efficient Hibernate applications. These features provide better support for JDBC 4.0, such as facilitating the use of annotations with EntityManager, providing support for column level read/write, and providing support for fetching profiles. Let's now discuss all these features in further subsections.

Annotations with EntityManager

Hibernate 3.5 facilitates the developers to use annotations and EntityManager together in an application. In earlier versions of Hibernate, developers were not able to use annotations and EntityManager together due to the following reasons:

- Different subversions
- Independent release cycles (different versioning)
- Different Java projects

Support for Read/Write Columns

To update data in an application, you need to implement the read-write strategy in the application. Whenever serializable transaction isolation level is required then, read-write strategy should not be used. Whenever a cache is used in a JTA environment, then you have to define a value for the hibernate.transaction.manager_lookup_class property and also needed to specify a naming strategy that would be used to obtain the JTA transaction manager. However, in other environment you have to confirm the completion of transaction when the Session.close() or Session.disconnect() method is executed. In case if you

want to use this strategy in a cluster, then you have to ensure that the cache implementation that is used, supports locking.

Support for Fetch Profiles

If the database supports American National Standards Institute (ANSI), Oracle or Sybase style outer joins, the use of outer joins to retrieve records often increases the execution performance by reducing the continuous flow of data to and from the database. The use of the outer join allows you to use many-to-one, one-to-many, many-to-many, and one-to-one associations to retrieve the records from database in a single SQL SELECT query.

You can disable the outer join fetching by setting the hibernate.max_fetch_depth property to 0. You can enable outer join fetching for one-to-one and many-to-one associations that have been mapped with fetch="join" by setting 1 or higher for the hibernate.max_fetch_depth property. Hibernate also provides the fetching strategy that facilitates searching an association in an application. This strategy can be used to fetch the records from a database and can be declared in the O/R mapping metadata or overridden by a particular HQL or Criteria query.

Exploring the Architecture of Hibernate

The Hibernate architecture consists of two technologies— Hibernate and Java EE. Hibernate makes use of the existing Java APIs, including JDBC, JTA, JNDI, JDBC loads database driver and establishes a connection for accessing relational database. It uses all types of database drivers that are supported by Hibernate. JNDI and JTA allow Hibernate to be integrated with application servers.

Figure 15.1 shows the architecture of Hibernate:

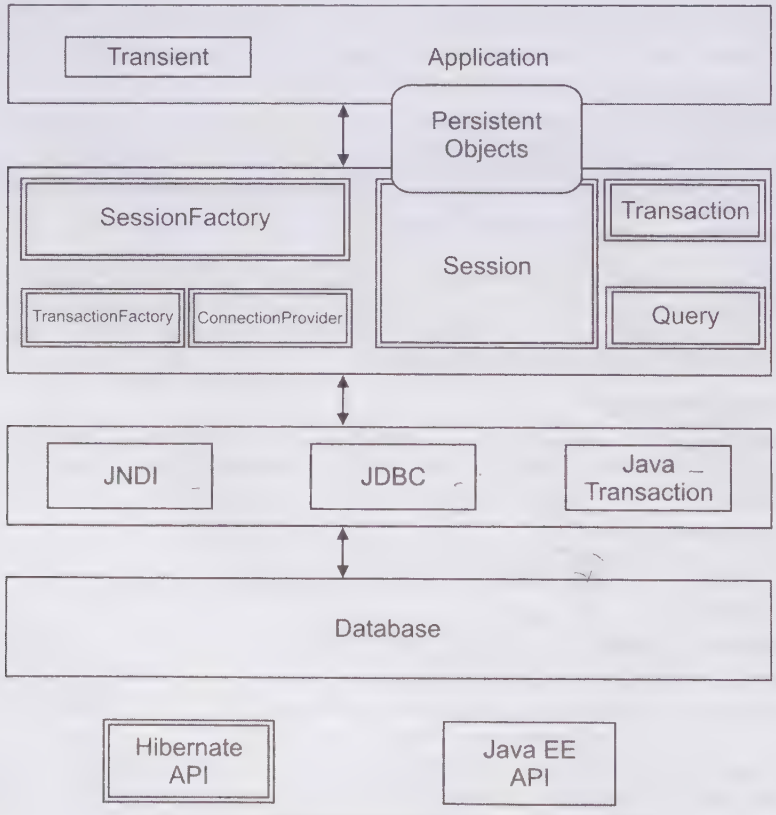


Figure 15.1: Displaying the Hibernate Architecture

Now, let's explore the important interfaces of the Hibernate architecture in detail.

Noteworthy Interfaces of Hibernate

Hibernate provides various interfaces, such as org.hibernate and org.hibernate.cfg, to develop applications. Some of the important interfaces which are necessary to understand the architecture of Hibernate are:

- **Session interface (org.hibernate.Session)** – Maintains a session between a connection and transaction. All applications need to maintain a session for a particular process. Sessions are lightweight and thread safe and can be created or destroyed several times in an application. A session caches all the loaded objects in a session and can keep track of any changes made to these objects. In this way, it maintains the persistence of these objects and is called the Persistence manager.

The org.hibernate.Session interface represents a Hibernate session and abstracts the notion of a persistence service. The Hibernate session performs mainly CRUD operations for objects of the mapped entity classes.

- **SessionFactory interface (org.hibernate.SessionFactory)** – Creates a session factory interface during application initialization. The SQL statements and other mapping metadata that Hibernate uses at runtime are cached by the SessionFactory interface. It holds the cached data that has been read in the current operation and may be further reused in other operations.
- **Configuration interface (org.hibernate.cfg.Configuration)** – Allows you to configure the `Configuration` object. A configuration instance is used to specify the location of the configuration files, such as Hibernate-specific properties. Further, the configuration instance is used to create the SessionFactory object.
- **Query and Criteria interfaces** – Helps to execute a query. The developer needs to obtain an instance of one of these interfaces to query the database:
  - **Query interface** – Allows the developer to perform queries against the database and control how the query is executed. Queries are written in HQL or in the native SQL. A Query instance binds concrete values to query parameters, limits the number of results returned by the query, and finally executes the query. A Query instance is lightweight in nature and can not be used outside the session that creates this instance.
  - **Criteria interface** – Enables the developer to create and execute the object-oriented criteria queries.
- **Transaction interface (org.hibernate.Transaction)** – Refers to an optional interface. This interface is used to create a Hibernate Web application that is portable among different kinds of Web containers and execution environments. It provides a consistent API that can control the transaction boundaries. A Transaction abstracts application code from the transaction implementation code that is used in application. The transaction can be a JDBC transaction, a JTA user transaction, or even a Common Object Request Broker Architecture (CORBA) transaction.

The Hibernate Cache Architecture

Hibernate has a dual-level cache architecture. A cache helps to keep the current database state (local copy of the data) available to the application whenever a lookup is performed or lazy initialization is needed. The elements of Hibernate cache architecture are as follows:

- First-level cache
- Second-level cache

The dual-level architecture of Hibernate provides the following features:

- **Thread safeness** – Ensures thread safety by providing a unique set of persistent instances for each session
- **Non-blocking data access** – Provides a continuous data access by avoiding threads in the wait state throughout an application

The various elements of the dual-level cache architecture can be seen in Figure 15.2:

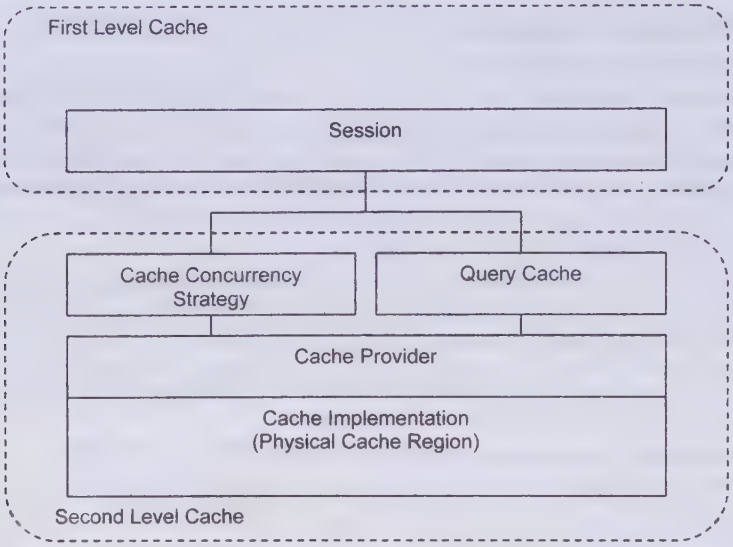


Figure 15.2: Displaying the Hibernate Dual-level Cache Architecture

Let's now discuss about both first-level and second-level cache.

First-Level Cache

The first-level cache is always associated with the session object. It is active for a single user that handles a database transaction or an application transaction. The first-level cache is mandatory and cannot be turned off. It also maintains the object identity inside a transaction and helps in avoiding database traffic. The benefits of first-level cache are as follows:

- Can be used with repeated requests for the same instance in a particular session.
- Can be used in O/R mapping, which facilitates mapping a database row to a single object. This results in updating the changes made to the object in the database.

When the operations, such as save() or update() are used on an object or if an object is retrieved using the load(), find(), list(), iterate(), or filter() method, then that object is added to the session cache.

To synchronize the object with the database, the flush() method can be used. In order to manage memory and avoid synchronization, the evict() method can be used. The evict() method removes the object (invokes the evict() method) and its data from the first-level cache. The Session.clear() method completely removes all the objects from the session cache.

Second-Level Cache

The second-level cache in Hibernate has process scope cache or cluster scope cache. The process scope cache stores either the persistent context or persistent state of the processes in a disassembled format. The cluster scope cache shares multiple processes on the same machine or between multiple machines in a cluster. In the second-level cache, network communication is very important and requires some kind of remote process communication to maintain consistency of the processes among multiple machines. The second-level cache stores the persistent instances in the disassembled form, which is similar to the deserialization of an object (save the state of an object) in Java programming. The cache policies include caching strategies and physical cache providers, which are used on the basis of various factors, such as the ratio of reads to writes, the size of the database tables, and the number of tables shared with external applications. The second-level cache can be used for both immutable and mutable data. The Hibernate second-level cache can be implemented in an application by performing the following steps:

- Selecting the concurrency strategy that can be used in the application
- Configuring cache expiration and physical cache attributes using the cache provider

According to the cache policy, you need to set up the following aspects in an application:

- The Hibernate concurrency strategy
- The cache expiration policies, such as timeout, least recently used (LRU), and memory-sensitive
- The physical format of the cache (memory, indexed files, and cluster-replicated)

While configuring Hibernate in an application, the caching strategy to be used can be defined by implementing the org.hibernate.cache.CacheProvider interface and setting the value for the hibernate.cache.provider_class property.

Caching Strategies

You can use either of the following caching strategies in an application to retrieve the data from a database:

- **Read-only**—Refers to the strategy in which the data is frequently read but never updated. This is simple and safe to use in a cluster.
- **Read/write**—Refers to the strategy in which the data is to be read as well as updated. This type of cache is used in non-JTA environments, where the completion of each transaction should be done before the invocation of the `Session.close()` or `Session.disConnect()` methods.
- **Nonstrict read/write**—Refers to the strategy in which an application sometimes needs to update the data along with reading a large amount of data. This strategy can also be used to restrict two or more transactions from modifying the same data simultaneously.
- **Transactional**—Refers to a fully transactional cache that can only be used in a JTA environment.

Query Cache

Hibernate also implements a cache for query resultsets that integrate closely with the second-level cache. This is an optional feature and is useful for applications involving few insert, delete, or update operations. The query cache caches just the identifier values and not the returned query results. To enable the query cache, the hibernate.cache.use_query_cache property is set to true.

Hibernate uses the timestamp cache to decide whether or not the resultset should be cached. The most recent resultset obtained by executing the insert, update, or delete queries is cached. If the timestamp of a resultset query is more than the timestamp of the cached query results, then the cached results are discarded and a new query is issued.

Cache Provider

Hibernate provides various cache providers from which you can select the appropriate cache provider for your application. It also supports some of the following open source cache providers:

- **ECHCache**—Supports the read-only, nonstrict read/write, and read/write caching strategies. This cache provider is used for the simple process scope cache in a single Java Virtual Machine (JVM). This type of cache provider can be in memory or on disk.
- **OpenSymphony cache (OSCache)**—Enables caching to memory as well as disk in a single JVM with a query cache. The caching strategy, such as read-only, nonstrict read/write, and read/write, are supported by the OSCache cache provider.
- **JBoss Cache**—Refers to a fully transactional replicated clustered cache that is based on JGroup, which is a communication protocol that is purely written in Java and used to create groups of processes whose members can send messages to each other. It supports read-only and transactional caching strategies.

Downloading Hibernate

Hibernate is a free open source software, which can be downloaded from http://www.hibernate.org/. Visit http://www.hibernate.org/ and then click the 'Download' link to go to the download page. Click the 'distribution bundles' link from the download page to download the current latest release of the Hibernate Core (hibernate-distribution-3.5.2-Final-dist.zip, (Hibernate 3.5)).

Let's now understand one of the most powerful features of Hibernate— HQL.

Exploring HQL

HQL is an easy-to-learn and powerful query language designed as an object-oriented extension to SQL that bridges the gap between the object-oriented systems and relational databases. The HQL syntax is very similar to the SQL syntax. It has a rich and powerful object-oriented query language available with the Hibernate ORM, which is a technique of mapping the objects to the databases. The data from object-oriented systems are mapped to relational databases with a SQL-based schema.

HQL queries are case insensitive; however, the names of Java classes and properties are case-sensitive. HQL is used to execute queries against database. If HQL is used in an application to define a query for a database, the Hibernate framework automatically generates the SQL query and executes it. Unlike SQL, HQL uses classes and properties in lieu of tables and columns. HQL supports polymorphism as well as associations, which in turn allows developers to write queries using less code as compared to SQL. In addition, HQL supports many other SQL statements and aggregate functions, such as sum() and max() and clauses, such as group by and order by.

HQL can also be used to retrieve objects from database through O/R mapping by performing the following tasks:

- Apply restrictions to properties of objects
- Arrange the results returned by a query by using the order by clause
- Paginate the results
- Aggregate the records by using group by and having clauses
- Use Joins
- Create user-defined functions
- Execute subqueries

Need of HQL

As already discussed, it is more beneficial to use HQL instead of native SQL to retrieve data from databases. The following are some of the reasons why HQL is preferred over SQL:

- Provides full support for relational operations. It is possible to represent SQL queries in the form of objects in HQL, which uses classes and properties instead of tables and columns.
- Returns results as objects. In other words, the query results are in the form of objects rather than the plain text. These objects can be used to manipulate or retrieve data in an application. This eliminates the need of explicitly creating objects and populating the data from the resultset retrieved from the execution of a query.
- Supports polymorphic queries. Polymorphic queries return the query results along with all the child objects (objects of subclasses), if any.
- Easy to learn and use. The syntax of HQL is quite similar to that of SQL. This makes Hibernate queries easy to learn and implement in the applications.
- Support for advanced features. HQL provides many advanced features as compared to SQL, such as pagination, fetch join with dynamic profiling (initialization the associations or collection of values with their parent objects can be done using a single select statement), inner/outer/full joins, and Cartesian product. It also supports projection, aggregation (max, avg), grouping, ordering, sub queries, and SQL function calls.
- Provides database independency. HQL helps in writing database independent queries that are converted into the native SQL syntax of the database at runtime. This approach helps to use the additional features that the native SQL query provides.

Now, let's learn about the HQL syntax and its usage.

HQL Syntax

HQL is considered to be the most powerful query language designed as a minimal object-oriented extension to SQL. HQL queries are easy to understand and use persistent class and property names, instead of table and column names. HQL consists of the following elements:

- Clauses
  - From
  - Select
  - Where
  - Order by
  - Group by
- Aggregate functions
- Subqueries

Clauses in HQL

Clauses are HQL keywords that make up queries. HQL allows you to express a SQL query in object-oriented terms, i.e. by using classes and their properties.

The from Clause

The from clause is the simplest form of HQL query used with the select statement that specifies the object whose instances are to be returned as the query result. The syntax of the from clause is:

```
from object [as object_alias]
```

In the preceding syntax, the object_alias refers to an object.

Consider the following code snippet which depicts the use of the from clause:

```
from org.kogent.baseclass.User
or
from User
```

In the preceding code snippet, the from clause is used to retrieve all the instances of the org.kogent.baseclass.User class.

If there is a need to refer to the `User` class in other parts of the query, an alias name can be assigned to the class by using the following code snippet:

```
from User as user
```

The preceding code snippet assigns the alias name user to the User instances. The alias name user can be used as reference in other queries for the User class. The as keyword is optional, so the preceding code snippet can be written as follows:

```
from User user
```

In case of cross join, you can use the following code snippet to assign an alias name for multiple tables:

```
from User, Group
from User as user, Group as group
```

In the preceding code snippet, the user alias name is assigned for the `User` class and the group alias name is assigned for the `Group` class.

The select Clause

The select clause refers to the objects and properties returned as query resultset. The returned properties may be of any value type, including the properties of the component type. The syntax of the select clause is shown in the following code snippet:

```
select [object.property]
```

The following code snippet shows the use of the select clause along with the from clause:

```
select user.name from User user
or
select customer.contact.firstName from Customer as cust
```

In the preceding code snippet, the execution of the first query returns the names of all the users from the details stored in the User table. The next query returns the firstname from the details of the customers stored in the Customer table.

The where Clause

The where clause is used to specify a condition based on which the resultset is obtained from a database. The syntax of the where clause is:

```
where condition
```

In the preceding syntax, condition refers to a combination of logical and relational operators, such as >, <, AND, and NOT used to specify the type of value you want to retrieve from the database. The where clause can be used with the select and/or select, and/or, and from clause.

The following code snippet returns the details of the user whose name is mary:

```
from User as user where user.name='mary'
```

Compound path expressions can also be used with the where clause to get classified results. Consider the following example in which the cust.contact.name expression is used to retrieve the details of a customer whose first name is not null in the database:

```
from org.applabs.base.Customer cust where cust.contact.name is not null
```

The preceding example results in an SQL query with a table (inner) join condition. In the where clause, the = operator can be used to compare properties and instances, as shown in the following code snippet:

```
from Document doc, User user where doc.user.name = user.name
```

In the preceding code snippet, the value of an expression (doc.user.name) is assigned with the help of the user.name expression. Instead of using an expression for assigning a value, you can directly provide a unique value, as shown in the following code snippet:

```
from Document as doc where doc.id = 131512
or
from Document as doc where doc.author.id = 69
```

In the preceding code snippet, a unique value is assigned for the id on the basis of which the data is retrieved.

The order by Clause

The order by clause is used to order (ascending/descending) the properties of objects that are returned as query resultset. It is used with the select and from clauses. The syntax of the order by clause is shown in the following code snippet:

```
Order by object1.property1 [asc|desc] [object1.property2]
```

By default, the order is ascending.

The following code snippet shows the use of the order by clause:

```
from User user order by user.name asc, user.creationDate desc, user.email
```

The group by Clause

The group by clause is used to group the object properties (returned as query resultset) on a specific criteria. This clause is used with the select, and/or, and from clause. The syntax of the group by clause is:

```
Group by object1.property0, object1.property1
```

The following code snippet shows the use of the group by clause:

```
select dept.emp_no from department as dept group by dept.mgr
```

The preceding code snippet returns a list of all the emp_no instances from the department group corresponding to the values of the mgr instances.

The `having` clause is used with either the select and from clause or the select or from clause, as shown in the following code snippet:

```
select sum(document) from Document document group by document.category
having document.category in (Category.HIBERNATE, Category.STRUTS)
```

Associations and Joins

Join is used to assign aliases to associated entities or elements of a collection of values. The join types supported by HQL are as follows:

- Inner join
- Left outer join
- Right outer join

- Full join

Aggregate Functions

HQL queries use aggregate functions to perform mathematical operations in a query to retrieve the desired output from the database. You can use these functions along with the `distinct` or `all` option. The supported aggregate functions are as follows:

- avg(...)
- sum(...)
- min(...)
- max(...)
- count()

There are four variant or subfunctions of count() function that are given as follows:

- count(*)
- count(...)
- count(distinct ...)
- count(all...)

Expressions

Expressions are used with the where clause to extract some rows from the database table. The where clause uses the following expressions:

- Mathematical operators—+, -, *, /
- Binary comparison operators—=, >=, <=, <>, !=
- Logical operators—and, or, not
- String concatenation—||
- SQL scalar functions— upper() and lower()
- Parentheses indicate grouping —( )
- In, between, is null
- JDBC IN parameters?
- Named parameters—:name, :end_date, : x1
- SQL literals—'abc', 60, '1990-01-01 10:00:01.0'
- Java public static final constants, such as Color.LOBBY

Subqueries

A query within a query is known as a subquery and is surrounded by parentheses. Hibernate supports creation of a subquery (a query within another query), which is an important and powerful feature of SQL. An example of a subquery is subselect in which a select clause is embedded in another clause, such as select, from, and where clause.

You should note that a subquery is executed before the execution of the main query. You can use the following quantifiers in HQL:

- any
- all
- some (a synonym for any)
- in (similar to the = operator)

For example, the following query returns items where all bids are less than 100:

```
from Item item where 100 > all ( select b.amount from item.bids b )
```

To retrieve the items with bids greater than 100, you can use the following query:

```
from Item item where 100 < any ( select b.amount from item.bids b )
```

Now, let's understand the concept of O/R mapping that provides mapping between the object and database table.

Understanding Hibernate O/R Mapping

ORM is a technique to map object-oriented data with the relational data. In other words, it is used to convert the datatype supported in object-oriented programming language to a datatype supported by a database. This facilitates the transfer of object-oriented data to the database without the occurrence of an exception due to incompatibility of datatypes.

ORM is also known as O/RM, ORM, and O/R mapping. ORM involves mapping of the object-oriented data to relational data. Mappings should be in a format that can define the mapping of the classes with tables, properties with columns, foreign keys with associations, and SQL types with Java types. The document should be in such a mapping format that it can be easily read and edited. ORM is used to reduce programming code and map the object-oriented programming objects with relational databases that can be oracle, DB2, MySql, Sybase, and any other relational database software. The programming code can be reduced through caching, and by using an embedded SQL or call-level interface, such as JDBC. XML documents are used to define O/R mapping. They can be easily manipulated by version-control systems and text editors, and may be customized at deployment time or runtime. XML mapping documents allows you to:

- Bridge the gap between object-oriented systems and relational databases
- Define the object-relational mapping
- Help to generate and export database table
- Use XDoclet support that allows mappings to be generated from javadoc-style source comments

In complex applications, a class can be inherited by other classes, which may lead to the problem of class mismatching. To overcome such problems, the following approaches can be used:

- **table-per-class-hierarchy** – Removes polymorphism and inheritance relationships completely from the relational model
- **table-per-subclass (normalized mapping)** – Denormalizes the relational model and enables polymorphism
- table-per-concrete-class: Represents inheritance relationships as foreign key relationships

Multiple-objects can be mapped to a single row. Polymorphic associations for the three mapping strategies are as follows:

- **Many-to-one** – Serves as an association in which an object reference is mapped to a foreign key association
- **One-to-many** – Serves as an association in which a collection of objects is mapped to a foreign key association
- **Many-to-many** – Serves as an association in which a collection of objects is transparently mapped to a match table
- **One-to-one** – Serves as an association in which an object reference is mapped to a primary key or unique foreign key association
- **Ternary** – Serves as an association in which a Map contained and keyed by objects is mapped to a ternary association (with or without match table)

Indexes collections, such as Maps, Lists, and Arrays can be maintained and persisted as key-value pairs. Mappings are done using persistent class declarations and not by table declarations. The example of Hibernate XML mapping file is shown in the following code snippet:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">------DTD
                                                              declaration
<hibernate-mapping>----Mapping declaration
<class
name="com.kogent.hibernate.Employee" table="EMPLOYEE">---Class Mapped to
                                                                  table
<id name="id" column="EMP_ID " type="long">----Identifier Mapped
```

```
<generator class="native"/>
</id>
<property name="name" column="NAME" type="string"/>----Property Mapped
</class>
</hibernate-mapping>
```

The preceding code snippet shows a Hibernate mapping file in which the Employee class is mapped to the EMPLOYEE table. The important elements of the hibernate.cfg.xml Hibernate mapping file are as follows:

- **DOCTYPE**—Refers to the Hibernate mapping Document Type Declaration (DTD) that should be declared in every mapping file for syntactic validation of the XML.
- **<hibernate-mapping> element**—Refers as the first or root element of Hibernate. Inside the <hibernate-mapping> tag, any number of class elements may be present. The attributes of the <hibernate-mapping> tag are as follows:
  - **schema (optional)**—Specifies the name of a database schema.
  - **catalog (optional)**—Specifies the name of a database catalog.
  - **default-cascade (optional-defaults to none)**—Specifies a cascade style.
  - **default-access (optional-defaults to property)**—Defines a strategy by using which Hibernate can access all the properties. In addition, this strategy can be a custom implementation of the PropertyAccessor interface.
  - **default-lazy (optional-defaults to true)**—Specifies the default value for lazy attributes of class and collection mappings.
  - **auto-import (optional-defaults to true)**—Specifies whether unqualified class names (of classes in this mapping) can be used in the query language.
  - **package (optional)**—Specifies a default package prefix so that the unqualified class names can be located in the mapping document.

The following code snippet shows the syntax of the <hibernate-mapping> element:

```
<hibernate-mapping
   schema="schemaName"
   catalog="catalogName"
   default-cascade="cascade_style"
   default-access="field|property|ClassName"
   default-lazy="true|false"
   auto-import="true|false"
   package="package.name"
 />
```

- **<class> element**—Maps a class object with its corresponding entity in the database. It also specifies the table name in the database that has to be accessed along with the column of that table. Within one <hibernate-mapping> element, several <class> mappings are possible. All attributes of the class element and their description are described as follows:
  - **name (optional)**—Specifies the fully qualified Java class name of the persistent class (or interface). In the absence of this attribute, it is assumed that the mapping is for a non-POJO entity.
  - **table (optional)**—Specifies the name of its database table.
  - **discriminator-value (optional)**—Specifies a value. This value defines polymorphic behavior to distinguish individual subclasses. Its values are null and not null.
  - **mutable (optional, defaults to true)**—Specifies whether instances of the class are mutable or immutable.
  - **schema (optional)**—Overrides the schema name that is defined by the root element (<hibernate-mapping>).
  - **catalog (optional)**—Overrides the catalog name that is defined by the root element (<hibernate-mapping>).
  - **proxy (optional)**—Specifies the name of the interface to use for lazy initializing proxies. The name of the class itself may be specified.

- **dynamic-update (optional, defaults to false)**—Specifies that generation of UPDATE SQL should be done at the runtime and contains only those columns whose values have been changed.
- **dynamic-insert (optional, defaults to false)**—Specifies that generation of INSERT SQL should be done at the runtime and contains only those columns whose values are not null.
- **select-before-update (optional, defaults to false)**—Specifies that SQL UPDATE operation can not be performed by Hibernate, until the object is not modified. In some cases (when a transient object become associated with a new session by using update() method), Hibernate should perform an extra SQL SELECT that determines whether an UPDATE operation is required or not.
- **polymorphism (optional, defaults to implicit)**—Specifies values to determine whether implicit or explicit query polymorphism is used.
- **where (optional)**—Specifies an arbitrary SQL WHERE condition to be used when retrieving objects of this class.
- **persister (optional)**—Specifies a custom ClassPersister interface. Refers to a custom interface of ClassPersister interface.
- **batch-size (optional, defaults to 1)**—Specifies a batch size for fetching instances of the class implements ClassPersister interface.
- **optimistic-lock (optional, defaults to version)**—Specifies the optimistic locking strategy.
- lazy (optional): Disables the lazy fetching by setting lazy="false".
- **entity-name (optional)**—Allows multiple mapping of a class (a class to multiple tables). It also allows entity mappings between Bean classes and databases.
- **check (optional)**—Specifies the SQL expression. This attribute is also used to generate a multi row check constraint whenever the schema generation is automatic.
- **rowid (optional)**—Specifies the physical location of a stored tuple. If this option is set to true, Hibernate can use ROWIDs on databases (Oracle) for fast updates.
- **subselect (optional)**—Maps an immutable and read-only entity to a database subselect. This is useful when there is a need to have a view instead of a base table.
- **abstract (optional)**—Marks abstract superclasses in <union-subclass> hierarchies.

The following code snippet shows the syntax of the <class> element:

```
<class
   name="ClassName"
   table="tableName"
   discriminator-value="discriminator_value"
   mutable="true|false"
   schema="owner"
   catalog="catalog"
   proxy="ProxyInterface"
   dynamic-update="true|false"
   dynamic-insert="true|false"
   select-before-update="true|false"
   polymorphism="implicit|explicit"
   where="arbitrary sql where condition"
   persister="PersisterClass"
   batch-size="N"
   optimistic-lock="none|version|dirty|all"
   lazy="true|false"
   entity-name="EntityName"
   check="arbitrary sql check condition"
   rowid="rowid"
   subselect="SQL expression"
   abstract="true|false"
   node="element-name"
/>
```

- **<id> element**—Serves as a unique identifier used to map primary key column of database table. The attributes of the id element are as follows:
  - **name**—Specifies the property name that is used by the persistent class.
  - **column**—Specifies the column, which is used to store the primary key value.
  - **type**—Specifies the Java data type used.
  - **unsaved-value**—Specifies the value that is used to determine if a class has been made persistent. The null value for the id attribute indicates that the object has not been persisted.
  - **access (optional-defaults to property)**—Specifies the strategy Hibernate used for accessing the property value.

The following code snippet shows the syntax of the <id> element:

```
<id
   name="propertyName"
   type="typename"
   column="column_name"
   unsaved-value="null|any|none|undefined|id_value"
   access="field|property|ClassName">
   node="element-name|@attribute-name|element/@attribute|."
   <generator class="generatorClass"/>
</id>
```

- **<generator> element**—Helps in generation of the primary key for the new record. The following are some of the commonly used generators:
  - **Increment**—Helps to generate primary keys of the long, short, and int type. It should not be used in the clustered deployment environment. This generator takes the maximium primary key column value from the table and increments this value by one each time when a row is inserted. It returns an identifier of type long, short, or int. This generator is useful in case when the Hibernate Web application has exclusive access to the database and it can not be used in other scenarios.
  - **Sequence**—Generates the primary key. It can be used with DB2, PostgreSQL, Oracle, and SAP DB databases.
  - **Assigned**—Requires invocation when application code generates the primary key.
  - **Identity**—Returns an identifier of type long, short, or int. In addition, this is used for identity columns in DB2, MySQL, MS SQL Server, Sybase, and HypersonicSQL.
  - **hilo (A high/low algorithm)**—Refers to an algorithm that helps to generate identifiers of int, long, or short type that are given in a table and its column. By default these identifiers are hibernate_unique_key and nrxt_hi values and acts as a source of hi value. The identifiers that are generated by hilo are unique for a particular database only.
  - **uuid**—Allows you to use 128-bit UUID to return identifier of the CHAR type. 128-bit UUID algorithm is used to generate String type identifiers and these identifiers are unique with in a network. The IP address is used in combination with a unique timestamp. The UUID attribute is encoded in a hexadecimal digit of length 32 bit.
  - **Native**—Selects any one of the identity generators, such as sequence or hilo, depending on the database.
  - **Guid**—Returns a String MS SQL Server and MySQL.

The following code snippet shows the syntax of the <generator> element:

```
<id name="id" type="long" column="cat_id">
   <generator class="org.hibernate.id.TableHiLoGenerator">
         <param name="table">uid_table</param>

         <param name="column">next_hi_value_column</param>
   </generator>
</id>
```

The developer may create a user-defined identifier generator by implementing Hibernate's IdentifierGenerator interface.

- **<property> element**—Defines standard Java attributes and their mapping into database schema. This element supports the column child element that specifies additional properties, such as the index name given on a column or a specific column type. The property name of the String type is mapped to a database column name. The attributes of the <property> element are described as follows:
  - **name**—Specifies the name of the property, started with a lowercase letter.
  - **column (optional-defaults to the property name)**—Specifies the name of the mapped database table column.
  - **type (optional)**—Specifies the Hibernate type.
  - **update, insert ()**—Refers to optional attributes, whose values are by default true. These attributes specified that all the mapped columns should be included in the UPDATE and/or INSERT statements of a SQL query.
  - **formula (optional)**—Defines a value for stored attribute, and this expression is specified as an arbitrary SQL expression.
  - **not-null (optional)**—Enables a column to have a null or not null constraint.
  - **unique (optional)**—Enables a column to have a unique constraint.
  - **lazy (optional-defaults to false)**—Specifies that lazy fetching of this property should be done when the instance variable is first accessed.
  - **access (optional-defaults to property)**—Specifies how the container accesses the property at runtime. For example, when access is set to property, the container calls the property of getter or setter method and when access is set to field, the container bypasses the getter or setter method and calls the corresponding field directly.

The following code snippet shows the <property> attribute:

```
<property
    name="propertyName"
    column="column_name"
    type="typename"
    update="true|false"
    insert="true|false"
    formula="arbitrary SQL expression"
    access="field|property|ClassName"
    lazy="true|false"
    unique="true|false"
    not-null="true|false"
    optimistic-lock="true|false"
    generated="never|insert|always"
    node="element-name|@attribute-name|element/@attribute|."
    index="index_name"
    unique_key="unique_key_id"
    length="L"
    precision="P"
    scale="S"
/>
```

- **<many-to-one> element**—Specifies a many-to-one association to a persistent class that is declared using a many-to-one element. The relational model is a many-to-one relationship, i.e. a foreign key is referenced as the primary key column in the target table. The attributes of the <many-to-one> element are described as follows:
  - **name**—Specifies the name of the property.
  - **column (optional)**—Specifies the name of the foreign key columns, which may also be specified by nested <column> element(s).
  - **class (optional-defaults to the property type determined by reflection)**—Specifies associated class name.
  - **cascade (optional)**—Specifies the operations to be cascaded from parent object to the associated object.

- **update, insert**—Refers to the optional attributes, by default whose values are true. This attribute specified that all the mapped columns should be included in SQL UPDATE and/or INSERT statements. If both the attributes update and insert are set to be "false", then a pure "derived" association is allowed.
- **property-ref (optional)**—Specifies the name of a property of the associated class that is joined to this foreign key. By default, the primary key of the associated class is used.
- **access (optional-defaults to property)**—Specifies the strategy to be used for accessing the property value.
- **unique (optional)**—Allows the DDL of a unique constraint to be generated for the foreign-key column.
- **not-null (optional)**—Enables a foreign key column to have a null or not null value.
- **optimistic-lock (optional-defaults to true)**—Specifies if the updates to this property require acquisition of the optimistic lock.
- **lazy (optional-defaults to false)**—Specifies that the lazy fetching of this property should be done when the instance variable is first accessed.

The following code snippet shows the <many-to-one> attribute:

```
<many-to-one
    name="propertyName"
    column="column_name"
    class="ClassName"
    cascade="cascade_style"
    fetch="join|select"
    update="true|false"
    insert="true|false"
    property-ref="propertyNameFromAssociatedClass"
    access="field|property|ClassName"
    unique="true|false"
    not-null="true|false"
    optimistic-lock="true|false"
    lazy="proxy|no-proxy|false"
    not-found="ignore|exception"
    entity-name="EntityName"
    formula="arbitrary SQL expression"
    node="element-name|@attribute-name|element/@attribute|."
    embed-xml="true|false"
    index="index_name"
    unique_key="unique_key_id"
    foreign-key="foreign_key_name"
/>
```

- **<one-to-one> element**—Specifies a one-to-one association to a persistent class and that is declared by using one-to-one element. The attributes of the <one-to-one> element are described as follows:
  - **name**—Specifies the name of the property.
  - **class (optional-defaults to the property type determined by reflection)**—Specifies the name of an associated class with this attribute.
  - **cascade (optional)**—Specifies the operations to be cascaded from the parent object to the associated object.
  - **constrained (optional)**—Specifies a foreign key constraint on the primary key of mapped table. This mapped table references the table of the associated class. This option specifies the order in which the save() and delete() methods are cascaded.
  - **fetch (optional-defaults to select)**—Specifies whether to choose outer-join fetching or sequential select fetching.
  - **property-ref (optional)**—Specifies the name of a property of an associated class. This property is to be joined with primary key of this class. If the property is not specified then the primary key of the class should be used.

- **access (optional-defaults to property)**—Specifies a strategy. This strategy is then used by Hibernate for accessing the property value.
- **formula (optional)**—Specifies a value for a computed property in an arbitrary SQL expression. The following code snippet shows <one-to-one> attribute:

```
<one-to-one
   name="propertyName"
   class="ClassName"
   cascade="cascade_style"
   constrained="true|false"
   fetch="join|select"
   property-ref="propertyNameFromAssociatedClass"
   access="field|property|ClassName"
   formula="any SQL expression"
   lazy="proxy|no-proxy|false"
   entity-name="EntityName"
   node="element-name|@attribute-name|element/@attribute|."
   embed-xml="true|false"
   foreign-key="foreign_key_name"
/>
```

Hibernate can persist instances of java.util.Map, java.util.Set, java.util.SortedMap, java.util.SortedSet, java.util.List, and any array of persistent entities or values. Properties of the java.util.Collection or java.util.List type may also be persisted with the bag semantic.

The elements, such as <set>, <list>, <map>, <bag>, <array>, and <primitive-array> are used to declare Collections. The <map> element contains the following attributes:

- **name**—Specifies the collection property name.
- **table (optional-defaults to property name)**—Specifies the name of the collection table (not used for one-to-many associations).
- **schema (optional)**—Specifies the name of a table schema to override the schema declared in the root element.
- **lazy (optional-defaults to false)**—Enables lazy initialization (not used for arrays).
- **inverse (optional-defaults to false)**—Marks this collection as the inverse end of a bidirectional association.
- **cascade (optional-defaults to none)**—Specifies operations to cascade to the child entities.
- **sort (optional)**—Specifies a Collection, defined with either natural order or with a given comparator class.
- **order-by (optional, JDK1.4 only)**—Specifies a single column or multiple columns of a table that defines the iteration order of the Map, Set, or Bag with either an optional asc or desc.
- **where (optional)**—Refers to an arbitrary SQL condition that is to be used when query is executed for retrieving or removing the collection (it is useful in case when the collection should contain only a subset of the available data).
- **fetch (optional, defaults to select)**—Provides the capability to choose among outer-join fetching, fetching by sequential select, and fetching by sequential subselect.
- **batch-size (optional, defaults to 1)**—Specifies the batch size for the fetching instances of a Collection.
- **access (optional-defaults to property)**—Specifies the strategy that Hibernate should use for accessing the property value.
- **optimistic-lock (optional-default to true)**—Specifies the changes that are made to the state of the collection results.
- **mutable (optional - defaults to true)**—Specifies that whether the elements of the Collection can be altered or not. The following code snippet shows the <map> element:

```
<map
name="propertyName"
table="table_name"
schema="schema_name"
lazy="true|extra|false"
inverse="true|false"
```

```
cascade="all|none|save-update|delete|all-delete-orphan|delete-orphan"
sort="unsorted|natural|comparatorClass"
order-by="column_name asc|desc"
where="arbitrary sql where condition"
fetch="join|select|subselect"
batch-size="N"
access="field|property|ClassName"
optimistic-lock="true|false"
mutable="true|false"
node="element-name|."
>
```

After being familiar with HQL and O/R mapping, let's understand how a JavaBean is configured in Hibernate Web application.

Working with Hibernate

Let's consider a Hibernate example in which a JavaBean sets and retrieves the employee details from a database. For this example, let's create EmployeeData table to store employee details and configure the connection details, such as dialect, connection Uniform Resource Locator (URL), username, and password in hibernate.cfg.xml file. Then a JavaBean is created to set or retrieve the employee details from the database table. The EmployeeData.hbm.xml file is created to configure the `Employeedata` table to the JavaBean. The following steps help in understanding configuration in Hibernate application:

- Setting up the Development Environment
- Creating database table
- Writing Hibernate configuration file, JavaBean, and Hibernate mapping file

Now, let's study each of them in detail.

Setting up the Development Environment

First, let's create necessary directories and move the required files to the appropriate directory by performing the following steps:

- Create a directory named chapter15 to represent the chapter number.
- Create the `MyHibernateApp` directory under the chapter 15 directory.
- Now, add the Hibernate code to the application development environment. Extract hibernate-distribution-3.5.2-Final-dist.zip in a directory on your system.
- Copy all the JAR files, such as hibernate3.jar from the lib directory of the Hibernate 3.5 dowloaded directory into the lib of your application. In our case, we have copied the JAR files into the lib directory of the `MyHibernateApp` directory.

After setting the development environment of the `MyHibernateApp` application, let's create a database table to be used in O/R mapping.

Creating Database Table

To work with the Hibernate application discussed in this chapter, create a table named Employeedata. The following code snippet shows the query to create the Employeedata table:

```
CREATE TABLE Employeedata (Employee_id NUMBER(5),
  Employee_name VARCHAR(25), Employee_title VARCHAR(20));
```

This table is required to be mapped with the Hibernate mapping file.

Writing Hibernate Configuration File, JavaBean, and Hibernate Mapping File

In this section, the code samples of the files required to configure and develop a Hibernate Web application are provided. These configuration files are hibernate.cfg.xml, Hibernate mapping (in our case it is EmployeeData.hbm.xml), and JavaBean (POJO) class that is required to map database table. You can configure the variables of JavaBean to map the column of database table.

Creating the hibernate.cfg.xml File

Let's create the hibernate.cfg.xml file to configure Hibernate by declaring the environmental details. In the hibernate.cfg.xml file, you need to provide various details, such as table name that maps to a class, and JavaBean properties mapping to column names of the specified table. The hibernate.cfg.xml file should be placed in \WEB-INF\classes folder of your application. The Hibernate configuration file (hibernate.cfg.xml) is shown in the following code snippet:

```
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
  <!-- properties -->
  <property name="connection.username">scott</property>
  <property
  name="connection.url"> jdbc:oracle:thin:@192.168.1.123:1521:XE</property>
  <property name="dialect"> org.hibernate.dialect.OracleDialect</property>
  <property name="connection.password">tiger</property>
  <property name="connection.driver_class"> oracle.jdbc.driver.OracleDriver </property>
<!-- mapping files -->


<mapping resource="com/kogent/hibernate/EmployeeData.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

Creating a JavaBean

A JavaBean stores the data retrieved from the database table. The AbstractEmployeedata class is a sample JavaBean class for the MyHibernateApp application. The AbstractEmployeedata class has three attributes—EmployeeId, EmployeeName, and EmployeeTitle. The EmployeeId variable allows the application to access database identity that serves as the primary key of the Employeedata table. This JavaBean also has getter methods, such as `getEmployeeId()`and `getEmployeeName()`, which are used to get data from the Employeedata table and also has some setter methods, such as setEmployeeId() and setEmployeeName(), which are used to set data in the Employeedata table. The AbstractEmployeedata JavaBean is shown in the following code snippet:

```
package com.kogent.hibernate;
import java.io.Serializable;
public abstract class AbstractEmployeedata
implements Serializable
{
/** The cached hash code value for this instance. Settting to 0 triggers re-
  calculation. */
  private int hashValue = 0;

  /** The composite primary key value. */
  private java.lang.Integer EmployeeId;

  /** The value of the simple EmployeeName property. */
  private java.lang.String EmployeeName;

  /** The value of the simple EmployeeTitle property. */
  private java.lang.String EmployeeTitle;

  /**
  * Simple constructor of AbstractEmployeedata instances.
  */
  public AbstractEmployeedata()
  {
  }

  /**
  * Constructor of AbstractEmployeedata instances given a simple primary key.
```

```
* @param EmployeeId
*/
public AbstractEmployeedata(java.lang.Integer EmployeeId)
{
      this.setEmployeeId(EmployeeId);
}

/**
* Return the simple primary key value that identifies this object.
* @return java.lang.Integer
*/


public java.lang.Integer getEmployeeId()
{
      return EmployeeId;
}

/**
* Set the simple primary key value that identifies this object.
* @param EmployeeId
*/
public void setEmployeeId(java.lang.Integer EmployeeId)
{
this.hashValue = 0;
this.EmployeeId = EmployeeId;
}

/**
* Return the value of the Employee_name column.
* @return java.lang.String
*/
public java.lang.String getEmployeeName()
{
      return this.EmployeeName;
}

/**
* Set the value of the Employee_name column.
* @param EmployeeName
*/
public void setEmployeeName(java.lang.String EmployeeName)
{
      this.EmployeeName = EmployeeName;
}

/**
* Return the value of the Employee_title column.
* @return java.lang.String
*/
public java.lang.String getEmployeeTitle()
{
      return this.EmployeeTitle;
}

/**
* Set the value of the Employee_title column.
* @param EmployeeTitle
*/
public void setEmployeeTitle(java.lang.String EmployeeTitle)
{
      this.EmployeeTitle = EmployeeTitle;
}

/**
```

```
    * Implementation of the equals comparison on the basis of equality of the
    primary key values.
    * @param rhs

    * @return boolean
    */
    public boolean equals(Object rhs)
    {
          if (rhs == null)
                return false;
          if (! (rhs instanceof Employeedata))
                return false;
          Employeedata that = (Employeedata) rhs;
          if (this.getEmployeeId() != null && that.getEmployeeId() != null)
          {
                if (! this.getEmployeeId().equals(that.getEmployeeId())) {
                      return false;
                }
          }
          return true;
    }

    public int hashCode() {
          if (this.hashValue == 0) {
                int result = 17;
                int EmployeeIdValue = this.getEmployeeId() == null ? 0 :
                 this.getEmployeeId().hashCode();
                result = result * 37 + EmployeeIdValue;
                this.hashValue = result;
          }
          return this.hashValue;
    }
}
```

Now, you need to map this JavaBean with the database table.

Creating the EmployeeData.hbm.xml File

Hibernate maps the columns of the Employeedata table to the properties of the JavaBean, so that you can use HQL query language. The following code sinippet shows Hibernate mapping in the Hibernate mapping file (EmployeeData.hbm.xml):

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping package="com.kogent.hibernate">
  <class name="AbstractEmployeedata" table="Employeedata">
  <id name="EmployeeId" column="Employee_id" type="java.lang.Integer">
  <generator class="native"/>
  </id>
  <property name="EmployeeName" column="Employee_name"
  type="java.lang.String" not-null="true" />
  <property name="EmployeeTitle" column="Employee_title"
  type="java.lang.String" not-null="true" />
  </class>
</hibernate-mapping>
```

Let's now learn to implement the O/R mapping using Hibernate by developing the HibernateApplication application.

Implementing O/R Mapping with Hibernate

Hibernate Web application, named HibernateApplication, acts as an employee directory service that allows a user to create, read, update, and delete employee details stored in the company database. In this section, you learn to define all the components of the HibernateApplication application including the Hibernate configuration (hibernate.cfg.xml, provided as Listing 15.2) , Hibernate mapping file (in this Web application it is

Employee.hbm.xml, provided as Listing 15.3), and JavaBean (Employee.java, provided in Listing 15.1). You also learn to develop some controller components with the help of Java Servlet and the View components by using the JSP technology. The Oracle database is used as backend that stores employee record by implementing Hibernate O/R mapping and HQL.

Developing a JavaBean

The first step in creating Hibernate application (HibernateApplication) is to build the JavaBean that is used to store the data persistently in a database. This Java class contains the getter and setter methods that maps to the EMPLOYEE table. The Employee JavaBean is shown in Listing 15.1 (you can find Employee.java file on CD in the `code\JavaEE\Chapter15\HibernateApplication\src\com\kogent\hibernate` folder):

Listing 15.1: Displaying the Employee JavaBean File

```
package com.kogent.hibernate;
public class Employee

{
  int employeeId;
  String name;
  int age;
  double salary;
  public int getEmployeeId()

  {
        return employeeId;
  }
  public String getName()
  {
        return name;
  }
  public int getAge()

  {
        return age;
  }
  public double getSalary()
  {
        return salary;
  }
  public void setEmployeeId(int employeeId)
  {
        this.employeeId = employeeId;
  }
  public void setName(String name)
  {
        this.name = name;
  }
  public void setAge(int age)
  {
        this.age = age;
  }
  public void setSalary(Double salary)
  {
        this.salary = salary;
  }
}
```

The code shown in Listing 15.1 (Employee JavaBean) sets the values of the fields of a table that can be created by executing the following query on the Oracle datasource:

```
Create table EMPLOYEE (EMPLOYEE_ID number(5), NAME varchar2(20),
  AGE number(3), SALARY number(7,2),
  constraint Pk_Employee_EmployeeId Primary key(EMPLOYEE_Id));
```

Developing Hibernate Configuration File

As discussed ealier, you can provide the Hibernate configuration details in the hibernate.cfg.xml file. This file contains connection details, such as driver class, username, and password of database software (Oracle), and Hibernate mapping file that you are using in the HibernateApplication application. This file must be in the HibernateApplication\WEB-INF\classes folder. The Hibernate Configuration file is shown in Listing 15.2 (you can find the hibernate.cfg.xml file on CD in the code\JavaEE\Chapter15\HibernateApplication\WEB-INF\classes folder):

Listing 15.2: Showing the Code for the hibernate.cfg.xml File

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
        <property name="connection.username">scott</property>
        <property name="connection.password">tiger</property>
        <property name="connection.url">
             jdbc:oracle:thin:@192.168.1.123:1521:XE
        </property>
        <property name="dialect">
              org.hibernate.dialect.Oracle10gDialect

        </property>
        <property name="connection.driver_class">
              oracle.jdbc.driver.OracleDriver
        </property>
        <mapping resource="com/kogent/hibernate/Employee.hbm.xml" />
  </session-factory>
</hibernate-configuration>
```

Developing Hibernate Mapping File

As discussed earlier, the Hibernate mapping file maps the columns of the EMPLOYEE table to the properties of the Employee JavaBean. The mapping of the EMPLOYEE table to the Employee JavaBean is known as O/R mapping. This mapping file must be in the HibernateApplication\WEB-INF\classes\com\kogent\hibernate folder.

The Hibernate mapping file (Employee.hbm.xml) is shown in Listing 15.3 (you can find the Employee.hbm.xml file on CD in the code\JavaEE\Chapter15\HibernateApplication\WEB-INF\classes\com\kogent\hibernate folder):

Listing 15.3: Showing the Code for the Employee.hbm.xml File

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="com.kogent.hibernate.Employee" table="EMPLOYEE" schema="SCOTT">
  <id name="employeeId" type="java.lang.Integer">
  <column name="EMPLOYEE_ID" precision="5" scale="0" />
  <generator class="assigned" />
  </id>
  <property name="name" type="java.lang.String">
        <column name="NAME" length="20" />
  </property>
  <property name="age" type="java.lang.Integer">
        <column name="AGE" precision="3" scale="0" />
  </property>
  <property name="salary" type="java.lang.Double">
        <column name="SALARY" precision="7" scale="0" />
  </property>
  </class>
</hibernate-mapping>
```

Creating the EmployeeData.java File

The EmployeeData.java file is used to develop business objects that store the data of an employee. This Java file contains methods, such as getEmployees(), addEmployee(), editEmployee(), and deleteEmployee() that provide the CRUD operation to access data from the EMPLOYEE table through the Hibernate SessionFactory interface. This interface is used to get the connection detail information, such as connection URL, username, and password from the hibernate.cfg.xml configuration file and build the factory for session instances. This session provides transaction to query the database table. The code for the EmployeeData.java file is shown in Listing 15.4 (you can find the EmployeeData.java file on CD in the `code\JavaEE\Chapter15\HibernateApplication\src\com\kogent\hibernate` folder):

Listing 15.4: Showing the Code for the EmployeeData Class

```
package com.kogent.hibernate;
import java.util.ArrayList;
import java.util.Iterator;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.Query;
import org.hibernate.cfg.Configuration;
import com.kogent.hibernate.Employee;
public class EmployeeData
{
  public static Employee getEmployee(String employeeId) throws Exception
  {
        Session session = null;
        Employee employee=null;
        try
        {
                // This step will read hibernate.cfg.xml
                //and prepare hibernate for use
                SessionFactory sessionFactory = new
                Configuration().configure().buildSessionFactory();
                session =sessionFactory.openSession();
                //Create new instance of Employee and set
                //values in it by reading them from form object
                Transaction tx = session.beginTransaction();
                System.out.println("Getting Record");
                employee = new Employee();
                tx.commit();
                System.out.println("Done");
        Query query = session.createQuery("select employee1.employeeId, employee1.name,
        employee1.age, employee1.salary from Employee employee1 where employeeId =
        '"+employeeId +"'");
                for(Iterator it=query.iterate();;)
                {
                        Object[] row = (Object[]) it.next();
                        employee.setEmployeeId((int) new Integer(row[0].toString()));
                        employee.setName((String) row[1]);
                        employee.setAge((int) new Integer(row[2].toString()));
                        employee.setSalary((Double) row[3]);
                }
        }
        catch(Exception e)
        {
                System.out.println(e.getMessage());
        }
        finally
        {
                session.flush();
                session.close();
        }
        return employee;
  }
```

```
        // method to read employee of CRUD operation
        public static ArrayList getEmployees()
        {
                Session session = null;
                Employee employee=null;
                ArrayList<Employee> employees = new ArrayList<Employee>();
                try
                {
                        // This step will read hibernate.cfg.xml
                        //and prepare hibernate for use
                        SessionFactory sessionFactory = new
                        Configuration().configure().buildSessionFactory();
                        session =sessionFactory.openSession();
    Query query = session.createQuery("select employee1.employeeId, employee1.name,
employee1.age, employee1.salary from Employee employee1");
                        for(Iterator it=query.iterate();it.hasNext();)
                        {
                                employee = new Employee();
                                Object[] row = (Object[]) it.next();

                                employee.setEmployeeId((int) new Integer(row[0].toString()));
                                employee.setName((String) row[1]);
                                employee.setAge((int) new Integer(row[2].toString()));
                                employee.setSalary((Double) row[3]);
                                employees.add(employee);
                        }
                }
                catch(Exception e)
                {
                        System.out.println(e.getMessage());
                }
                finally
                {
                        session.flush();
                        session.close();
                }
                return employees;
        }
        // method to create employee of CRUD operation
        public void addEmployee(Employee emp) throws Exception
        {
                Employee employee = null;
                Session session = null;
                try
                {
                        // This step will read hibernate.cfg.xml
                        //and prepare hibernate for use
                        SessionFactory sessionFactory = new
                        Configuration().configure().buildSessionFactory();
                        session =sessionFactory.openSession();
                        Transaction tx = session.beginTransaction();
                        System.out.println("Selecting Record");
                        employee = new Employee();
                        employee.setEmployeeId(emp.getEmployeeId());
                        employee.setName(emp.getName());
                        employee.setAge(emp.getAge());
                        employee.setSalary(emp.getSalary());
                        session.save(employee);
                        tx.commit();
                        System.out.println("Done");
                System.out.println("Employee fffId Name Age Salary");
                }
                catch(Exception e)
                {
                System.out.println("Exception caught in EmployeeData.addEmployee" + e);
                }
```

```
    }
    // method to delete employee of CRUD operation
    public static void deleteEmployee(String employeeId) throws Exception
    {
          Session session = null;
          try
          {
                // This step will read hibernate.cfg.xml
                //and prepare hibernate for use
                SessionFactory sessionFactory = new
                Configuration().configure().buildSessionFactory();
                session =sessionFactory.openSession();
     String hqlDelete ="delete Employee employee where employeeId='" + employeeId + "'";
                Transaction tx = session.beginTransaction();
                System.out.println("Deleting Record");
                Query query = session.createQuery(hqlDelete);
                int row = query.executeUpdate();
                System.out.println("Number of rolw deleted " + row);
                tx.commit();
                System.out.println("Done");
          }
          catch(Exception e)
          {
     System.out.println("Exception caught in EmployeeData.deleteEmployee" + e);
          }
    }
    // method to edit employee of CRUD operation
    public void editEmployee(Employee emp) throws Exception
    {
          Session session = null;
          try
          {
                // This step will read hibernate.cfg.xml
                //and prepare hibernate for use
                SessionFactory sessionFactory = new
                Configuration().configure().buildSessionFactory();
                session =sessionFactory.openSession();
                Transaction tx = session.beginTransaction();
                System.out.println("Updating Record...");
                String hqlUpdate = "update Employee employee set name = :newName, age =
                :newAge, salary = :newSalary where employeeId = :NewEmployeeId";
                Query query = session.createQuery(hqlUpdate);
                query.setInteger("NewEmployeeId", emp.getEmployeeId());
                query.setString("newName", emp.getName());
                query.setInteger("newAge", emp.getAge());
                query.setDouble("newSalary", emp.getSalary());
                int rowCount = query.executeUpdate();
                System.out.println("Rows affected: " + rowCount);
                tx.commit();
                System.out.println("Done");
          }
          catch(Exception e)
          {
          System.out.println("Exception caught in EmployeeData.EditEmployee " + e);
          }
    }
}
```

Developing Controller Component

In a Web application, the business logic is handled by controller components. In our case, the business logic of the HibernateApplication application is to perform the CRUD operations. Let's now develop servlets as controller components that are used to retrieve data from the database by using the EmployeeData.java file. The following files are required to perform the CRUD operations in the HibernateApplication application:

- AddEmployeeServlet.java

- DeleteEmployeeServlet.java
- EditEmployeeServlet.java
- EmployeeListServlet.java

Creating the AddEmployeeServlet.java File

The AddEmployeeServlet servlet is used to add a row (or record) in the EMPLOYEE table by using the addEmployee() method of the EmployeeData.java file and forward it to the EmployeeList JSP page to display a list of the employees working in a company. The code for the AddEmployeeServlet servlet is shown in Listing 15.5 (you can find the AddEmployeeServlet.java file on CD in the `code\JavaEE\Chapter15\HibernateApplication\src\com\kogent\hibernate` folder):

Listing 15.5: Showing the Code for the AddEmployeeServlet.java File

```
package com.kogent.hibernate;
import com.kogent.hibernate.Employee;
import com.kogent.hibernate.EmployeeData;
import java.io.IOException;
import java.util.ArrayList;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import javax.servlet.RequestDispatcher;
public class AddEmployeeServlet extends HttpServlet {

  public void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException
  {
        String employeeId = request.getParameter("employeeId");
        String name = request.getParameter("name");
        String age = request.getParameter("age");
        String salary = request.getParameter("salary");
        Employee emp = new Employee();
        EmployeeData emplData = new EmployeeData();
        try
        {
              emp.setEmployeeId(Integer.parseInt(employeeId));
              emp.setName(name);
              emp.setAge(Integer.parseInt(age));
              emp.setSalary(Double.parseDouble(salary));
              emplData.addEmployee(emp);
              ArrayList employees = EmployeeData.getEmployees();
              HttpSession session = request.getSession(true);
              session.setAttribute("employees", employees);
        }

        catch(Exception e)
        {
        System.out.println("Exception caught in AddEmployeeServlet" + e);
        }
        RequestDispatcher rd=request.getRequestDispatcher("/EmployeeList.jsp");
        rd.forward(request,response);
  }
}
```

Creating the DeleteEmployeeServlet.java File

This DeleteEmployeeServlet servlet is used to delete a row from the EMPLOYEE table and forward the EmployeeList JSP page to display the remaining row of the EMPLOYEE table. The deletion is done by the delete HQL query mentioned in the deleteEmployee() method in the EmployeeData.java file (Listing 15.4). The code for the DeleteEmployeeServlet servlet is shown in Listing 15.6 (you can find DeleteEmployeeServlet.java file on the CD in the `code\JavaEE\Chapter15\HibernateApplication\src\com\kogent\hibernate` folder):

Listing 15.6: Showing the Code for the DeleteEmployeeServlet.java File

```
package com.kogent.hibernate;
import java.io.IOException;
import java.io.PrintWriter;
```

```
import java.util.ArrayList;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
public class DeleteEmployeeServlet extends HttpServlet
{
  public void doGet(HttpServletRequest request, HttpServletResponse
   response) throws ServletException, IOException
   {
        String employeeId = request.getParameter("employeeId");
        System.out.println("employee Idddddd  "+employeeId);
        try {
                EmployeeData.deleteEmployee(employeeId);
                ArrayList employees = EmployeeData.getEmployees();
                HttpSession session = request.getSession(true);
                session.setAttribute("employees", employees);
        }

        catch(Exception e)
        {
        System.out.println("Exception caught in AddEmployeeServlet" + e);
        }
        RequestDispatcher
        rd=request.getRequestDispatcher("/EmployeeList.jsp");
        rd.forward(request,response);
  }
}
```

Creating the EditEmployeeServlet.java File

The `EditEmployeeServlet` servlet is used to edit a row of the EMPLOYEE table and display the edited row by using the EmployeeList JSP page. The `EditEmployeeServlet` servlet calls the editEmployee() method of the EmployeeData.java file and updates the desired row by using the HQL update query. The code for the EditEmployeeServlet servlet is shown in Listing 15.7 (you can find the EditEmployeeServlet.java file on CD in the `code\JavaEE\Chapter15\HibernateApplication\src\com\kogent\hibernate` folder):

Listing 15.7: Showing the EditEmployeeServlet.java File

```
package com.kogent.hibernate;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class EditEmployeeServlet extends HttpServlet
{
  public void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException
  {
        String employeeId = request.getParameter("employeeId");
        String name = request.getParameter("name");
        String age = request.getParameter("age");
        String salary = request.getParameter("salary");
        System.out.println("dsfdsf empid  "+employeeId);
        Employee emp = new Employee();
        EmployeeData emplData = new EmployeeData();
        try
        {
                emp.setEmployeeId(Integer.parseInt(employeeId));
                emp.setName(name);
                emp.setAge(Integer.parseInt(age));
                emp.setSalary(Double.parseDouble(salary));
```

```
                emplData.editEmployee(emp);
                ArrayList employees = EmployeeData.getEmployees();
                HttpSession session = request.getSession(true);
                session.setAttribute("employees", employees);
            }
            catch(Exception e)
            {
            System.out.println("Exception caught in EditEmployeeServlet " + e);
            }
            RequestDispatcher rd=request.getRequestDispatcher("/EmployeeList.jsp");
            rd.forward(request,response);
    }
}
```

Creating the EmployeeListServlet.java File

The EmployeeListServlet servlet is used for getting all the rows of the EMPLOYEE table by using getEmployees() method of the EmployeeData.java file and forwarding to the EmployeeList JSP page to display a list of the employees working in a company. The code for the EmployeeListServlet servlet is shown in Listing 15.8 (you can find the EmployeeListServlet.java file on CD in the `code\JavaEE\Chapter15\HibernateApplication\src\com\kogent\hibernate` folder):

Listing 15.8: Showing the Code for the EmployeeListServlet.java File

```
package com.kogent.hibernate;

import java.io.IOException;
import java.util.ArrayList;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class EmployeeListServlet extends HttpServlet

{
  public void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException

  {
        ArrayList employees = EmployeeData.getEmployees();
        HttpSession session = request.getSession(true);
        session.setAttribute("employees", employees);
        ArrayList employees1=((ArrayList) session.getAttribute("employees"));
        if(employees1!= null && employees1.size()>0)
        {
                for(int i=0;i<employees1.size();i++)
                {
                        Employee emp= (Employee) employees1.get(i);
                        System.out.println(emp.getEmployeeId());
                        System.out.println(emp.getName());
                        System.out.println(emp.getAge());
                        System.out.println(emp.getSalary());

                 }
        }
        RequestDispatcher rd=request.getRequestDispatcher("/EmployeeList.jsp");
        rd.forward(request,response);
  }

}
```

After developing all the servlets for performing CRUD operation of EMPLOYEE table, let's now develop some JSP pages that display the CRUD operation of the EMPLOYEE table.

Developing View Components

The view components are JSP pages that are used to display the resultant view page of the CRUD operations of the EMPLOYEE table in a company. The following JSP files are developed as view components of the HibernateApplication application:

- AddEmployee.jsp
- EditEmployee.jsp
- EmployeeList.jsp

Now, let's develop these JSP files one by one.

Creating the AddEmployee.jsp File

The AddEmployee.jsp file displays the Add Employee Page screen that is used to add a row (or record) of an employee. This page contains four text fields, such as Id, Name, Age, and Salary along with a submit button. The code for the AddEmployee page is shown in Listing 15.9 (you can find the AddEmployee.jsp file on CD in the `code\JavaEE\Chapter15\HibernateApplication` folder):

Listing 15.9: Showing the Code for the AddEmployee.jsp File

```
<%@ page language="java" %>
<html>
<head>
   <title>Add Employee Page</title>
   <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
   <form action="/HibernateApplication/AddEmployeeServlet">
   <table width="500" border="0">
   <tr>
         <td>Employee Id</td>
         <td><input type="text" name="employeeId"></td>
   </tr>
   <tr>
         <td>Employee Name</td>
         <td><input type="text" name="name"></td>
   </tr>
   <tr>
         <td>Employee Age</td>
         <td><input type="text" name="age"></td>
   </tr>
   <tr>
         <td>Employee Salary</td>
         <td><input type="text" name="salary"></td>
   </tr>
   <tr>
         <td> <input type="submit" name ="submit" value="submit"> </td>
   </tr>
   </table>
</form>
</body>
</html>
```

Creating the EditEmployee.jsp File

The EditEmployee JSP page displays the Edit Employee Page screen that is used to edit the details of an employee. The code for the EditEmployee JSP page is shown in Listing 15.10 (you can find the EditEmployee.jsp file on CD in the `code\JavaEE\Chapter15\HibernateApplication` folder):

Listing 15.10: Showing the Code for the EditEmployee.jsp File

```
<%@ page language="java" %>
<%@ page import="com.kogent.hibernate.EmployeeData,
com.kogent.hibernate.Employee" %>
<html>
<head>
```

```
    <title>Edit Employee Page</title>
    <link rel="stylesheet" type="text/css" href="styles.css">
  </head>
  <body>
    <%
    String employeeId = request.getParameter("employeeId");
    Employee emp = null;
    emp= EmployeeData.getEmployee(employeeId);
    %>
    <form action="/HibernateApplication/EditEmployeeServlet">
    <table width="500" border="0">
    <tr>
          <td>Employee Id</td>
          <td> <%=emp.getEmployeeId() %> <input type = "hidden" name ="employeeId"
          value=<%=emp.getEmployeeId() %>></td>
    </tr>
    <tr>
          <td>Employee Name</td>
          <td><input type="text" name="name" value=<%=emp.getName()%>></td>
    </tr>
    <tr>
          <td>Employee Age</td>
          <td><input type="text" name="age" value=<%=emp.getAge() %>></td>
    </tr>
    <tr>
          <td>Employee Salary</td>
  <td><input type="text" name="salary" value=<%=emp.getSalary() %>></td>
    </tr>
    <tr>
          <td> <input type="submit" name ="submit" value="submit"> </td>
    </tr>
    </table>
  </form>
  </body>
  </html>
```

Creating the EmployeeList.jsp File

The EmployeeList JSP page displays the Employee List Page screen that is used to display a list of the employees working in a company. This JSP page contains three hyperlinks that are used to add, delete, and update the details of an employee. The code for the EmployeeList JSP page is shown in Listing 15.11 (you can find the EmployeeList.jsp file on CD in the `code\JavaEE\Chapter15\HibernateApplication` folder):

Listing 15.11: Showing the **Code for the** EmployeeList.jsp File

```
<%@ page import="java.util.ArrayList, com.kogent.hibernate.Employee" %>
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
<head>
  <title>Employee List Page</title>
  <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
  <table width="700"
  border="0" cellspacing="0" cellpadding="0">
  <tr align="left">
        <th>Employee Id</th>
        <th>Name</th>
        <th>Age</th>
        <th>Salary</th>
  </tr>
  <!-- iterate over the results of the query -->
<% ArrayList employees=((ArrayList) session.getAttribute("employees"));
if(employees!= null && employees.size()>0)
{
  for(int i=0;i<employees.size();i++)
  {
```