

Enterprise Application Development in Java EE

Are you registered with
Onlinevarsity.com?

Yes



No



Did you download this book
from **Onlinevarsity.com**?

Yes



No



Scores

For each **YES** you score **50**

For each **NO** you score **0**

If you score less than 100 this book is illegal.

Register on **www.onlinevarsity.com**

Enterprise Application Development in Java EE

Learner's Guide

© 2015 Aptech Limited

All rights reserved.

No part of this book may be reproduced or copied in any form or by any means – graphic, electronic or mechanical, including photocopying, recording, taping, or storing in information retrieval system or sent or transferred without the prior written permission of copyright owner Aptech Limited.

All trademarks acknowledged.

APTECH LIMITED

Contact E-mail: ov-support@onlinevarsity.com

First Edition - 2015



Dear Learner,

We congratulate you on your decision to pursue an Aptech course.

Aptech Ltd. designs its courses using a sound instructional design model – from conceptualization to execution, incorporating the following key aspects:

- Scanning the user system and needs assessment

Needs assessment is carried out to find the educational and training needs of the learner

Technology trends are regularly scanned and tracked by core teams at Aptech Ltd. TAG* analyzes these on a monthly basis to understand the emerging technology training needs for the Industry.

An annual Industry Recruitment Profile Survey# is conducted during August - October to understand the technologies that Industries would be adapting in the next 2 to 3 years. An analysis of these trends & recruitment needs is then carried out to understand the skill requirements for different roles & career opportunities.

The skill requirements are then mapped with the learner profile (user system) to derive the Learning objectives for the different roles.

- Needs analysis and design of curriculum

The Learning objectives are then analyzed and translated into learning tasks. Each learning task or activity is analyzed in terms of knowledge, skills and attitudes that are required to perform that task. Teachers and domain experts do this jointly. These are then grouped in clusters to form the subjects to be covered by the curriculum.

In addition, the society, the teachers, and the industry expect certain knowledge and skills that are related to abilities such as learning-to-learn, thinking, adaptability, problem solving, positive attitude etc. These competencies would cover both cognitive and affective domains.

A precedence diagram for the subjects is drawn where the prerequisites for each subject are graphically illustrated. The number of levels in this diagram is determined by the duration of the course in terms of number of semesters etc. Using the precedence diagram and the time duration for each subject, the curriculum is organized.

- Design & development of instructional materials

The content outlines are developed by including additional topics that are required for the completion of the domain and for the logical development of the competencies identified. Evaluation strategy and scheme is developed for the subject. The topics are arranged/organized in a meaningful sequence.

The detailed instructional material – Training aids, Learner material, reference material, project guidelines, etc.- are then developed. Rigorous quality checks are conducted at every stage.

➤ Strategies for delivery of instruction

Careful consideration is given for the integral development of abilities like thinking, problem solving, learning-to-learn etc. by selecting appropriate instructional strategies (training methodology), instructional activities and instructional materials.

The area of IT is fast changing and nebulous. Hence considerable flexibility is provided in the instructional process by specially including creative activities with group interaction between the students and the trainer. The positive aspects of web based learning –acquiring information, organizing information and acting on the basis of insufficient information are some of the aspects, which are incorporated, in the instructional process.

➤ Assessment of learning

The learning is assessed through different modes – tests, assignments & projects. The assessment system is designed to evaluate the level of knowledge & skills as defined by the learning objectives.

➤ Evaluation of instructional process and instructional materials

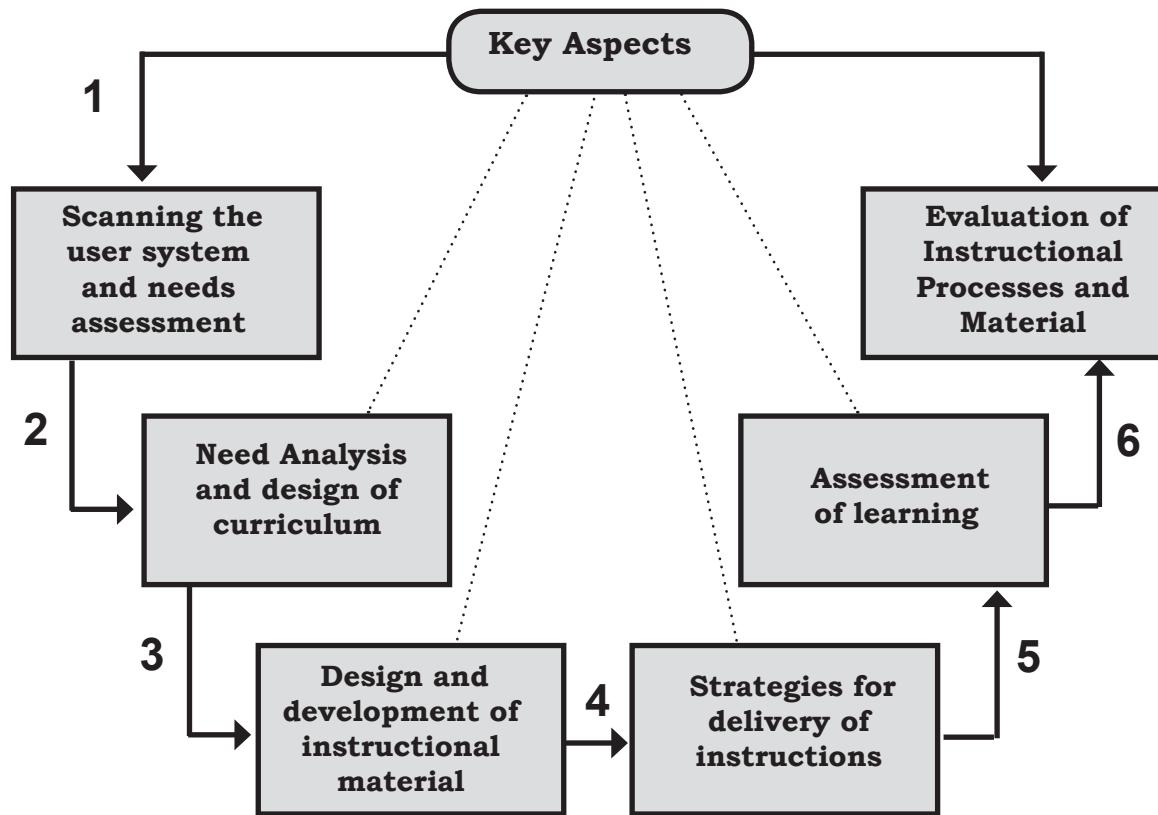
The instructional process is backed by an elaborate monitoring system to evaluate - on-time delivery, understanding of a subject module, ability of the instructor to impart learning. As an integral part of this process, we request you to kindly send us your feedback in the reply pre-paid form appended at the end of each module.

*TAG – Technology & Academics Group comprises members from Aptech Ltd., professors from reputed Academic Institutions, Senior Managers from Industry, Technical gurus from Software Majors & representatives from regulatory organizations/forums.

Technology heads of Aptech Ltd. meet on a monthly basis to share and evaluate the technology trends. The group interfaces with the representatives of the TAG thrice a year to review and validate the technology and academic directions and endeavors of Aptech Ltd.

Industry Recruitment Profile Survey - The Industry Recruitment Profile Survey was conducted across 1581 companies in August/September 2000, representing the Software, Manufacturing, Process Industry, Insurance, Finance & Service Sectors.

Aptech New Products Design Model



WRITE-UPS BY

EXPERTS AND LEARNERS

TO PROMOTE NEW AVENUES AND
ENHANCE THE LEARNING EXPERIENCE



FOR FURTHER READING, LOGIN TO

www.onlinevarsity.com

Preface

The primary aim of 'Enterprise Application Development in Java EE' book is to teach the students how to design and develop enterprise applications on Java Enterprise Edition (Java EE) platform.

The book discusses about Enterprise JavaBeans (EJB) technology used in developing business components for large enterprise applications. It explains the concepts, methodology, and development process of EJB components on Java EE platform.

The book covers the current version of EJB, that is, EJB 3.0 which has made component development easier by supporting techniques such as Java annotations, metadata programming dependency injection, interceptors, asynchronous session beans, singleton session beans, EJB Timer service, and so on.

The book discusses how to use Java Persistence API (JPA) for persisting data of enterprise applications in databases. It covers various features supported by entities such as relationships, inheritance, polymorphism, and Java Persistence Query Language (JPQL).

The book also covers various services such as transaction, security, and so on supported in Java EE applications. Finally, it concludes with the introduction of various EJB design patterns used for designing enterprise-level applications on Java EE platform.

The knowledge and information in this book is the result of the concentrated effort of the Design Team, which is continuously striving to bring to you the latest, the best, and the most relevant subject matter in Information Technology. As a part of Aptech's quality drive, this team does intensive research and curriculum enrichment to keep it in line with industry trends and learner requirements.

We will be glad to receive your suggestions. Please send us your feedback, addressed to the Design Centre at Aptech's corporate office, Mumbai.

Design Team

BIG
B

I
G

Balanced Learner-Oriented Guide

for enriched learning available



Table of Contents

Sessions

1. Introduction to Business Components
2. Enterprise JavaBeans
3. Session Beans
4. Stateful Session Beans
5. Singleton Session Beans
6. Introduction to Messaging
7. Interceptors and Dependency Injection
8. Transactions
9. Persistence of Entities
10. Advanced Persistence Concepts
11. Query and Criteria API
12. Concurrency, Listeners, and Caching
13. Security
14. EJB Timer Service
15. EJB Design Patterns

GROWTH
RESEARCH
OBSERVATION
UPDATES
PARTICIPATION





Welcome to the Session, **Introduction to Business Components**.

This session describes various aspects of application development with respect to distributed enterprise applications. The session explains the Java Enterprise Edition (Java EE) platform with its features and services for developing enterprise applications on Java platform. Further, the session also explains the various Java EE application servers used for enterprise application development and the services provided by containers in the application servers. Finally, the session concludes with the introduction of Java EE profiles for deploying enterprise applications.

In this Session, you will learn to:

- Explain enterprise applications
- Describe the problem faced by large enterprise applications
- Explain distributed application architecture used for developing business components
- Describe Java EE platform and its API
- Describe application development architecture of enterprise application
- Describe application server containers and their services
- List various Java EE application servers
- Describe Java EE profiles and the use of EJBLite on application servers

1.1 Introduction

Java has come a long way from its humble beginning as a platform-neutral programming language for consumer electronic devices. Today, it has grown in various directions with the objective of fulfilling the requirements of various industries and domains.

The different flavors of Java are differentiated based on the packages provided for the development of various kinds of applications. One of the most interesting packages provided by Java is Java Enterprise Edition (Java EE) platform that enables building of enterprise applications.

1.1.1 Enterprise Applications

An enterprise application is a large business application. It is usually developed to fulfil the needs of large business domains that involve complex functionalities, such as complex business logics, scalability to extend based on work load, concurrent access by multiple users and heterogeneous databases, deployed across multiple platforms, and so forth.

An enterprise application is usually hosted on server and simultaneously provides services to a large number of users over a computer network. This is in contrast to the single-user software applications which run on a user's own local computer and serve only one user at a time.

The process of enterprise application development begins with understanding of the problem domain, refining the requirements, developing the desired components, assembling the components, testing the application as a whole, and finally deploying the application in the production environment.

1.1.2 Need for Enterprise Applications

Businesses have undergone several changes in the last few decades to service the highly complex and customized customers' requirements. The rapid changes have made businesses to meet the requirement of customers, communicate with other business processes, and incorporate business-to-business services.

Enterprise applications are developed to satisfy the needs of such businesses. Instead of dealing with the requirement of an individual user, enterprise applications are developed to meet the requirements of large businesses or domains.

The different types of domains involving complex business functionalities include: order management, production scheduling, customer information management, bank account maintenance, and so on.

Some of the concerns and requirements of large businesses are as follows:

- They should be long-lived applications and perform parallel processing.
- They should be functional across different platforms.
- They need to support complex business processes with domain-based constraints.
- They must follow strict rules and procedures with respect to the security, administration, and maintenance of the applications.

- Complex business requirements are defined through policies, constraints, rules, processes, and entities in the domain which are to be implemented while developing applications.

All these concerns and requirements led to the development of enterprise applications. The designing and developing of enterprise applications must also meet the requirements of other legacy systems dependent on them. The legacy systems are existing applications or services already running within the system.

1.2

Enterprise Application Architecture

Application architecture divides application components into tiers or layers based on their functionality in the environment.

Initially, applications were designed based on two-tier architecture. The two-tier architecture is similar to the client-server application. Normally, in two-tier architecture, functionality is as follows:

Client Tier - The first tier is the client tier. It is primarily responsible for presentation of data to the client.

Server Tier - The server is responsible for providing data services to the client.

Figure 1.1 shows typical client-server architecture.

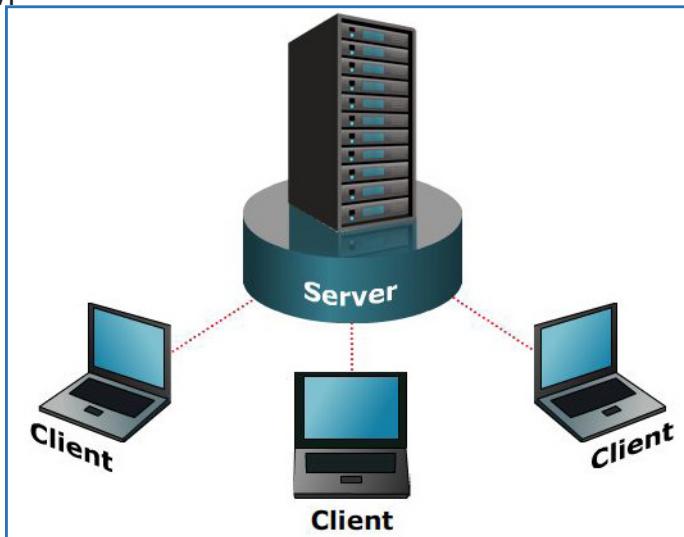


Figure 1.1: Client-Server Architecture

In the client-server architecture, the services are hosted on the server and the client can remotely access it over the network.

There is one more form of client-server architecture, where the client can contain some business logic residing on it. This type of client is referred to as thick client, as significant part of the executable code resides on the client-tier.

Figure 1.2 shows the client-server architecture with a thick client.

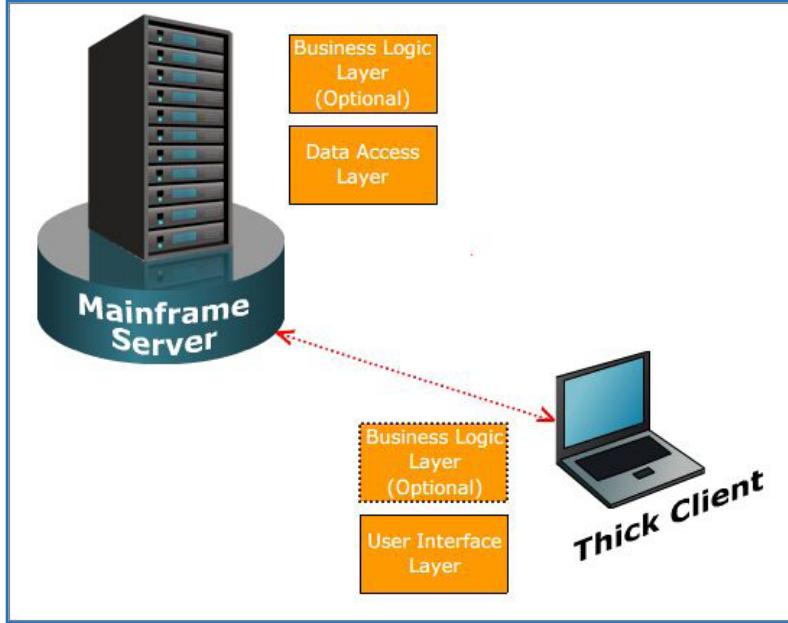


Figure 1.2: Client-Server Architecture – Thick Client

Some of the drawbacks with the client-server architecture are as follows:

1. If the server has multiple clients trying to access the server, then there might be resource contention resulting in response delays.
2. Any changes in the user interface have to be updated at all client nodes.

To overcome these problems, distributed application architecture has been introduced. This architecture divides the large monolithic application into layers. It introduces a middleware layer whose main purpose is to bridge the gap between different hardware systems. The middleware components of an application are located on these hardware systems and can be accessed over the network through the layer.

Figure 1.3 shows distributed architecture of applications.

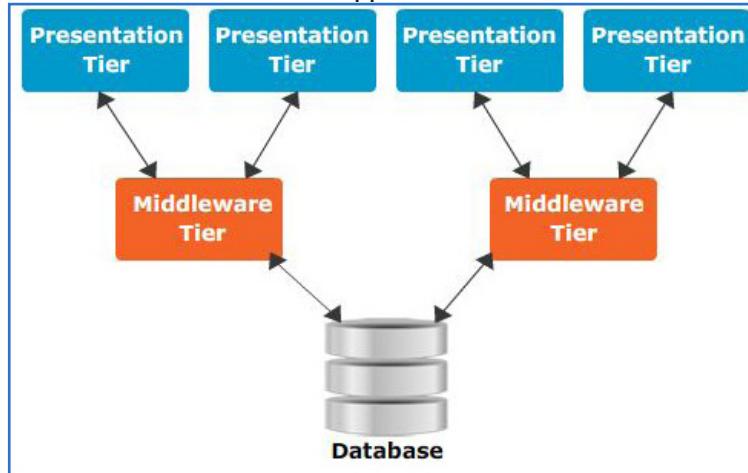


Figure 1.3: Distributed Architecture

In figure 1.3, there are different layers in which the application components are organized.

The different layers of the distributed architecture are as follows:

Presentation tier - It is the client tier; service requests for the application are initiated in this layer. It consists of the screens and forms that the user is going to view and manipulate the data in.

Middleware tier – It is also referred to as business tier. It implements the business logic of the application; it can communicate with both the client and the database.

Database tier – It is also be termed as Enterprise Information Systems (EIS) layer which holds all the data pertaining to the application.

When application processing is distributed among separate layers, it results in better resource utilization.

Java enterprise application development is based on distributed application architecture. It enables development of components that can be distributed on the multiple layers and can be accessed on the network through appropriate protocols.

This architecture improves the efficiency of the application and makes the remote location of the components transparent to the end user of the application.

The distribution of components on different layers has further evolved into n-tier architecture.

1.2.1 Requirements of Distributed Applications

As discussed, enterprise application execution is distributed across multiple tiers. However, there are various issues or concerns related with respect to the design of the distributed applications.

Figure 1.4 depicts some of the issues which should be considered while creating an enterprise application based on distributed architecture.

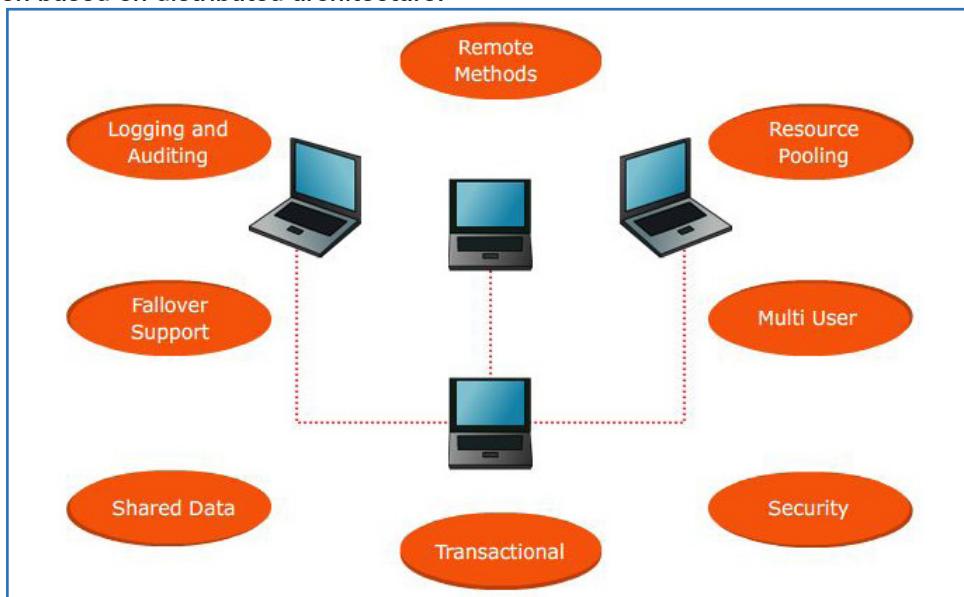


Figure 1.4: Issues with Distributed Architecture

Enterprise applications that are based on distributed architecture must fulfil the following requirements:

- Reusability** - Reusability specifies that the same business object may be used by different sub-systems of the same application as well as by different applications. For instance, both order management and invoicing applications may make use of the same product business object.
- Remote access** – Various components of the enterprise applications can interact with each other remotely. Accessing objects remotely involves invocation of methods, dispatching the method requests, passing the method parameters, dispatching SQL commands, and so on.
- Resource pooling** – Java EE application server provides resource pooling. The resources in an application are database connections, enterprise beans, and so on. Resource pooling in distributed applications ensures that these resources are available to any object on the application server.
- Multi-user** - A business object should be able to provide its services to a large number of clients at the same time. Each client should get the impression that it is being serviced by a dedicated business object.
- Access control** - The services offered by the business objects are often subjected to access control. A manager for example can view as well as modify a business object and a junior-level operator may just view the business object.
- Failover support** – When a distributed application is deployed on a set of application servers, then the failure of one of the application servers should not be visible to the end user of the application. All the requests being processed by failed server are transferred to another server in the cluster. Distributed applications can have clustered servers. Server clustering is used in the applications to ensure high availability. The client requests of the application are handled by the cluster. When one of the servers in the cluster fails, then the requests coming from the clients are transferred to the remaining servers in the cluster.
- Transactional** - Business objects often need to participate in transactions. A transaction can be described as a set of functions that need to be completed as a unit. If one of the functions fails, all the functions in the unit will be cancelled or rolled-back. If all of them succeed, the transaction is said to be committed.
- Shared data** - Business objects usually operate on the same data. They should therefore support measures so that the data integrity is maintained. For instance, if two users attempt to update the same account information, the built-in mechanisms should ensure that data remains consistent.
- Logging and auditing** – Logging is the activity of the application. Auditing of the log is essential to monitor the performance of the application. Enterprise applications maintain an application log which holds the record of all the requests received and service provided by the application. This application log can be regularly audited to analyze the performance of the application and optimize it.
- Security** – In distributed applications, since the components are remotely accessed, the components should be secured from malicious users.

1.3 Java in Enterprise Development

There are three variants of Java platforms – Java Standard Edition (SE), Java Enterprise Edition (EE), and Java Micro Edition (ME). There are different editions of Java platform which are bundled with related sets of API. The bundled APIs help in developing and executing applications on Java platform.

Figure 1.5 shows different editions of Java.



Figure 1.5: Editions of Java

Java SE – Java Standard Edition (Java SE) is used to develop and deploy Java applications. It provides the basic functionality of Java programming language. It also provides support for networking, database access, graphical user interface, and so on. Apart from the core API, Java SE has various development and deployment tools, virtual machines, and class libraries.

Java EE - Java Enterprise Edition (Java EE) comprises all the programming support provided in Java SE. The purpose of Java EE is to build large scalable, multi user applications. Though, the basic programming tools are the same, the tools in Java EE are meant for larger applications. Applications developed using Java EE are also known as enterprise applications.

Java ME - Java Micro Edition (Java ME) is used to develop applications for devices with limited resources such as limited RAM and processor capabilities. This version is used for developing applications for mobile and handheld devices.

1.3.1 Java EE Platform Stack

Java EE platform refers to the application environment where developers can write Java enterprise applications. It consists of Java EE specification, Test Suite for testing the applications, and enables the developers to write code against reference implementations present as a part of the Java EE platform.

Java EE platform also comprises the following components:

- **A set of specifications for the platform** – The specification of the Java EE platform are defined in terms of architecture, security, transaction management, resources, Application Programming Interfaces (APIs), interoperability, and so on.
- **Implementation of specifications** - The specification only defines the standard way in which the component can be implemented. Actual implementation of the specification has to be developed. These implementations can further be open source or proprietary products. The implementations are supported through a set of tools required for managing and monitoring the implementation.
- **Java EE Software Development Kit** – The Java EE SDK provides a basic set of necessary implementations and tools required to develop and build a Java EE application.
- **Java EE components and applications** – The Java EE platform is built to develop and build application and respective components.

1.3.2 Java EE Application Model

An enterprise application is usually composed of a distributed multitier architecture. The functionality of the application is separated into multiple isolated functional areas which can be independently implemented. Each of the isolated functional areas is termed as a tier in the application.

Following are the application tiers implemented in most of the enterprise applications:

- Client tier
- Middle tier which in turn may have a Web tier and business tier
- Enterprise Information Systems tier

Figure 1.6 shows a graphical representation of distributed multitier application model for enterprise applications in Java.

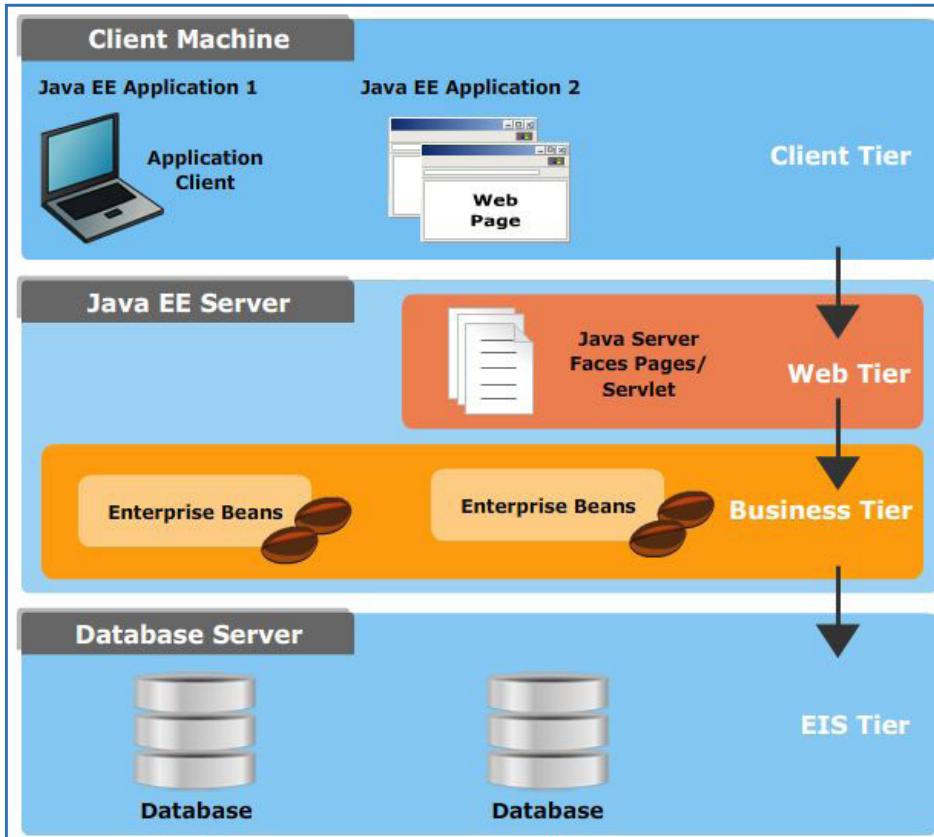


Figure 1.6: Enterprise Application Architecture

Functional components of each tier may reside on a different machine of the application infrastructure. Following are the functions of each of the tiers:

- **Client tier** comprises the application clients or Web pages which access the application. End users of the application access it through the client tier. A request is initiated from the client tier which is further forwarded to the business tier in case of enterprise applications and Web tier in case of Web applications. The client tier also receives the response from the business tier and returns it to the end user.
- **Web tier** as the name suggests is responsible for handling communication of the application components over the Internet. The Web tier contains Java EE technologies such as JavaServer Faces (JSF), Servlets, Expression language, Contexts, and Dependency injection.
- **Business tier** of the application implements the business logic of the application. This tier is customized and developed according to the application requirement. The technologies used in business tier are Enterprise Java Beans (EJB) with respect to enterprise applications, JAX-RS Restful Web services in case of Web services, and Java Persistence API entities.
- **Enterprise Information Systems tier** comprises the data repositories of the application that is, the database servers and legacy systems such as mainframes. The EIS tier is accessed through the business tier of the application. It cannot be directly accessed from the client tier. The Java EE technologies used in this tier are Java Database Connectivity

(JDBC), Java Persistence API, Java EE Connector Architecture, and Java Transaction API.

A Java EE component can be defined as a self-contained functional software unit which is one of the application components that forms the application. It is independently developed and assembled into the enterprise application. An enterprise application can have different types of components.

Java EE components are written in Java programming language. They are compiled in the same way as other Java classes. The application components are developed to be compliant with the Java EE specification. They are run and managed by Java EE server.

1.3.3 EJB as Business Components

EJBs are server-side components in the application. An enterprise application has a user interface which can be application clients, Web pages, and so on through which the end user puts the application to utility. The data received from the user has to be processed by the application and then either stored on the database or returned to the client. Processing of data is defined by the business requirements of the application and implemented through EJBs in case of Java. EJB's can communicate with both the end user and the database of the application.

Following are the functions that can be carried out by EJBs:

- **Implement business logic:** The business logic of an application is defined by the requirements of the application. It is implemented by EJBs by using various supporting APIs provided by Java EE. Java EE provides APIs for generic tasks such as connecting to the database, sending messages to the components, application security and so on. The developer can make use of these pre-existing APIs and concentrate on creating the business logic of the application.
- **Access database:** While processing the data received from the user the EJBs might have to access the database. EJBs access the database to read and write data from it and do not allow the application client to access the database directly ensuring data security. The authentication to data access and retrieval are also implemented by EJBs.
- **Integrate with other systems:** An application when deployed has to work with existing legacy systems and also be compatible with emerging systems. This integration of the application is performed by the EJBs. EJBs make use of Java EE connector architecture to achieve this.

EJB enables development and deployment of distributed components in the enterprise development done in Java language.

Remote components were earlier accessed through Remote Method Invocation (RMI). Following are the steps based on which RMI is implemented:

1. When a component has to access another component located over the network, the client invokes a stub. The stub at the client end is a proxy which is responsible for masking the network communications from the end user.
2. The stub invokes the component through a skeleton.

The skeleton on the server-side is responsible for receiving requests from other components and invoking appropriate methods on the server.

3. The skeleton extracts the parameters from the request and invokes right methods to generate the response for the request received. Once the response is generated, the response is sent to the client.

There are some drawbacks of RMI such as loss of object identity which creates performance bottlenecks and so on.

EJB overcomes the drawbacks of RMI and other technologies like Common Object Request Broken Architecture (CORBA) through component and container framework model. It enables deployment and execution of business components in a distributed, multi-user environment.

1.3.4 Java EE Containers

Java EE provides various underlying services for enterprise application components through containers. A Java container provides support services such as transaction management, security, and so on, thus allowing application developers to concentrate on the business logic specific for the application, rather than implementation of services for the application.

Figure 1.7 shows various services provided by the containers.

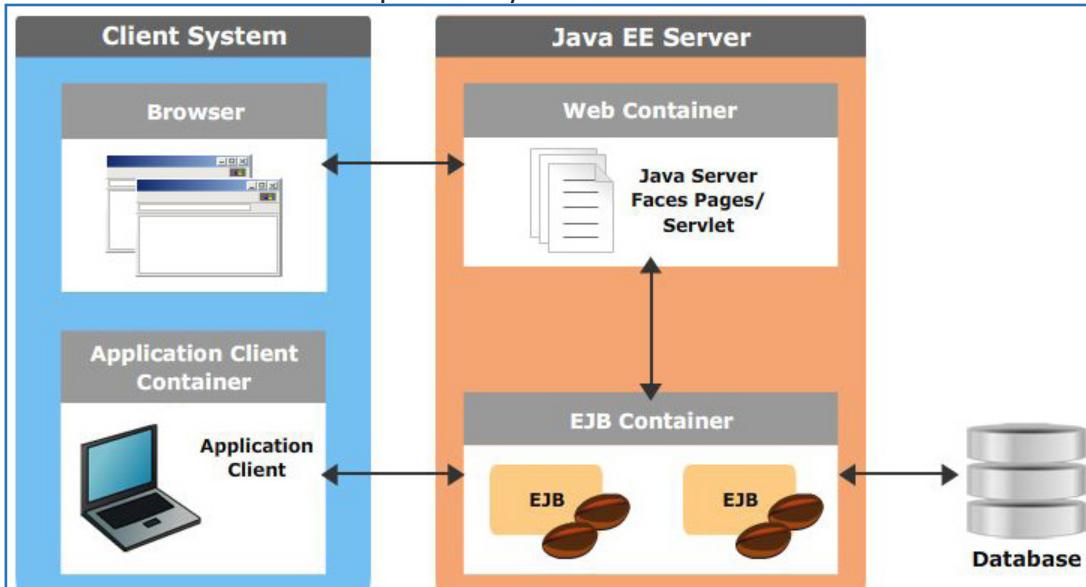


Figure 1.7: Container Services

Figure 1.7 shows different types of containers and the services provided by these containers.

Container can be defined as an interface between the application component and the underlying hardware/software on which the component is residing. All the application components are deployed into a container before execution.

The containers in an enterprise application can be categorized as:

- EJB container** – Manages the execution of enterprise beans on Java EE server.
- Web container** – Manages the execution of dynamic Web pages, Servlets, and other Web components.
- Application client container** – Manages the execution of application clients.
- Applet container** – Applets are independent applications which access the application, this container manages the applets.

The services provided by the container to the application component are as follows:

- Security** – Java EE provides for security model by configuring the container. The authentication and authorization mechanisms can be configured in the container while deploying the application.
- Transaction support** – The container provides transaction support to the application component. The Java EE transaction model links various methods which execute as a part of transaction and treats them as a single unit. Transaction management is crucial for integrity of the data sources of the application.
- Java Naming and Directory Interface (JNDI)** – The container provides naming services through the container to ensure that the objects of the application are appropriately accessed.
- The Java EE communication model is implemented by the container to provide low level communication between clients and enterprise beans.
- The container also manages enterprise bean and servlet lifecycles, database connection, resource pooling, data persistence, and access to the Java EE platform APIs.

Apart from these services, J2EE container also provides the following services to the applications it hosts:

- Deployment-based services** provided by the container enable deployment of the application components based on deployment descriptors. Deployment descriptors are declarative XML files which contain deployment information.
- API-based services** are those provided by different APIs which are part of Java. These APIs are used by the application and the container manages these services.
- Inherent services** are those services which manage the lifecycle of the components deployed in the container.
- Vendor-specific services** provided by the container are those which are essential for the application end user. These are services such as load balancing, scalability, high availability and so on.

1.3.5 Java EE APIs and Services

Java EE provides various APIs for creating enterprise applications. These APIs are used for developing different components of the application. APIs also provide a specification for development of components so that the assembling process of the enterprise application is simpler.

Figure 1.8 shows all the APIs which are part of Java EE 7.

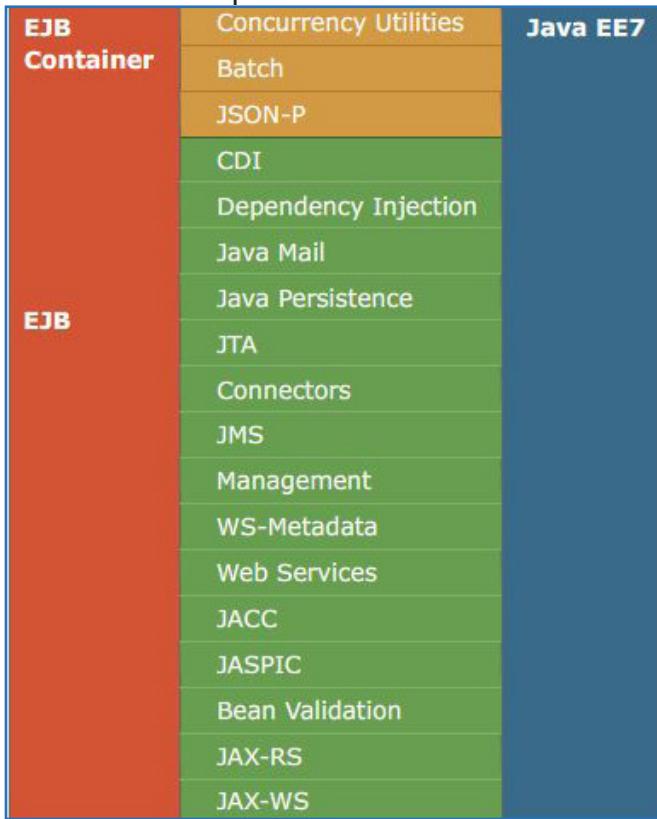


Figure 1.8: Java EE APIs in EJB Container

In figure 1.8, the APIs highlighted in yellow are the new ones introduced in Java EE 7.

Following are some of the commonly used APIs in application development:

- **Java Persistence API** – The persistence API uses an object-relational mapping approach to enable interaction between the beans and the database.
- **Java Transaction API** – The transaction API sets the boundaries of the transactions in the application and manages multiple transactions among the components of the application.
- **Java API for RESTful Web Services** – Defines APIs for the development of Web services based on Representation State Transfer architectural style.
- **Contexts and Dependency Injection** – This defines a set of services provided by the container to enable the enterprise beans to function along with the JSF pages.
- **Bean Validation** – This defines a Meta data model and an API for validating Java bean components.
- **Java Message Service API** – Enables communication between application components in a loosely coupled, reliable and asynchronous manner.
- **Java EE Connector Architecture** – It is primarily used for system integration to create resource adapters that access the enterprise information system.
- **Java Mail API** – The Java Mail API provides interfaces for sending and receiving e-mail notifications. The Java Mail API has two interfaces – application level interface and a service provider interface.

1.3.6 Java EE 7 Software Development Kit

Java EE 7 Software Development Kit acts as a base for Java application development. This is a pre-requisite for installation of GlassFish server and NetBeans IDE. It comprises the runtime libraries and basic components for executing, debugging, and compiling of Java programs.

1.3.7 Java EE Application Servers

Application servers are entities of applications on which enterprise applications are deployed and run. Application servers consist of components such as database connectors, Web server connectors, runtime libraries, and so on.

All the components of the enterprise application are deployed on different containers on the application server. The application server comprises an EJB container and Web container. All the bean components of the application are deployed in the EJB container and all the Web components of the application are deployed in the Web container.

Following are the Java EE application servers:

- WildFly
- GlassFish
- WebSphereAS
- WebLogic server
- Apache TomEE

Among these servers, GlassFish is the reference implementation of Java EE and supports Enterprise Java Beans, JavaServer Faces (JSF), Java Persistence API (JPA) and Java Messaging Service (JMS). Here, GlassFish 4.0 is being used along with NetBeans IDE.

Figure 1.7 shows the components deployed in an application server and their interaction with the client. It also depicts components of a generic Java EE server and their interaction with other components of an enterprise application.

NetBeans is an integrated development environment which is used for development in Java. The IDE is written in Java and supports modular development of applications.

Figure 1.9 shows the NetBeans IDE interface.

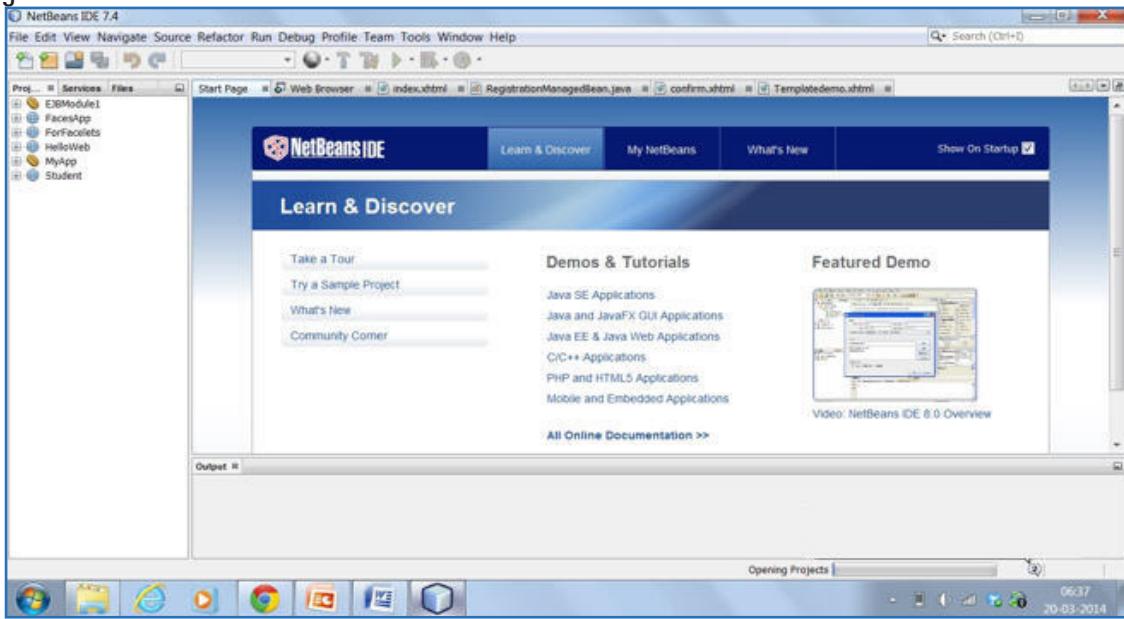


Figure 1.9: NetBeans IDE for Application Development

1.4

Java EE Profiles

Java Runtime Environment comprises the Java Virtual Machine (JVM), core Java classes, and various other libraries. However, every application does not require all the libraries provided by the runtime environment. Therefore, various profiles are defined for the runtime environment. A profile is a subset of the libraries of the Java Runtime Environment (JRE), being a subset of libraries requires less compute resources.

EJBLite is a Java EE profile defined in Java EE7 platform. It allows delivering a subset of EJB. EJBLite has the following features implemented:

- It implements all the three variants of session beans – Stateless, Stateful, and Singleton.
- It implements local EJB interfaces or no interfaces.
- Interceptors are part of EJBLite.
- It supports container-managed and bean-managed transactions.
- The security mechanisms are declarative.
- It also supports Embedded API.

Table 1.1 shows the comparison of the components in EJB 3.1Lite and Complete EJB 3.1 API.

Components	EJB 3.1Lite	EJB 3.1
Session	Yes	Yes
Message-driven	No	Yes
2.x/1.x	No	Yes
Java Persistence 2.0	Yes	Yes

Table 1.1: Components of EJB 3.1Lite and Complete EJB 3.1

Table 1.2 shows the comparison of the Session Bean Client Views in EJB 3.1Lite and Complete EJB 3.1API.

Session Bean Client Views	EJB 3.1 Lite	Complete EJB 3.1
Local	Yes	Yes
No-interface	Yes	Yes
3.0 Remote interface	No	Yes
2.0 Remote interface	No	Yes
Web Service End point	No	Yes

Table 1.2: Session Bean Client Views of EJB 3.1Lite and EJB3.1

Table 1.3 compares the services provided in the EJB 3.1Lite and Complete EJB 3.1.

Services	EJB 3.1Lite	EJB 3.1
Timer	No	Yes
Asynchronous Session Beans	No	Yes
Interceptors	Yes	Yes
RMI-IIOP interoperability	No	Yes
Transactions	Yes	Yes
Security	Yes	Yes
Embeddable API	Yes	Yes

Table 1.3: Services Provided by EJB 3.1Lite and EJB 3.1

Check Your Progress

1. Which of the following is not newly introduced in Java EE 7?

(A)	JMS	(C)	Concurrency utilities
(B)	JSON	(D)	Batch processing

2. Which of the following technologies are not based on component frameworks?

(A)	Remote Method Invocation	(C)	Enterprise JavaBeans
(B)	Common Object Request Broker Architecture	(D)	None of these

3. Which of the following options are not a part of EJB specification in Java EE 7?

(A)	Session beans	(C)	Entities
(B)	Message-driven beans	(D)	None of these

4. Which of the following components is part of the client tier in enterprise application?

(A)	Web browser	(C)	Servlets
(B)	Database	(D)	Beans

5. Which of the following is neither an application server nor a Web server?

(A)	WildFly	(C)	Apache
(B)	GlassFish	(D)	NetBeans

Answer

1.	A
2.	A
3.	C
4.	A
5.	D

Summary

- An enterprise application is a large business application. It is usually hosted on servers and simultaneously provides services to a large number of users over a computer network.
- Application architecture divides application components into tiers or layers based on their functionality in the environment.
- Distributed application architecture divides the large monolithic application into layers.
- As enterprise application execution is distributed across multiple tiers, there are various issues or concerns with respect to the design of these applications.
- The purpose of Java EE platform is to build large scalable, multi user applications.
- The business logic of an application is implemented by EJBs by using various supporting APIs provided by Java EE.
- EJB is based on component framework model where the application comprises communicating components.
- Enterprise application components are deployed into containers that provide supporting services such as transaction management, security, and so on to the application.
- Application servers are entities of applications on which enterprise applications are deployed and run.
- EJBLite is a Java EE profile, which is a compact runtime environment including all the essential application entities.

GROWTH
RESEARCH
OBSERVATION
UPDATES
PARTICIPATION



www.onlinevarsity.com



Welcome to the Session, **Enterprise JavaBeans**.

This session explains the component-based development approach followed by enterprise applications. The session describes the server-side Enterprise JavaBeans (EJBs) components and their characteristics. Then, the session explains how to register the components using Java Naming and Directory Services (JNDI) on the application server to be accessed remotely by the clients. The session describes the various tools used for developing EJB components and packaging of EJBs components on the Java EE platform.

In this Session, you will learn to:

- Describe the principles of component-based development
- Define Enterprise JavaBeans
- List the characteristics of enterprise JavaBeans
- Describe the evolution of enterprise JavaBeans
- Explain the features of EJB 3.0
- Explain the different types of Enterprise JavaBeans
- Explain the JNDI service on Java EE platform
- Explain JNDI APIs
- Describe the various roles involved in EJB application development
- Explain the various steps involved in developing and packaging an enterprise application
- Describe tools used for developing enterprise application

2.1 Introduction

Enterprise applications are developed to fulfil the needs of a specific domains such as banking, finance, and so on. They can be used by large number of users in that domain. The enterprise application uses Enterprise JavaBeans (EJBs) as business components on the middleware tier of the multilayered application architecture. The business components are used to implement the business logic of the application. The implementation of EJBs on the middleware tier is laid by the EJB specification provided by Sun Microsystems.

Enterprise applications developed using EJB specification are based on component-based development framework.

According to the specification, applications are implemented as independent components which interact with each other. These components interact with each other in a loosely coupled manner to implement application related tasks.

2.1.1 Principles of Component-Based Development

Components can be defined as a reusable program building blocks which are used to implement functionality in the enterprise applications.

Features of components are:

- They have an interface defined so that other application components can access it.
- They have a lifecycle mechanism with well-defined initiation, process, and termination phases.
- They should be configurable and should also have a third party integration scheme.
- Components can be assembled with other similar program blocks to build a complete application.
- They are portable and reusable.
- They can function independently or when assembled with other components.

Advantages of creating a component-based application are:

- Reduction of cost and time in building large complicated systems as there is a reuse of components.
- Better quality of software is ensured because the components are tested and stabilized through multiple iterations of testing.
- A developer creating an application component does not require knowledge of the entire application. Since, the components are independently developed, the application development process is quick.



Apart from EJB, CORBA is also a component-based application development framework.

CORBA is a standard defined to enable interoperation of different application components developed on different platforms.

2.2 Enterprise JavaBeans (EJBs)

EJBs are server-side components of an enterprise application. When an application client requests the service of an application, then the server provides the service through the enterprise beans.

EJB technology provides a platform for developing portable, reusable, and scalable business applications. The EJB components of the application are built according to the EJB specification. EJB specification has evolved from EJB 1.0 to EJB 3.2, which is the current version.

All the beans on the server are deployed into a container, which in turn provides generic services to these server-side components. The services provided by the container include security, persistence, transaction management, and so on.

There are two ways of looking at EJBs that are as follows:

- EJB are specifications** - The EJB specifications are provided by Sun Microsystems. These specifications lay out rules and standards on how you should code your EJBs.
- EJBs are Java interfaces** - EJBs are coded by the implementing the EJB interfaces. The EJB interfaces are exposed to the EJB clients.

2.2.1 Characteristics of Enterprise JavaBeans

Following are the important characteristics of Enterprise JavaBeans:

- EJBs implement the business logic of the application.
- EJBs are deployed in the EJB container on the Java-enabled application server.
- Deployment descriptors are XML files which define the deployment of EJBs on the server declaratively. They comprise XML tags which are used to define the configuration of the beans.
- EJBs communicate with the application client through a no-interface view or business interface.
- Container on the application server provides services such as transaction management, security services, lifecycle management of the beans, providing access to the remote clients, and so on for the EJBs deployed in it.

2.2.2 EJB as a Component

An application can be developed consisting of several reusable components. The main requirement of a component is that it should encapsulate the behavior of an application. Users are not aware of the internal processes of the components in an application, however, they are aware of what they need to pass in as input and what to expect as output.

Figure 2.1 displays the different tiers used for deploying an application.

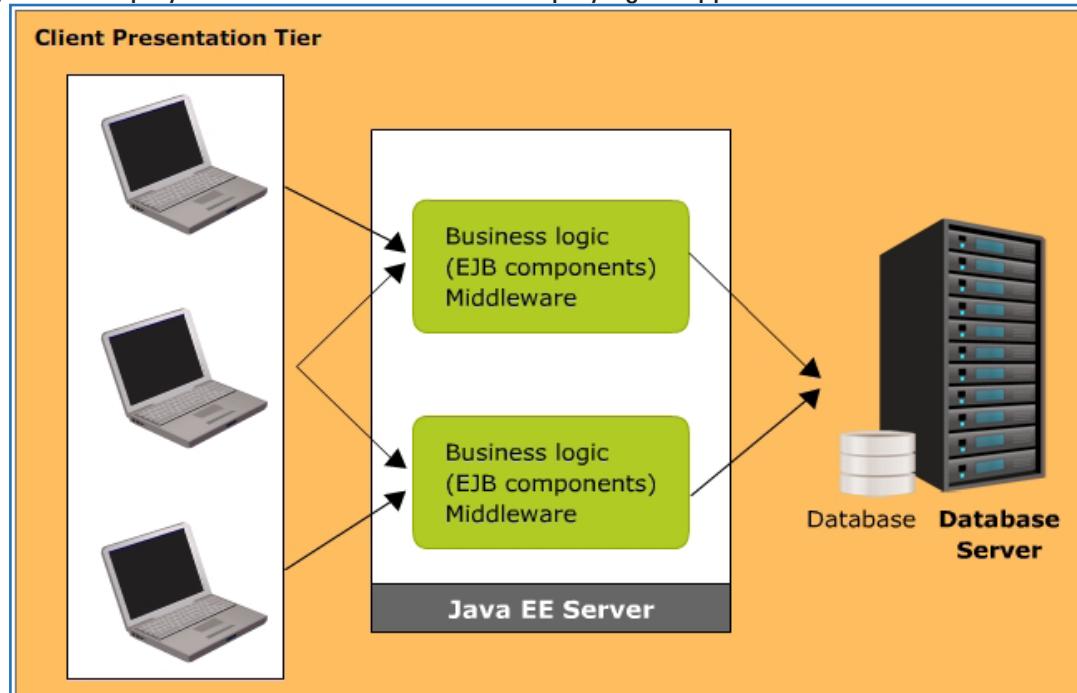


Figure 2.1: Application Deployment in Multi-tier Environment

In figure 2.1, the presentation layer contains the logic for displaying a user interface to the client, the middle-tier contains the actual business logic, and the database tier provides data services.

The foundation of an application is the data that an application contains utilizes. The data layer consists of a database and the access mechanisms used by the application for accessing the database. JDBC API is the low-level interface for accessing data from the relational databases.

The application data on its own is not of any use till some processing logic is applied on it. The logic is applied in the business layer and the code in this layer maps to a company's business processes. For instance, a retail chain gives a 5% discount to each 100th customer in its stores. This type of requirement needs to be implemented in the business components running on the business layer.

The EJBs therefore sit behind presentation layer components and provide services to them. These presentation layer components are referred as clients. Java Applets and applications, Web pages, and Web services can act as EJB clients.

Figure 2.2 shows the functions of EJB.

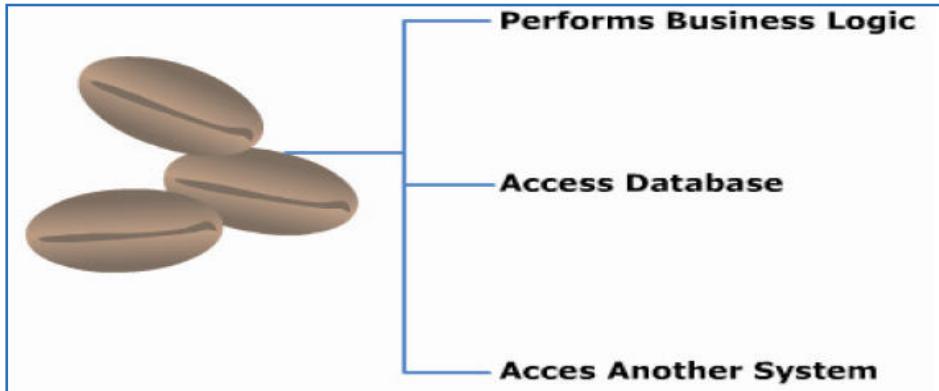


Figure 2.2: Functions of EJB

EJB components exist in a container. The container and the components together can be viewed as a framework, where EJB container provides valuable services for enterprise components.

When an EJB is deployed in the EJB container, the EJB container creates multiple instances of the EJB to serve multiple clients. Collection of bean instances or beans in the EJB container is referred as bean pool. The properties of the bean pool depend on the settings that are provided with the EJB during deployment. These settings pertain to the number of beans in the pool at a time, how many new beans should be added when the container runs short of beans, the maximum bean pool size, and so on. The container creates separate bean pools for each EJB. Figure 2.3 shows a container with bean pools.

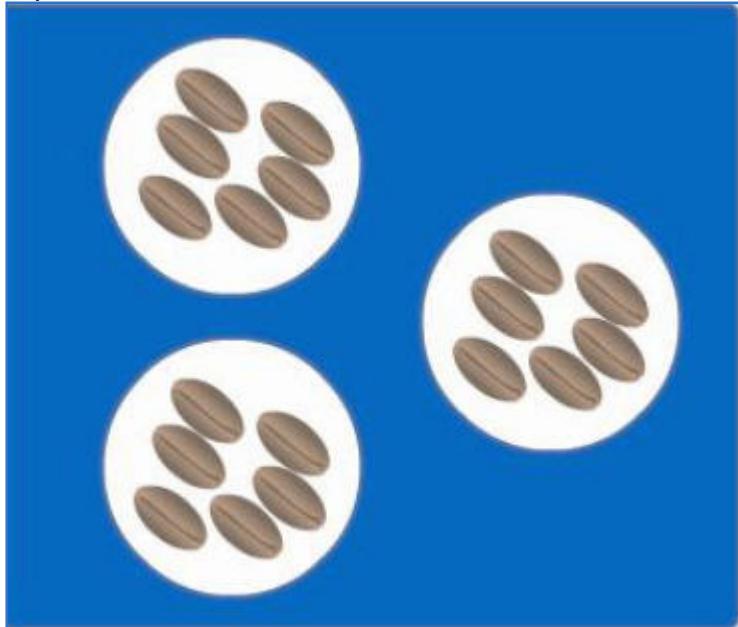


Figure 2.3: EJB Container with Bean Pool

2.2.3 Evolution of EJB

The EJB specification provides a framework for component-based application development. The business logic is implemented through independent EJB components which are deployed on the server. These components are accessed by the application clients through interface. However, the access method in the earlier versions of EJB was different. EJB has evolved according to the changing needs of application development process.

The evolution of application architecture since EJB 1.0 is as follows:

- **EJB 1.0** - In EJB 1.0, there were two types of enterprise beans – Session beans which established a communication session among the application components and Entity beans which were responsible for database related operations in the application.
- **EJB 1.1** – EJB 1.1 introduced XML-based deployment of the application. Earlier, this was done through metadata embedded in the application code.
- **EJB 2.0** – EJB 2.0 introduced local interfaces to simplify the communication between the local client and the bean component. Message-driven beans were introduced in this version, which were enterprise beans based on messaging system.
- **EJB 2.1** – EJB 2.1 added features such as timer service and support to Web services. The deployment descriptor had XML schema to define the deployment of the application.
- **EJB 3.0** introduced annotation-based programming model. All the enterprise beans are POJOs. These objects have methods to implement application functionality. The beans in the application are injected through annotations from this version of EJB. The entities in the earlier versions which were handling the database operations were now deprecated and replaced by the Java Persistence API.
- **EJB 3.1** – EJB 3.1 introduced no-interface view of accessing the beans. It also offered Singleton pattern of development. It introduced a Java EE profile, EJBLite, which is a subset of EJB container functionality.
- **EJB 3.2** – EJB 3.2 has made enhancements in handling the timer service, managing the bean lifecycle methods, activation, and passivation of Stateful Session beans.

2.3 EJB 3.0

In EJB 3.0, components are implemented as independent components. A component is a Plain Old Java Object (POJO) class with some additional features. EJB components comprise Session beans, Message-driven beans, and Entity beans. Session and Message-driven beans are used for implementing the business logic, whereas the Entity beans are used for implementing persistence of an EJB object to the external storage.

EJB 3.0 provides several benefits to applications. Some of these benefits are:

- **Simple** - The developer can focus on developing the business logic instead of concentrating on other services such as transaction, security, resource pooling, and so forth. EJB 3.0 provides a practical outlook and does not demand much understanding of theoretical intricacies. These system level issues are taken care of by the EJB container.
- **Reusable** - The need of reusability aroused as EJB 2 required re-configuration in the components for their reuse. EJB 3.0 is a reusable component. It can be used by multiple applications that can make calls to the deployed enterprise bean.

- **Scalable** - EJB is used when application needs to scale beyond initial low level usage level and support multiple concurrent users. Scalability is the ability of an application to function properly without degradation in performance even when the load on application changes.
- **Transactional** - EJB helps in transaction management. The developers use EJB for transaction concept to provide systems with atomicity, consistency, isolation, and durability properties. This helps to maintain the consistent state of the database when an error takes place. Transaction management is performed by the EJB container which is a component in the EJB architecture.
- **Vendor support** - EJB 3.0 is supported by most of the well-known organizations such as Oracle, IBM, JBoss, and so on. The advantages of being used by different vendors are as follows:
 - Rise or fall of a company will not affect the growth of the technology.
 - Developers can take advantage of other technologies both inside and outside the Java world.
 - Vendors compete with each other by providing value-added non standard features.

Some of the important features introduced in EJB 3.0 are as follows:

- Use of annotations
- Callback Methods
- Elimination of Home interface
- Elimination of component interface
- Dependency Injection
- Interceptors
- Java Persistence API (JPA)
- Timer service

2.3.1 Use of Annotations

EJB 3.0 uses metadata annotations to specify the services that the EJB components will use when it is deployed in the container. Annotations help the developer to provide the specifications and based on that specifications, the system automatically adds code. Metadata annotations simplify the development and testing of the application. Annotations, which are used extensively in the Java enterprise platform that help the developers to transform a simple POJO to an EJB. Besides specifying the required services, annotations can also be used to specify the component type.

Annotations are processed at compile time. They are also used during application deployment.

Code Snippet 1 shows the use of annotations.

Code Snippet 1:

```
...
@Stateful
public class HelloBean implements HelloInterface {
    @Remove
    public void removeBean() {
        //close all resources
    }
    ...
}
```

In Code Snippet 1, `@Stateful` annotation indicates that **HelloBean** is a Stateful Session bean. It also indicates that the required code (bean artifacts) for the bean will be generated by the IDE, when a bean is created. The `@Remove` annotation indicates that the `removeBean()` method will be invoked by the container when it is about to destroy the bean instance.

Some of the advantages of using annotations are as follows:

- Ease of use** – Annotations are checked and compiled by the Java language compiler and are simple to use. Many vendors such as IBM and BEA have introduced the annotation feature in attributes for the deployment descriptor.
- Portability** – Annotations are portable.
- Type Checking** – Annotations are instances of annotation types and are compiled in their own class files.
- Runtime Reflection** – Annotations are stored in the `.class` files and accessed for runtime access.

If the bean developer and deployer are two separate entities, then before the developer generates the bean deployment descriptor, they need to read the code so that the deployment descriptor does not override the bean provider's deployment metadata-specified configuration. Another important issue with metadata is that each time a change is made to the bean code, the bean needs to be recompiled and repackaged.



Deployment descriptor is not mandatory in EJB 3.0, annotations can serve the purpose.

2.3.2 Dependency Injection

Dependency injection is a means through which the container creates the required operational environment for the application. It provides the environment by injecting required resources into the application. Appropriate environment variables are injected into the bean's variables or methods. The developer specifies the required resources either through deployment descriptor or annotations.

`@Inject` and `@Resource` annotations are used for dependency injection.

Bean files also support dependency injection through JNDI lookup. They use `lookup()` method of the `EJBContext` interface for dependency injection.

2.3.3 Callback Methods

Callback methods are those methods which are invoked when various lifecycle events with respect to the enterprise bean occur in the application. In versions earlier to EJB 3.0, developers had to define these methods irrespective of whether they were used in the application or not. EJB 3.0 makes the implementation of callback methods such as `ejbPassivate`, `ejbActivate`, `ejbLoad`, `ejbStore`, and so on optional. If the developer defines one of these methods, the container will invoke the method.

Another important change introduced in EJB 3.0 is that any method can be designated as callback method to listen to lifecycle events. The developer can also use the callback listener class instead of defining the callback methods in the bean class.

Figure 2.4 shows the EJB architecture of EJB 2.0.

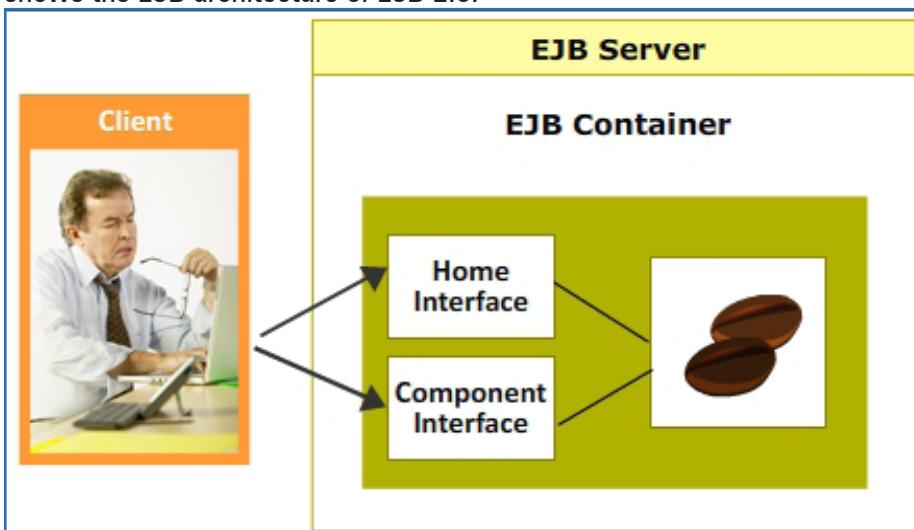


Figure 2.4: Architecture of EJB 2.0

Figure 2.4 demonstrates that a client could access an enterprise bean either through a home interface or component interface.

These were replaced by business interface and no-interface view in EJB 3.0.

2.3.4 Elimination of Home Interface

EJB 3.0 has simplified the development of bean by removing the home and object interfaces. The home interface and the home objects have been replaced by Plain Old Java Interfaces (POJI) and POJO respectively which together form the business interface. The new Session bean uses a business interface to contain all the business methods. This business interface is designated by the bean developer as remote business interface or local business interface. The business methods present in the remote business interface can raise any exception except `java.rmi.RemoteException`. All other exceptions such as protocol, system level, and so on are encapsulated in the `javax.ejb.EJBException` class which is a subclass of the `java.lang.RuntimeException`. This exception is thrown to the client by the container.

A Message-driven bean does not require any client invocation. Hence, there is no need for business interface and business methods in it. In case of errors, the container logs the error and communicates to the corresponding resource adapter instead of the client.

2.3.5 Elimination of Component Interface

EJB 3.0 has done away with the use of component interface. In earlier versions, the component interfaces were used as they provided a way for the container to notify the bean instance of the various lifecycle events affecting it. These component interfaces contained the declaration of the various lifecycle methods such as `ejbPassivate`, `ejbActivate`, `ejbLoad`, and so on. It was mandatory to implement these methods in the bean class. These lifecycle methods were invoked by the container whenever an event occurred.

EJB 3.0 onwards the developer need not implement the lifecycle callback methods of the beans. For example, the `ejbDestroy()` method enables the container to notify the Message-driven bean instance that it will destroy. The `ejbDestroy()` method and the bean class perform tasks such as closing the JDBC connection and free resources held by the bean. The bean is represented as a simple POJO class implementing the business interface if it is a session bean.

The two ways in which a bean class can receive notification from the container are as follows:

- First, the developer can write a separate class containing the implementation of callback notification method. The container is then informed to treat the class as the bean's callback listener class.
- Second, the developer can implement the callback notification methods within the bean class and designate each of these methods to handle appropriate events.

Both these approaches require the use of annotations. Annotations are not only used for notifications, but also used for other purposes.

2.3.6 Interceptors

Interceptors are methods used to intercept business method calls or lifecycle callback method calls. They are methods which are executed before the event is actually intercepting. Interceptors can be used by Stateless, Stateful, and Message-driven beans. They are used to perform operations such as application auditing, logging, and so on to monitor the performance of the application. Interceptors can be defined as methods within the bean class or as a separate Interceptor class in the application.

2.3.7 Java Persistence API (JPA)

JPA is an enhanced EJB query language which can support execution of EJB queries with named and positional parameters. Java Persistence API enables the developer to write code independent of the underlying database provider. The business logic of the application can thus be developed independent of the database. It also bridges the gap between the object oriented interpretation of data in Java programs and relation based interpretation in databases.

2.3.8 Timer Service

Timer service is introduced in EJB 3.0 where the applications can execute in certain time interval. Timer service enables the developer to schedule various events of the application without manual intervention. Developers can setup timers and define the timer callback methods in the application which are to be invoked when the timer expires.

2.4 Types of Enterprise JavaBeans

There are two categories of enterprise beans – Session beans and Message-driven beans.

Figure 2.5 shows various categories of enterprise beans and their utility.

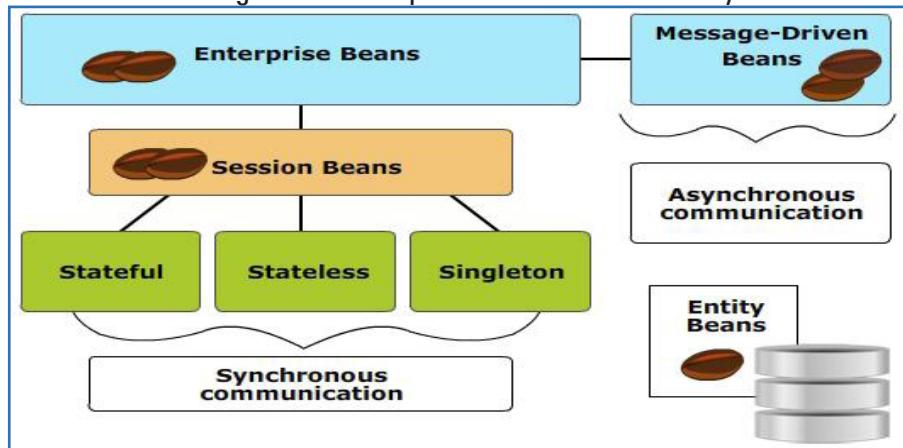


Figure 2.5: Types of Enterprise Beans

2.4.1 Session Beans

Session beans are used to implement a specific application task or a use case of the application. They implement business logic of the application. Session beans are deployed in the EJB container. They can be invoked by the client through a local, remote, or Web service through a business interface. They synchronously communicate with the client applications.

There are three variants of session beans that are as follows:

- **Stateful** – Stateful session beans are enterprise beans which retain the conversational state of the session between the client and server. Every client component has a unique stateful session bean instance which service his request.
- **Stateless** – A session bean is said to be stateless, if it does not maintain any conversational state with a client. Since there is no data to be maintained for each session, the same instance of a Stateless Session bean can support multiple clients.
- **Singleton** - Singleton session beans are invoked only once during the lifecycle of the application and are retained as long as the application is active.

2.4.2 Message-Driven Beans

Message-driven beans are enterprise beans which are asynchronously invoked through Java Message Service (JMS). These messages can be sent by an application client or another component of the application. Unlike Session beans, Message-driven beans cannot be invoked through interfaces, they can only be invoked through messages.

2.4.3 Entity Beans

Entity bean is an object whose data gets persisted in the database storage. It also implements business logic pertaining to the data in the application database. It manages application data and returns required data from the database. It is a server-side component.

2.5 Container Services for EJB

All the application components are deployed into EJB containers. Containers manage the lifecycle of the application components and provide services such as transaction management, security, and so on.

Following are the services provided by the container to the application components deployed in it:

- According to the security model implemented by a Java EE container, the container authenticates and authorizes users wanting to access the application and system resources.
- The transaction model of the container ensures that the methods in a single transaction are appropriately executed without leading to an inconsistent application state.

- The container provides naming services to access different components of the application without any conflict.
- The container also provides connectivity services to the application. When an application component deployed in the container attempts to connect to an external component, then the container manages all the lower-level communication tasks.

Enterprise beans are developed as Java classes. These beans are accessed by clients either through local interface or remote interface. Figure 2.6 shows the accessing of EJBs through remote interface.

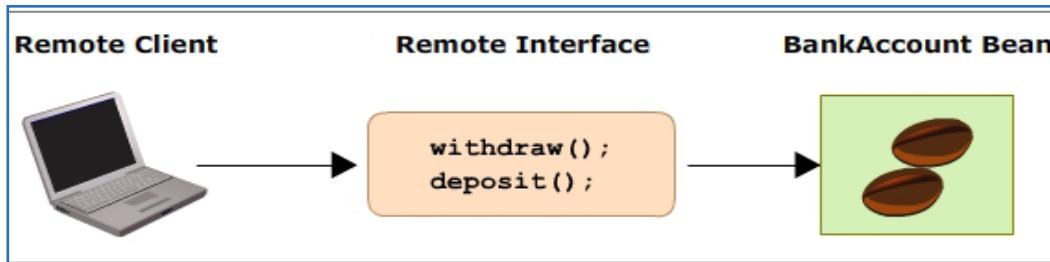


Figure 2.6: Enterprise Java Bean

2.5.1 Accessing Enterprise JavaBeans

Enterprise beans can be accessed through the interfaces provided by the bean class. They can be accessed either through the business interface view or no-interface view.

Business interface refers to the set of methods provided by the bean class through which the enterprise bean can be invoked.

No-interface view refers to all the public methods of the bean class which can be used to access the bean. The implementation of the interface methods is hidden from the client.

Consider an enterprise bean implementing the function of a calculator as shown in Code Snippet 2.

Code Snippet 2:

```

import javax.ejb.Remote;

@Remote

public interface Calculator {
    public String sayHello(String name);
    public int addition(int a, int b);
    int multiplication(int a, int b);
    int subtraction(int a, int b);
}
    
```

In Code Snippet 2, an interface has been demonstrated which is used to access a Calculatorbean in an application. The interface has four methods `sayHello()`, `addition()`, `multiplication()`, and `subtraction()`.

The methods `sayHello()` and `addition()` are part of the no-interface view to access the bean and the methods `multiplication()` and `subtraction()` are the business interface view to access the bean. The interface has to be implemented like any other interface in Java. When a client intends to access the `Calculatorbean`, it uses either of these two interfaces.

In order to access the bean methods, the application client should first obtain the reference of the bean instance. Remote or standalone clients can also obtain the bean reference through JNDI lookup and dependency injection using annotations.

An enterprise bean can be accessed either through a local client or through a remote client.

2.5.2 Local Clients

A local client runs in the same application where enterprise bean it is accessing. A local client can either be a bean component or a Web component. Local clients and the EJB are always located within the same JVM. The client can invoke methods on the bean, just as it will on any other Java object using normal Java method calls. However, exposing an enterprise bean to a local client does have a few drawbacks in terms of losing location transparency. This is due to the fact that a local client and the enterprise bean that it accesses must always be located in the same JVM.

A local client can access the enterprise bean through any one of the following methods:

- Through no-interface view. The local client can access the enterprise bean through the public methods defined in the bean class.
- The business interface can be annotated with `@Local`. The interface can be implemented through a different class, where the interface class name is provided as a property of the `@Local` annotation as, `@Local(interface.class)`.

2.5.3 Remote Clients

A remote client can run in a different application, different machine, or a different JVM. A remote client can be a bean component, a Web component, or an application client.

The location of the enterprise bean is transparent to the remote client. Remote clients can access the enterprise bean through a remote interface. It cannot access the enterprise bean through no-interface view.

The interface which accesses a remote enterprise bean is annotated with `@Remote`.

2.6 Java Naming and Directory Interface (JNDI)

A large scale enterprise domain has large number of objects interacting with each other. These applications require a well-defined naming infrastructure to access objects as per the requirement. JNDI is an API which provides these naming services. JNDI binds a name with an object in the application.

Figure 2.7 demonstrates how clients access the objects through a naming service.

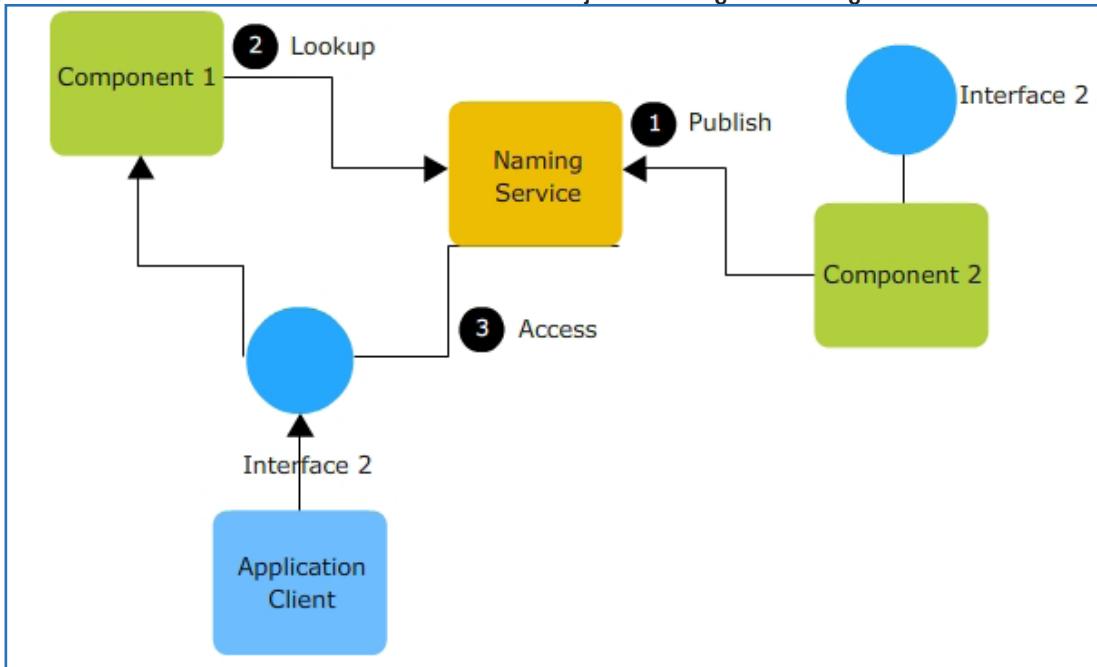


Figure 2.7: Naming Service

Figure 2.7 demonstrates the function of the naming service. In figure 2.7, component 2 is publishing its identity to the naming service. When component 1 tries to access component 2 it will perform a lookup operation onto the naming service.

An application client which is trying to access the component 2 will access it through the interface provided.

All the application components publish their identity to the naming service. Naming service binds a component with a JNDI name. When an application client or component has to access a bean it has to first perform a lookup based on the JNDI name. The naming service then provides reference of the component.

The client refers to an object. Through an object, the naming service implemented by JNDI is responsible for accessing the object using appropriate object references. All the remote clients of the application access the object through JNDI name. JNDI organizes the objects in the namespace in a hierarchy. The resource objects which can be accessed through JNDI names are enterprise beans, databases, and so on.

Figure 2.8 shows the JNDI architecture.

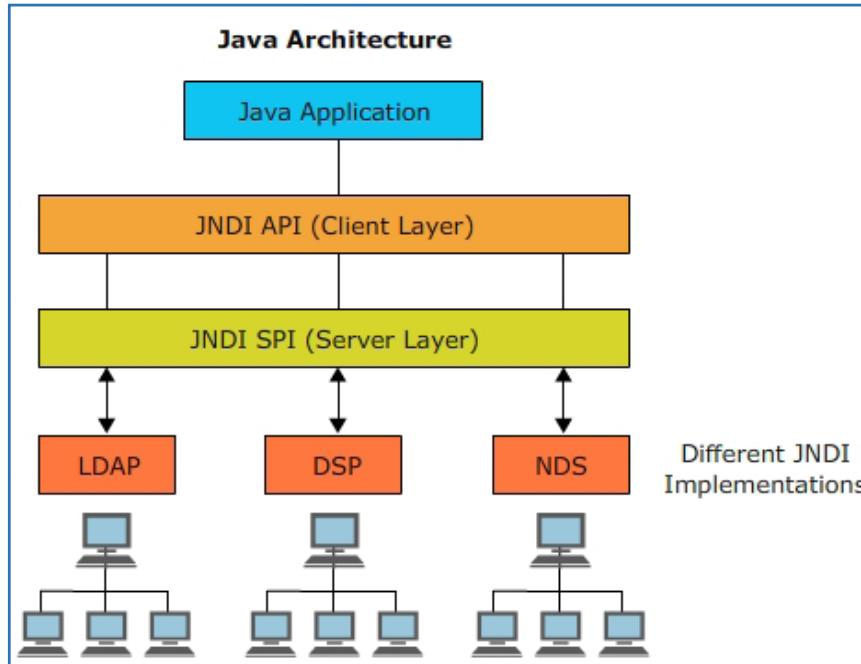


Figure 2.8: JNDI Architecture

JNDI is a naming service which is part of Java EE specification. It organizes the components of the application according to hierarchy and enables unique identification of the application components.

Applications access the JNDI service using JNDI API which provides a standard way for accessing the underlying objects or directory service. The JNDI architecture contains two parts:

- **JNDI API** – Are used by the remote applications to access the mapped components or objects from JNDI registry.
- **JNDI Service Provider Interface (SPI)** – Is a mechanism to plugin naming and directory services of different vendors on the Java EE platform.

2.6.1 JNDI API

The main package in the JNDI API is `javax.naming`. The `javax.naming` package contains two interfaces namely `Context` and `Name`. It also contains a class named `InitialContext` which represents the root node of JNDI hierarchy.

JNDI arranges all the JNDI names in the `Context` in a hierarchy and the access is relative to the `InitialContext`. The `InitialContext` class implements the `Context` interface and provides the root of the hierarchy, relative to which all the objects in the namespace are named.

Firstly, client establishes a connection to the JNDI service, also referred to as JNDI tree. After getting the connection to the JNDI service running on the Java EE platform, an object of `Context` is created. The `Context` object allows access to the system components and resources. The context representing a naming context consists of name-to-object bindings.

Thus, establishing the context is necessary, before accessing the resources from the JNDI hierarchy.

The client uses the `InitialContext` to establish the connection and retrieving the `Context` object.

Code Snippet 3 shows how to retrieve the `Context` object.

Code Snippet 3:

```
Context ctx = new InitialContext();
```

The `Context` class provides methods for binding and unbinding JNDI names with objects. It also allows creating sub contexts and provides methods for creating sub contexts. The methods provided by the `Context` class are as follows:

- `bind()` - It accepts two parameters, the JNDI name in string format and the object to which it has to be bound.
- `unbind()` - It accepts a string parameter and is used to unbind the specific object from the JNDI name.
- `lookup()` - It returns an object on which the lookup operation is performed. It accepts the JNDI name as parameter.
- `rebind()` - It also accepts two parameters, the JNDI name and the object.

Figure 2.9 shows how to obtain a JNDI context using `InitialContext()`.

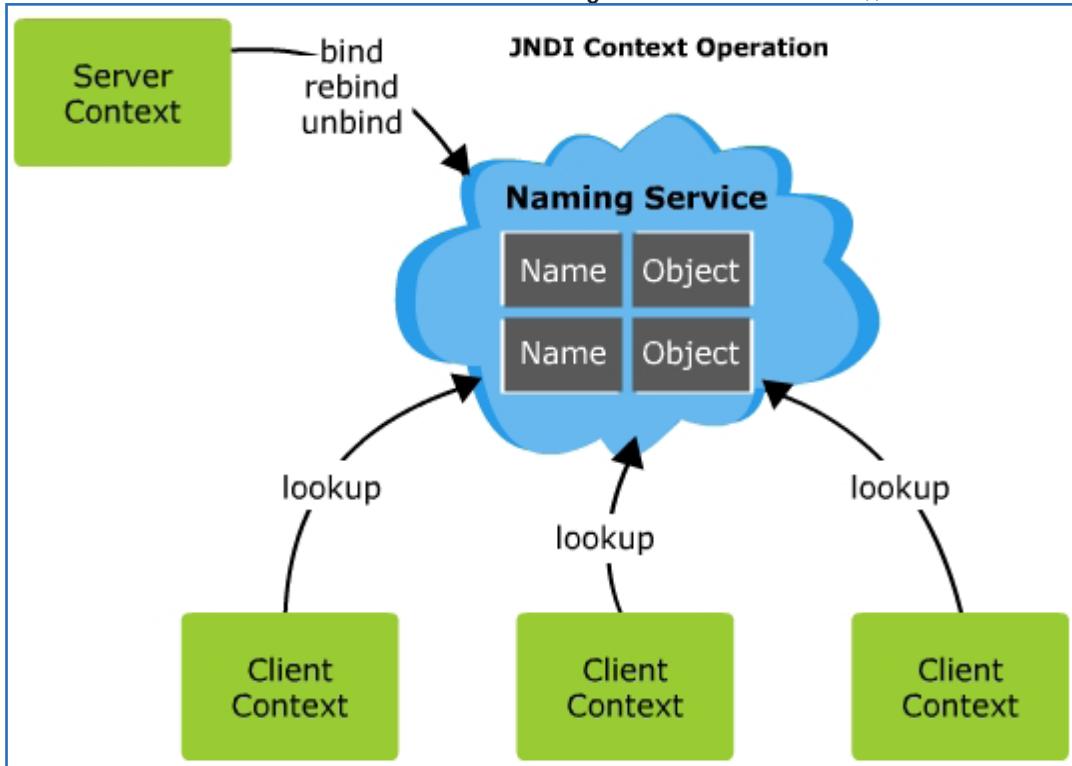


Figure 2.9: JNDI Context

The client performs the lookup of the objects registered in the JNDI registry and the application server provides the services for binding and unbinding the objects to the JNDI registry.

JNDI provides three namespaces based on which a component lookup is possible:

- **java:global** – This namespace identifies the component through a hierarchy of application **name** → **module name** → **enterprise bean name** → **interface name**. Following is the format of **java:global** namespace:

```
java:global[/application name]/module name /enterprise bean  
name[/interface name ]
```

- **java:module** – It identifies the component from the module-level of the naming hierarchy which is **module name** → **enterprise bean name** → **interface name**. The format of the **java:module** namespace is given as follows:

```
java:module/enterprise bean name/[interface name]
```

- **java:app** – This namespace is localised and identifies the enterprise beans packaged within the same application. The hierarchy of identification in this case also starts from **enterprise bean name** → **interface name**. The hierarchy can also start from the module name as **module name** → **enterprise bean name** → **interface name**, however it is optional. The format of the **java:app** namespace is as shown:

```
java:app[/module name]/enterprise bean name [/interface name]
```

2.7 EJB Roles

An application development process has various stages such as design, development, deployment, and administration. The design phase involves identifying the classes and objects that should be implemented for the application. It also defines the interactions among the objects of the domain. The development stage involves coding to implement the components using different technologies. The components developed in the development stage are deployed on the application server. Once the application is deployed, the access to the application should be given to different users of the application based on their roles and responsibilities in the application domain.

At every stage of application development there is a task to be implemented which is carried out by different roles assigned.

Figure 2.10 demonstrates various roles in the application and their mutual interaction in application development and deployment.

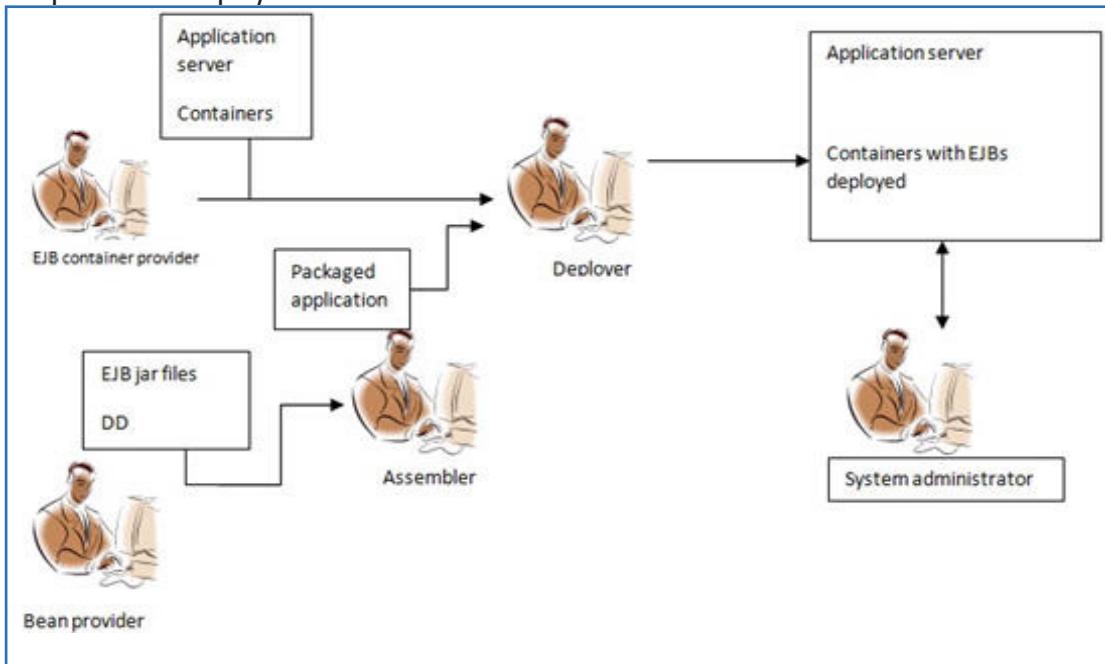


Figure 2.10: EJB Roles in Application Development

- **EJB container provider** - It is an infrastructure provider for the application development process. Container provider is responsible for implementing all the container services upon which the application can be deployed. This role is not application specific, but implements all the generic tasks which can be used among multiple applications.
- **Bean provider** – It is the application developer who implements application specific business logic. The bean provider packages the application beans into Java archive files to be deployed as part of the application. A bean provider can be a single developer or a team of developers working on the application development and coding.
- **Application assembler** – It has the knowledge of how all the components of the application interact with each other and is responsible for assembling all the application components into the final application. The assembler may have to write application integration code to make these components compatible with each other.
- **EJB deployer** – It follows the instructions provided by the application component provider and deploys the application onto the infrastructure. The application deployer puts the application onto the production environment. The deployer is responsible for securing the application through a firewall and makes hardware choices for appropriate application deployment. Deployer is also responsible for performance tuning of the application if hardware and software infrastructure of the application is given.
- **System administrator** – Its role is activated after the application is deployed into the production environment. The responsibility of an administrator is to provide access rights to the application end users according to their role in the application context. The administrator is also responsible for monitoring the application performance and informing other stakeholders if there are performance related issues.

2.8 Developing a Java EE Application

Developing a Java enterprise application involves various steps to be performed, that are as follows:

- Designing of the application
- Developing the components of the application
- Assembling and packing of the components as a single unit
- Deploying the file on the Java-enabled application server

2.8.1 Application Design

Apart from being used in traditional object-oriented design, Java EE design patterns are used for application designing as well. Enterprise application design should also consider issues such as application scalability and availability.

Design patterns are reusable solutions for commonly occurring software design problems. They provide tried and tested solutions for commonly occurring design problems. Each design pattern is described based on a consistent format and is divided into sections based on a template.

The output of this phase is a set of UML class diagrams which can be directly translated into classes and objects. A good design will result in simple conversion into the application code.

Apart from the classes, the designer should also identify whether the application requires a database component. During the analysis phase if the need for a database component is identified then, during the design phase the database structure of application should also be designed.

2.8.2 Coding

This phase of application development involves writing code to implement various components of the application defined during the design phase. The application components are user interface, database, enterprise beans, and so on.

If the application is developed using Netbeans IDE, then these components are organized into appropriate directories. Developers develop the components, debug, and perform unit testing during this phase of application development.

Developers can also create independent components which can be used by multiple applications. Interfaces are provided for the components through which they can be reused.

2.8.3 Deployment Descriptor

Deployment descriptor declaratively describes the interaction between various components of the application. It also defines the interactions between the components and the container in which they are deployed. The deployment descriptor is an XML file used by Java-enabled application server to obtain the references of the components.

Following information is specified in the deployment descriptor:

- The transaction management policy of the application.
- Security specifications and authorization constraints.
- Configuration variables used during application deployment.
- Resource access dependencies of the application.

The deployment descriptor separates the application configuration information from the application code, so that the application administration and management is simple.

Code Snippet 4 shows a deployment descriptor. It is created with the name `glassfish-ejb-jar.xml`. This is a standard name used for all the deployment descriptors of GlassFish server.

Code Snippet 4:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE glassfish-ejb-jar PUBLIC "-//GlassFish.org//DTD GlassFish
Application Server 3.1 EJB 3.1//EN" "http://glassfish.org/dtds/
glassfish-ejb-jar_3_1-1.dtd">

<glassfish-ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>CalculatorImplBean</ejb-name>
      <jndi-name>caljndi</jndi-name>
      <business-local>CalculatorLocal</business-local>
      <business-remote>CalculatorRemote</business-remote>
      <ejb-class>beans.CalculatorImpl</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
</glassfish-ejb-jar>
```

In the given deployment descriptor, a JNDI name '`caljndi`' is assigned to the ejb '`CalculatorBean`', When a client invokes a `lookup()` method with '`caljndi`' as parameter then a reference to the `CalculatorBean` is returned. The session bean is implemented through the class '`CalculatorImpl`'. It can be accessed through the local interface '`CalculatorLocal`' and remote interface '`CalculatorRemote`'. This is a Stateless Session bean which is managed by the EJB container.

2.8.4 Packaging

The steps for packaging and deploying a Java EE application are as follows:

- The Application Component Providers create Java EE modules, such as EJB, Web, Resource Adapter, and Application clients. These modules can be deployed independently without being packaged into a Java EE enterprise application.
- The Application Assembler packages these modules to create a complete Java EE enterprise application.
- Deployer deploys the deployable unit.

During the packaging of the modules, the Application Component Providers ensures that all the coding and file-naming conventions are met and the code of each module is frozen. The Application Assembler resolves dependencies between modules by creating references in the corresponding modules' deployment descriptors. Each module may have dependencies on other modules within the same archive, on modules in different archives, or both. The Application Assembler must resolve all such dependencies before deployment.

The Java EE specification provides different types of archive files to package the modules. These are as follows:

- EJB modules comprise all the class files pertaining to the business logic of the application. EJB modules are packaged as .jar files and may have a deployment descriptor.
- Web modules comprise the HTML files, JSPs, JSFs, and Servlet class files pertaining to the application. All the Web application files are packaged into a .war file for deployment and distribution. The archive file of the application may or may not have the deployment descriptor.
- Resource adapter modules comprise components which implement resource adaptation. In scenarios where the application is communicating with a database, resource adapter objects play an important role to translate the database interpretation of data to object-oriented interpretation. It also includes the native libraries used by the application. The resource adapter files are packaged as archive files using .rar extension.
- Application client modules are modules like user forms for the application; these modules are packaged as .jar files.

2.8.5 Assembling and Deployment

An enterprise application is developed as independent components. These components have to be assembled together and deployed into a single unit which can work as an application. The assembled application has to be deployed on an application server which can further be accessed by clients. The deployment is done either manually or through a tool. Deployment process also involves vendor specific configuration with respect to load balancing and performance tuning of application components.

According to the instructions provided in the deployment descriptors the archive files are deployed onto the application server.

The deployment descriptor for the ejb modules is `ejb-jar.xml` and for Web modules it is `web.xml`. These deployment descriptors are declarative XML files.

The application should follow proper naming standards so that the deployment can happen without any conflict. The JNDI lookup names for the EJB components should be unique across the application. Therefore, a consistent naming convention must be used across all the objects in the application.

The Application Assembler packages the components into an Enterprise Archive (EAR) file. All modules created in the application are packaged together in an `.ear` file. When packaging the components into an `.ear` file, the assembler should ensure the root contains a folder called `META-INF`. This folder contains a deployment descriptor called `application.xml`, one or more container-specific deployment descriptors, and `manifest.mf` file. The `application.xml` file contains the names of all the Java EE archives that are packaged in the EAR file. The `manifest.mf` file contains additional meta-information about the EAR file.

Besides the `META-INF` folder, the root of the file structure contains the archives of the Java EE modules that constitute an EAR file. These archives can be: `.jar` files for EJB module, `.war` files for Web modules, `.jar` files for Application Clients, `.rar` files for Resource Adapters.

These archives can be either placed in the root or in one or more sub-folders.

To manually package the components in an EAR file, the command can be run as:

```
jar cvf <name>.ear *
```

This command creates an ear file with the contents from the current folder. Figure 2.11 shows the file structure of EAR file.

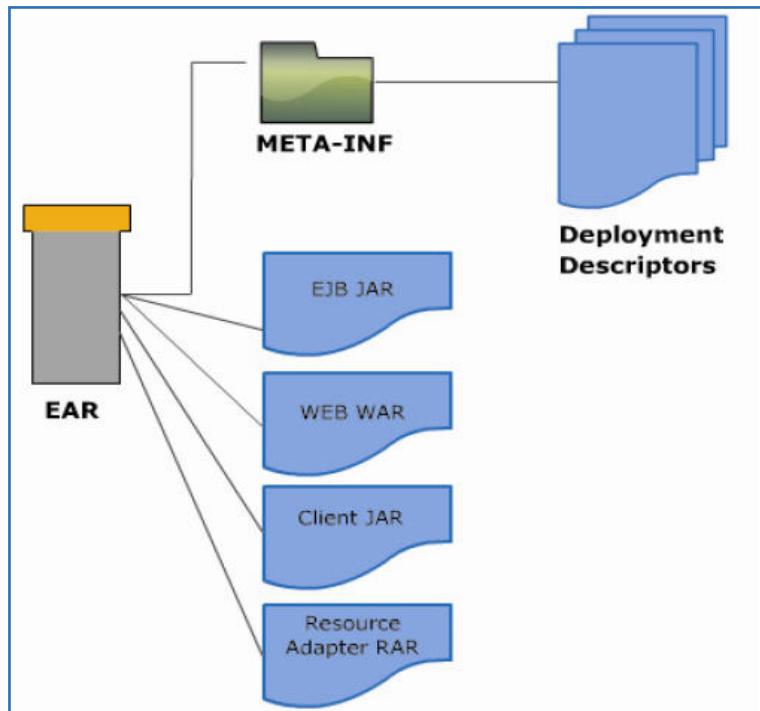


Figure 2.11: File Structure of EAR File

2.9 EJB Development Tools

The EJB development environment is provided by Integrated Development Environments (IDE) such as NetBeans, Eclipse, and so on. The IDE provide various tools to support the application development process. They also enable testing and debugging of the application.

Following are the categories of tools provided by the development environment:

- J2EE Perspective editor
- Tools for creating and accessing enterprise beans
- Tools for accessing the database and building persistence into the enterprise applications
- Tools for generating deployment code from the annotations
- Tools for creating applications based on design patterns such as session facades and so on

IDE provide these features to simplify the application development process. EJB being an open source specification has a spectrum of free tools to support these functions of application development.

Figure 2.12 shows a typical interface of NetBeans IDE.

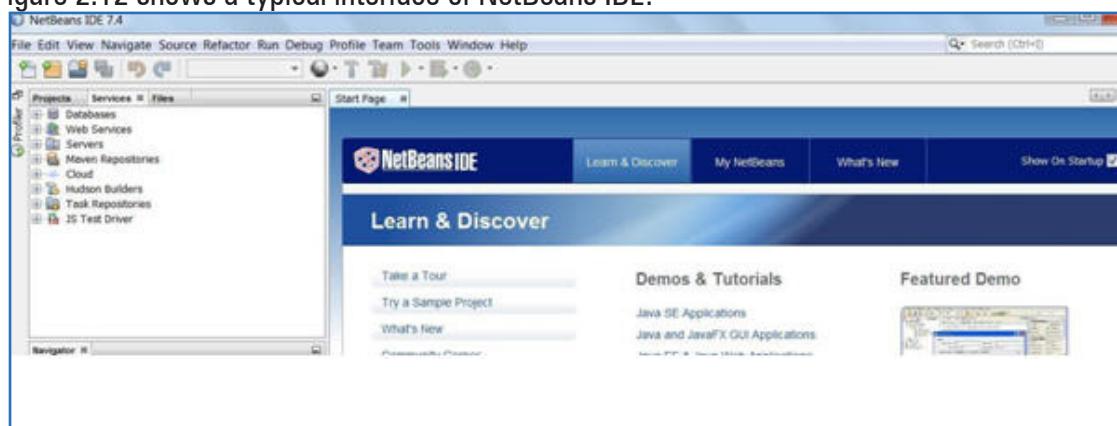


Figure 2.12: NetBeans IDE

The default application server used by NetBeans IDE is GlassFish. Apart from GlassFish there are other application servers such as Red Hat JBoss, Bean Weblogic, and so on.

Check Your Progress

1. Which of the following statements about enterprise beans are true?

(A) Bean container manages the bean lifecycle	(C) Stateless session beans are invoked by singleton session beans
(B) Communication with external components is not managed by the container	(D) None of these

2. Which enterprise bean is invoked asynchronously?

(A) Session bean	(C) Stateless Session bean
(B) Message driven bean	(D) None of these

3. JSF pages are stored in which of the following modules in an enterprise application?

(A) Resource adapter module	(C) Application client module
(B) EJB module	(D) None of these

4. Which of the following are true about deployment descriptors?

(A) It can map the enterprise bean classes onto jndi names	(C) It cannot provide servlet mapping
(B) Being an xml file it can be used to implement user interface	(D) None of these

5. Which of the following EJB roles is responsible for managing the application in the production environment?

(A) Deployer	(C) System administrator
(B) Assembler	(D) None of these

Check Your Progress

6. Which of the following is not a feature of EJB 3.0?

(A)	Home interface	(C)	Business interface
(B)	No view interface	(D)	None of these

Answer

1.	A
2.	B
3.	D
4.	A
5.	C
6.	A

Summary

- Enterprise JavaBeans are application components which implement business logic of the application.
- There are three types of enterprise beans – Session beans, Entity Beans, and Message-driven beans.
- Session beans can be stateless, stateful and singleton, they are synchronously invoked by the application client.
- Message-driven beans are enterprise beans which are asynchronously invoked through JMS messages.
- Entity beans are used to implement persistence in enterprise applications.
- Entities are enterprise beans which implement database operations, they are deprecated and entity persistence to database is implemented by Java Persistence API in EJB 3.0.
- Enterprise beans are deployed in a container which provides services such as transaction management, component security, naming services, and so on.
- Enterprise beans can be accessed through business interface view or no interface view of the application.
- Annotations are metadata used in the application for deployment and dependency injection.
- Enterprise beans are packaged into EAR file.



Balanced Learner-Oriented Guide

for enriched learning available



www.onlinevarsity.com



Visit

Frequently Asked Questions

@

www.onlinevarsity.com



Welcome to the Session, **Session Beans**.

This session introduces Session beans. It explains different types of Session beans and their use in developing Java enterprise application. Further, the session explains the Stateless Session bean along with its characteristics and lifecycle. It also explains how to perform asynchronous communication through Stateless Session bean. Finally, the session demonstrates the development of Stateless Session bean.

In this Session, you will learn to:

- Describe Session beans
- Describe the characteristics of Session bean
- List the different types of Session beans
- Explain Stateless Session beans
- Describe the lifecycle of Stateless Session beans
- Explain the process for developing a Stateless Session bean in NetBeans IDE
- Describe the types of communication with Session beans
- Explain asynchronous communication in Stateless Session bean

3.1 Introduction

EJB are server-side components that are hosted on Java-enabled application servers. They are developed as a part of enterprise applications to assist in providing solution to complex requirements of the business. Normally, EJB's are placed on the middle layer of the application and are accessed by different types of clients such as a Servlet or a Web service.

The Java EE platform supports different types of EJB components based on the functionality and integration with other components in the application. Enterprise JavaBeans are categorized into various types of bean development. They are Session beans and Message-driven beans.

Session beans represent business processes that are used to handle business logics for the application. A business logic can be performing addition on two numbers, connecting to a database, performing transaction on a bank account, or invoking other Session beans or Message-driven beans.

3.1.1 Session Bean

Session beans are reusable Java components that encapsulate the business logic of an enterprise application. They execute on the server-side in an EJB container. They are deployed on Java-enabled application servers and provide services to their clients which can access them programmatically.

Session bean objects are conversational. A conversation is an interaction between a bean and a client and it is composed of a number of method calls between them. For example, Cookies can be used to maintain the client data in Web applications. Similarly, Session beans can hold the state of the client during the method calls.

There can be different types of clients to a Session bean. These clients can access the Session bean either remotely or locally.

- Remote client** – It can be a stand-alone Java application, a Web service, or other Session bean executing as a part of enterprise application in different JVM.
- Local client** – It can be components residing within the same enterprise application. It can either be another Session bean or a Web component Servlet.

Session bean components implement `javax.ejb.SessionBean` interface. They are not persistent across system failures.

Figure 3.1 demonstrates the different perspectives for Session beans based on who is accessing the bean.

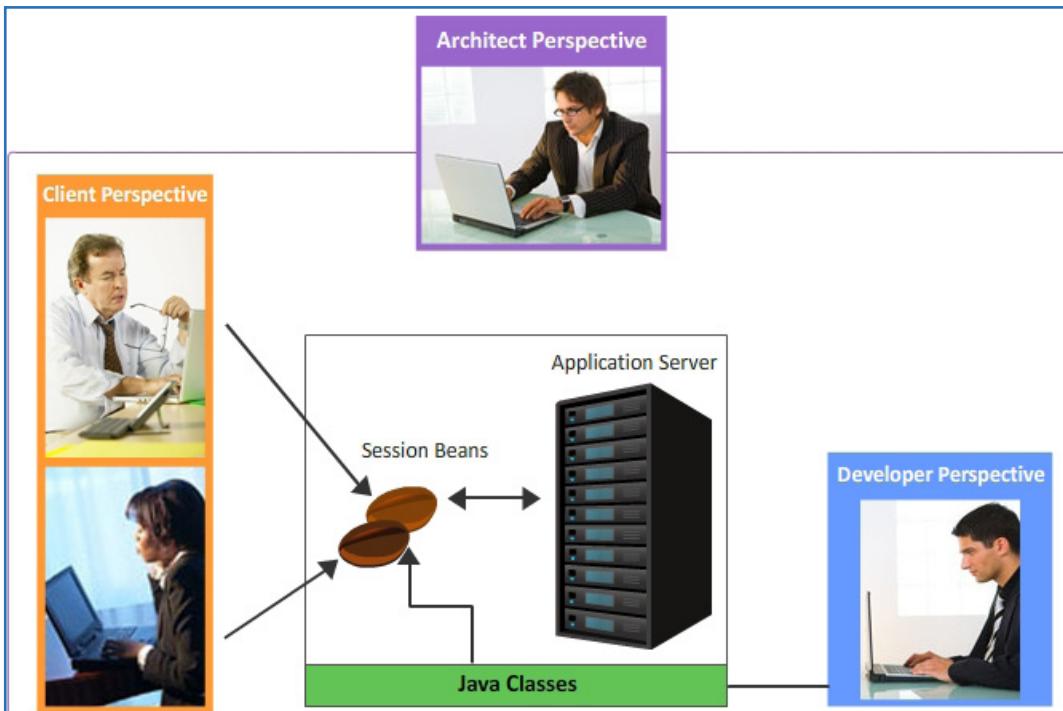


Figure 3.1: Different Perspectives of Session Beans

In figure 3.1, the application architect views the Session beans as implementation of business logic, the Session beans here are accessed by multiple clients. The architect designs the business logic according to the client requirements. The developer perspective of the Session beans is implementation of Java classes. These Java classes implement the application functionality.

The client perspective of the Session beans is extension of the application server. The client accesses the services provided by the application server through Session beans. Clients access the Session bean through the business interface provided by the developers.

3.1.2 Types of Session Beans

The main difference between the Session bean and JavaBeans is the lifetime of the Session bean. An instance of a Session bean performs a task on behalf of the client. In other words, it represents client on the server-side and acts as an extension of client on the server. An instance of a Session bean is alive as long as the client is present. Hence, the lifetime of a Session bean is equivalent to the lifetime of a client. It maintains the conversation state for the client. Conversation can be defined as the period between the client accessing a Session bean instance and client invoking the remote method on the instance.

Following are the two main aspects of lifetime of a Session bean:

- ❑ Multiple clients cannot share instances of a Session bean. Length of a session is dependent upon the length of time the bean instance has been used.

- Lifecycle of a Session bean's instance is controlled by the container and the instance of the Session bean is removed by the container when timeout occurs or the bean is explicitly destroyed.

State of the instances of a Session bean cannot be saved in a database or a file system. Hence, Session beans are not persistent, but they can perform database operations.

Based on the lifetime of the Session bean, there are three variants of Session beans:

- **Stateless Session beans** – As the name suggest, the Stateless Session bean does not maintain the client state on the server during conversation. This means it does not stores the value of the instance variables associated with the client conversation. Whenever a method of the Stateless Session bean is invoked, it uses the instance variables of the bean, but the values of these variables are retained only for the duration of method invocation. After the method is completed, the values are not retained, that is, they are deleted by the container.

Some of the situations which do not require the client's data to be maintained and are suitable for Stateless Session bean development, they are as follows:

- SalaryCalculator
- MoneyConverter
- CardVerification

- **Stateful Session beans** – Stateful Session beans maintain the state of the clients on the server. This means that they store information obtained from the specific client between the method invocations. In other words, Stateful Session beans are responsible to retain the client state across various method conversations.

Some of the situations where it is necessary to store the client's information to maintain its identity and are suitable for Stateful Session bean, they are as follows:

- Shopping cart
- Online banking
- Product order processing system

- **Singleton Session beans** – Singleton Session beans are similar to Stateless Session bean. However, they are instantiated once for the entire application. They are accessed concurrently by multiple clients and they are similar to Stateless Session beans.

The single instance of the Singleton bean is share by all the clients. Some of the scenarios where Singleton Session bean can be used for executing initialization and clean tasks of the application when it shuts down.

3.1.3 Characteristics of Session Beans

A Session bean executes the business logic on the server, hiding the complexities from the client accessing it. Regardless of the type of Session Beans developed in the enterprise application, they possess the following characteristics:

- Session beans are short-lived objects. This means Session beans cannot survive when the application server crashes.
- A session bean instance is associated with a single client and cannot be shared by multiple clients.
- Session beans are managed by the container. This means the execution of the Session bean is controlled by the container.
- Session beans are executed by the client code and represent the state of communication between the client and bean.
- Session beans are instantiated and managed by the EJB container.
- Session beans supports synchronous and asynchronous communication. This means they can be invoked synchronously or asynchronously by the clients.
- Session beans are non-persistent beans which means they cannot be used to represent data in a database.
- Session beans can implement transaction boundaries and security mechanisms.

The Session bean is used in the application when any of the following conditions hold good:

- When the application has to retain the state across multiple method invocations.
- When there is a requirement of communication between the client and other components of the application.

3.2

Stateless Session Bean

A Stateless Session bean is invoked by the client when it requires service of an application server. It does not maintain the conversational state of the session with the client. It is used in scenarios where the application requires scalability. Stateless Session beans can be maintained as a pool of Session beans to support Web services. Session beans in an application are used to perform independent operations.

Figure 3.2 demonstrates a pool of beans instantiated and managed by the container on the application server.

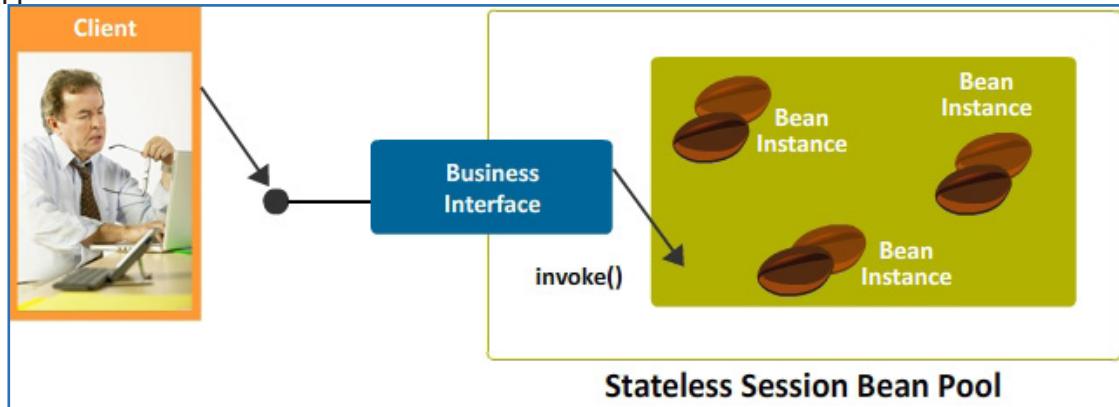


Figure 3.2: Pooled Stateless Session Bean Instances

Figure 3.2 demonstrates the functioning of a Stateless Session bean. A pool of Stateless Session beans is instantiated into the container during application deployment. When a client has to access the Session bean, it uses the business interface of the Stateless Session bean. Through the business interface defined, the application client accesses the Session bean from the pool of available beans. The Session bean then provides the required service. The bean is returned to the pool of available bean instances after the client request is serviced.

The client requests do not actually reach the Session bean, but they reach the container in which the Session bean is deployed. The container assigns the client request to an appropriate instance of bean. The container manages which Session bean has to be invoked. If all the instances are assigned to other clients, the container may instantiate a new bean for the new client request. These choices are made by the container based on its configuration.

3.2.1 Callback Methods for Stateless Session Bean

Callback methods are the methods used by the container to manage the lifecycle of a Session bean. These methods are prefixed with callback annotations.

Stateless Session beans have two types of callback methods, PostConstruct and PreDestroy methods. These methods are prefixed with `@PostConstruct` and `@PreDestroy` annotations respectively.

The PostConstruct methods implement the functionality or tasks which should execute immediately after the Session bean is instantiated such as initializing the variables and so on.

The PreDestroy methods implement the functionality or tasks which should execute before the Session bean is removed from the container such as freeing all the resources held by the Session bean and so on.

These methods do not return any value; hence the return type is `void`.

3.2.2 Lifecycle of a Stateless Session Bean

Stateless Session beans are maintained as a pool of beans in the container. Whenever a client request is received, the container assigns a bean from the pool to service the request. Once the session is complete, the bean is returned to the pool of beans maintained by the container.

The Stateless Session bean can therefore be in either of the two stages during its lifecycle:

1. **Instantiated** – In this stage, the Session bean is instantiated and added to the bean pool.
2. **Removed** – In this stage, the instance of the bean is destroyed and removed from the container.

Figure 3.3 graphically represents the lifecycle of a Stateless Session bean.

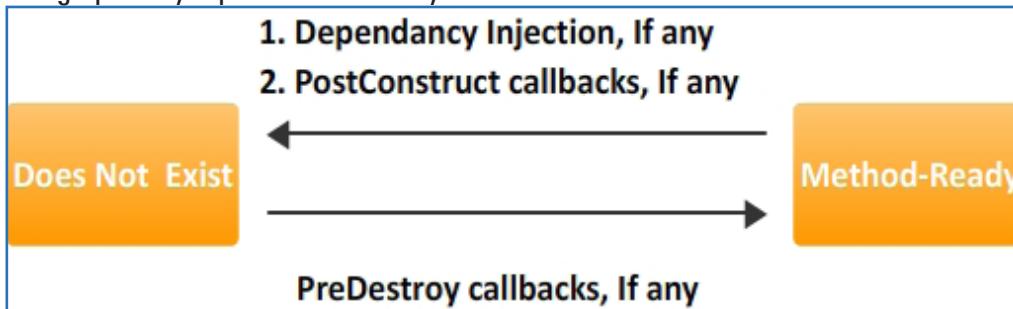


Figure 3.3: Lifecycle of a Stateless Session Bean

The application server initially does not have any Session beans in the container. Then, a bean is said to be in 'Does not exist' state. Whenever there is a client request for the Session bean, the container instantiates the Session bean. However, the container may also instantiate the Session beans even if there is no client request for the Session bean, such instantiation happens during the application startup.

The container creates an instance of the Session bean through `newInstance()` method. After instantiating, the bean dependencies declared through annotations and deployment descriptors are injected into the bean.

Invoking a post construct callback method is optional. The function of a post construct callback method is to perform additional initialization operations after the Stateless Session bean is instantiated. These methods are invoked by the container not the client. When a client invokes a bean, the client acquires the bean from the pool and returns the bean to the session bean pool.

After instantiating, all the Session beans are placed in the Method-Ready pool. The Session beans in this pool are ready to take requests from the client and respond accordingly.

When a container has to remove the Session bean from the container, it may have to execute pre-destroy callback methods. These methods are responsible for performing clean up tasks for the bean, such as handing over the state of the client to another session, deallocating all the resources used by the bean and so on. It is optional to have a pre-destroy method for the Session bean.

3.2.3 Annotations in Stateless Session Beans

Annotations are metadata which are used by the container during application deployment. They provide directives regarding application configuration.

A Stateless Session bean can be created by annotating the specific Java class with `@Stateless` annotation.

The interfaces of the Session bean can be annotated with `@Local` and `@Remote` annotations. A local interface is annotated with `@Local` and remote interface is annotated with `@Remote`.

A Stateless Session bean can have callback methods associated with their lifecycle events. The callback methods can be annotated with `@PostConstruct`, `@PreDestroy` annotations.

3.3 Developing Stateless Session Beans

When a developer intends to create a Stateless Session bean, they have to create the following components:

- Bean class
- Business interface

Bean class refers to the Java class implementation of the business logic expected from the bean. It has a set of instance variables and methods defined to accomplish the task expected.

Business interface refers to the set of methods which are exposed to the client through which the client can access the Session bean. Creating a business interface is not mandatory for creating Session beans; they can be accessed through no-interface view also.

Some of the rules to be followed while developing the Stateless Session bean:

- There should be at least one business interface for the Session bean.
- Session bean class cannot be declared as `final` or `abstract` since the container needs to manipulate the class.
- Session bean class should implement a no argument constructor as this constructor is invoked when an instance of the Session bean is created.
- Session bean class can be a subclass of another Session bean or a POJO class.
- Business and life cycle callback methods are defined in the Session bean class or superclass.
- Business method names present in the business interface and in the Session bean class must not begin with `ejb`.
- Business methods should be declared as `public` and cannot be declared as `final` or `static`.

To create a Stateless Session bean in NetBeans IDE, perform the following steps:

1. Open NetBeans IDE.

2. Select **File** → **New Project** to create an enterprise application. Choose New Project from the File menu, it will lead you to the screen shown in figure 3.4.

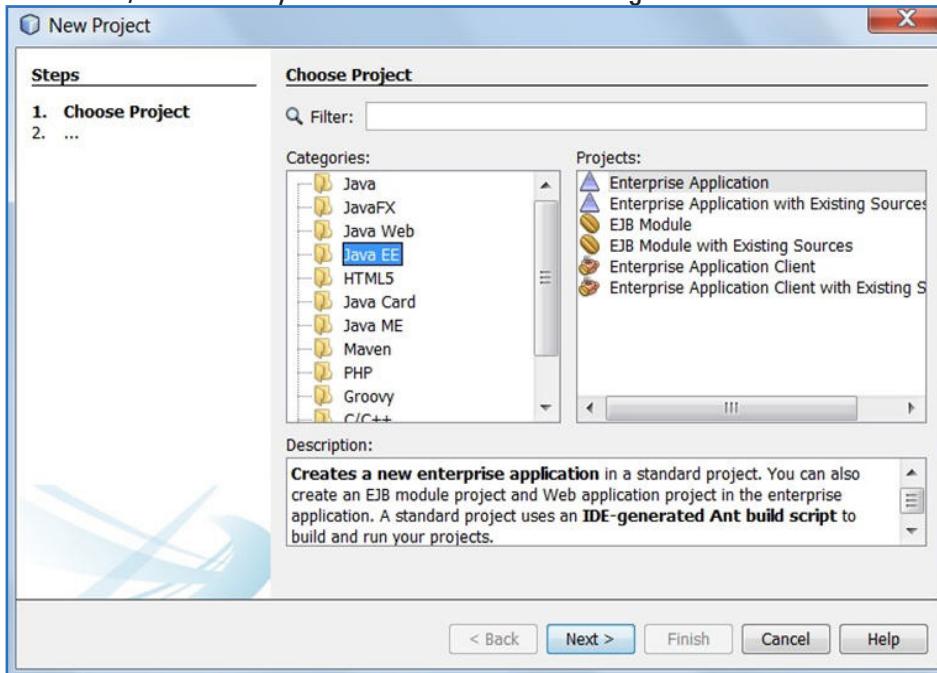


Figure 3.4: Creating an Enterprise Application

3. Select the appropriate options as shown in figure 3.4 to create an enterprise application. Choose **Java EE** → **Enterprise Application** in this screen.

Following screen prompts the developer for the enterprise application name, select the appropriate name and click **Next**. It will lead to the screen as shown in figure 3.5.

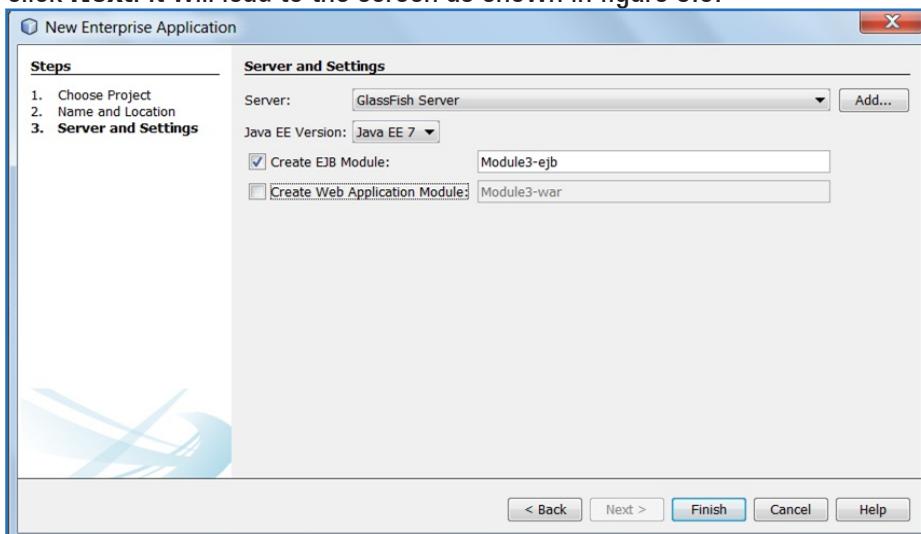


Figure 3.5: Selecting the Application Server

In the screen shown in figure 3.5 the developer has to select the application server onto which the application has to be deployed. If there are multiple versions of Java EE, then the developer has to choose the version the developer intends to use for the application.

Also, choose what modules of the application are to be created. The IDE by default creates an EJB module and Web module. The focus here is only on the EJB module.

- Once the project is created it appears in the project window as shown in figure 3.6.

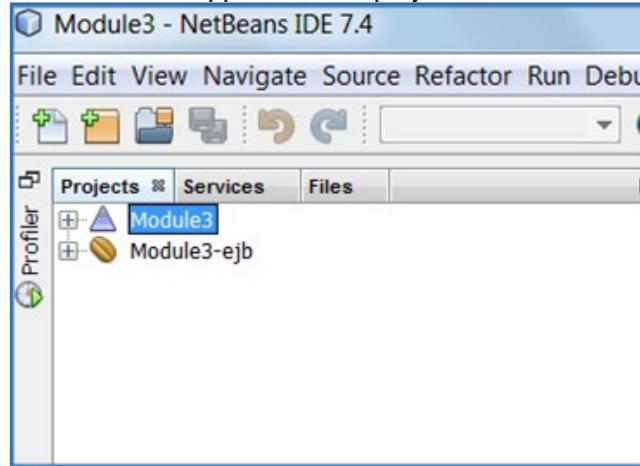


Figure 3.6: Enterprise Application in NetBeans IDE

Figure 3.6 shows the enterprise application 'Module3' and its corresponding EJB module. Now create a Stateless Session bean in the EJB module for the application.

- Create the Session bean now. Select the EJB module and select **New File** from the File menu. It leads to the screen shown in figure 3.7.

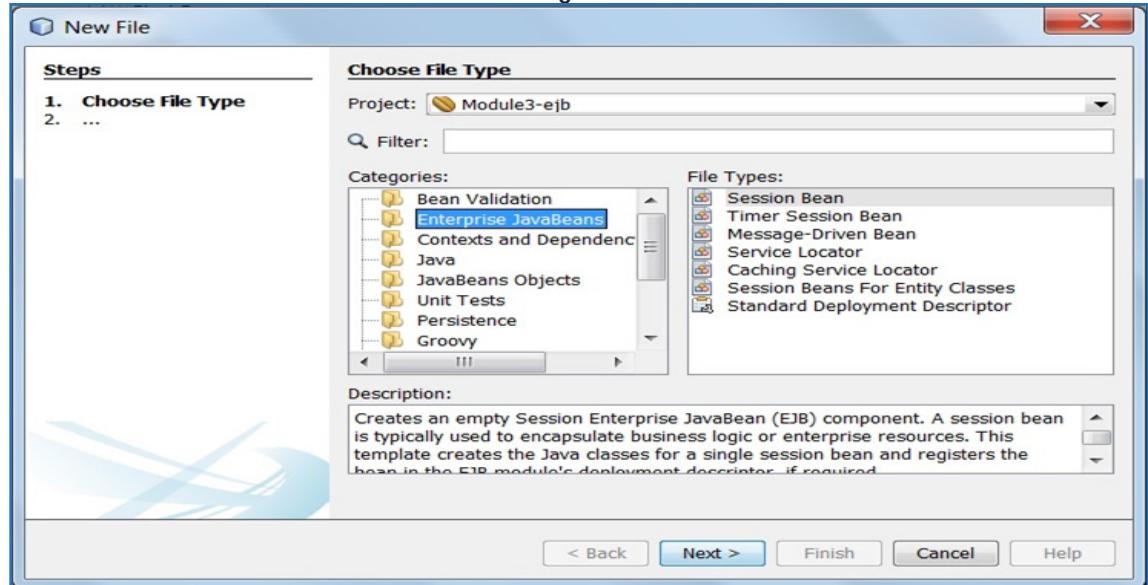


Figure 3.7: Creating a Session Bean

Choose Enterprise **JavaBeans** → **Session Bean** in the current screen and click Next. This will lead to the screen shown in figure 3.8.

6. Select options of the Stateless Session bean using the wizard shown in figure 3.8.

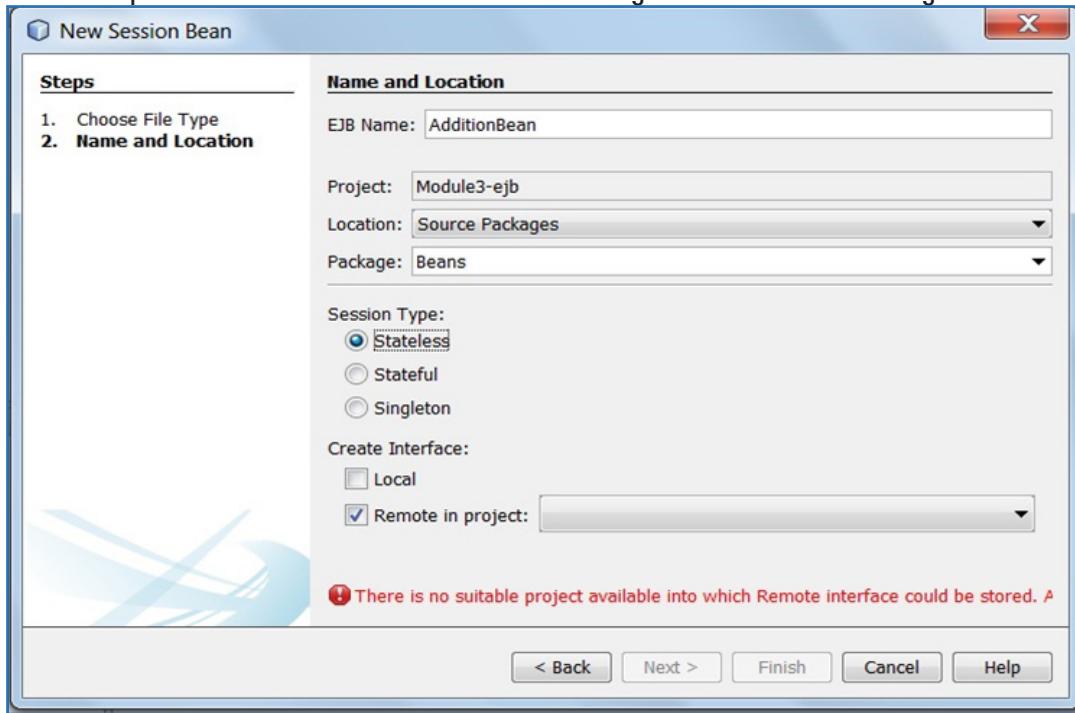


Figure 3.8: Selecting Options to Create Stateless Session Beans

In figure 3.8, select options for the Stateless Session bean. First provide the name of the bean and a package in which the bean can be stored. It is a good practice to store the beans in different directories according to their functionality.

Choose the type of Session bean being created. Here, a Stateless Session bean is created with remote interface to access it. When remote interface is selected, the application prompts for a suitable project to store the remote interface. Hence, firstly create a remote interface.

3.3.1 Creating Remote Interface

Remote interface is the set of methods through which the Session bean can be remotely accessed. Following are the steps to be followed to create a remote interface:

1. To create a remote interface, create a new Java Class Library project in the NetBeans IDE by clicking **File** → **New Project** → **Java** → **Java Class Library** from the **New Project** dialog box as shown in figure 3.9.

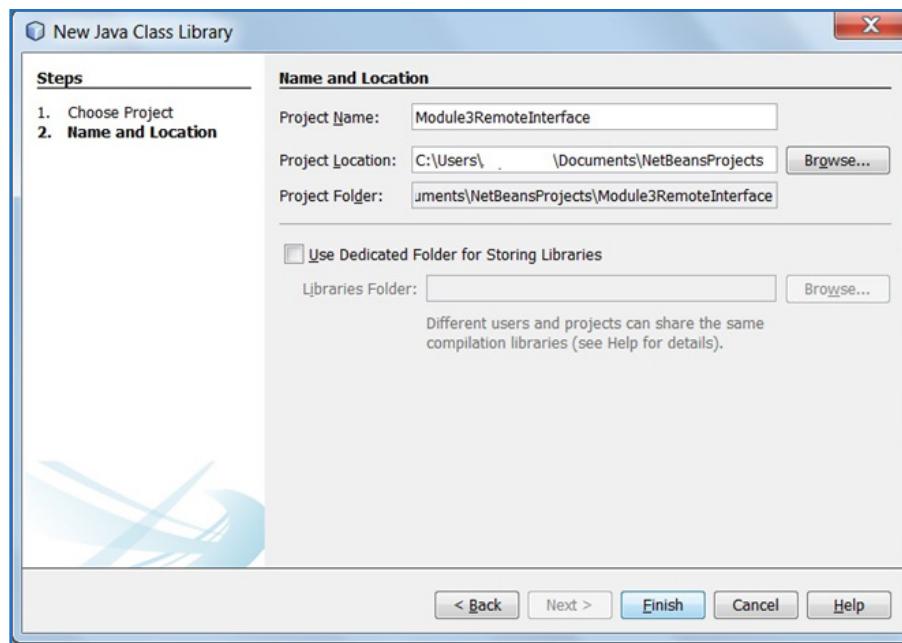


Figure 3.9: Creating a Remote Interface

Now complete the bean creation process by selecting the options as shown in figure 3.10.

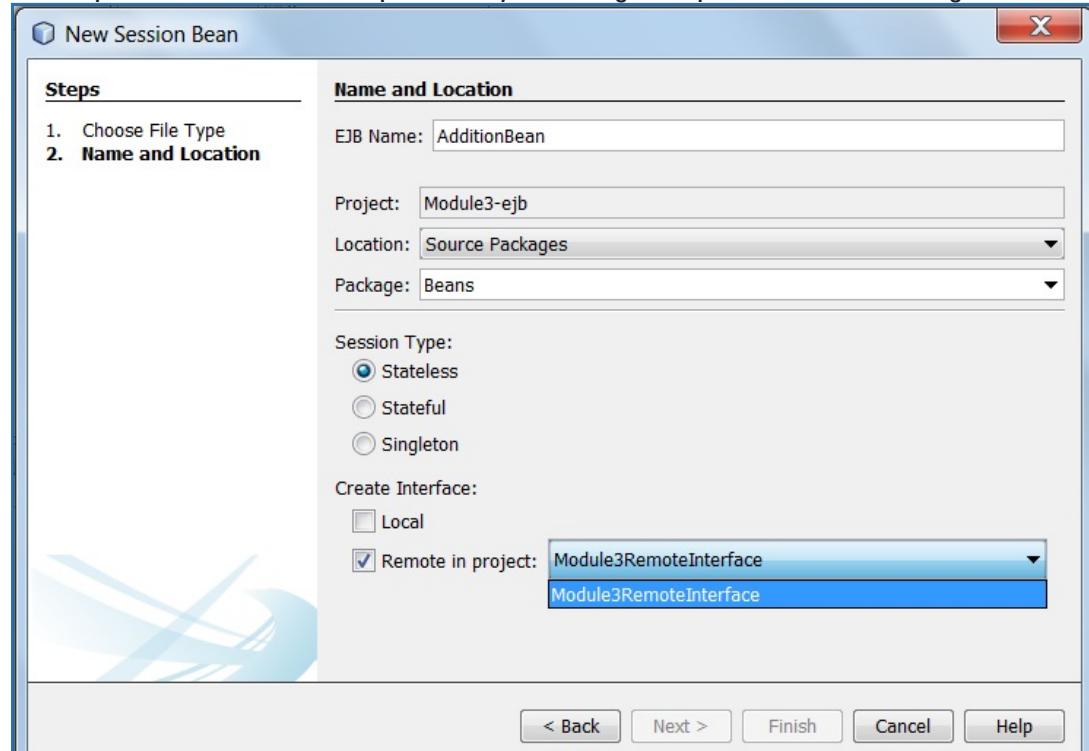


Figure 3.10: Selecting the Remote Interface for Session Bean

2. On creating the session bean with the remote interface, the directories are created as shown in figure 3.11 in the NetBeans IDE.

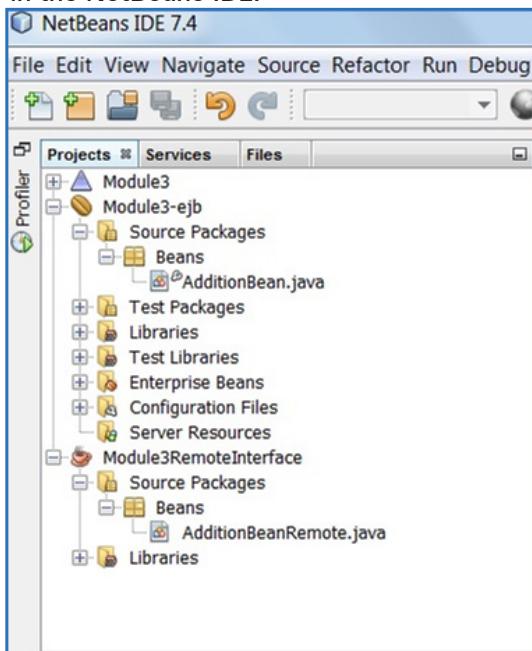


Figure 3.11: Directory Structure of Module3

3.3.2 Creating the Business Methods

Following are the steps to be followed for creating the business methods in the session bean.

1. Double-click the bean file, `AdditionBean.java` will open in the code editor and add bean implementation code to it. In the editor, right-click in the area where the business methods are defined and it will pop out a menu as shown in figure 3.12.

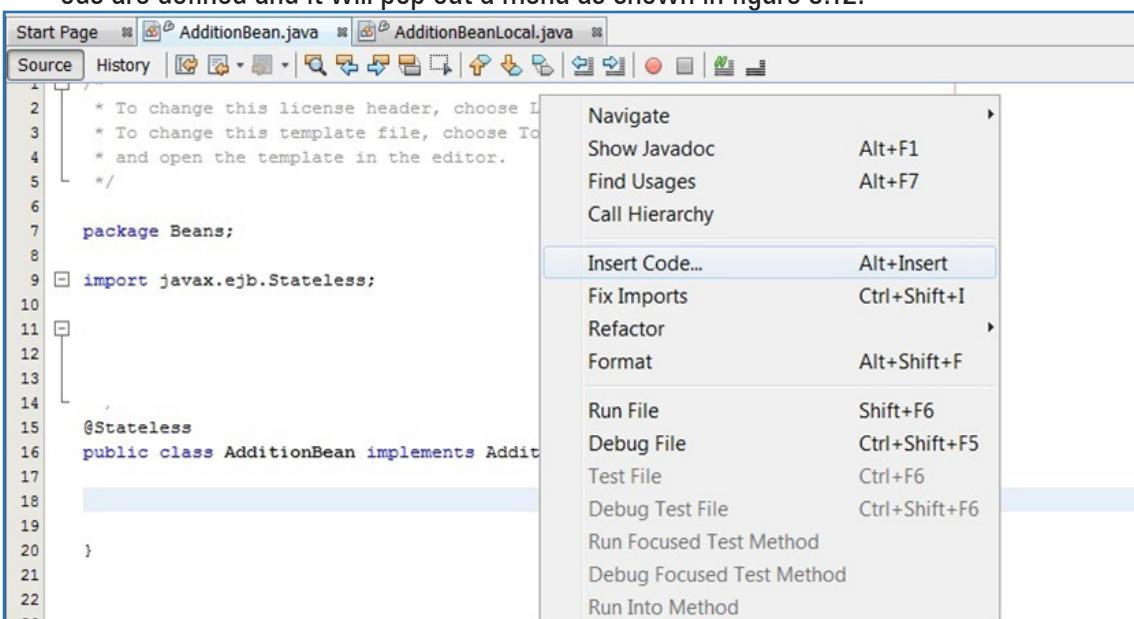


Figure 3.12: Inserting Code into the Session Bean

2. Select Insert Code option in the menu. It will lead to another menu as shown in figure 3.13.

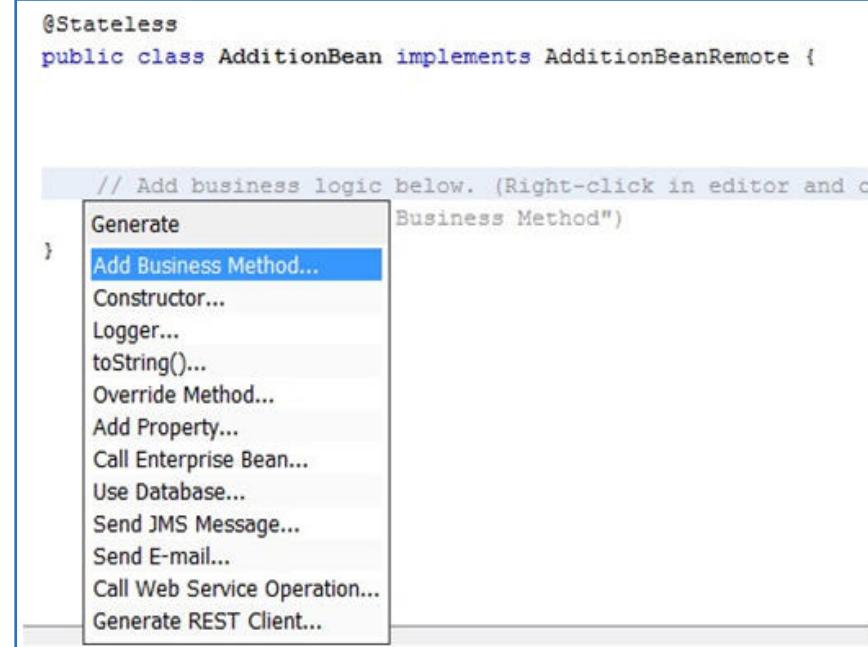


Figure 3.13: Adding the Business Method

3. Select Add Business Method option from the menu. It will lead to a screen shown in figure 3.14.

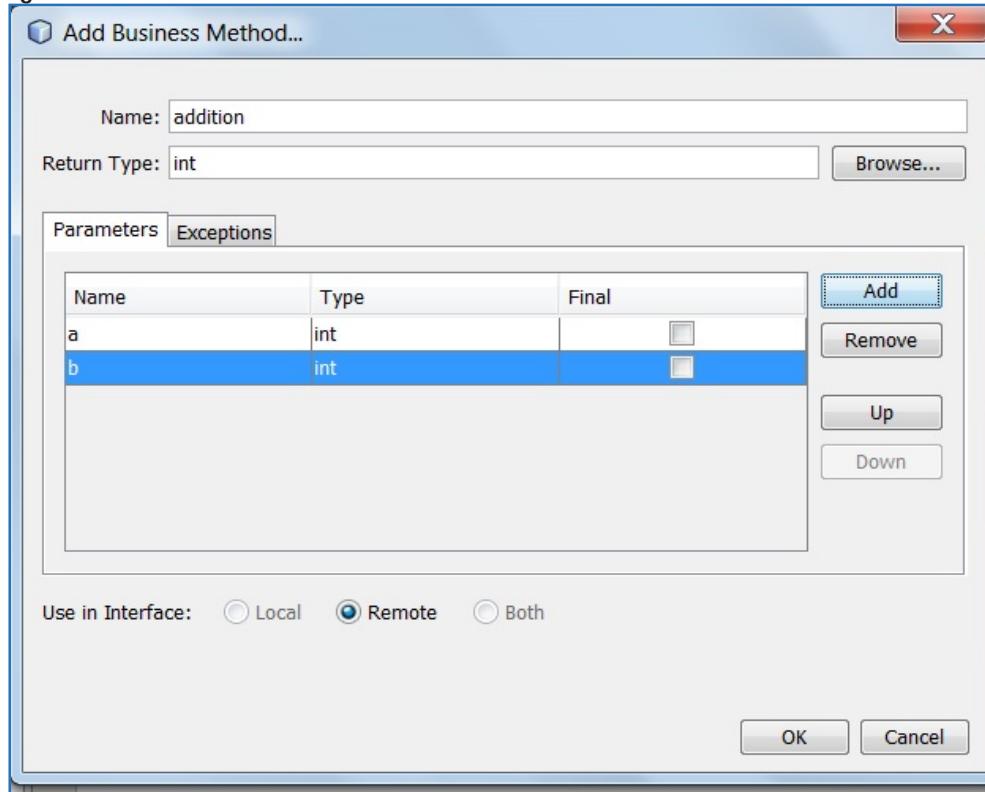


Figure 3.14: Creating a Business Method

Select the characteristics of the bean method. According to the definition shown in figure 3.14, a business method named 'addition' is being created whose return type is 'int'. There are two input parameters of the method, a and b of type 'int'. Define these characteristics and click OK.

This will create code in the `AdditionBean` and `AdditionBeanRemote` classes as shown in figures 3.15 and 3.16 respectively.

```
package Beans;

import javax.ejb.Stateless;

@Stateless
public class AdditionBean implements AdditionBeanRemote {

    @Override
    public int addition(int a, int b) {
        return 0;
    }
}
```

Figure 3.15: AdditionBean Class

```
package Beans;

import javax.ejb.Remote;

@Remote
public interface AdditionBeanRemote {

    int addition(int a, int b);
}
```

Figure 3.16: AdditionBeanRemote Interface

4. Modify the default code created by the IDE in the `AdditionBean` to perform the addition operation of two int variables. Code Snippet 1 shows the code after modification.

Code Snippet 1:

```
package Beans;

import javax.ejb.Stateless;

@Stateless

public class AdditionBean implements AdditionBeanRemote {

    @Override

    public int addition(int a, int b) {

        return a+b;

    }

}
```

Code Snippet 1 shows the package to which the current bean is located. The annotation `@Stateless` implies that it is a stateless bean so that it can be appropriately deployed. The implementation of a stateless bean is imported from `javax.ejb.Stateless` package. The bean implements the remote interface `AdditionBeanRemote`.

3.3.3 Creating a Client for a Session Bean

1. Create an application client which can access the session bean. To create an application client, create a new project with options as shown in figure 3.17.

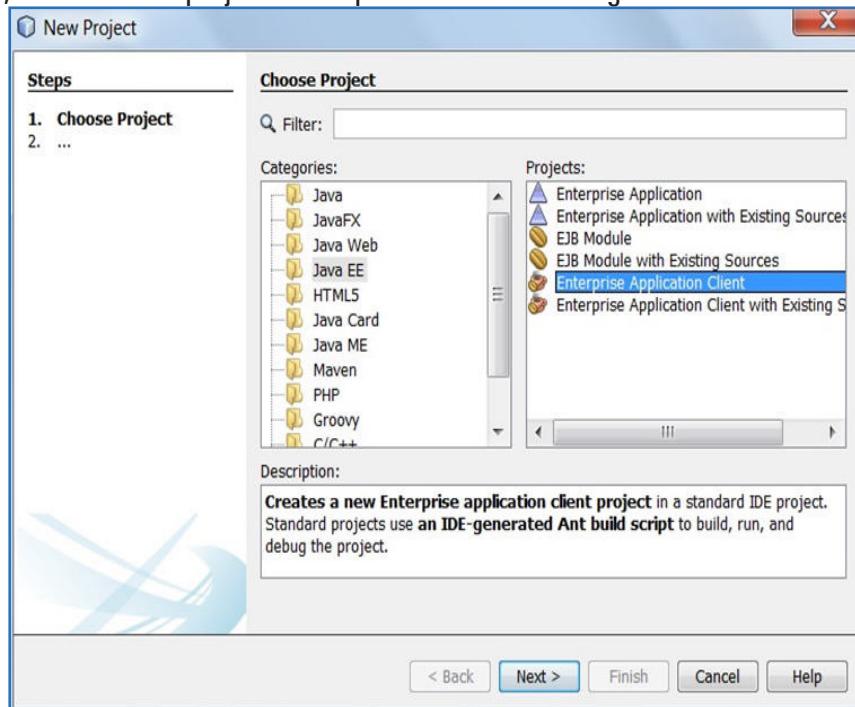


Figure 3.17: Creating an Application Client

2. Choose Module3client1 as the name for the application client and other options as shown in figure 3.18.

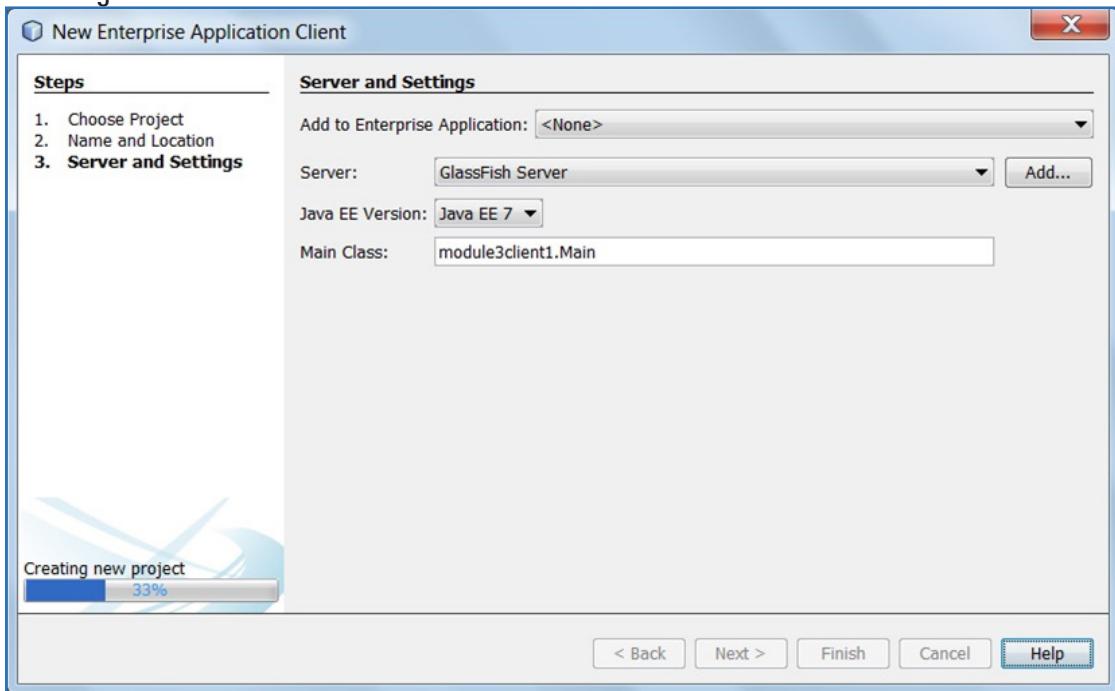


Figure 3.18: Configuring the Application Client

3. Creating the application, client will result in the main class as shown in figure 3.19. Invoke the enterprise bean from the client, to do so right-click in the client code and choose the option.

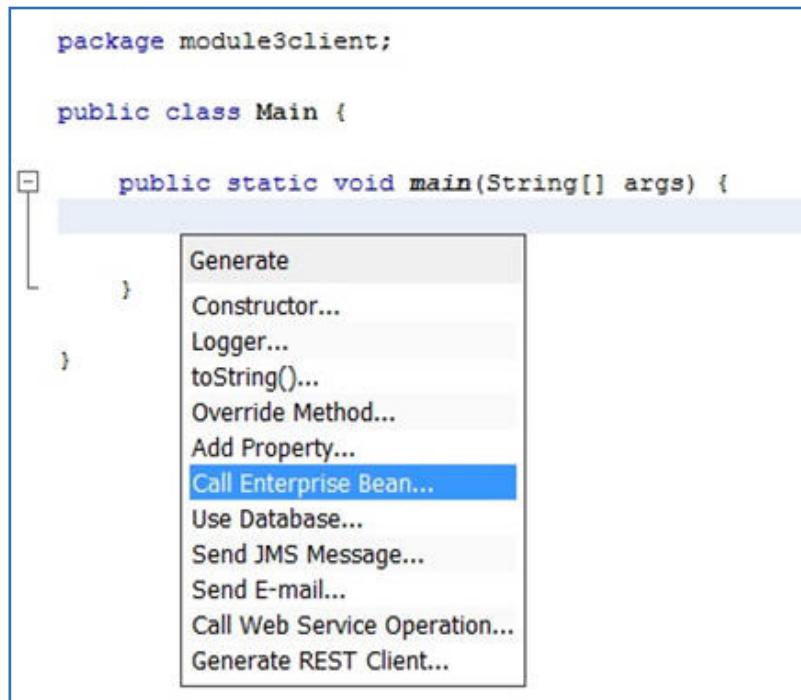


Figure 3.19: Calling the Enterprise Bean from the Application Client

4. On choosing the Call Enterprise Bean option, the wizard shown in figure 3.20 appears which lists all the available enterprise beans.



Figure 3.20: Selecting the Enterprise Bean to be Accessed

5. Modify the code of the application client as shown in Code Snippet 2.

Code Snippet 2:

```
package module3client;

import Beans.AdditionBeanRemote;
import javax.ejb.EJB;
public class Main {

    @EJB
    private static AdditionBeanRemote additionBean;

    public static void main(String[] args) {
        System.out.println("Result:" + additionBean.addition(4, 2));
    }
}
```

In Code Snippet 2, the remote interface is imported into the client module through import Beans.AdditionBean accessing the additionBean used to invoke the method.

On deploying and running the application client, the output is as shown in figure 3.21.

```
Result: 6
run:
BUILD SUCCESSFUL (total time: 18 seconds)
```

Figure 3.21: Output on Running the Application Client

3.4 Configuring and Deploying Session Beans

A Session bean can be configured through deployment descriptor and annotations. ejb-jar.xml is the deployment descriptor for the enterprise application. It is a declarative XML file which specifies the required configuration of the application.

The deployment information determines how the EJB container manages the bean when deployed, this information includes the type of session which the Session bean will establish, transaction type supported, and so on.

Code Snippet 3 shows a generic deployment descriptor.

Code Snippet 3:

```
<?xml version="1.0"?>

<!DOCTYPE ejb-jar PUBLIC
'-/Sun Microsystems, Inc./DTD Enterprise JavaBeans 1.1//EN'
'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>

<ejb-jar>

<enterprise-beans>

<session>
  <ejb-name>AdditionBean</ejb-name>

  <ejb-class>Beans.AdditionBean</ejb-class>
  <session-type>Stateless</session-type>
  <transaction-type>Container</transaction-type>
</session>
...
</ejb-jar>
```

In Code Snippet 3, the configuration of an `AdditionBean` is defined. The class implementing the EJB is `Beans.AdditionBean`.

The `<ejb-class>` element represents the Java class name through which the Session bean is implemented. The `<session-type>` element identifies the type of the bean being implemented, in this case, it is a Stateless Session bean. The `<transaction-type>` element identifies the transaction management method applicable. In this case, the transaction management is defined through the container.

The session type is stateless and the transaction is managed by the container. However, this deployment descriptor code is partial code.

3.5

Asynchronous Communication with the Session Beans

A Session bean can interact with the client synchronously or asynchronously. In case of synchronous communication, both the client and bean are actively involved in the communication. When client sends a request to bean, it waits for the bean to respond, similarly when the bean sends information to the client, it actively waits for the acknowledgement. Synchronous communication is thus more reliable.

In case of asynchronous communication, the client sends a request to the bean and does not wait for the response from the bean and vice-versa. A Session bean is invoked by a client either through business interface view or no-interface view. The communication between the client and Session bean can be both synchronous and asynchronous.

Asynchronous communication is used in case of long running operations. The client can perform other operations before the bean method execution is complete. Session beans which implement Web services cannot be asynchronous.

3.5.1

Creating Asynchronous Method Invocation in Session Beans

To mark a method as an asynchronous method, it has to be annotated with `@Asynchronous` annotation. `javax.ejb.Asynchronous` must be imported to use this annotation.

Asynchronous methods are expected to return void type or an object of `Future<V>` interface.

`Future` interface is defined by Java to hold the results of an asynchronous method. It has methods to check whether the asynchronous method is complete or not, methods to wait for the completion of the computation and also to store the result of the method.

The class `javax.ejb.AsyncResult<V>` is an implementation of `Future` interface. A method which returns an object of `Future` interface can throw application exceptions. Code Snippet 4 shows a sample interface.

Code Snippet 4:

```
package beans;

import java.util.Date;

import java.util.concurrent.ExecutionException;

import java.util.concurrent.Future;

import javax.ejb.AsyncResult;

import javax.ejb.Asynchronous;

import javax.ejb.Stateless;

@Stateless

public class AsyncMethods {

    @Asynchronous

    public Future<String> sayHello(String name) {

        System.out.println(new Date().toString() + "Welcome to the
application " + name);

        try{

            Thread.sleep(5 * 1000);

        } catch (Exception e){

            e.printStackTrace();

        }

        System.out.println(new Date().toString() + " Bye " + name);

        return new AsyncResult<String>("Hello " + name);

    }

    public static void main(String args[]) throws InterruptedException,
ExecutionException{

        AsyncMethods AS = new AsyncMethods();

        Future<String> S = AS.sayHello("Harry");

        System.out.println("In main method " + S.get());

    }

}
```

Code Snippet 4 shows an asynchronous method which returns a `Future<String>` object. The `sayHello()` method in the code is annotated with `@Asynchronous` annotation and can be invoked asynchronously by the client. The `main()` method in the class can be seen as the client method invoking the asynchronous method. An asynchronous method can be invoked by another bean or an application client in an enterprise application. The asynchronous method returns an object of Future type.

The object is returned to the `main()` method. In order to retrieve the `String` data from the `Future` object, `get()` method is used as shown in the code.

Figure 3.22 shows the output of the executed code.

```

Output ✎
Java DB Database Process ✎ GlassFish Server ✎ AsynchronousApplication (run) ✎
run:
Mon Jul 21 18:08:14 IST 2014Welcome to the application Harry
Mon Jul 21 18:08:19 IST 2014 Bye Harry
In main method Hello Harry
BUILD SUCCESSFUL (total time: 5 seconds)

```

Figure 3.22: Execution of Asynchronous Method

3.5.2 Using Asynchronous Method Invocation in Session Beans

When a client invokes an asynchronous method, it receives an instance of `Future<V>` interface as response from the container. This interface can be used by the client to check the status of the method, receive information about the exceptions thrown by the method and so on.

Following are the methods in `Future` interface:

- `get()` – This method waits if the method is not complete, if the method completes, it returns an object of result type V.
- `get(long TimeOut, TimeUnit unit)`- This method waits for the result from the asynchronous method for the duration specified in the `TimeOut` parameter, the timeout value is determined according to the `TimeUnit` objects.
- `cancel()` – This method attempts to cancel an asynchronous method, if it is not complete and returns a boolean value to specify whether the cancel operation was successful or not.
- `isCancelled()` – This method checks whether a method is cancelled or not and returns a boolean value.
- `isDone()` – This method checks whether a method is complete or not and returns a boolean value.

Check Your Progress

1. Which of the following scenarios is most suitable for using Stateless Session beans?

(A)	Counting the number of hits for the application	(C)	Manipulating user's mail box
(B)	Manipulating customer's bank account	(D)	Calculating interest for a given tenure and rate of interest of loan

2. Which of the following callback methods are not associated with a Stateless Session bean?

(A)	PreDestroy	(C)	PreConstruct
(B)	PostConstruct	(D)	None of these

3. A Session bean can be accessed through:

(A)	Business interface view	(C)	Both a and b
(B)	No-interface view	(D)	Application interface

4. _____ decides when the session bean has to be instantiated.

(A)	Client	(C)	Server
(B)	Container	(D)	None of these

5. When the session beans are instantiated, they are deployed on _____.

(A)	Server	(C)	GlassFish server
(B)	Container	(D)	Application client

Answer

1.	D
2.	C
3.	C
4.	B
5.	B

Summary

- Stateless Session beans are used when the Session bean does not need to maintain the conversational state.
- Each Session bean is associated with one client.
- Container maintains a pool of Stateless Session beans.
- There are two stages in the lifecycle of the application – instantiated and removed.
- The container invokes the required callback methods for the Session bean.
- The enterprise application is deployed and configured through the deployment descriptor and annotations.
- ejb-jar.xml is the deployment descriptor for enterprise applications using EJB specification.
- Bean methods can be asynchronously invoked with the help of Future<V> interface.



To enhance your knowledge,

visit **REFERENCES**



www.onlinevarsity.com



Welcome to the Session, **Stateful Session Beans**.

This session discusses the concept of Stateful Session Beans and their utility. It also describes how to develop, configure, and deploy Stateful Session Beans. The session describes the lifecycle of Stateful Session beans and different methods of accessing Stateful Session Beans. It also explains the different types of clients accessing the Stateful Session bean and exception handling in case of session beans.

In this Session, you will learn to:

- Explain the working of Stateful Session beans
- Explain the different elements of Stateful Session beans
- Describe the lifecycle of Stateful Session beans and associated callback methods
- Explain the Implementation of Stateful Session beans
- Explain the different types of clients accessing Stateful Session bean
- Explain exception handling in session beans

4.1 Introduction

Stateless Session beans do not hold the state of the data between the method invocations. This means after every method call, the container decides to either destroy the bean instance or pool the instance by clearing all the information related to the method invocation.

However, consider a situation where business processes invoked by the client needs the data to be maintained between the conversations over several requests. For example, while performing transactions on a bank account, you as a customer can use the teller machine to complete the transaction operation on your account. You may deposit as well as withdraw money from your account. To achieve this on your behalf, the bank teller machine performs several operations such as checking the account balance, depositing the money in the account, and finally withdrawing the money from the account.

Figure 4.1 shows the business processes invoked by the client for performing transactions on a bank account.

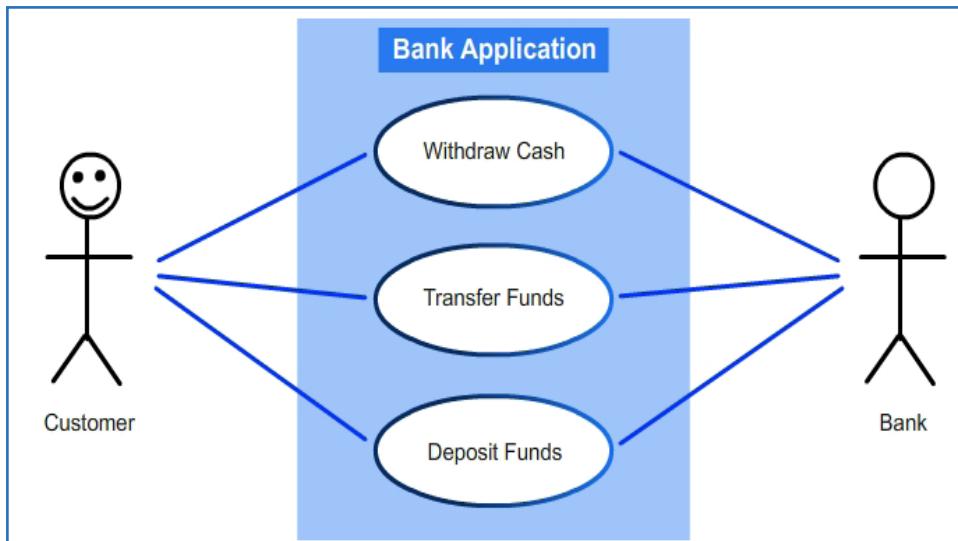


Figure 4.1: Typical Use Case Diagram for a Bank Application

Figure 4.1 shows a typical use case diagram for Bank application. The customer can perform operations such as cash withdrawal, fund transfer, and fund deposit through the banking application.

All the mentioned processes need separate method invocations in the application and the state of the account data has to be maintained in order to perform the transactions. Thus, to process multiple requests, the enterprise bean specification has provided Stateful Session Beans.

4.1.1 Stateful Session Bean

Stateful Session bean can be defined as a bean that services business processes spanning over multiple business method requests.

Stateful Session Bean maintains the conversational state of the application client with which it is associated. This means that the Stateful Session beans retain state on behalf of the client. In other words, if a Stateful Session bean's state is changed during a method invocation, the same state would be available to the same client upon the subsequent invocation.

An example of a Stateful session bean is a shopping cart on an e-commerce Web site. You can add multiple products to the shopping cart on different pages. Each time you add a product to the cart or go to the next Web page, a new request is performed while retaining the state of the previous requests.

Figure 4.2 shows a Stateful Session bean.

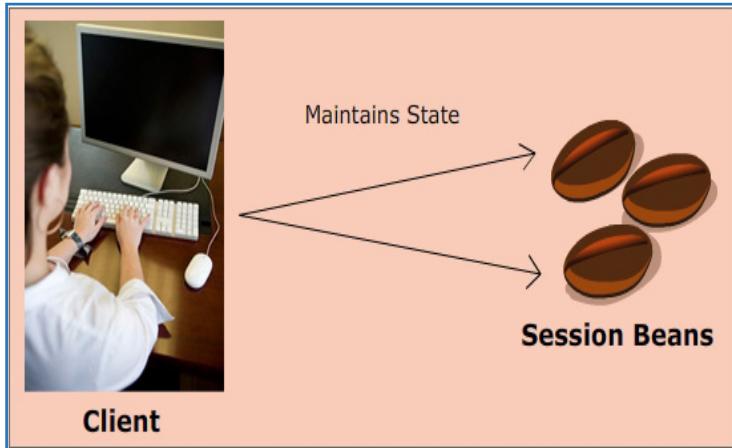


Figure 4.2: Stateful Session Bean

A Stateful Session bean acts on behalf of a client and is dedicated to one client for the life of the bean instance. It retains the state on behalf of the client, until the Stateful Session bean instance is explicitly removed by the container or there is a timeout. The client can access the EJB instance of the Stateful Session Bean through the EJB object.

Figure 4.3 shows accessing of Account Bean instances by the clients.

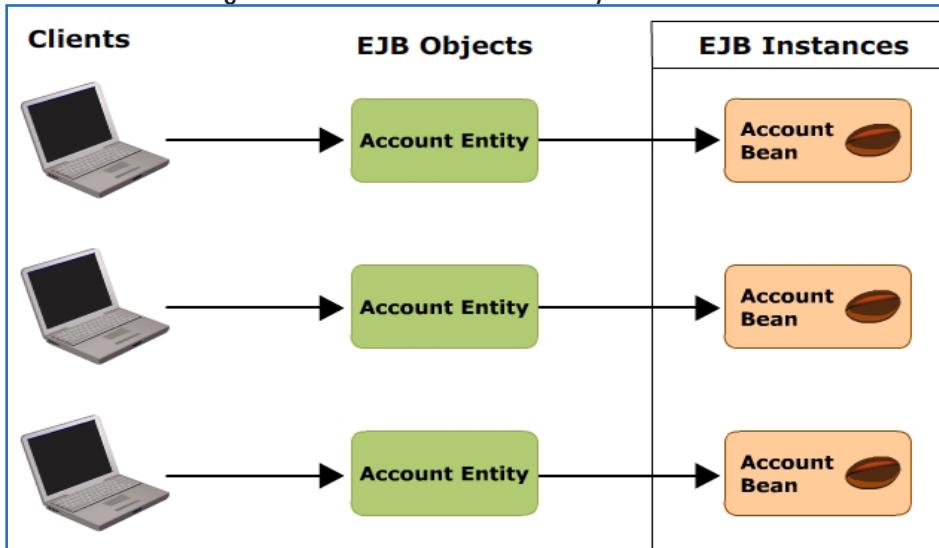


Figure 4.3: Account Stateful Session Bean Instances

The only difference between the Stateful and Stateless Session Beans is that Stateful Session Bean maintains the conversational state of its interaction with the client, while the Stateless Session bean does not maintain the conversational state.

4.1.2 Characteristics of Stateful Session Beans

Bean instance save the client information between method invocations. The bean state represents the conversation between the client and the bean.

Following are the points that states the characteristics of a Stateful Session bean:

- Every instance of an application client is associated with a single instance of Stateful Session bean and the Session bean is associated with the particular client during the entire application session. Stateful Session beans are activated and passivated based on application requirements. The activation and passivation processes are associated with respective callback methods.
- They are transaction aware and short lived.
- They are managed by the EJB container. The EJB container manages separate bean instances for each client.
- Stateful Session beans can access the database, retrieve data from the database, and update data in the database as per application requirement.

4.1.3 Stateful Session Bean Conversational State

When a Stateful Session bean is swapped out of the container its conversational state is written to the permanent storage. This process of writing the conversational state onto permanent storage and removing the Stateful Session bean from the container is known as passivation.

Figure 4.4 shows the passivation of a Stateful Session bean.

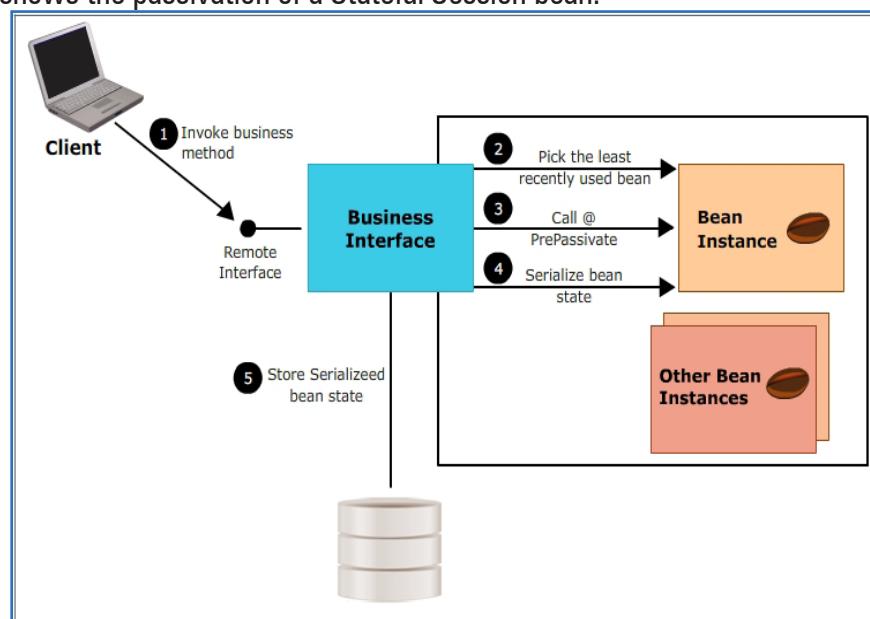


Figure 4.4: Passivation of a Session Bean

To choose which Stateful Session Bean must be removed from the container, the container generally uses Least Recently Used (LRU) strategy. A Stateful Session Bean is said to be least recently used if it has not serviced any client requests for a long time.

When there is a request for the swapped out bean, then it is again brought back into the container, this process of bringing back the bean into container is known as activation.

Figure 4.5 shows the activation of a Stateful Session bean.

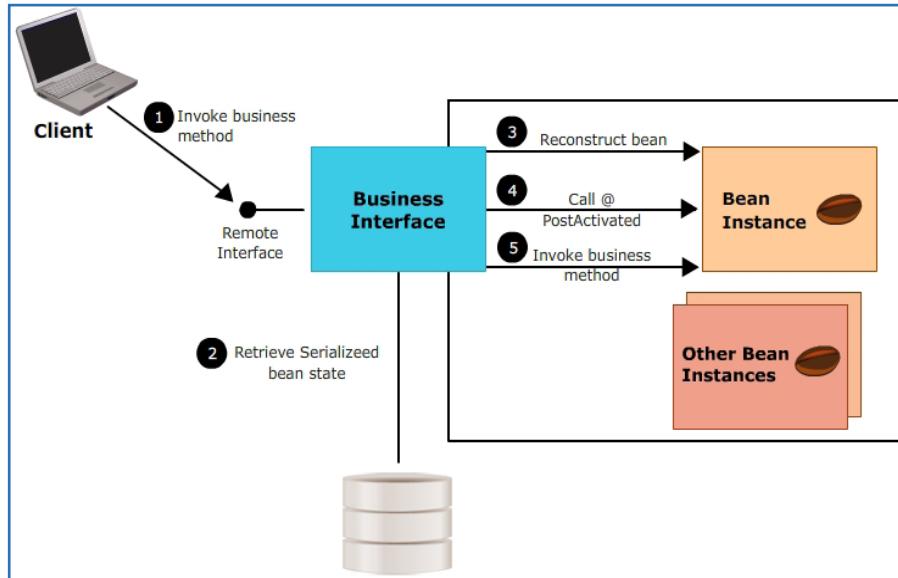


Figure 4.5: Activation of a Stateful Session Bean

4.1.4 Elements of a Stateful Session Bean

Similar to Stateless Session beans, Stateful Session beans also consist of a bean class and a business interface. The business methods are defined in a bean class implementing the business logic.

Bean Class

It is a simple Java class that uses a class level annotation or deployment descriptor to specify the bean type. Annotation is done using `@Stateful` annotation. In a mixed mode, the `@Stateful` annotation is specified only if other member of class level annotations are present.

Business Interface

They are similar to Stateless Session bean and are annotated using `@Local` and `@Remote` annotations. Since, SOAP-based Web services are Stateless, Stateful Session Bean cannot have a Web service endpoint interface.

Business Methods

The business methods defined in a Stateful Session Bean are similar to those defined in the Stateless Session Bean.

4.2 Lifecycle of a Stateful Session Bean

Stateful Session beans do not use instance pooling. The instances are dedicated to one client for the entire application. The bean instances are removed from the memory when they are not in use. EJB object remains connected with the client. Bean instances are passivated before they are removed from the memory and when the EJB object becomes active the instance is activated to restore its state. Container performs various operations to manage Stateful Session Beans.

Following are the steps performed by the container:

- Uses the default constructor to create a bean instance
- Injects the resources such as database connections
- Stores the bean instance in the memory
- Executes the business method invoked by the client
- Waits and executes further requests
- Passivates the bean instance when the client is idle
- Activates the bean instance on receiving a call from the client
- Destroys a bean instance when the bean is not invoked for a period of time
- Requests for removal of bean instance from the client require the activation of the bean instance followed by destruction

There are three states in the lifecycle of Stateful Session bean: **Does Not Exist, Method-Ready, and Passive**.

Figure 4.6 shows various states of a Stateful Session bean.

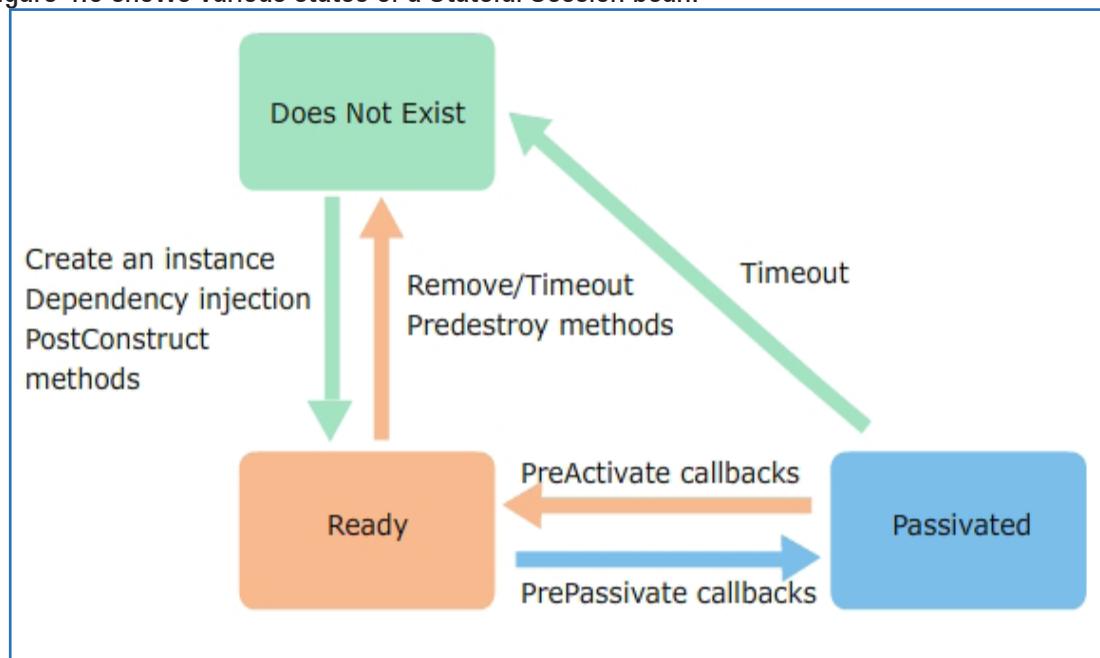


Figure 4.6: Lifecycle of a Stateful Session Bean

Does Not Exist State – In this state, a Stateful Session bean has not been instantiated by the container. It does not hold any application resources in this state.

Ready State – A Session bean is in ready state when it is instantiated by the container. In Method-Ready state, a bean instance services the requests from its clients. The bean's lifecycle begins when the client invokes the first method on the bean class. In this transition to the Method-Ready State, the `newInstance()` method is invoked by the container to create an instance of the bean class.

Once the bean is instantiated, the container allocates required resources to the bean through dependency injection. The resources to be injected are identified through annotations applied in the bean code or through bean configuration supplied in deployment descriptor. Session bean may also require post construct methods to be executed. These methods are annotated with `@PostConstruct` annotation.

The session bean may transit to **Passivated State** or **Does Not Exist State** from the **Ready State**. In **Ready State**, the session bean can accept client requests and perform application tasks. The session bean can transit to passivate, when the container has to persist session data onto secondary storage or when the client is waiting for some event to occur.

The session can transit from the **Ready State** to **Does Not Exist State** when either the client removes the session bean or there is a time out on the session bean. The container can decide on when to remove the session bean from the container based on bean replacement algorithm used by the container. When the container removes the bean to transit the bean into **Does Not Exist State** it may have to execute `PreDestroy` methods based on the application requirement. These methods are annotated with `@PreDestroy` annotation.

The container may transit the session bean into a passivated state, to conserve the resources.

Passivated State – During phases of inactivity the container may transform the session bean into a passive state. The container stores the conversational state and places the session bean in a passivated state. The session bean is removed from the memory during this state temporarily and brought back to active state when the client requests for it.

Before transitioning the session bean to passive state the bean may have to execute prepassivate methods. These methods are annotated with `@PrePassivate` methods. When the container receives a request from the client, the container transits the session bean from **Passivated State** to **Ready State**. The container executes `PostActivate` methods after bringing the session bean from **Passivated State** to **Ready State**. These methods are annotated with `@PostActivate` annotations.

Every Stateful session bean is associated with a timeout value, which is the maximum amount of time period for which the application can stay idle in container or in passivated state. The Stateful Session bean can transit from both **Ready State** and Passivated state after the timeout period.

4.2.1 Lifecycle Callback Methods

Lifecycle callback methods are used to control the lifecycle events of the application. These methods may be used to perform operations such as initializing database for the session bean, close database connections, and so on. These methods are defined similar to any other session bean method prefixed with the callback annotation.

Following are the different callback annotations used in the Stateful Session bean:

- **PostConstruct** – These callback methods are executed after the session bean is instantiated by the container. They are used in both Stateful and Stateless Session beans. PostConstruct methods are prefixed with `@PostConstruct` annotation.
- **PreDestroy** – These callback methods are executed before the session bean is removed from the container and transits to **Does not Exist** state. These methods are also used by both Stateful and Stateless Session beans and are prefixed with `@PreDestroy` annotation.
- **PrePassivate** – These callback methods are used by the session bean, before the bean is passivated, that is the bean transits from **Ready State** to **Passivated State**. They perform all the required operations before passivating the bean such as persisting the session bean state to the secondary storage and so on. These methods are annotated with `@PrePassivate` annotation.
- **PostActivate** – These callback methods which are used to perform tasks after the container transits the session bean from the passivated state to active state. These methods are annotated with `@PostActivate` annotation.

4.2.2 Programming Rules for Stateful Session Bean

Following are the rules to be implemented by a Stateful Session beans:

- Instances of Stateful Session bean should be of Java primitive data types or Serializable objects.
- Stateful Session bean class should define a method that would destroy the bean instance by the bean class using the `@Remove` annotation.
- Stateful Session bean have the `PostActivate` and `PrePassivate` lifecycle callback methods. The `PostActivate` method is invoked once the bean is brought back in the memory and the `PrePassivate` method is invoked before the bean instance is passivated.

4.2.3 Developing Stateful Session Bean

A Stateful Session bean is created along with the remote interface for managing products in a catalog.

Code Snippet 1 shows the code for the class `ProductCatalogBean` class.

Code Snippet 1:

```
...
@Stateful
public class ProductCatalogBean implements ProductCatalogBeanRemote {
    List<String> products;
    public ProductCatalogBean () {
        products = new ArrayList<String>();
    }
    public void addProduct (String productName) {
        products.add(productName);
    }
    public List<String> getProducts () {
        return products;
    }
}
```

In Code Snippet 1, two business methods `addProduct()` and `getProducts()` are being created. The `addProduct()` method is used to add new products to the shopping portal. The products added to the portal are stored in an array list. `getProducts()` method retrieves the products added to the `ProductCatalogBean`. The methods added to the `ProductCatalogBean` are accessed by a remote client through the remote interface.

Code Snippet 2 shows the remote interface for the `ProductCatalogBean`.

Code Snippet 2:

```
...
@Remote
public interface ProductCatalogBeanRemote {
    void addProduct (String productName);
    List getProducts ();
}
```

Code Snippet 2 shows the remote interface defined by the developer with the method prototypes which can be used as a part of the application. Remote interface refers to the set of methods provided by the session bean which can be accessed by the remote clients.

In the code, a remote client of `ProductCatalogBean` can access the methods `addProduct()` and `getProducts()`. The annotation `@Remote` indicates that these methods can be accessed only by the remote client.

4.3 Clients Interfaces for Stateful Session Bean

An EJB by itself cannot perform any function till it is called by a client. An EJB client is an object that interacts with an EJB so that the bean performs some service on behalf of the client. This interaction is in the form of the client invoking operations on the bean's interfaces. The interface of an EJB defines the methods that are available for the clients to invoke.

A client never gets access to the actual EJB. The EJB resides in the EJB container and only the container has the access to the bean. The client gets access only to the EJB object which contains the signature of all the methods that the client can call. The EJB object acts as a factory and distributor of EJB objects to the clients.

When creating an enterprise application the developer has to decide on how application clients can access the application. An application client can access the application through local access, remote access, or through Web service.

The decision of providing local access or remote access is dependent on the following factors:

- **Type of Service** – Since application clients can access the session beans of the application either through the local interface or through remote interface, they can be termed as a local client or a remote client. Based on the type of the service provided by the application, an appropriate client is developed.
- **Relation among the bean components** – Enterprise beans also take services from other beans. If the enterprise beans are tightly coupled, that is they have dependencies on each other it is a good design choice to create them as one logical unit. In such a scenario, the beans are provided with local access. If the bean components are loosely coupled then the beans can access each other remotely.
- **Location of components on the enterprise network** – A bean can be accessed either locally or remotely based on the distribution of application components over the enterprise network. If the remaining components of the application which have to access the enterprise beans are remotely located, then the bean component should be remotely accessed.
- **Performance demands** - Based on the performance demand of the application, the developer has to decide on how to locate and access the bean components of the application. Remotely located application components have performance degradation due to delays introduced by network traversal of data. The developer should consider these factors while deciding the type of access to the enterprise beans.

Based on the choice of access to the bean, the bean access methods are prefixed with `@Local` and `@Remote` annotations.

4.3.1 Local Client

Local clients and the EJB are always located within the same JVM. The client can invoke methods on the bean, which is similar to accessing methods through Java objects. Exposing an enterprise bean to a local client have a few drawbacks in terms of losing location transparency when local client and the enterprise bean are present in the same JVM.

A local client can be another enterprise bean or Web component of the application.

Local clients can access the session beans either through no-interface or through local interface view. The no-interface view comprises all the public methods of the session bean class and the local interface comprises methods declared as part of the local interface. If the business interface of a session bean is not explicitly annotated with `@Local` annotations, it is considered to be a local interface.

4.3.2 Remote Client

A remote client to an EJB need not be located within the same JVM to access the methods of the bean. It can reside in a different JVM on another physical machine. The remote client does not care about the physical location of the bean that it accesses.

The remote client can be another enterprise bean residing in the same or different location, or it can also be a Web application, an applet, or a Java console application. The remote client can even be a non-Java program, such as a CORBA application written in C++.

Remote client cannot access the enterprise bean through no-interface view. It has to implement a business interface which is annotated with `@Remote` annotation. An enterprise bean allows access by remote clients through remote interface. The business and lifecycle methods of an enterprise bean are defined in the remote interface.

Code Snippet 3 demonstrates the code for a remote client accessing the shopping portal application.

Code Snippet 3:

```
...
public class StatefulClient{
    @EJB
    private static ProductCatalogBeanRemote productCatalogBean;

    public static void main(String[] args) {
        List PList = new ArrayList();
        productCatalogBean.addProduct("Laptop");
    }
}
```

```
productCatalogBean.addProduct("MobilePhone");
productCatalogBean.addProduct("Personal Digital Assistant");
PList=productCatalogBean.getProducts();
Iterator itr=PList.iterator();

while (itr.hasNext()) {
    String str=(String) itr.next();
    System.out.print(str + "\n");
}

System.out.println();
}

}
```

Figure 4.7 demonstrates the execution of a remote client of a Shopping Portal application.

The screenshot shows the NetBeans IDE's Output window. The title bar of the window reads "Output". Inside, there are three tabs: "ShoppingPortalClient (run-single)", "Java DB Database Process", and "GlassFish Server". The "ShoppingPortalClient" tab is active and contains the following text:
warning: C:\users\vayashree\documents\netbeansprojects\shoppingportalclient\src
Laptop
MobilePhone
Personal Digital Assistant

run-single:
BUILD SUCCESSFUL (total time: 1 minute 31 seconds)

Figure 4.7: Executing a Stateful Session Bean Remotely

4.3.3 Accessing JNDI Lookups

The client can access the local or remote interface of an enterprise bean either through EJB objects or through lookup services such as JNDI. JNDI is a naming and directory service that is used to register enterprise beans for remote invocation.

To perform the lookup on EJB object using JNDI service, the following components needs to be set that are as follows:

- **JNDI initialization parameters** – The JNDI initialization parameters includes JNDI driver which is used to invoke the registered enterprise bean from the naming directory. The JNDI driver varies from container to container used for managing the bean instance.

Thus, to access the JNDI service, the client must provide the manual properties for accessing the container specific JNDI driver in the client program. However, hard coding the JNDI driver destroys the code portability. Thus, JNDI API provides a default initial context which is used by the container to set the container specific JNDI properties. These are set for the client environment before the bean is accessed.

- **InitialContext Class**

An `InitialContext` class is used to create an entry point to the naming system used in the application. The objects of the application are bound with JNDI names for reference and used in the application. JNDI organizes all the objects of the application into a hierarchy and the `InitialContext` instance is used to create the root of the hierarchy.

The `javax.naming.InitialContext` class implements the `Context` interface.

Code Snippet 4 demonstrates usage of `InitialContext` object.

Code Snippet 4:

```
...
Context c = new InitialContext();
return (ProductCatalogBeanRemote) c.lookup("java:global/
ShoppingPortal/ShoppingPortal-ejb/ProductCatalogBean!beans.
ProductCatalogBeanRemote");
...
```

Code Snippet 4 shows the usage of `InitialContext` object which returns the reference of `ProductCatalogBeanRemote` interface.

- **Configuring JNDI**

A lookup operation in the JNDI namespace can be performed through `lookup` method. In Code Snippet 4, a lookup operation was performed to access the remote interface of the `ProductCatalogBean`. The namespace in the code performs lookup on global namespace.

The `lookup` method uses the deployment descriptor to obtain the JNDI name and object binding.

Code Snippet 5 shows a sample deployment descriptor with JNDI name bindings defined.

Code Snippet 5:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE glassfish-ejb-jar PUBLIC "-//GlassFish.org//DTD
GlassFish Application Server 3.1 EJB 3.1//EN" "http://glassfish.
org/dtds/glassfish-ejb-jar_3_1-1.dtd">
<glassfish-ejb-jar>
  <enterprise-beans>
    <ejb>
      <ejb-name>ProductCatalogBean</ejb-name>
      <jndi-name>pro</jndi-name>
    </ejb>
  </enterprise-beans>
</glassfish-ejb-jar>
```

Deployment descriptor is a declarative XML file used to configure an application. In Code Snippet 5, the ProductCatalogBean is bound to the JNDI name 'pro'.

4.3.4 Web Clients

Web clients are different Web components of the application that can access the enterprise bean through remote interface or local interface based on interfaces defined by the enterprise bean.

The Web service clients can invoke the business methods of the Stateless Session beans. A Web client can access all the public methods of the enterprise bean.

4.4 Injecting Resources in Enterprise Beans

Resource injection enables developers to inject resources such as databases, connectors, and so on into container-managed components such as Web components and bean components. The resources must be defined in the JNDI namespace so that it can be injected into the component.

The `javax.annotation.Resource` is used to declare a reference to a resource. The annotation is added in the code as `@Resource`. The resource injection can be class based injection, field/method based injection.

A Resource annotation has five elements namely, `name`, `type`, `authenticationType`, `shareable`, `mappedName`, and `description`. The description of the elements is as follows:

- `name` field has the JNDI name of the resource being accessed.
- `type` implies the type of the resource.

- ❑ `authenticationType` refers to the method of authentication used by the resource, if any.
- ❑ `shareable` implies whether the resource can be shared with other components.
- ❑ `mappedName` implies any system specific name onto which the resource has to be mapped.
- ❑ `Description` describes the resource.

In order to use field-based injection, the field declaration has to be preceded by the `Resource` annotation.

For method based injection, a setter method can be declared preceded by the `@Resource` annotation. The setter method must follow the bean naming convention; the method name should start with 'set'.

Class-based resource injection can be done by providing the `@Resource` annotation with appropriate name and type values.

4.5

Exception Handling in Session Beans

Exceptions in Java refers to unusual conditions that arise during execution of the application. Java provides exception handling mechanisms to make the application robust. An exception is an event, which occurs during the execution of a program which disrupts the normal flow of program's instructions.

Some of these exceptions are caused by user error, some as a result of programming error, and others by physical resources that have failed in some manner.

Exception handling is done through mechanisms such as try-catch block and throws statement in Java code.

In case of a `try-catch` block the code which is likely to raise an exception is enclosed in the `try` block and the `catch` block comprises code to be executed to handle the exception which occurred in the `try` block.

In a given block of code if there is a probability of an exception then, it can be explicitly mentioned with `throws` statement. The `catch` block is responsible for handling the exception.

4.5.1

Types of Exceptions

There are three categories of exceptions:

- ❑ **Checked exceptions** - A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer – invalid conditions in areas outside the immediate control of the program (invalid user input, database problems, network outages, and absent files).
- ❑ **Unchecked/Runtime exceptions** – Unchecked exceptions occur due to error in the programming logic, they cannot be detected at the compile time therefore are known as runtime exceptions.

- **Runtime Exceptions** – Runtime exceptions occur due to incorrect logic implemented in the application. These are detected only when the application executes. They cannot be detected during the application compile time.

4.5.2 Java EE Exceptions

According to EJB specification, an enterprise application may have two types of exceptions – Application exceptions and System exceptions.

Application exceptions arise due to certain conditions in the business logic of the application.

System exceptions arise due to faults in the application infrastructure or failure of database connections.

Application exceptions in case of enterprise applications are – CreateException, FinderException, and RemoveException.

Commonly occurring system exceptions are RemoteException, NamingExceptions, SQLException, and so on.

An EJBException is thrown when either an Application exception or System exception occurs.

Table 4.1 lists different types of exceptions that may occur in an EJB application.

Exception	Description
CreateException	Raised when the instantiation of an object or bean fails.
FinderException	Raised when the application is unable to find an object.
RemoveException	Raised when the application cannot remove an instance of bean cannot be removed from the container.
RemoteException	Raised when there is a network failure and the client cannot access a remote object.
NamingException	Raised when the jndi name could not be resolved to the object.
SQLException	Raised when the bean could not get response from the application database.
EJBException	It can be raised when the bean could not respond to the application clients appropriately.

Table 4.1: Exceptions in EJB

4.6

Exception Handling in Enterprise Applications

In enterprise applications, exception handling can be done either by the container or through an enterprise bean. Both application exceptions and system exceptions can be handled by either a container or a bean.

4.6.1 Handling Exceptions Through a Container

Application exceptions can be thrown by enterprise bean methods invoked by clients or other enterprise beans. When an exception is thrown the transaction in which the exception is thrown need not be rolled back. Instead the exception thrown is passed onto the caller bean or client. Enterprise applications do not roll back the actions performed. The exception thrown is returned back to the caller and then, container continues executing the exception handling code.

Whenever a container encounters system exceptions, then it performs the following operations:

- The container first logs the system exception.
- It then deallocates all the allocated resources and performs all the cleanup operations on the bean instance.
- Removes the bean instance from the memory which has caused the system exception.
- The calling method is informed about the system exception by throwing an appropriate exception such as `javax.ejb.EJBException`, `javax.EJB.NoSuchEJBException`, and so on.

4.6.2 Handling Exceptions Through Beans

Exceptions can be handled in the bean by writing explicit code for exception handling. If the exceptions are handled through bean classes then the bean tries to recover from the application. If the application can be recovered then the code for recovery is invoked. If the exception is not recoverable then the transaction is rolled back.

Code Snippet 6 demonstrates the usage of exceptions in an enterprise application.

Code Snippet 6:

```
...
public class ExceptionDemo {
    public static void main(String[] args) throws
        FileNotFoundException, IOException {
        try{
            testException(-5);
            testException(-10);
        }catch(FileNotFoundException e){
            e.printStackTrace();
        }catch(IOException e){
            e.printStackTrace();
        }finally{
```

```
System.out.println("Releasing resources");

}

testException(15);

}

public static void testException(int i) throws
FileNotFoundException, IOException{

    if(i < 0){

        FileNotFoundException myException = new
FileNotFoundException("Negative Integer "+i);

        throw myException;

    }elseif(i > 10){

        throw new IOException("Only supported for index 0 to 10");

    }

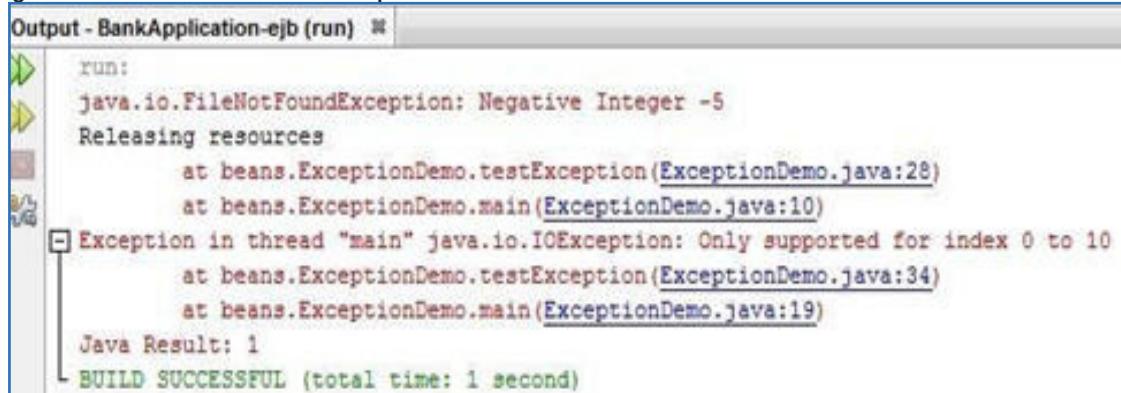
}

}

}
```

In Code Snippet 6, the usage of `FileNotFoundException` and `IOException` has been demonstrated. In order to demonstrate the exception the method `testException()` has been used. An `FileNotFoundException` is thrown when the parameter to the `testException()` method is negative. When the parameter of the `testException()` method is greater than 15 an `IOException` is thrown.

Figure 4.8 demonstrates the output of the code.



The screenshot shows the NetBeans IDE's Output window titled "Output - BankApplication-ejb (run)". The window displays the following log entries:

```
RUN:
java.io.FileNotFoundException: Negative Integer -5
Releasing resources
    at beans.ExceptionDemo.testException(ExceptionDemo.java:28)
    at beans.ExceptionDemo.main(ExceptionDemo.java:10)
Exception in thread "main" java.io.IOException: Only supported for index 0 to 10
    at beans.ExceptionDemo.testException(ExceptionDemo.java:34)
    at beans.ExceptionDemo.main(ExceptionDemo.java:19)
Java Result: 1
BUILD SUCCESSFUL (total time: 1 second)
```

Figure 4.8: Output of Exception Handling Code

The runtime exceptions can be application exceptions and system exceptions. Beans can handle application exceptions by redirecting to the exception handling code in the bean. It is a good design choice to handle exceptions other than the application exceptions through the container.

Whenever an application exception is thrown, the bean handles them in the following ways:

- If the exception handling code for the application exception is defined then, the exception handling code is executed.
- If the application exception is unrecoverable, then the bean class throws an exception and exit.
- If the exception causes loss of data integrity then the transaction is rolled back.

System exceptions such as `EJBException` and `RuntimeException` are not handled through bean methods. They are generally handled through the container.

4.7

Exception Logging

Applications log exceptions for further analysis by the developer and for rectifying the cause of an exception. Exceptions can be logged at multiple levels – Component-level logging, middle level logging, and top level logging.

Component level logging of the exceptions refer to exception logging at each component of the application. Whenever, a component throws an exception it also makes an entry into the exception log. This mechanism of exception logging requires logging code to be written for each component.

In case of middle level exception logging, the exception log is maintained in a location common to different components where sufficient information about all the components is available such as a container. Middle level logging does not have information about the entire application but a subset of application components.

Exception log can also be maintained at the top level, this refers to an exception log of the application. Top level logging of exceptions has information about all the exceptions thrown in the application. Such exception logs are useful to assess the performance of the entire application.

Code Snippet 7 demonstrates exception logging in enterprise applications.

Code Snippet 7:

```
private com.card.CardValidationRemote lookupCardValidationBean() {  
    try {  
        javax.naming.Context c = new javax.naming.  
InitialContext();  
        Object remote = c.lookup(...);  
        com.card.CardValidationRemoteHome rv = ... return  
rv.create();  
    } catch (javax.naming.NamingException ne) {
```

```
java.util.logging.Logger.getLogger(getClass()).
getName()).log(java.util.logging.Level.SEVERE,"exception caught",
ne);

throw new RuntimeException(ne);

} catch (javax.ejb.CreateException ce) {

java.util.logging.Logger.getLogger(getClass().getName()).
log(java.util.logging.Level.SEVERE,"exception caught",ce);

throw new RuntimeException(ce);

} catch (java.rmi.RemoteException re) {

java.util.logging.Logger.getLogger(getClass().getName()).
log(java.util.logging.Level.SEVERE,"exception caught",re);

throw new RuntimeException(re);

}

}
```

In Code Snippet 7, there is an additional `Logger` statement for each exception, which logs the exception state to the system log.

Check Your Progress

1. Which of the following is not an element of @Resource annotation?

(A)	name	(C)	shared
(B)	type	(D)	None of these

2. Which of the following are System exceptions?

(A)	EJBException	(C)	ArithmetiException
(B)	FileNotFoundException	(D)	None of these

3. Which of the following callback methods are only used by Stateful Session Beans?

(A)	PreDestroy	(C)	PostActivate
(B)	PostConstruct	(D)	None of these

4. Which of the following is not true about Stateful Session Beans?

(A)	Stateful Session Beans store the conversational state of the bean	(C)	Maintaining a pool of Stateful Session Beans is simpler than in Stateless Session Beans
(B)	Stateful Session Beans have a passivated state	(D)	None of these

5. Which of the following is an invalid exception?

(A)	RemoveException	(C)	RemoteException
(B)	RestoreException	(D)	None of these

Check Your Progress

6. Which of the following objects are used for exception logging?

(A)	Exception	(C)	RuntimeException
(B)	Logger	(D)	None of these

Answer

1.	C
2.	A
3.	C
4.	D
5.	B
6.	B

Summary

- Stateful Session Beans store the conversational state of the session.
- Each Stateful Session Bean has a unique identity and is associated with a single client.
- There are three states in the lifecycle of a Stateful Session Bean – Does not Exist, Activated, and Passivated.
- There are four categories of lifecycle callback methods – PostConstruct, PrePassivate, PostActivate, and PreDestroy.
- Stateful Session Beans can be accessed through local and remote interface.
- Stateful Session Beans can be activated through local, remote, and Web service clients.
- Application and system exceptions are two types of exceptions in an enterprise application according to EJB specification.

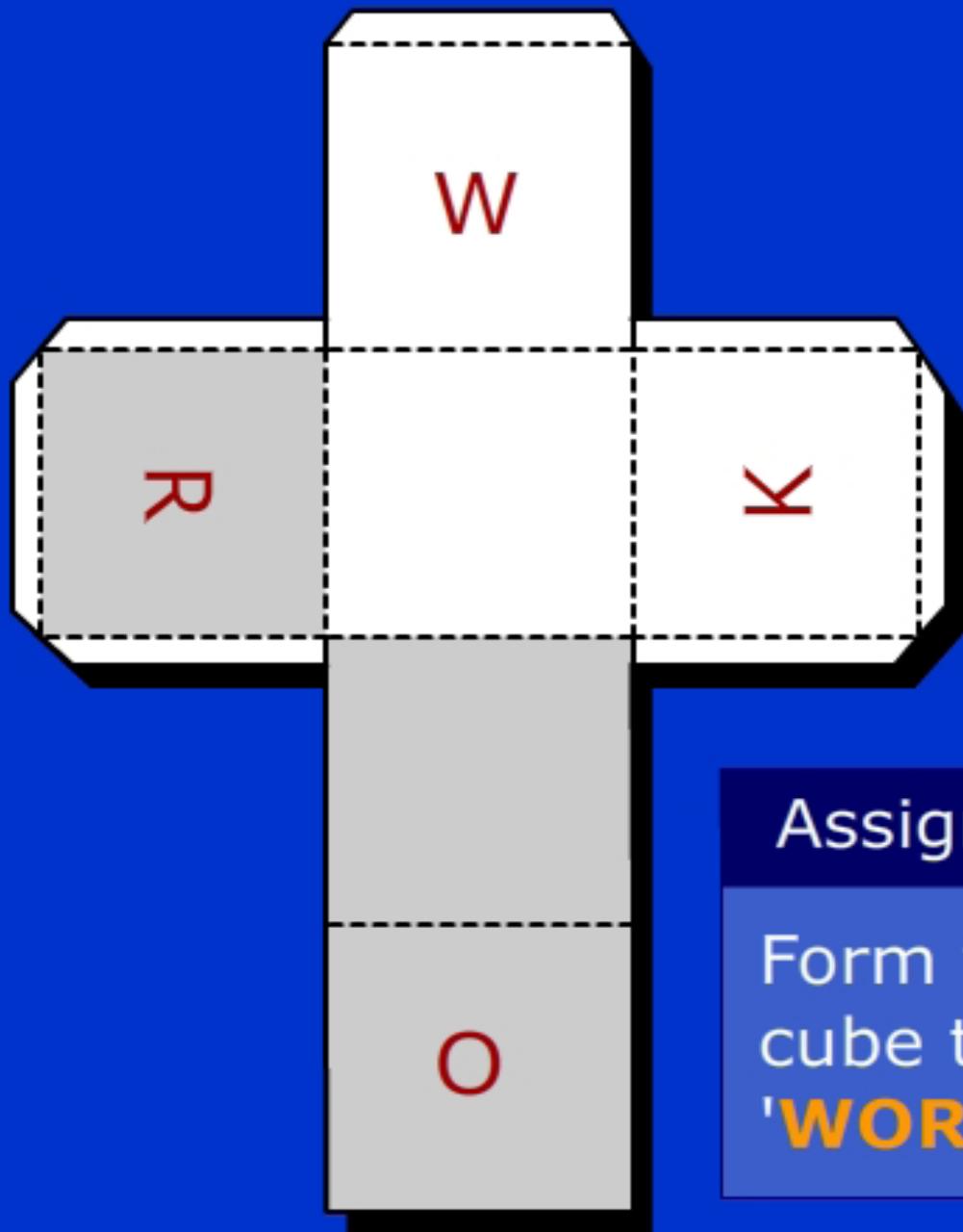
Technowise



Are you a
TECHNO GEEK
looking for updates?

Login to

www.onlinevarsity.com



Assignment

Form the
cube to read
'WORK'.

"Practice does not make perfect. Only perfect practice makes perfect."
- Vince Lombardi

For perfection, solve assignments @

www.onlinevarsity.com



Welcome to the Session, **Singleton Session Beans**.

This session explains the working of Singleton Session bean. It describes the characteristics and various stages in the lifecycle of Singleton Session bean. Further, it discusses on how to manage concurrent access to the Singleton Session bean. Finally, it explains how to develop Singleton Session bean in the Java EE application.

In this Session, you will learn to:

- Explain the need of Singleton Session bean
- Describe the characteristics of Singleton Session bean
- Describe the various stages in the lifecycle of Singleton Session bean
- Explain Singleton Session bean specification and its initialization strategies
- Describe how to achieve concurrency access to Singleton Session bean
- Explain container-managed concurrent access to Singleton Session bean
- Explain bean-managed concurrent access to Singleton Session bean
- Explain the mechanism to configure access time out in concurrency
- Explain how to implement Singleton Session bean in an enterprise application

5.1 Introduction

Session beans are used to implement business logics in an enterprise application. Based on the requirements of the enterprise application, Stateless Session beans or Stateful Session beans are designed. Stateless Session bean offer better scalability for the applications, as few instances can support large number of clients. However, Stateful Session beans are associated with specific clients, hence, each new clients request, creates a new instance of the Stateful Session bean.

In both, the Stateless Session bean and Stateful Session bean model, only single request can access the bean instance at any time. This means these beans are thread-safe, as each request is represented by the invocation of a single thread.

Figure 5.1 shows the Session beans pulled from the pool of instances.

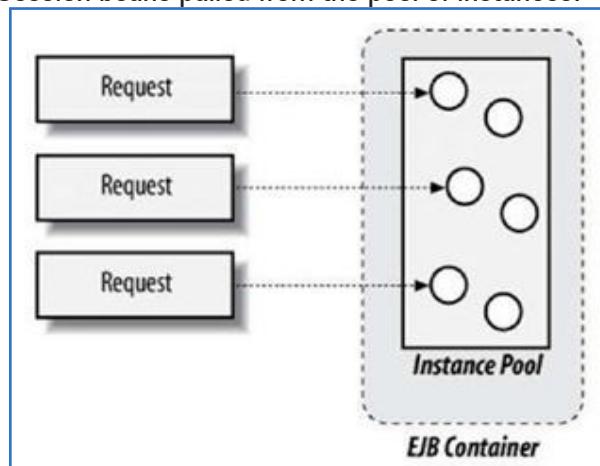


Figure 5.1: Pool of Session Bean Instances

There are some situations when a single shared instances is used by all the clients. This lead to the concurrence access to the bean instance. EJB 3.1 specification have introduced Singleton Session bean which provides concurrent access to the clients.

5.1.1 Singleton Session Bean

A Singleton Session bean is one which gets instantiated only once for every application. This means that only once instance of Singleton Session bean exists during the entire lifecycle of an application. It supports concurrent access which means one or more clients can simultaneously access the same bean instance at the same time.

Figure 5.2 shows how a Singleton Session bean is accessed by multiple clients.

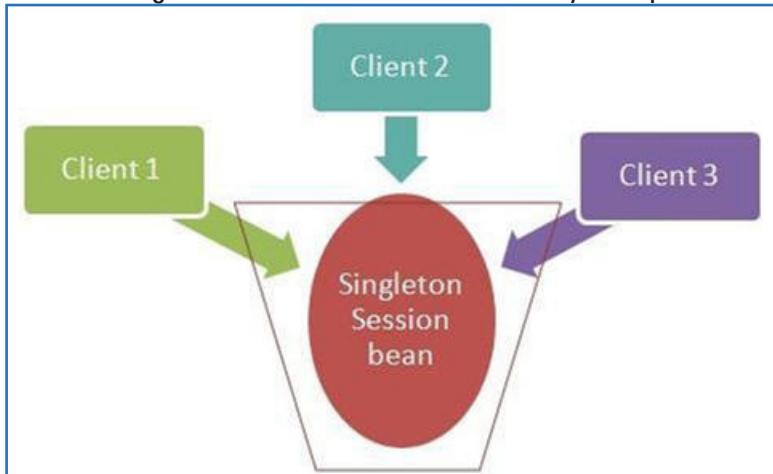


Figure 5.2: Singleton Session Bean

As shown in Figure 5.2, a Singleton Session bean instance lives for the entire lifecycle of the application. It maintains the state between the client's methods invocation, however, the state is not survived if the container crashes or application is shutdown.

5.1.2 Characteristics of Singleton Session Bean

A Singleton Session bean has the following characteristics:

- A single instance of the Singleton Session bean is shared by all request.
- A Singleton Session bean is similar to a Stateless Session bean. Like Stateless Session beans, the conversational state of Singleton Session beans cannot be persisted onto permanent storage. However, they retain the state of the client during multiple method invocations.
- Singleton Session bean instance cannot be passivated.
- The bean instance must be thread-safe, as it is handled by multiple clients simultaneously.
- The container is responsible for deciding when to initialize a Singleton bean instance.
- It has less memory footprint as compared to other session beans.

In case of large enterprise applications, the application container may not reside on a single server. The application server function may be distributed over multiple servers and therefore, the application container is also scaled over multiple servers. In such a situation, every instance of application container on different server machines will have an instance of the Singleton Session bean.

5.2

Lifecycle of Singleton Session Bean

Lifecycle of Singleton Session bean is similar to that of a Stateless Session bean. It comprises two stages – Does not Exist and Ready.

Figure 5.3 shows the lifecycle of a Singleton Session bean.

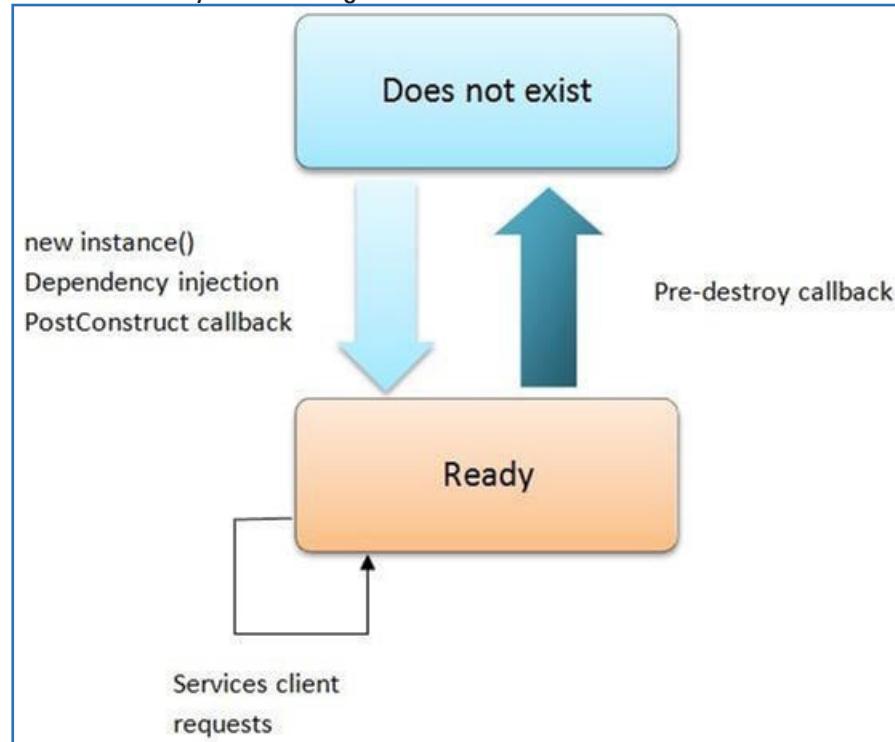


Figure 5.3: Lifecycle of a Singleton Session Bean

A Singleton Session bean is in 'Does not Exist' state until the bean has been instantiated by the container. After the container instantiates the bean instance, it moves to the 'Ready' state.

A Singleton Session bean can be explicitly started by the container during the application startup by invoking a method annotated with `@javax.ejb.Startup`. This annotation indicates the bean instance must be invoked with the application start up. In addition, the annotation `@javax.ejb.Singleton` is also used to mark a singleton session bean class.

The Singleton Session bean comes into existence when the container instantiates it. Like a Stateless Session bean, it is ready to accept client requests, once it is instantiated.

Once the session bean is instantiated all the required resources for the bean class are acquired through dependency injection. PostConstruct callback methods are invoked by the container over the bean instance. The methods annotated with `@PostConstruct` are invoked.

After instantiation, the session bean is in 'Ready' state as long as the application runs in the application container. The bean in the 'Ready' state and responds to client requests.

Finally, when the application shut downs, the `@PreDestroy` callback method is invoked by the application container, before removing the bean instance from the container.

5.3

Singleton Session Bean Specification

A Singleton Session bean is similar to Stateless and Stateful Session bean. It comprises a bean class and one or more business interfaces that are optional.

Bean Class

It is a standard Java class which needs to be marked with `@Singleton` annotation.

 Business Interface

Business interface of a Singleton Session bean is similar to Stateless and Stateful bean interfaces. The business interface can be a local or remote. Singleton Session bean also supports no-interface local view for the clients deployed on the application server.

Code Snippet 1 demonstrates the creation of a Singleton Session bean.

Code Snippet 1:

```
import javax.ejb.Singleton;
import javax.ejb.Startup;

@Singleton(name = "ItemCount")
public class ShoppingItemCount {
    private int counter = 0;

    // Increment number of shopper counter
    public void incrementCounter() {
        shopperCounter++;
    }

    // Return number of shoppers
    public int getCounter() {
        return Counter;
    }

    // Reset counter
    public void initializeCounter() {
        shopperCounter = 0;
    }
}
```

Code Snippet 1 annotates the `ShoppingItemCount` class with `@Singleton` annotation. The `name` attribute of the annotation defines the ejb-name of the bean by which it is referenced by other beans in the container.

The class defines three methods namely, `incrementCounter()`, `getCounter()`, and `initializeCounter()` that increment the counter variable by 1, returns the value of the counter variable, and initializes the counter value to zero.

5.3.1 Initialize Singleton Session Bean

By default, EJB container is responsible for initializing the Singleton Session bean instance. However, the bean developer can also configure when to initialize the Singleton Session bean instance.

If the Singleton Session bean class is annotated with `@Startup` annotation, then the bean is initialized during the application startup sequence. This initialization is also known as eager initialization, where the container initializes the Singleton Session bean as soon as the application startup.

Code Snippet 2 shows how to initialize the Singleton Session bean with application startup.

Code Snippet 2:

```
@Startup  
@Singleton  
public class ShoppingItemCount {  
    ...  
}
```

In Code Snippet 2, for eager initialization, the bean class is annotated with `@Startup` annotation. This instructs the container to configure and initialize the Singleton Session bean during the startup process of the application.

Alternatively, Singleton beans can also be instantiated by creating the object of the Singleton bean class. However, the container would instantiate it only once. The initialization parameters of the Singleton bean are passed through the constructor method.

5.3.2 Startup Initialization Dependencies

When the container is initializing multiple Singleton Session beans components in an application, they can have explicit initialization ordering dependencies upon each other. In such cases, the container must define the sequence in which these beans have to be initialized. The sequence of initialization is essential to address the dependencies among different Singleton Session beans.

`javax.ejb.DependsOn` annotation is used to ensure that a specific Singleton Session bean is initialized, before other Singleton Session beans dependent on it are initialized by the container.

The `@DependsOn` annotation has a `value` attribute which defines the target and holds the Singleton Session bean class name. Based on this annotation, the container ensures that the dependency is satisfied, before the dependent beans are initialized.

Consider a scenario of shopping an item from a Website and earning points based on the number of items purchased. The ShoppingItemPoints Singleton Session bean has dependency on the item shopped from that Website. To achieve this functionality, two beans – ShoppingItemCount and ShoppingItemPoints are created. The ShoppingItemPoints bean cannot update the points, until the shopping is not successful. This results in the use of @DependsOn annotation which declares the dependency of the Singleton Session bean.

Code Snippet 3 shows the implementation of ShoppingItemPoints dependent on ShoppingItemCount session bean.

Code Snippet 3:

```
//class definition of ShoppingItemPoints  
  
@Singleton  
  
@Startup  
  
@DependsOn("ItemCount")  
  
public class ShoppingItemPoints {  
  
    private final int points;  
  
    public void AddPoints() {  
  
        // Business logic for adding points  
  
    }  
  
}
```

Code Snippet 3 shows two Singleton bean classes annotated with @Singleton annotation. The ShoppingItemPoints class has been annotated with the @DependsOn annotation indicating that the instantiation of the bean object is dependent on the ShoppingItemCount object.

Kindly, note that the dependencies can extend to multiple levels of Singleton Session beans. In that case, the value attribute of @DependsOn annotation holds one or more strings.

For example, @DependsOn({ "ItemCount", "PaymentType" }). Here, each value specifies the ejb-name of the target Singleton Session bean.

5.3.3 LifeCycle Callback Methods

The callback events handle the construction and destruction of a Singleton Session bean. The callback events are mapped to the following events:

- PostConstruct** – It is denoted by @PostConstruct annotation and is fired after a bean instance is instantiated by the container, and before the invocation of the first business method defined in the class.

- **PreDestroy** – It is denoted by `@PreDestroy` annotation and is fired when the application is shutting down. During shut down, the container invokes the PreDestroy lifecycle interceptor method on the Singleton Session bean instance. After the method completion, the container destroys all the dependent Singleton Session bean instances that were created for `@DependsOn` relationship in the reverse order of their creation.

Code Snippet 4 demonstrates the PostConstruct and PreDestroy interceptor methods.

Code Snippet 4:

```
...
@Singleton(name = "ItemCount")
@Startup
public class ShoppingItemCount {
    ...
    @PostConstruct
    public void applicationStartup() {
        System.out.println("ApplicationStartup - Initializing the
countervariable to zero.");
        Counter = 0;
    }
    @PreDestroy
    public void applicationShutdown() {
        System.out.println("ApplicationShutdown Happening");
    }
}
```

Code Snippet 4 handles the application startup and application shutdown in the callback interceptor methods.



Unlike Stateless Session bean, Singleton Session beans are never passivated and hence callback such as `@PrePassivate` and `@PostActivate` are not applicable on it.

5.4**Concurrency in a Singleton Session Bean**

Every application has only one instance of a Singleton Session bean. This instance can be accessed by multiple clients. When multiple clients access the same instance of the Singleton Session bean, then this results into concurrency. The concurrent access handling of the bean is managed by the container and hence, a client may not be concerned about other clients accessing the same bean instance.

There are two ways by which the concurrent access to a singleton session bean can be controlled. These are as followed:

- **Container-managed concurrent** - As the name suggests, the container is responsible for managing the concurrent access to the Singleton Session bean data or methods. This is the default concurrency management type.
- **Bean-managed concurrent** – This container provides full access of the bean to the bean developer. Even the synchronization of bean state is managed by the bean developer.

A Singleton Session bean can use either container-managed concurrency or bean-managed concurrency, however, the bean developer cannot use both the techniques.

The concurrency management method used for the current enterprise bean can be defined through the annotation `javax.ejbConcurrencyManagement`. The type of concurrency management can be specified through a `type` attribute whose value can be either set to `javax.ejbConcurrencyManagementType.CONTAINER` or `javax.ejbConcurrencyManagementType.BEAN`.

5.4.1 Container Managed Concurrency

The container manages the concurrent access to the Singleton Session bean by associating the business methods with a lock. The methods can be defined with either a shared read lock or exclusive write lock.

Annotations are used to define the type of lock to be acquired, while accessing the business method. The annotation, `javax.ejb.Lock` is used to specify the required locks. It has an attribute `javax.ejb.LockType`, which accepts the lock type as `READ` or `WRITE` for the bean methods.

Marking the business method with `@Lock(LockType.READ)` grants shared access to the bean method. This means any number of concurrent invocation to the methods with read locks are allowed simultaneously on the bean instance. Annotating the bean method with `@Lock(LockType.WRITE)` grants exclusive access to the bean method. This means no concurrent invocation of the method is allowed, until the method holding the lock completes its task.

If the annotation is not specified, then the default lock type is `@Lock(@LockType.WRITE)`.

Code Snippet 5 demonstrates the container-managed concurrency applied to the Singleton Session bean.

Code Snippet 5:

```
...
@Singleton(name = "WebsiteVisitCount")
@Startup
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)
public class WebsitevisitCount{
    ...
    private int Counter;

    // Increment number of visitors
    @Lock(LockType.WRITE)
    public void incrementCounter() {
        Counter++;
    }

    // Return number of visitors
    @Lock(LockType.READ)
    public int getVisitorCount() {
        return Counter;
    }
}
```

Code Snippet 5 shows the implementation of a Singleton Session bean on the number of visitors who accessed the Web page. The `@ConcurrencyManagement` annotation specifies that the concurrency type is set to CONTAINER. The `@Lock(LockType.READ)` annotation is specified at the method-level to the `getVisitorCount()` which returns the number of users visited to the site.

The `@Lock(LockType.WRITE)` acquires the WRITE lock which blocks the access of the method for all other clients, while one client is accessing it.



The `@Lock` annotation can also be specified at the class-level which means all methods of the Singleton Session bean are specified with the same lock type. However, you can override this, by providing the method-level annotation.

5.4.2 Bean Managed Concurrency

Bean managed concurrency is specified through the annotation `ConcurrentManagementType`. BEAN. When bean managed concurrency control mechanism is used the developer is responsible for incorporating the concurrent access mechanism in the bean class. It is the responsibility of the bean developer to synchronize the state of the Singleton Session bean to avoid synchronization errors occurring due to the concurrent access.

The bean developer can use the `synchronized` and `volatile` primitive types for synchronization.

5.4.3 Concurrent Access Timeout

When multiple clients are trying to access the Singleton Session bean, then if one client is holding an exclusive write lock on the method, then access by other clients is blocked. However, the client requests cannot be blocked indefinitely, therefore, an access timeout value is associated with each blocked request.

The `@AccessTimeOut` annotation can be used to specify the maximum time period for which a request can be blocked, before time out happens. After the mentioned timeout period is crossed, the EJB container generates the `ConcurrentAccessTimeOutException` which returns the control to the client waiting for the method access.

The `@AccessTimeOut` has two attributes namely, `value` and `timeUnit`. The `timeUnit` attribute is optional. The default time unit specified in the `value` attribute is in milliseconds. However, the developer can change it to one of the constants provided in the `java.util.concurrent.TimeUnit`. The constants are `MILLISECONDS`, `MICROSECONDS`, or `SECONDS` which can be specified in the `timeUnit` attribute.

If the `AccessTimeOut` value specified is -1, then it indicates that the client will block indefinitely until it gains access to the bean. When `AccessTimeOut` value is 0, it indicates that the concurrent access is not allowed.

Code Snippet 6 shows the configuration of access time value for the Singleton Session bean.

Code Snippet 6:

```
// Increment number of visitors
@Lock(LockType.WRITE)
@AccessTimeout(value=120000)
public void incrementCounter() {
    Counter++;
}
```

Code Snippet 6 specifies the access time out for the `incrementCounter()` method to 120000 milliseconds.

5.5 Developing Singleton Session Bean

A Singleton Session bean can be created by choosing the option **Singleton**, while creating the session bean in NetBeans IDE as shown in figure 5.4.

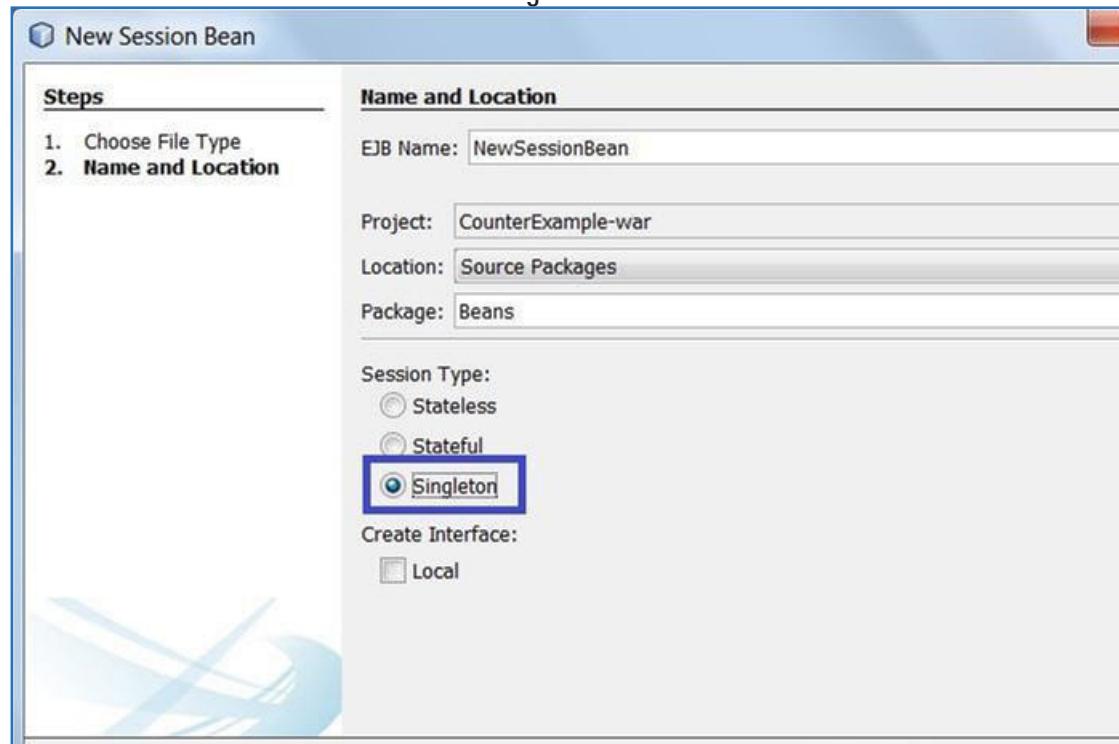


Figure 5.4: Creating Singleton Session Bean in NetBeans IDE

The singleton session bean is created with the bean class name and appropriate annotations.

Modify the code as shown in Code Snippet 7 to add a variable to the bean.

Code Snippet 7:

```
package Beans;

import javax.ejbConcurrencyManagement;
import static javax.ejbConcurrencyManagementType.CONAINER;
import javax.ejbLocalBean;
import javax.ejbLock;
import static javax.ejbLockType.WRITE;
import javax.ejbSingleton;

@Singleton
@ConcurrencyManagement(CONAINER)
@LocalBean

public class CountBean {
    static int visits=0;
    @Lock(WRITE)
    public int incrementCounter() {
        return ++visits;
    }
}
```

Code Snippet 7 defines a container managed Singleton Session bean. The session bean has a property visits which is incremented through the incrementCounter() method. The @LocalBean designates that the bean exposes a no-interface view. Two client classes are defined which access the visits property of the Singleton Session bean.

Code Snippets 8 and 9 shows the code of two different clients that accesses the Singleton bean.

Code Snippet 8:

```
...
public class Module5Client1{
public Module5Client1() {
    CountBean C = new CountBean();
    System.out.println("Counter value seen by Client 1 "+C.incrementCounter());
}

}
```

Code Snippet 9:

```
...
public class Module5Client2{
public Module5Client2() {
    CountBean C = new CountBean();
    System.out.println("Counter value seen by Client 2 "+C.incrementCounter());
}

}
```

Code Snippets 8 and 9 create two Java main classes which invoke the Singleton Session bean in the constructor of the class. Whenever an object of the client class is invoked, the Singleton Session bean is accessed.

Code Snippet 10 shows another main class which creates multiple instances of clients to access the Singleton Session bean.

Code Snippet 10:

```
package Beans;  
  
public class ClientRequest {  
  
    public static void main(String[] args) {  
  
        Module5Client1 m1 = new Module5Client1();  
  
        Module5Client2 m3 = new Module5Client2();  
  
        Module5Client1 m2 = new Module5Client1();  
  
        Module5Client2 m4 = new Module5Client2();  
  
    }  
  
}
```

Code Snippet 10 creates four instances of clients which are equivalent to client requests to access the Singleton Session bean.

Figure 5.5 shows the output when the ClientRequest Java class is executed.



Figure 5.5: Output of ClientRequest

Check Your Progress

1. The lifecycle of a Singleton Session bean is similar to _____.

(A) Stateless Session bean	(C) Message Driven bean
(B) Stateful Session bean	(D) None of these

2. The default concurrency management type for a Singleton bean is _____.

(A) Bean managed concurrency	(C) Using synchronized blocks of code
(B) Container managed concurrency	(D) None of these

3. Which of the following callback interceptor methods are supported by Singleton Session beans?

(A) PostConstruct and PreDestroy	(C) PostConstruct and PreActivate
(B) PostPassivate and PreDestroy	(D) PostConstruct

4. What should be the value of `AccessTimeOut` annotation to indicate that the client will be blocked indefinitely?

(A) 1	(C) -1
(B) 0	(D) None of these

5. Which of the following attribute is supplied with `@Lock` annotation?

(A) LockType	(C) WriteLock
(B) ReadLock	(D) Value

Answer

1.	A
2.	B
3.	A
4.	C
5.	A

Summary

- A Singleton Session bean is instantiated only once in the application lifecycle and retained throughout the application lifecycle.
- Singleton Session bean cannot store the conversational state nor can it be passivated.
- Singleton Session bean has `@PreDestroy` and `@PostConstruct` callback methods.
- When there are multiple Singleton beans in the application and they are dependent on each other, then the EJB container should initialize them in right order.
- `@DependsOn` annotation is used to define the dependencies among the Singleton beans.
- Multiple clients can access the Singleton Session bean in the application; this concurrent access has to be managed.
- Container-managed and Bean-managed concurrent access are the two variants of concurrency management for Singleton Session beans.
- The client requests cannot be blocked indefinitely, therefore, an access time out value is associated with each blocked request.



Login to www.onlinevarsity.com

ASK to LEARN

Questions
in your
mind?



are here to HELP

Post your queries in **ASK to LEARN** @

www.onlinevarsity.com



Welcome to the Session, **Introduction to Messaging**.

This session introduces the concept of Message-Oriented Middleware (MOM) architecture used to send messages between the clients. It explains the use of Java Messaging Service (JMS) API as a messaging standard used on Java EE platform. Further, the session explains the use of Message-driven bean for performing asynchronous communication in enterprise applications.

In this Session, you will learn to:

- Describe the messaging concept and its architecture
- Describe Java Messaging Service API
- Describe various messaging models supported by JMS
- Explain the working of Message-driven beans
- Describe how to create and configure a Message-driven bean in Java EE application

6.1 Introduction

The Java application architecture is loosely coupled, which implies that all the components of the application are functionally independent of each other. Any changes made to one component of the application do not affect the remaining components of the application. In other words, the components do not know the type, location, or implementation of the other objects.

Then, a very obvious question arises, how the components communicate?

There are various mechanisms adopted by different technologies that allow communication between distributed components or objects. You can develop and deploy the distributed objects using many technologies such as Component Object Model (COM) or Distributed Component Object Model (DCOM) in Microsoft, Remote Method Invocation (RMI) provided by Java, Common Object Request Broker Architecture (CORBA) defined by Object Management Group (OMG).

Figure 6.1 depicts the RMI on a remote object.

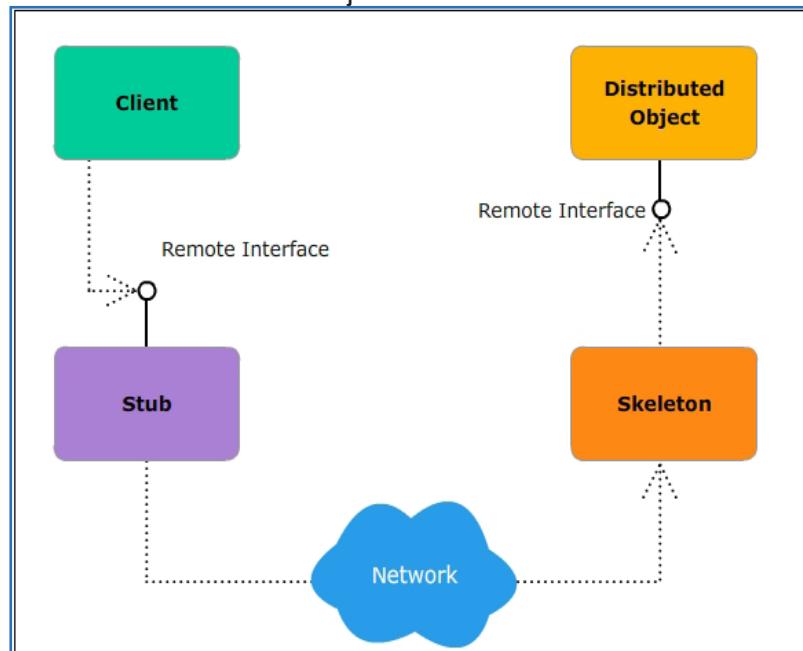


Figure 6.1: RMI

6.1.1

Distributed Communication in Enterprise Applications

Sun Microsystems introduced RMI mechanism which provides a native way to communicate distributed objects running in the different JVMs. The RMI resulted in accessing of objects on network. The Java Remote Invocation over the Internet Inter-ORB Protocol (RMI-IIOP) is an extension of RMI. The RMI-IIOP is the protocol of the enterprise Java Beans developed in the Java enterprise application.

The RMI-IIOP was written by OMG to provide a standard way of communication between the CORBA products provided from different vendors. The use of RMI-IIOP resulted in the

interoperability between the applications based on CORBA architecture. Later on, when the developers of EJB felt a similar need of achieving interoperability among the EJB containers provided by different vendors. They adopted RMI-IIOP over RMI for providing communication between the different EJB components. RMI-IIOP became the protocol for EJBs to communicate and hence, served as a foundation for EJB.

However, RMI-IIOP has several changes which can be described as follows:

- **Asynchronous Communication** – RMI-IIOP does not support asynchronous communication. This means the RMI-IIOP client has to wait for the server response, till the server completes its work and sends the result back to the client. Hence, client keeps on waiting, before continuing its processing.
- **Decoupling** – The RMI-IIOP client needs to be aware with the server for accessing the remote objects through references. This results in the dependency of client on the server. In case if the server has to be removed, then this will affect the client directly.
- **Reliability** – When the RMI-IIOP client invokes the component on the server, the server must be running. In case if the server crashes, then the data might be lost and client may not complete its operations.
- **Support for Multiple Producer and Consumer** – The RMI-IIOP restricts the communication between a single client and a single server at any given of time. It lacks the broadcasting of events from multiple clients to multiple servers.

Thus, enterprise applications supported the concept of messaging. It is a lightweight alternative for communication among the distributed components.

6.2

Messaging

Messaging is an alternative to RMI. It is based on the concept of Message-Oriented Middleware (MOM) which serves as a middleman placed between the client and the server. Generally, the messages are sent and received, so the terminology associated with messaging is a message producer and a message consumer.

The middleman receives messages from message producers and broadcasts these messages to one or multiple message consumers. The message producer can send a message, continue with other tasks, and later be notified of the response when the consumer completes the task of sending a message. This is known as asynchronous processing.

Figure 6.2 displays messaging as against RMI in Java.

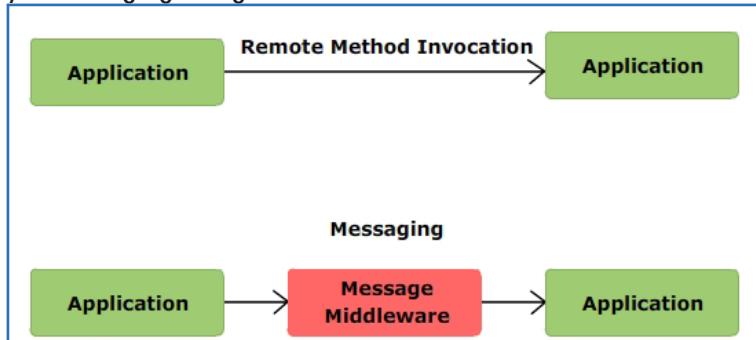


Figure 6.2: Messaging Versus RMI in Java

The implementation of RMI-IIOP has certain concerns which have been addressed by Messaging. These are as follows:

- **Non-blocking request processing or Asynchronous processing:** While executing a request, the messaging client need not block other tasks that it was processing.
- **Decoupling:** In Message-Oriented Middleware (MOM) system, the message sender is not required to know the message receivers as it addresses the messaging system. Message senders are decoupled from message consumers and are not affected by change in customers.
- **Reliability:** In MOM system, a message is delivered even if the customer is not available for a specific period of time. The message is sent to the MOM system which routes the message to the consumer when active. In RMI-IIOP, an exception is thrown when the server is down.
- **Support for Multiple Producer and Consumer:** MOM system can accept messages from multiple message producers and broadcast it to multiple consumers.

6.2.1 Message-oriented Middleware (MOM)

Over the years, there have been different models adopted to make it possible for a software component to communicate with the other over a network.

Some of these are as follows:

- **Remote Procedure Call** - It is also known as RPC-based middleware. In this middleware architecture, one procedure calls another procedure. However, the calling procedure has to wait for the response to be returned by the called procedure, before performing any other task. This type of communication is also referred as synchronous messaging. The same mechanism is adopted by Java to allow communication between two objects using RMI.
- **Object Request Broker** – It is also known as ORB-based middleware which is designed to allow the communication between two systems which may be running on different platforms. For example, Common Request Broker Architecture (CORBA) allows communication between the components running in different environments, such as different hardware, Operating System (OS), and programming languages.

MOM refers to an infrastructure that supports messaging. It can be defined as a software that enables asynchronous message exchange between system components. The software stores the message in the location specified by the sender and acknowledges immediately.

The message sender is known as the producer and the location where the message is stored is known as the destination. The software components can retrieve the stored messages and are known as message consumers.

Figure 6.3 displays the functioning of the MOM and the components.

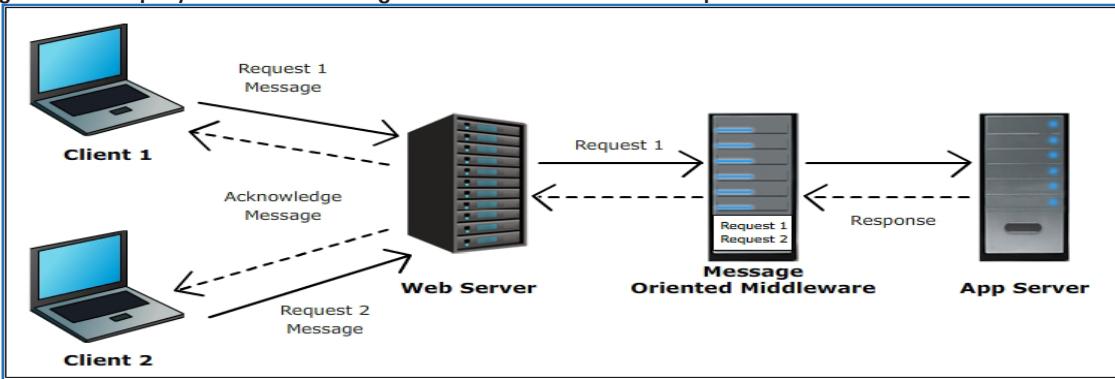


Figure 6.3: Message-oriented Middleware and its Components

MOM based architecture is present in products such as: Sun Java System Messaging Server, Microsoft MSMQ, IBM WebSphere, and so on. Value-added services provided by these products are fault tolerance, load balancing of destinations, inactive subscriber, and so on.

Some of the disadvantages of MOM architecture are as follows:

- Any server supporting MOM-based middleware has its own APIs which is specific to a vendor.
- MOM APIs are not portable to other messaging systems.
- Developers have to learn the proprietary messaging API to incorporate messaging functionality in the applications.

The Java EE platform provides a vendor-neutral API known as Java Messaging Service (JMS). This eliminates the need to learn the vendor-specific MOM-based APIs to perform communication between the components in the enterprise applications.

Java Message Service is a messaging standard which is designed to eliminate the disadvantages of the MOM based products.

6.3

Java Messaging Service (JMS) API

JMS supports messaging among the application components on the Java EE platform. The components can asynchronously communicate through messages. The advantage of asynchronous communication is that it reduces the dependency among the application components. It reduces the idle time which the application components may face while waiting for response from other application components.

The messaging is implemented through the MOM middleware that is in the middle tier of the three-tier application architecture. Any message sent is destined at the server, till it is read by the consumer or the receiver of the message.

The JMS API is divided into two parts:

- **JMS API** - The JMS API provides the functionality of creating, sending, receiving, and reading messages among the application components. It also defines a set of interfaces which can be used in applications to implement the messaging function.

- **Service Provider Interface (SPI)** – The SPI is used as a plug-in for the MOM implementation on the server. It is a JMS provider which talks to server specific MOM implementation.

6.3.1 JMS Communication Models

Messaging requires the developer to decide on the messaging style or domain to be adopted. The messaging domain defines the pattern of communication between the two components of the applications. The pattern of communication decides the technology used for exchanging information between the applications.

There are two messaging models supported by the JMS API:

- Publish-Subscribe model
- Point-to-Point model

□ Publish-Subscribe Model

In this type of messaging style, you can have multiple message producers talking to multiple message consumers. The messages are sent through a virtual channel called Topic. Topic is a logical destination object which contains messages from different sender components. The component which is sending the message to the topic is said to be a publisher.

In order to access messages from the topic, the components should subscribe to the topic. The topic sends a copy of the message received to all the subscribers. Subscriber is the term used to refer to a message consumer component which has subscribed to receive all the messages from topic destination. The MOM system is responsible to maintain the list of subscribers on the server.

Figure 6.4 shows the graphical representation of how different components communicate in a publish – subscribe model.

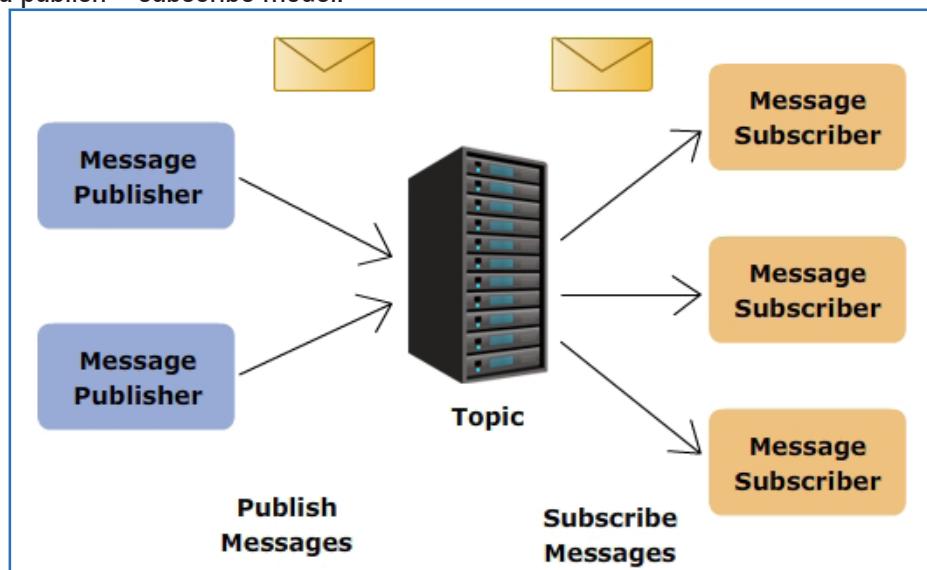


Figure 6.4: Publish-Subscribe Model of Messaging

There are different types of subscriptions; the subscriptions can be sharable or non-sharable subscriptions. A sharable subscription implies that the message can be received by a set of consumers who have agreed to share the subscription, whereas non-sharable subscription implies that only one subscriber can receive a message through subscription.

Subscriptions can also be classified as durable or non-durable. In case of non-durable subscriptions the messages are stored on the topic only when there is some client to receive the message. Once all the subscribed clients receive the message it is removed from the topic. In case of durable subscriptions the messages are retained by the topic for a certain time period according to the configuration of the application.

This messaging model is also referred to as push-based model and is used when the message has to be multicasted.

Point-to-Point Model

In Point-to-point messaging model, the communication is between a pair of components. The destination in this case is a queue object.

In this type of messaging style, you can have only a single consumer for each message. A consumer can grab the message in the queue, but a given message is consumed only once by a consumer. Multiple producers can send messages to a queue, but only a single consumer can consume a message. Messages are sent by the producers to a centralized queue and are distributed as First In First Out (FIFO). It is a pull or polling based model as messages are requested from a queue and not pushed to the client automatically.

The receivers access the queue to retrieve the messages. The message in the queue is retained until the consumer consumes the message.

Figure 6.5 graphically demonstrates the point-to-point messaging model.



Figure 6.5: Point-to-Point Messaging Model

Figure 6.5 shows point-to-point messaging model supported by JMS. The message consumer receives the message from the queue and acknowledges the queue after the message has been successfully consumed. The message consumers can also configure a message listener to listen to the message queue and invoke appropriate methods when a message is received on the queue.

The message consumption is by default asynchronous in both the messaging model as the message consumers do not actively wait on the destination objects.

6.3.2 Use of JMS API in EJB

JMS API is an integral part of Java Enterprise Edition. It was first part of J2EE 1.2 as JMS 1.0. Since then it has evolved to JMS 2.0 which is the most recent version in Java EE 7.

Following are the features offered by JMS API:

- Various components such as application clients, enterprise beans, and Web components can send messages and asynchronously receive JMS messages.
- The API allows the application components to set up a message listener for processing the JMS messages. Message listeners are objects which respond to messages received and invoke appropriate components on receiving a message.
- Java EE also provides Message-driven beans which are enterprise components. These Message-driven beans are used to process JMS messages.
- Containers can maintain a pool of message driven beans to enable concurrent processing of messages. This pool of Message-driven beans facilitates different JMS messages simultaneously.
- JMS messages can also be part of a Java transaction, which means that a Java transaction can send or receive a message.

6.3.3 Messaging Application Architecture

A typical JMS API application consists of the following parts:

1. JMS provider
2. JMS clients
3. Administered objects
4. Messages

The JMS provider is responsible for actual routing of messages between the components. It is also responsible for the delivery of messages at the destination. It implements the interfaces provided by the API. The JMS provider is also responsible for managing the messaging infrastructure of the enterprise application.

JMS clients are those application components which use the services provided by the JMS provider and exchange messages among themselves.

Administered objects refer to those entities which serve as communication infrastructure for the JMS application or JMS module of the application. JMS connection objects, sessions and destinations are administered objects. These objects are instantiated according to the requirement of the application which happens when the application components want to communicate among themselves.

Messages are objects which are exchanged during the communication process.

Figure 6.6 shows how different objects in the JMS application mutually interact with each other.

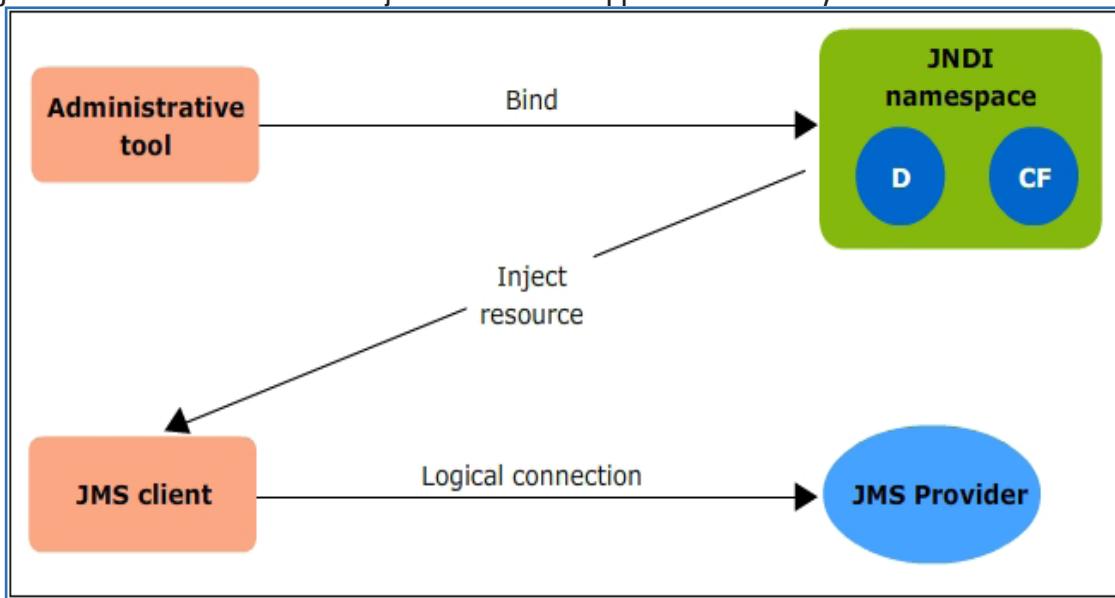


Figure 6.6: JMS Application

Figure 6.6 shows a JMS client injecting the resources required for communication. In this case it obtains a reference of a destination object or ConnectionFactory object. The objects are instantiated and bound to their respective JNDI names. The JMS client takes the service of the JMS provider to communicate with the destination.

6.4

JMS API Programming Model

Following are the objects which are a part of a JMS application:

- Administered objects are objects such as connection factories and destinations. These objects are created on Java EE server such as GlassFish server or other servers administratively on which the application is hosted.
- Connections are objects which logically connect two communicating entities.
- Session refers to a series of message exchanges between two communicating entities in a given context.
- JMSContext object is a communicating environment between the message sender and receiver with instances of connection and session objects.
- JMS Message Producers are entities which in certain context produce JMS messages.
- JMS Message Consumers are entities which receive JMS messages.
- Messages are the entities exchanged between the message producer and consumer.

Figure 6.7 shows the JMS API programming model and how different objects interact with each other.

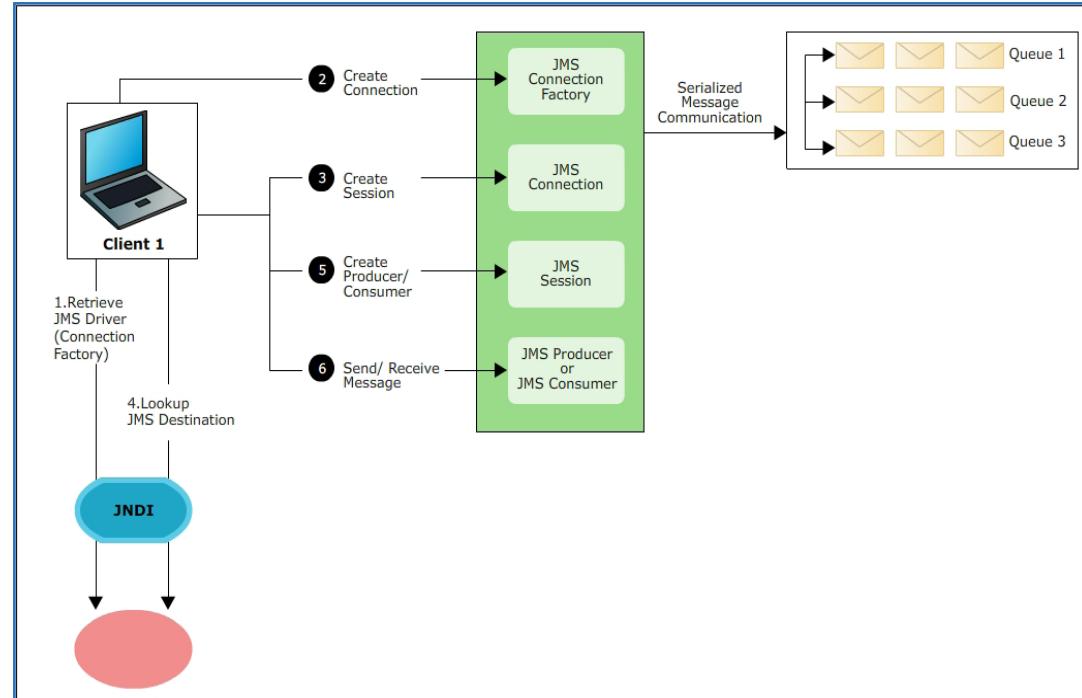


Figure 6.7: JMS API Programming Model

Figure 6.7 graphically represents the communication among different components of the JMS application. The `ConnectionFactory` instantiates a connection object to virtually connect two communicating entities. A `Session` object is also created which represents specific instance of communication between the message consumer and message producer. The connection object and the session object together form the `JMSCContext` which creates the environment for communication. The `JMSCContext` also creates a reference to the message producer and message consumer entities in the application.

The `session` object has message objects associated with the `MessageConsumer` and `MessageProducer` objects. The `MessageProducer` object writes messages to the destination and the `MessageConsumer` object receives messages from the destination.

JMSConnectionFactories are referred to as administered objects according to the JMS API programming model. The developer can obtain a connection instance from the `ConnectionFactory`. This connection object will further become part of the `JMSCContext`. A connection factory object can be an instance of `ConnectionFactory`, `QueueConnectionFactory` or `TopicConnectionFactory` interface.

The `JMSClient` object injects a connection factory object into the application. The Java EE server provides a JNDI name to access the `ConnectionFactory` object such as `java:comp/DefaultConnectionFactory`. The developer can provide actual JNDI name for the object.

JMS Destination is also an administered object in a JMS application. According to the messaging models supported by JMS, the destination objects can either be a queue or topic. The type of the destination object created is chosen as per the requirement of the application.

Administered objects can be created on the application server such as GlassFish through administrative tools provided by the server or IDE.

JMSContext objects provide an application execution environment to JMS applications. It combines a connection object and a session object to provide context for the message producer and message consumer to communicate.

A `JMSContext` can be created from a `ConnectionFactory` instance as shown:

```
JMSContext J = connectionFactory.createContext();
```

A `JMSContext` object can be used to create other objects of the applications. When there is a `JMSContext` in the application it has a `Connection` and `Session` instance associated with it, these can be further associated with other objects of the application. They are message producers, message consumers, messages, queue browsers, temporary and destination objects such as temporary queues and topics.

When a `JMSContext` object is created the mode of message acknowledgement can also be defined. The default message acknowledgement mode is `AUTO_ACKNOWLEDGE`, where the consumer sends an acknowledgement as soon as the message is received. Other acknowledgement modes are `CLIENT_ACKNOWLEDGE` where the client acknowledges the message received by invoking an `acknowledge()` method and `DUPS_OF_ACKNOWLEDGE` where a delayed acknowledgement is sent by the receiver.

JMS Message Producer objects are created based on the `JMSContext` of the application. They can be created by invoking `createProducer()` method of the `JMSContext`.

A JMS message producer can be created as follows:

```
...
JMSContext J = connectionFactory.createContext();
JMSProducer JP = J.createProducer();
...
```

JMS Message Consumers are created in a similar way as the message producers along with the `JMSContext` instance. While creating the message consumer, the destination from which the message is produced should access the messages that is specified as a parameter.

A JMS message consumer can be created as follows:

```
...
JMSContext J = connectionFactory.createContext();
JMSConsumer JC = J.createConsumer(destination);
...
...
```

Once the destination is configured, the consumer receives the messages from the destination object. This behavior can be disabled by invoking the method `setAutoStart(false)`. After setting the auto start behavior to false to receive a message `start()` method should be invoked by the consumer object and then `receive()` method must be explicitly called to receive the messages. JMS consumers can define message listeners on the destination objects.

Messages are the objects of communication among JMS application components. Every message has a standard format. There are three parts of a message: message header, properties, and body.

Message header holds all the control information used by the JMS provider and container to route the message from the message producer to consumer properly. Message header has information such as a unique identifier to identify the message, message destination, and so on. Most of the header fields are set through `send()` method of the message producer.

Table 6.1 describes the different message header fields.

Header field	Description
JMSDestination	This field defines the destination where the message is destined to. This is set by the <code>send()</code> method of message producer.
JMSDeliveryMode	This field defines the mode of message delivery, which is the method of persisting the message. This value is set by the <code>send()</code> method.
JMSDeliveryTime	This field of the message defines the time when the message sending process is initiated and also the delay time tolerable. This field is also set by the <code>send</code> method.
JMSExpiration	This field defines the maximum time duration for which the current message is valid. This value is set by the <code>send()</code> method.
JMSPriority	This field defines the priority of the message and is set by the <code>send()</code> method of the message producer.
JMSMessageID	It is a unique identifier to identify the message and is set by the <code>send()</code> method.
JMSTimestamp	This field specifies the time when the message was handed over to the JMS provider and is set by the <code>send()</code> method.
JMSCorrelationID	This field is set by the client and is used to link one message to the following message in a session.
JMSReplyTo	This field specifies the destination where the reply to the current message has to be sent.
JMSType	JMS messages can have data of different types. This field defines the data type of the message content and is set by the client.
JMSRedelivered	This field is set when the acknowledgement for the message is not received and as a result the message is redelivered.

Table 6.1: JMS Message Headers

If the application requires few other properties in addition to those defined by the message headers, the developers can define these properties and values.

Message body is the actual content of the message which is supposed to be transmitted between the sender and the receiver.

JMS supports the following types of data to be part of a message body.

- Text Message** – This comprises data as Java strings.
- Map Message** – The data has map objects, a map object is a key-value pair. The entities of a map object can be sequentially accessed through enumerator.
- Bytes Message** – A stream of un-interpreted bytes which match some data format such as JPEG and so on.
- Stream Message** – A stream of values which are of primitive types.
- Object Message** - A serializable object.
- Message** – Empty message body is also a valid message in JMS. This is used for administrative tasks.

JMS API has methods for creating and processing the messages of the mentioned message types. At the consumer end the message is received as a generic message object, which is later type casted onto appropriate data type.

6.5 Implementing JMS

When a JMS application is developed, the developer has to create JMS resources on the server for the application.

Following are the steps for creating a JMS resource on the server. Applications may require multiple JMS resources on the server, steps can be repeated for each resource created:

1. In order to add a JMS resource to an application, right-click the project to which you have to add the JMS resource and select '**Other**' from the menu. This will lead to the wizard as shown in figure 6.8.

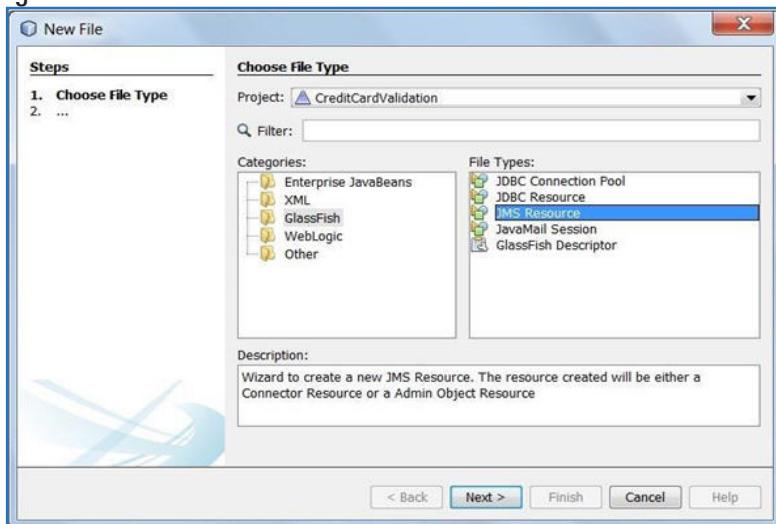


Figure 6.8: Creating a JMS Resource

2. In the wizard shown in figure 6.8. Select '**JMSResource**' from the menu.
3. In the following screen provide the JNDI name for the resource as '**jms/myQueue**' and choose the type of resource to be created as shown in figure 6.9.

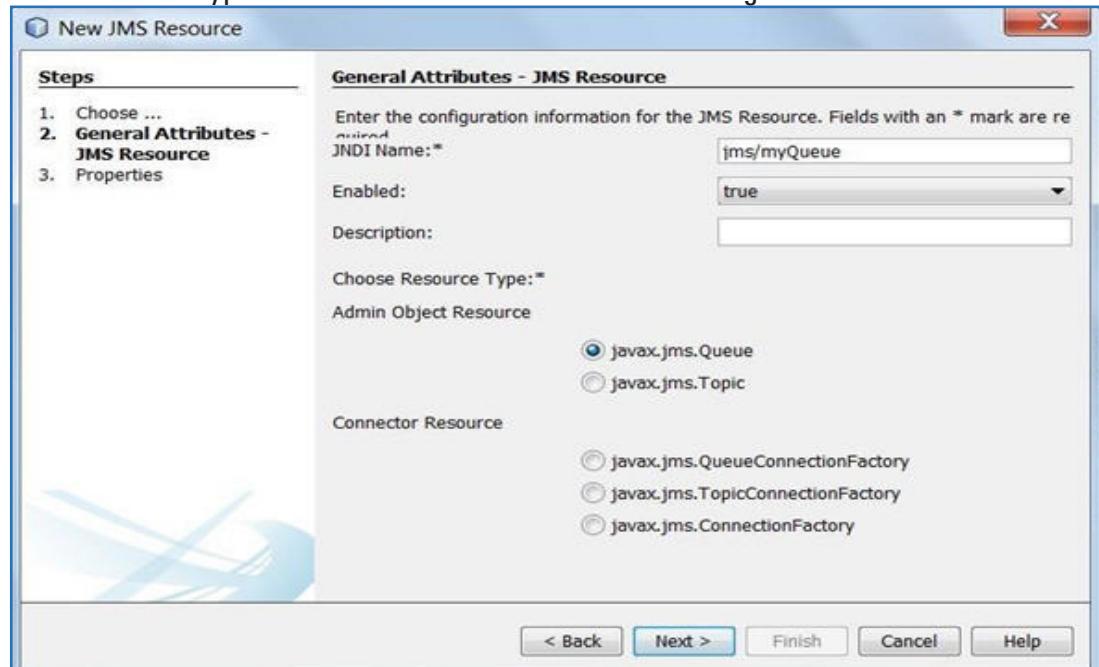


Figure 6.9: Selecting the JNDI Name and Type of Resource

4. Provide additional configuration information in the following screen, if required. Click '**Finish**' after adding the configuration process.

6.5.1 JMS Resources in an Application

A JMS application/module has a message producer which generates the messages and a message consumer which reads the messages generated by the message producer. As JMS supports asynchronous messages, the messages are not directly sent to the message consumer. Instead they are stored in a JMS resource such as Queue or Topic on the application server.

The process of creating a message producer and message consumer are as follows:

1. Create a message producer application. The message producer enterprise application sends messages to the message destination on the server.
2. Create the '**JMS Resource**' on the GlassFish server. When the message producer application sends messages, the messages are stored on these JMS resources located on the server. The JMS resources created on the server can either be a `JMSQueue` object or `JMSTopic` object.

Figure 6.10 demonstrates creation of a JMS resource on the server.

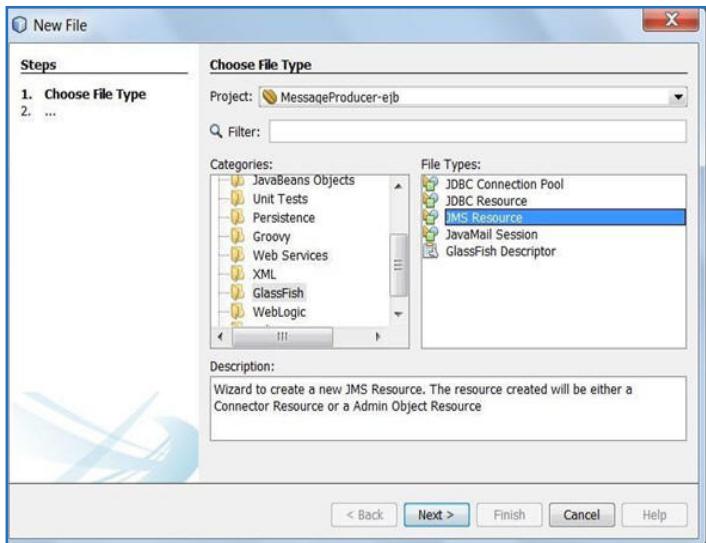


Figure 6.10: Creating a JMS Resource

The JMS resource can accept messages from the message producer application. Here a `Queue` object is being created as the destination object. A `Connector` object is required to connect the message producer with the `Queue` on the server.

3. Add a '`Queue`' object and a '`Connector`' object one after the other through the wizard used to add JMS resource to the server GlassFish server as shown in figure 6.11.

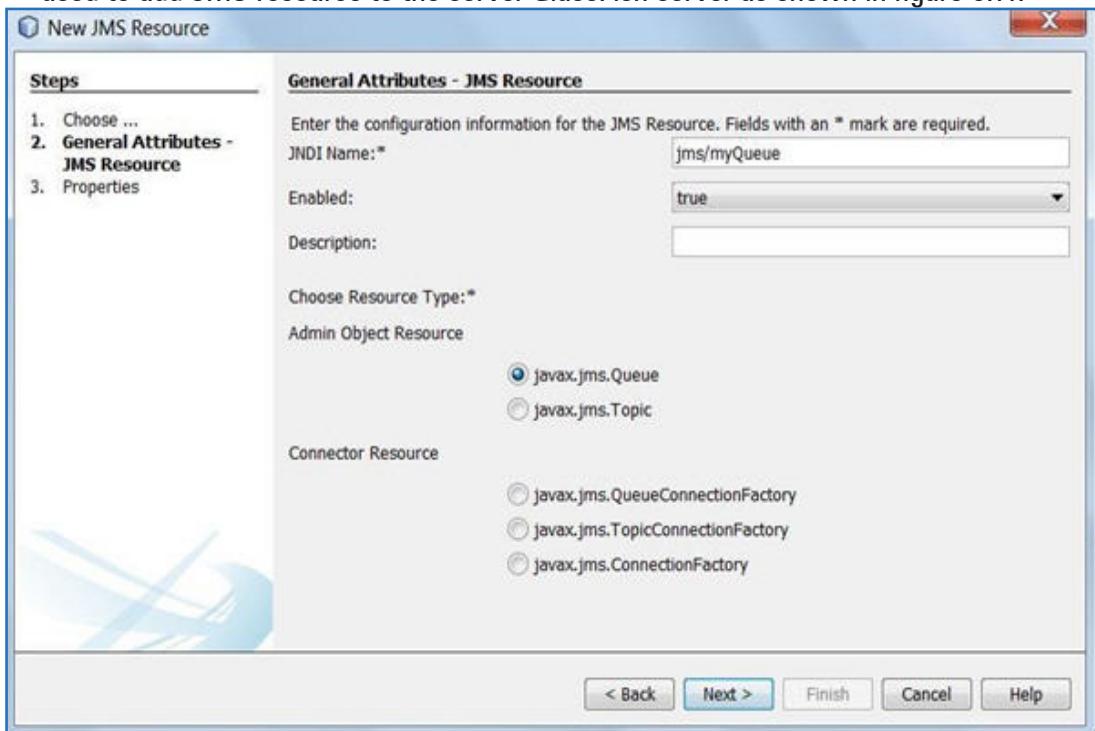


Figure 6.11: Creating a JMSQueue

4. Click 'Next'. On the next screen, provide 'myQueue' in the Value field of the Name property as shown in figure 6.12 and click 'Finish'.

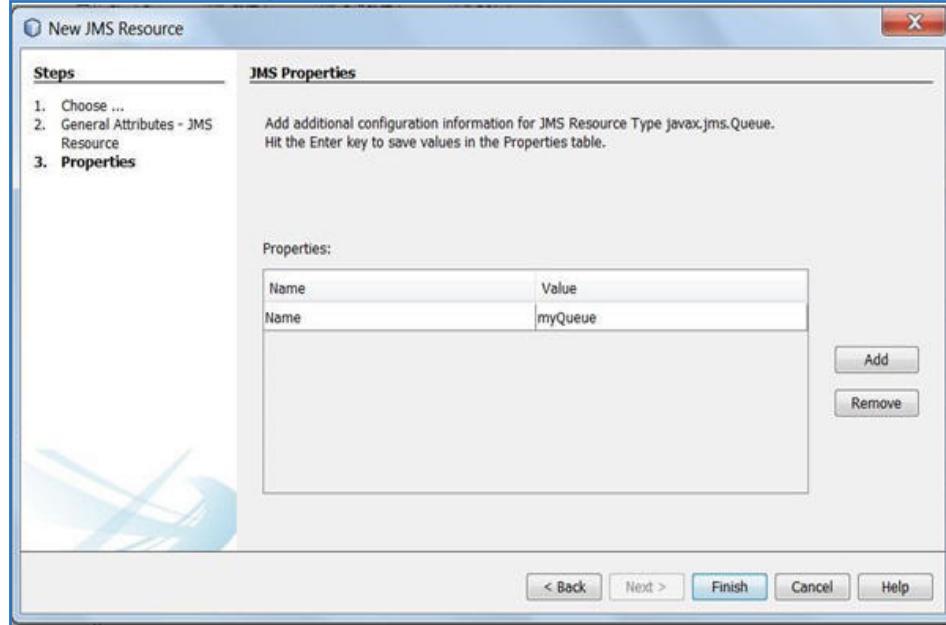


Figure 6.12: Adding Information to the Server

The information about `myQueue` is added to the `glassfish-resources.xml` file in the Server Resources folder.

Create a `ConnectionFactory` resource. The `ConnectionFactory` resource is required to make connections between the message producer module and the `Queue` object on the server.

5. Right-click the project and select **New → Other → GlassFish → JMS Resource** from the New File dialog box.
6. Click 'Next' and specify the options as shown in figure 6.13 to create a '`QueueConnectionFactory`' object.

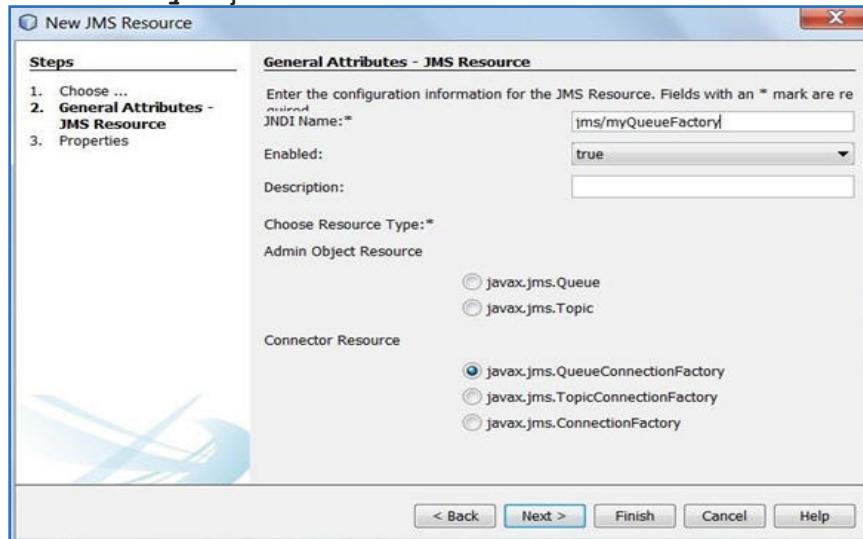


Figure 6.13: Creating a QueueConnectionFactory Object on the Server

The choice of messaging model to be used, that is, whether the system uses a point-to-point model or publish-subscribe model is defined by selecting the appropriate JMS resources at this stage.

7. Specify the JNDI Name as '`jms/myQueueFactory`' and select the Connector Resource as '`javax.jms.QueueConnectionFactory`'.
8. Click 'Next'. The JMS Properties screen is displayed. Here, you can specify additional configuration information.
9. Click 'Finish'. The information about `myQueueFactory` is added to the `glassfish-resources.xml` file in the Server Resources folder.

6.5.2 Checking JMS Resources on GlassFish Server

To check whether the JMS resources are added to the server:

1. Select the '**Services**' tab in the IDE.
2. Expand the '**Resources**' folder of the GlassFish server and then, expand the '**Connectors**' folder.
3. Right-click '**Admin Object Resources**' and select '**Refresh**'.
4. Right-click the '**Connector Resources**' and '**Connector Connection Pools**' folders and select '**Refresh**'.

The Admin Object Resource, '`jms/myQueue`', and a Connector Resource object, '`jms/myQueueFactory`' will appear in the respective folders as shown in figure 6.14.

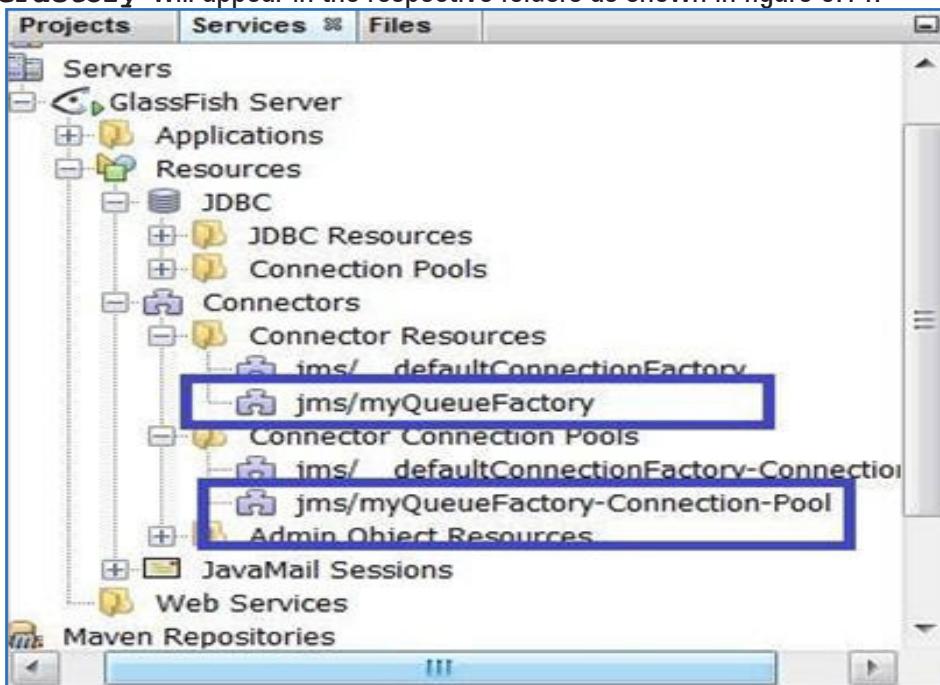


Figure 6.14: GlassFish Server with JMS Resource Objects

6.5.3 Creating Bean Objects in MessageProducer Application

You can create Message-driven beans in a way similar to creating session beans. To create a JSF ManagedBean that will act as a message producer:

1. Right-click the project name and select **New → Other → JavaServer Faces → JSF ManagedBean** from the New File dialog box.
2. Click '**Next**'. The New JSF ManagedBean dialog box displays on screen.
3. Specify the class name of the bean as '**MessageProducerBean**' and click '**Finish**'. The ManagedBean is created.

Add a property named '**message**' to the bean and create the respective getter and setter methods. The resultant code in the JSF ManagedBean is depicted in Code Snippet 1.

Code Snippet 1:

```
package mes;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;

@ManagedBean
@RequestScoped

public class MessageProducerBean {

    /**
     * Creates a new instance of MessageProducerBean
     */

    private String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public MessageProducerBean() {
    }
}
```

In Code Snippet 1, a JSF ManagedBean is created to which the attribute message has been added. The getter and setter methods for this property have also been generated.

6.5.4 Adding JMS Code to the ManagedBean

1. In the editor area of 'MessageProducerBean', press 'Alt+Insert' to open the NetBeans Code Generator feature and select 'Send JMS Message' option.
2. The Send JMS Message dialog box displays on screen. Ensure that Project Destination is set to 'jms/myQueue' and Connection Factory is set to 'jms/myQueueFactory'.
3. Click 'OK'. NetBeans adds the proper resource declarations to the code for the Queue and ConnectionFactory instances as depicted in Code Snippet 2.

Code Snippet 2:

```
@ManagedBean  
 @RequestScoped  
  
public class MessageProducerBean {  
  
    @Resource (mappedName = "jms/myQueue")  
    private Queue myQueue;  
  
    @Inject  
    @JMSConnectionFactory ("jms/myQueueFactory")  
    private JMSContext context;  
  
    /**  
     * Creates a new instance of MessageProducerBean  
     */  
  
    private String message;  
  
    public String getMessage () {  
        return message;  
    }  
  
    public void setMessage (String message) {  
        this.message = message;  
    }  
  
    public MessageProducerBean () {  
    }  
}
```

To send a JMS message, the message destination and connection to the message destination need to be configured. In Code Snippet 2, a `Queue` object has been added which is meant to be the destination of the message sent from the session bean. A `JMSConnectionFactory` object is also added as a resource which enables the message sender to connect to the destination on the application server. When the session bean intends to send a message to the destination object it establishes a connection with the server and sends the message onto the destination queue on the application server.

6.5.5 Creating `send()` Method in the MessageProducer Application

Once the JMS resource is configured on the server, add a new `send()` method to the `MessageProducerBean`. This method sends messages to the message destination object on the server.

Following are the steps for adding a new `send()` method:

1. Invoke the initial context of the application.
2. Acquire the current instance of `FacesContext`, to retrieve the user interface state of the application.
3. Use the admin objects created on the Web application server (glassfish) to bind the connection and queue objects onto jndi names.
4. Create JMS objects – `Connection`, `Session`, and `MessageProducer`.
5. Send the message received from the JSF page to the queue.

Code Snippet 3 depicts the `send()` method to be added to the `MessageProducerBean`.

Code Snippet 3:

```
...
public void send() throws NullPointerException,
NamingException, JMSEException {
    InitialContext initContext = new InitialContext();
    FacesContext facesContext = FacesContext.
    getCurrentInstance();
    ConnectionFactory factory = (ConnectionFactory) initContext.
    lookup("jms/myQueueFactory");
    Destination destination = (Destination) initContext.
    lookup("jms/myQueue");
    initContext.close();
```

```
//Create JMS objects  
  
Connection connection = factory.createConnection();  
  
Session session = connection.createSession(false, Session.  
AUTO_ACKNOWLEDGE);  
  
MessageProducer sender = session.createProducer(myQueue);  
  
  
//Send messages  
  
TextMessage msg = session.createTextMessage(message);  
sender.send(msg);  
  
FacesMessage facesMessage = new FacesMessage("Message sent: " +  
message);  
  
facesMessage.setSeverity(FacesMessage.SEVERITY_INFO);  
facesContext.addMessage(null, facesMessage);  
}
```

In the `send()` method, first lookup of the Queue and Connection objects on the server is performed. Once these objects are accessed, their instances are created to send a message to the Queue which is configured on the server. These message objects reside on the server. They can be accessed by a consumer application.

6.5.6 Creating a `receive()` Method

To receive a message from the destination, create a `MessageConsumer` application with a managed bean. The managed bean would have a `receive()` method as shown in Code Snippet 4.

Code Snippet 4:

```
public class ReceiverBean implements javax.ejb.SessionBean  
{  
  
InitialContext jndiContext;  
  
public String receiveMessage() {  
try{  
  
QueueConnectionFactory f = (QueueConnectionFactory) jndiContext.  
lookup("java:comp/env/jms/myQueueFactory");  
  
jndiContext.lookup("java:comp/env/jms/myQueue");  
}
```

```
QueueConnection connect = factory.createQueueConnection();
QueueSession session = connect.createQueueSession(true, 0);

QueueReceiver rec = session.createReceiver(myqueue);
TextMessage textMsg = (TextMessage) rec.receive();
connect.close();

return textMsg.getText();
}

catch (Exception e) {
throws new EJBException(e);

}

}
```

6.6 Introduction to Message-Driven Bean

Message-driven bean is an enterprise bean used to process asynchronous messages received from application components. A Message-driven bean is an EJB component that receives JMS message and other messages. It is a stateless, server-side component that is used for processing asynchronous messages delivered using JMS. The Message-driven bean is invoked by the container, and is responsible for processing messages. The container is responsible for managing the component's environment such as transactions, security, resources, concurrency, and message acknowledgement.

A Message-driven bean can process hundreds of JMS messages concurrently because numerous instances of the Message-driven bean can execute at the same time in the container.

A Message-driven bean does not have a remote or local business interface. Hence, a client cannot access a Message-driven bean through a business interface. A client can only interact with a Message-driven bean through the messaging system.

Every message driven bean contains an `onMessage()` method which has all the operations to be carried out on receiving a message.

A Message-driven bean has the following characteristics:

- A Message-driven bean is executed only upon receiving a message.
- They are short lived and are invoked asynchronously.
- Message-driven beans can access and update database, but do not represent the data in the database.
- Message-driven beans are stateless and transaction aware.

6.6.1 Lifecycle of a Message-Driven Bean

Lifecycle of a Message-driven bean is similar to that of a stateless session bean. There are only two states in the lifecycle of a Message-driven bean.

Figure 6.15 shows the lifecycle of a Message-driven bean.

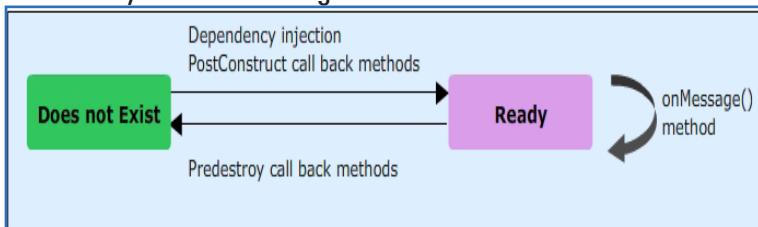


Figure 6.15: Lifecycle of a Message-Driven Bean

Figure 6.15 demonstrates how the Message-driven bean transits from a Does not exist state to a Ready state when instantiated by the container. Container handles Message-driven beans in a similar manner as it handles stateless session beans. Once instantiated, required resources are injected into the bean and the life cycle callback methods are also invoked. The bean transforms from a Does not exist state to Ready state.

A bean in Ready state is invoked by the messages received from clients. On receiving a message the `onMessage()` method of the Message-driven bean is invoked to perform the required operation. After handling the received message, the Message-driven bean is in Ready state and can handle client messages.

6.6.2 Lifecycle Event Handlers

When the Message-driven bean transits from one state of the lifecycle onto another state certain operations are required to be performed on the beans. This set of operations can also be referred to as lifecycle event handlers or lifecycle callback methods.

In case of message-driven beans there can be two variants of lifecycle event handlers – PostConstruct and PreDestroy event handlers. These event handlers are marked with `@PostConstruct` and `@PreDestroy` annotations.

PostConstruct event handlers are invoked after the bean object is instantiated by the container along with the required dependency injection.

PreDestroy event handlers are invoked before the bean object is removed from the container.

6.6.3 Comparing Session Beans with Message-Driven Beans

- Session beans can be invoked directly through the interface provided by the developer, whereas Message-driven beans cannot be invoked.
- Message-driven beans are similar to stateless session beans as they do not maintain any conversational state of the application.
- Message-driven beans are asynchronously invoked by the messages.

- Message-driven beans like stateless session beans have only two states in their life cycle. They cannot be passivated like Stateful Session beans.
- Message-driven beans like stateless session beans provide scalability in handling multiple clients.

6.6.4 Creating and Configuring Message-Driven Beans

A Message-driven bean is created in the application to respond to the asynchronous messages received by the application. A Message-driven bean instance must implement the `javax.jms.MessageListener` interface. In earlier versions, a Message-driven bean implemented the `javax.ejb.MessageDrivenBean` interface. However, in EJB 3.0, it is not compulsory to implement the `MessageDrivenBean` interface. A JMS Message-driven bean implementation class must define a default constructor without any argument.

Message listeners asynchronously receive messages from the destinations. A message listener class implements `MessageListener` interface and defines the actions to be performed on receiving a message in `onMessage()` method. The `onMessage()` method accepts a `Message` object as a parameter and is invoked on receiving a message of parameter type.

A message listener object can be defined on a consumer using `setMessageListener()` method as follows:

```
....  
JMSContext J = connectionFactory.createContext();  
JMSConsumer JC = J.createConsumer(destination);  
Listener L = new Listener();  
JC.setMessageListener(L);  
....
```

Message-driven beans like session beans are created in Netbeans IDE.

- Right-click the project in which you have to create the Message-driven bean and select the options as shown in figure 6.16.

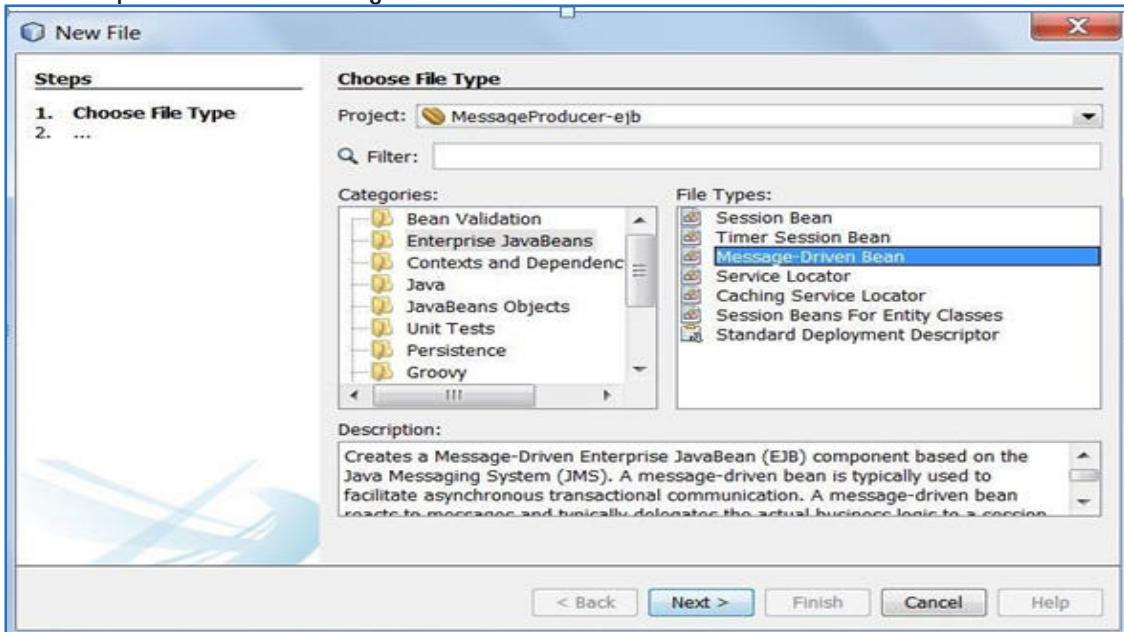


Figure 6.16: Creating Message-Driven Bean

- The Message-driven bean has to respond to messages received on a destination object present on the application server. When you create a Message-driven bean the IDE searches for available destinations on the server and retrieves them for the developer to select. Select the name of the Message-driven bean and configure the destination of the messages, as shown in figure 6.17.

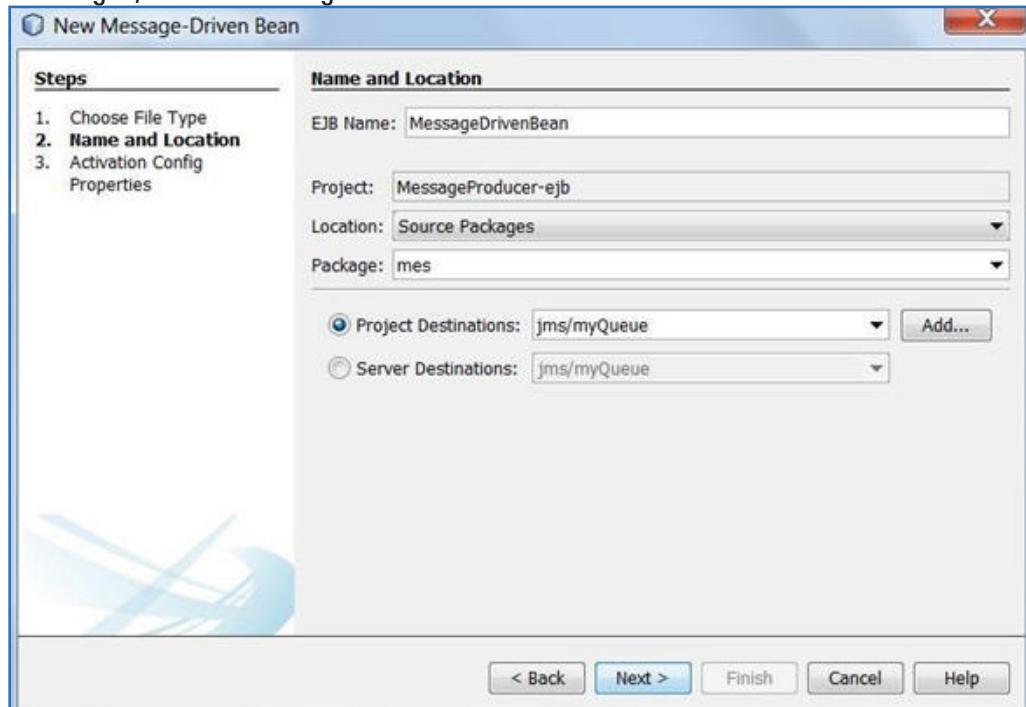


Figure 6.17: Configuring the Message-Driven Bean

The Message-driven bean is activated when a message is received on the configured destination. In figure 6.17, the Message-driven bean is configured to receive messages from the 'jms/myQueue' message queue created on the GlassFish server.

3. After selecting the message destination object on the application server, the Message-driven bean is configured as shown in figure 6.18.

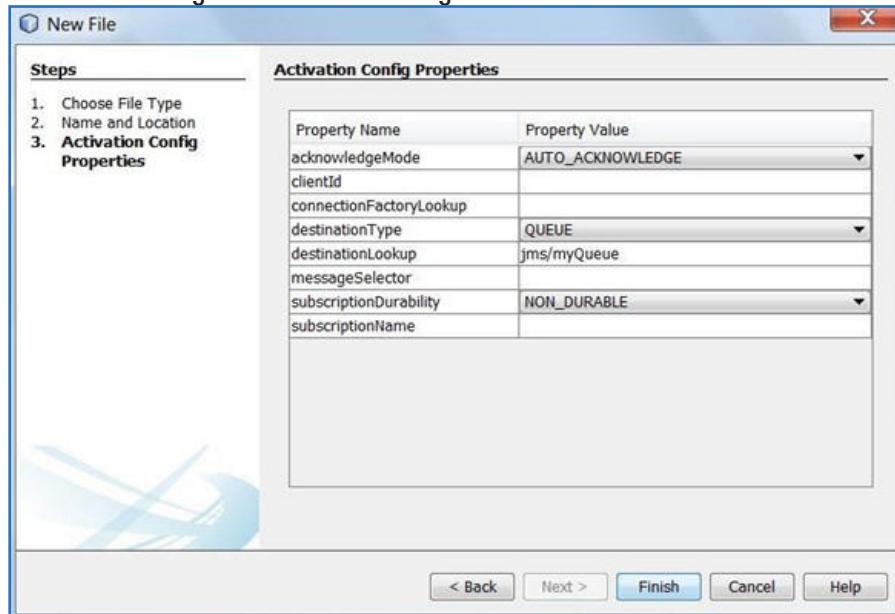


Figure 6.18: Configuring Message-Driven Bean Properties

In figure 6.18, the Message-driven bean is configured by defining the destination type, the JNDI name for which the destination is configured and the acknowledgement mode for the message received.

4. Message-driven bean is created using the code shown in Code Snippet 5. The `onMessage()` method is invoked in response to the message received at the message destination object on the server. Developer can add code to the `onMessage()` method to define the application behavior on receiving a message.

Code Snippet 5:

```
...
import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.Message;
import javax.jms.MessageListener;
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationType",
    propertyValue = "javax.jms.Queue"),
    @ActivationConfigProperty(propertyName = "destinationLookup",
    propertyValue = "jms/myQueue")
})
```

```
public class MessageDrivenBean implements MessageListener {
    public MessageDrivenBean() {
    }
    @Override
    public void onMessage(Message message) {
    }
}
```

The developer can also define the operations to be performed on receiving an asynchronous message in the `onMessage()` method.

Check Your Progress

1. _____ method is used by the message-driven bean to handle the messages received.

(A)	receive()	(C)	send()
(B)	onMessage()	(D)	None of these

2. Which of the following is not true about Message-driven beans?

(A)	Message-driven beans are asynchronously invoked	(C)	Every Message-driven bean has an onMessage() method
(B)	Message-driven beans can be explicitly invoked by stateless session beans	(D)	None of these

3. Which of the following destination objects are used when the application is using point-to-point messaging model?

(A)	Topic	(C)	JMSContext
(B)	Queue	(D)	Any of these

4. JMSContext comprises:

(A)	Connection	(C)	Both a and b
(B)	Session	(D)	ConnectionFactory

5. Which of the following are administered objects?

(A)	Connection	(C)	Destination
(B)	Session	(D)	None of these

Check Your Progress

6. Which of the following is true about JMS Resources?

(A)	JMS resources are created by the application during runtime	(C)	ConnectionFactory objects are JMS resources
(B)	Queue and Topic objects are not JMS resources	(D)	All of these

Answer

1.	B
2.	B
3.	B
4.	C
5.	C
6.	C

Summary

- JMS API can be used to create communication modules of enterprise applications. Apart from being part of an enterprise application, independent JMS applications can also be created.
- JMS primarily communicates through two messaging models, Publish-Subscribe model and Point-to-point model.
- The prime focus of JMS API is to provide asynchronous communication.
- Communicating components can define message listeners on the message destinations.
- JMS API Programming model defines how different Java classes are used to implement the JMS application.
- Message-driven beans are defined in Java EE that are invoked on receiving a message on the message destination.



To enhance your knowledge,

visit **REFERENCES**



www.onlinevarsity.com



Welcome to the Session, **Interceptors and Dependency Injection**.

This session discusses the concept of aspect-oriented programming used to separate concerns from the business code. The session explains how to use interceptors in Java EE applications. It discusses the purpose of interceptors and the different implementations of the interceptors in EJBs. Further, the session also discusses Context and Dependency Injection (CDI) with respect to the resources injected in the enterprise applications.

In this Session, you will learn to:

- Explain aspect-oriented programming
- List the advantages of aspect-oriented programming
- Explain interceptors and their usage in enterprise applications
- Describe the different types of interceptor methods
- Explain how to define the interceptors and their levels of description
- Explain the creation of interceptor class
- Explain how to implement business method and lifecycle callback method interceptors
- Describe various aspects of context dependency and Injection
- Describe how to use CDI in enterprise applications

7.1

Introduction to Aspect-oriented Programming

Every business logic has some dependencies on its execution. For example, the parameters passed to a business method must be validated, before they are processed further by the method.

In a software architecture design, the services provided to the business logics for their successful execution are termed as concerns. The concerns are interrelated to the business logics and are also referred to as business logic concerns. For example, consider a banking application that has a business method, which is performing transferring of an amount from one account to another.

Figure 7.1 shows the concerns incorporated in the banking application.

```

void transfer(Account fromAcc, Account toAcc, int amount, User user,
Logger logger) throws Exception {
    logger.info("Transferring money...");
    transaction.begin();

    if (!isUserAuthorised(user, fromAcc)) {
        logger.info("User has no permission.");
        throw new UnauthorisedUserException();
    }

    if (fromAcc.getBalance() < amount) {
        logger.info("Insufficient funds.");
        throw new InsufficientFundsException();
    }

    fromAcc.withdraw(amount);
    toAcc.deposit(amount);

    transaction.commit();
    logger.info("Transaction successful.");
    transaction.close();
}

```

The diagram illustrates the separation of concerns in the code. It shows a Java method `transfer` with several logging statements and exception handling. Three specific sections of code are highlighted with red boxes and labeled as "Concerns": 1) Logging the start of the transfer. 2) Checking if the user is authorized. 3) Checking if there are sufficient funds. Arrows point from these three concern blocks to a single label "Concerns" located on the right side of the diagram.

Figure 7.1: Encapsulation of Business Logic and Concerns

Now, consider if you have to incorporate a new concern such as security which would be span across multiple statements within the code, then, the business logic will be affected with the changes.

To handle these types of concerns separately from the business logic, modern languages and frameworks provide support for Aspect-oriented Programming (AOP). AOP allows the code developer to express cross-cutting concerns into stand-alone modules called aspects. The aspects intercept the request invocation executed for the object. Then, the aspects are injected into the functionality and finally, the request is delegated to the targeted object. Some of the common aspects include logging, transaction, security, connection, and so on.

Figure 7.2 shows the aspects injecting into various business codes of the enterprise application.

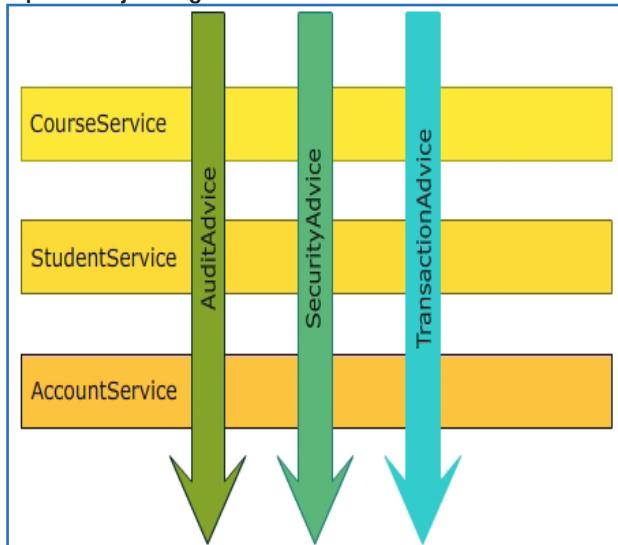


Figure 7.2: Injecting Aspects into the Business Processes

7.1.1 Advantages of Aspect-oriented Programming

Some of the advantages of Aspect-oriented programming are as follows:

- **Reusability** – By modularizing the code, it can be injected into multiple objects. This avoids redundancy of implementations.
- **Separation of Concern** – Separation of concerns from the business logic makes the implementation independent of the concerns. Thus, the developer can modify the aspect without having to change the implementation of the business processes.
- **Focus** – Developers can concentrate on aspects separately without merging them with the business logics. Also, aspects can be developed by different developers.

On Java EE platform, EJB 3 supports AOP framework through Interceptors. Interceptors are used to enable inclusion of aspects within the EJBs. They provide the capability to intercept business methods and lifecycle callback methods.

7.2

Interceptors

Interceptors allow the developers to interpose EJB business method. This means they allow the developer to add a wrapper code which is executed before or after the method is called. Interceptors are used to perform tasks such as logging information of bean method execution, auditing, and so on. Interceptors provide a fine grained control to the bean developer on the lifecycle methods of a bean.

Interceptors are used in association with enterprise bean classes to allow the invocation of interceptor methods on the associated target class. On Java EE platform, Interceptors may be used with Session beans or Message-driven beans.

Following are some of the scenarios in which interceptors can be executed:

- Method parameters are to be validated, before they are passed to the bean method.
- Security checks are required to be performed, before executing the bean method.
- Logging or profiling operations are performed, during the bean method execution.
- Certain operations required to be performed, before or after class instantiation in the application.

They can be defined through metadata annotations or in the deployment descriptor containing the elements to define the interceptors and target classes.

7.2.1 Types of Interceptors

The different types of interceptors are as follows:

- Business Method Interceptors**

Business method interceptors are also referred to as 'AroundInvoke' methods. They can intercept the invocations of the business methods of the session bean. They can also intercept the listener methods of the Message-driven bean.

The `javax.interceptor.AroundInvoke` annotation is used to designate the business method as an interceptor method.

- Lifecycle Callback Interceptors**

Lifecycle callback methods intercepts the lifecycle methods of the session bean or Message-driven bean instances.

- Timeout Callback Interceptors**

Timeout callback methods may be defined for EJB timer services provided for session bean and Message-driven bean. These methods are also referred to as 'AroundTimeout' methods.

7.2.2 Defining Interceptors

Developers can define the interceptors for a target class in one of the following two ways:

- Developer can define an interceptor method within a target class. The target class is the bean class. The methods of the target class which are supposed to be intercepted are annotated with `@javax.Interceptor.Interceptors` annotation. The bean can either impose interceptor class on all methods or impose interceptor class on explicit methods.
- The interceptor can also be defined in a separate interceptor class. The interceptor class contain methods which can be invoked along with the lifecycle callback methods of the target class or business methods of the target class. There can be only one Interceptor class defined for a particular target class.

When an interceptor is defined on a class or a method, the developer has to define when the interceptor has to be invoked by the application. An interceptor can be invoked at various instances such as before the target method invocation, after completing the execution of the target method, before instantiating the target class, and so on.

The time of interceptor action can also be defined through annotations.

7.2.3 Levels of Interceptors

Interceptors can be defined at different levels in an enterprise applications. These are as follows:

- Default interceptors
- Class-level interceptors
- Method-level interceptors

Default interceptors

These interceptors are declared in the deployment descriptors, that is, `ejb-jar.xml` of the application. They are applicable across all the EJBs deployed in the application.

Code Snippet 1 shows the definition of a default descriptor in the application.

Code Snippet 1:

```
<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_1.xsd"  
    version="3.1">  
  
<assembly-descriptor>  
    <interceptor-binding>  
        <ejb-name>*</ejb-name>  
        <interceptor-class>com.beans.MyInterceptor</interceptor-  
class>  
    </interceptor-binding>  
    ...  
</assembly-descriptor>
```

Code Snippet 1 shows how an interceptor is defined on all the bean classes of the application. The default interceptors are invoked, before every method invocation in the application. The wildcard character '*' in the `<ejb-name>` element applies the interceptor to every EJB deployed in the application.

□ Class-level interceptors

These interceptors are defined on all the methods in a bean class. Bean developers use `@Interceptors` annotation to define the interceptor for all the methods in the class as shown in Code Snippet 2.

Code Snippet 2:

```
@Stateless  
 @Interceptors (value=com.beans.MyInterceptor.class)  
 public class ApplicationBean {  
     ...  
 }
```

Code Snippet 2 shows an interceptor `@Interceptors` annotation declared at the class-level intercepts all the methods of an `ApplicationBean` class.

□ Method-level interceptors

These interceptors are defined to intercept only a method call in the EJB. Similar to class-level interceptors, these interceptors are defined through `@Interceptors` annotation on specific method. The method for which the interceptor is defined is known as a target method.

Code Snippet 3 demonstrates the method-level annotation.

Code Snippet 3:

```
...  
 @Interceptors ({AccountsConfirmInterceptor.class})  
     public void sendBookingConfirmationMessage (long  
 orderId)  
     {  
         ...  
     }  
 ...
```

7.2.4 Configuring an Interceptor Class

An Interceptor class is different from the bean class, as the invocation of an interceptor method is performed in response to the execution of the business method or lifecycle callback method of the bean.

After the Interceptor class is defined, the developer can associate it with Stateless Session bean or Stateful Session bean.

The steps to configure an interceptor class are as follows:

1. Create an Interceptor class which is a simple POJO class in Java.
2. Define a method in the Interceptor class which can be business method or lifecycle method.
3. Designate the method as the interceptor method with `@AroundInvoke` annotation or lifecycle annotations.
4. Apply or associate the defined interceptor class to the EJB session bean.

7.2.5 Creating Business Method Interceptor

To create business method interceptors the annotation `javax.interceptor.AroundInvoke` is used with the interceptor method. Code Snippet 4 shows the signature of the business interceptor method annotated with `@AroundInvoke` annotation.

Code Snippet 4:

```
@AroundInvoke  
Object <METHOD_NAME> (javax.Interceptor.InvocationContext) throws  
Exception  
{  
    ...  
}
```

Code Snippet 4 shows the `@AroundInvoke` annotation that wraps around the invoked business method and is invoked in the same context in which the bean is executed. The method designates as the interceptor method. The developer can use only one `@AroundInvoke` annotation per class.

The interceptor method can be declared with `public`, `private`, `protected`, or package level access. However, the method cannot be declared as `final` or `static`. The interceptors may throw runtime exceptions.

The interceptor method is invoked with an `InvocationContext` object as a parameter. The `InvocationContext` object is the generic representation of the business method for which the interceptor is defined. The `InvocationContext` object includes information such as target bean instance on which the interceptor is invoked, parameters with which the target bean is invoked, and so on. The security and transaction context of the `@AroundInvoke` interceptor is same as that of the target method.

Some of the methods of the `InvocationContext` object are as follows:

- `Object getBean():` Returns a reference to the bean on which the method is invoked.
- `Method getMethod():` Returns a reference to the invoked method.
- `Object[] getParameters():` Returns the parameters passed to the method.
- `void set Parameters(Object[] parameters):` Sets the parameters of the object.
- `Map getContextData():` Gets contextual data that can be shared in a chain.
- `Object proceed():` Proceeds to the next interceptor in the chain or the business method, if it is the last interceptor.

Code Snippet 5 demonstrates the definition of an interceptor class.

Code Snippet 5:

```
...
import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;
public class MyInterceptor {

    @AroundInvoke
    public Object businessIntercept(InvocationContext ctx)
        throws Exception {
        Object result=null;
        System.out.println("perform intercepting operations");
        try {
            result=ctx.proceed();
            return result;
        } finally { ...
        }
    }
}
```

Code Snippet 5 creates the method named `businessIntercept()` which is annotated by `@AroundInvoke` annotation. The method intercepted with `@AroundInvoke` annotation will be executed before any of the bean's business method gets invoked. The reference of `InvocationContext`, `ctx` is obtained. The method `proceed()` is invoked on the `ctx` reference to get the next configured interceptor in the chain. If another interceptor has to be invoked as a part of method call, then `proceed()` invokes the `@ AroundInvoke` method of the other interceptor.

The advantage of interceptors is that they provide the developer a way to add functionalities on the business methods without modifying the method code.

Figure 7.3 shows the wrapping of the business method with the chain of interceptors.

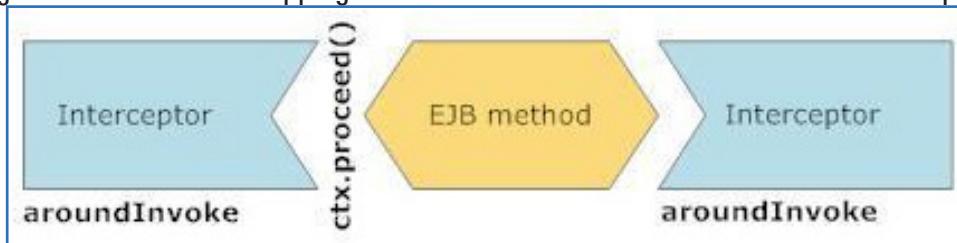


Figure 7.3: Business Method with AroundInvoke Wrapper

7.2.6 Applying Interceptor

After the interceptor class is created, it needs to be applied on the EJB bean class. The developer can apply one or more interceptors to all deployed EJB beans by configuring interceptor in deployment descriptor. Interceptor can be configured to be applied to a single method of the bean class.

Alternatively, interceptor can be applied through `@Interceptors` annotation. The syntax to add multiple interceptors can be declared for a target method through the `@Interceptors` annotation.

Syntax:

```

@Interceptors({Interceptor1.class, Interceptor2.class})

targetMethod() {

    ...

}
  
```

Code Snippet 6 shows the code of a target class which uses the interceptor.

Code Snippet 6:

```

...
@Stateless
@LocalBean
public class ExampleBean {

    @Interceptors(value=interceptors.MyInterceptor.class)

    public void sayHello() {
        System.out.println("SayHello");
    }
  
```

Code Snippet 6 shows the code of the target class on which the interceptor class is defined. The `businessIntercept()` method is invoked before any method of the `ExampleBean` class is invoked.

Code Snippet 7 shows the invocation of the `ExampleBean` method.

Code Snippet 7:

```
package Beans;

public class InterceptorDemo {

    public static void main(String[] args) {

        ExampleBean E = new ExampleBean();
        E.sayHello();
    }

}
```

Code Snippet 7 instantiates the `EmployeeBean` class and invokes the method `sayHello()` on it.

While defining the interceptor as an interceptor class, it must meet the following requirements:

- The interceptor class must have a default public constructor.
- The interceptor classes should be defined with `@Interceptors` annotation.
- When a target class has multiple interceptor classes defined for it, the order of interceptor invocation should be defined through annotations.
- The order of invoking the interceptors in case of multiple interceptors can be defined in the deployment descriptor of the application also. The order defined in the deployment descriptor overrides the order defined through annotations.

Code Snippet 8 shows the code to specify the interceptors in the deployment descriptor for a particular method.

Code Snippet 8:

```
...
<interceptor-binding>
<target-name>Target bean class </target-name>
<interceptor-class>First interceptor class</interceptor-
class>
<interceptor-class>Second interceptor class</interceptor-
class>
<interceptor-class>Third interceptor class</interceptor-
class>
<method-name>Method to be intercepted</method-name>
</interceptor-binding>
...
```

The order in which the interceptors are invoked is same as the order in which these interceptors are specified in the deployment descriptor. The `<method-name>` element includes the method for which the interceptors will be executed in the specified order.

7.2.7 Lifecycle of an Interceptor

The lifecycle of the interceptor is the same as that of the bean with which the interceptor is associated. When an instance of target class is created, then the interceptor class is also instantiated for the target class. If the target class uses multiple interceptors, then in that case, the object of each applied interceptor class is instantiated, before the business methods are invoked on the target bean class instance.

All the interceptor classes must be instantiated, before the invocation of the `PostConstruct` callback method. Similarly, the `PreDestroy` callback method is invoked before the instances of target class and interceptor class are destroyed.

7.3 Lifecycle CallBack Interceptors

Lifecycle callback events occur whenever the bean object changes from Does Not Exist state to Ready state or when the bean object transits from activated state to passivated state, and so on. The lifecycle callback events are handled through the lifecycle callback methods.

Some of the annotations related to the lifecycle methods of the bean class are as follows:

- `javax.interceptor.AroundConstruct` – This annotation is used when the developer intends to invoke the interceptor method during a lifecycle event. When an enterprise bean is instantiated, if the developer intends to invoke an interceptor on it, then the `@AroundConstruct` annotation can be used.
- `javax.annotation.PostConstruct` – Interceptor methods prefixed with the annotation `@PostConstruct` are invoked after the bean object is instantiated into the container. These methods are used to perform the tasks required after the bean object is instantiated by the container. They are invoked after the lifecycle event of instantiation. These methods may perform tasks such as dependency injection and so on.
- `javax.annotation.PreDestroy` – Interceptor methods annotated with `@PreDestroy` annotation are invoked when the bean object has to be removed from the container. These methods perform tasks such as deallocating the resources allocated for the bean and so on.

Apart from this, Stateful Session bean also have methods annotated with `@PrePassivate` and `@PostActivate`.

- `@PrePassivate` annotation is used to annotate interceptors defined on Stateful session beans. It is invoked before the bean is passivated.
- `@PostActivate` annotation is also used to annotate interceptors defined on the Stateful session beans. It is invoked when the bean is removed from a passivated state.

If the lifecycle callback interceptors are defined within the target class they are annotated with appropriate annotations. If the interceptors are defined in an interceptor class then the interceptors have an `InvocationContext` parameter.

The lifecycle callback interceptors cannot be `static` or `final`. These interceptors may throw runtime exceptions.

The syntax for declaring the lifecycle interceptor methods is as follows:

Syntax:

```
@callback-annotation  
void method-name (InvocationContext ctx);
```

The method defined in the interceptor class will be annotated with the callback annotation. The return value of these method is `void`, as EJB callback methods do not return any value. The method takes the `InvocationContext` as the parameter.

Code Snippet 9 shows an interceptor class for a Stateful Session bean with lifecycle callback interceptor methods.

Code Snippet 9:

```
public class MyStatefulSessionBeanInterceptor {  
    ...  
    protected void myInterceptorMethod (InvocationContext ctx) {  
        ...  
        ctx.proceed();  
        ...  
    }  
  
    @PostConstruct  
    @PostActivate  
    protected void  
    myPostConstructInterceptorMethod (InvocationContext ctx) {  
        ...  
        ctx.proceed();  
        ...  
    }  
    @PrePassivate  
    protected void myPrePassivateInterceptorMethod (InvocationContext  
    ctx) {  
        ...  
        ctx.proceed();  
        ...  
    }  
}
```

In the code, the `myPrePassivateInterceptorMethod()` is defined as the lifecycle callback interceptor method for the pre-passive lifecycle event. Then, the method `myPostConstructInterceptorMethod()` is defined to handle the PostConstruct and PostActivate lifecycle events.

7.4 Timeout Interceptors

Enterprise beans cannot be retained in the container indefinitely. Inactive enterprise beans are removed from the container after a certain timeout period defined according to the configuration of the application.

When an object is removed from the container due to timeout, interceptors can be defined to handle this event.

The timeout events of the bean objects may have interceptors annotated with `@AroundTimeout`. Timeout interceptors can be defined either in the target class or in an interceptor class.

The syntax to define the `@AroundTimeOut` is as follows:

Syntax:

```
@AroundTimeOut  
public Object InterceptorMethod() {  
    ...  
}
```

Like other interceptors the timeout event interceptors cannot be `static` or `final`. A timeout event interceptor can access all the resources accessed by target timeout method and has the same security constraints and transaction context as the target method.

Multiple timeout event interceptors can be used on the target methods, where the order of interceptors is specified through `@Interceptors` annotation or through the deployment descriptor.

7.5 Interceptor Chaining

A class or method can have multiple interceptors defined on it. When there are multiple interceptors defined, it is important to specify an order in which these interceptors will execute. The order of interceptor execution can be specified with the interceptors or through the deployment descriptor.

Following are the rules which determine the order in which the interceptors are invoked:

- There are a set of default interceptors which are determined on a target class or bean, these are executed first. Default interceptors are specified in the deployment descriptor. These default interceptors are executed in the order in which they are specified in the deployment descriptor.
- The order of the interceptors can also be defined with the `@Interceptors` annotation. The order of execution is the order in which they are specified with the annotations. Priorities assigned to the interceptors are not applicable to the `@Interceptors` annotation.
- When interceptors are defined as classes, hierarchy is applicable among these classes. In such a scenario, the super class interceptors are executed first and then, the subclass interceptors.

- `javax.annotation.priority` annotation can be used to prioritize the interceptors. This annotation can be used to set priority to the interceptor methods.
- For interceptors annotated with `@AroundConstruct` annotation, the interceptors are invoked first and then, the constructor of the target bean class.

7.6

Context and Dependency Injection

Context and Dependency Injection (CDI) is a standard introduced in Java EE 6 used to define the structure of application code in the enterprise application. CDI is a type safe, dependency injection mechanism which supports Java EE modularity and Java EE component architecture. CDI enables integration of various components of the application in a loosely coupled manner.

CDI is a set of services that make it easy for developers to use enterprise beans along with JavaServer Faces technology in Web applications. CDI allows the EJBs to be used as managed beans for JSF applications. This helps the Web tier to interact directly with the business and persistence tiers.

Various application components make use of managed beans for implementation of business logic. Managed beans implement the business logic for both enterprise and Web applications. CDI is responsible for resolving the dependencies on the managed beans.

Figure 7.4 shows the interrelationships among different specifications in Java EE using CDI.

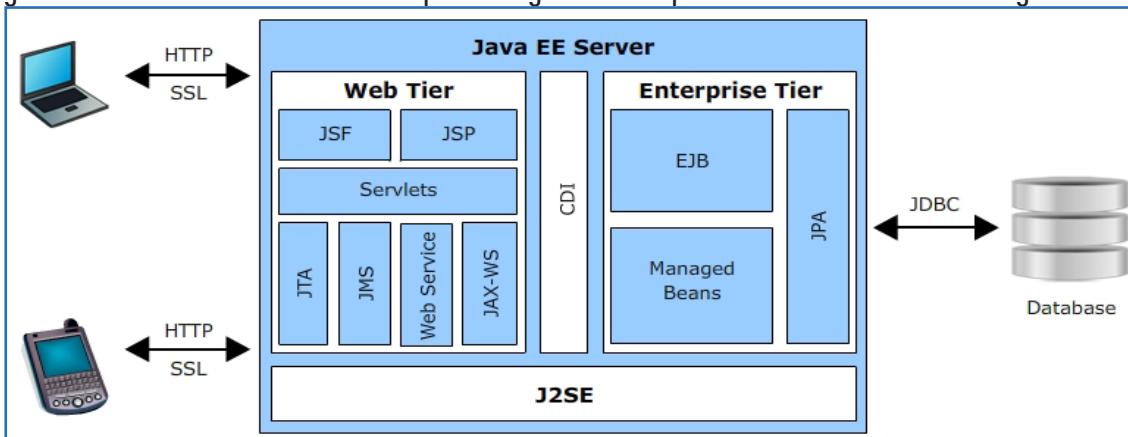


Figure 7.4: Interrelationship among Java EE Specifications

Figure 7.4 demonstrates that various specifications of Java EE platform which depend on CDI for injecting the required dependencies in the application. In figure 7.4, Servlets, EJBs, JSP, and JSF depend on CDI for injecting the dependencies on managed beans in the application. The service of CDI is implemented through the container. Apart from the managed beans other Java EE resources are also injected into the application through CDI.

Following are the services provided by the CDI in Java EE:

- CDI provides contexts for the lifecycle of Stateful Session bean. Context refers to the environment of creation for the stateful objects.
- CDI provides for dependency injection which allows resources to be injected into the application in a type-safe way.

- CDI allows for integration of different components of the applications by using expression language. Using expression language also enables compatibility among different application components such as JSP, JSF, and so on.
- It allows the bean developer to associate interceptors with bean components in the application. It provides an event-notification model.
- It also provides a Service Provider Interface (SPI) where third party frameworks can be integrated into the Java EE environment.
- CDI provides for loose coupling of the application components:
 - It decouples the client and the server through well-defined basic datatypes or object types and their respective qualifiers.
 - It decouples the lifecycle of the collaborating components, by allowing the stateful components to interact with services by interchanging messages.

7.6.1

Implementing Managed Beans

All enterprise beans are implemented as Java classes. Therefore, the managed beans are also implemented as Java classes. Following are the requirements of a managed bean class:

- The managed bean class should not be a non-static inner class.
- The managed bean class should be a concrete class or should be annotated with `@Decorator`.
- The managed bean class should not be annotated with an EJB component defining annotation or declared as an EJB bean class in the deployment descriptor `ejb-jar.xml`.
- The managed bean class should have a default constructor and the class should declare a constructor annotated `@Inject`.

Code Snippet 10 demonstrates the usage of `@Inject` annotation.

Code Snippet 10:

```
...
import javax.ejb.LocalBean;
import javax.ejb.Stateless;
import javax.inject.Inject;
@Stateless
@LocalBean
public class InjectBeanDemo {
    @Inject
    private HelloBean H;
    public InjectBeanDemo () {
        H= new HelloBean();
    }
    public static void main (String args[]) {
        InjectBeanDemo I = new InjectBeanDemo ();
    }
}
```

In Code Snippet 10, a `HelloBean` is injected into the current bean through `@Inject` annotation.

Code Snippet 11 shows the code in the `HelloBean`.

Code Snippet 11:

```
...
@Stateless
@LocalBean
public class HelloBean {

    public HelloBean() {
        System.out.println("In Hello Bean");
    }
}
```

Code Snippet 11 shows the `HelloBean` which is injected into the `InjectBeanDemo` class.

7.6.2 Beans as Injectable Objects

CDI makes it possible to inject objects into applications which are not container managed. Following are the objects which CDI enables to be injected into the application:

- CDI allows any Java class to be injected as a resource into the application.
- CDI enables injection of Session beans as resources.
- CDI enables injection of Java EE resources such as data sources, Java Message Service topics, queues, and so on.
- It allows injection of persistence contexts.
- It allows injection of producer fields.

7.6.3 Injecting Beans

Enterprise beans are used as resources in other bean classes. In order to perform this kind of injection, bean classes make use of the annotation `javax.inject.Inject`.

Consider a bank application, where there are two enterprise beans `CustomerInformation` and `LoanApproval`. In this scenario the bank is offering pre-approved loans to a subset of its customers.

```
class CustomerInformation{
    ...
}
```

When `LoanApproval` bean requires the customer information, the `CustomerInformation` bean is injected.

Code Snippet 12 demonstrates the process of injecting the `CustomerInformation` into the `LoanApproval` bean.

Code Snippet 12:

```
import javax.inject.Inject;  
  
class LoanApproval{  
  
    @Inject CustomerInformation;  
  
    ...  
}
```

7.6.4 Configuring a CDI Application

To configure a CDI application, the scope of the bean has to be appropriately defined for the bean class. The scope can be set by using annotations on the session bean class or it can be defined through a deployment descriptor.

The scope of a session bean can have any one of the following values:

- Request** – When the scope of the bean is defined to be a request, then the bean is active only till one request from the client is served.
- Session** – The scope of the bean in this case is for a set of HTTP requests during an interaction with the client.
- Application** – When the scope value is set to be application then the bean is active as long as the application is in the container.
- Dependent** – Dependent is the default scope when no other scope is specified, which implies that the bean object exists to serve exactly one client bean and has the same lifecycle as that of the client bean.
- Conversation** – This scope is set when there is an interaction with a servlet or similar entities which have long running sessions.

7.6.5 Packaging CDI Applications

When an enterprise application is deployed, all the beans managed by CDI are packaged as bean archive files. There are two variants of bean archive files namely, implicit bean archive files and explicit bean archive files.

An explicit bean archive comprises a `beans.xml` deployment descriptor. The `beans.xml` file keeps track of all the enterprise beans in the application. This kind of archiving is essential to simplify the bean discovery process in the applications. The `beans.xml` descriptor also defines the interceptors, decorators, and alternatives associated with each bean.

Code Snippet 13 shows an empty beans.xml descriptor.

Code Snippet 13:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans 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/beans_1_0.xsd">
</beans>
```

The implicit bean archive is one which contains some beans annotated with a scope type. It does not contain beans.xml deployment descriptor. The container checks all the Java classes in the application for bean defining annotation and discovers the beans through them.

Check Your Progress

1. Which of the following specifications use the services of CDI?

(A)	Servlets	(C)	EJB
(B)	JSF	(D)	All of these

2. Which of the following define the order of invoking the interceptors?

(A)	Deployment descriptor	(C)	Both a and b
(B)	@Interceptors annotation	(D)	@InterceptorClass annotation

3. Which of the following is an implicit bean archive deployment descriptor?

(A)	beans.xml	(C)	persistence.xml
(B)	web.xml	(D)	All of these

4. Which of the following is a valid value for scope attribute of a Session bean? A, B, C => true

(A)	Request	(C)	Conversation
(B)	Dependent	(D)	Local

5. Which of the following annotations are associated with interceptors on lifecycle callback events?

(A)	@AroundTimeOut	(C)	@PreDestroy
(B)	@PostConstruct	(D)	Both b and c

6. Which of the following statements is not true about interceptors?

(A)	Interceptors can be defined as a method in the target class	(C)	Interceptors can be defined as an interface
(B)	Interceptors can be defined as an independent class	(D)	Each target class can have multiple interceptors

Answer

1.	1
2.	2
3.	3
4.	4
5.	5
6.	6

Summary

- Interceptors are invoked when either business methods are invoked or when bean objects are instantiated.
- Interceptors can be implemented as classes or methods.
- When interceptors are defined as a class then interceptor method has an InitialContext parameter.
- There can be more than one interceptor defined for the bean class.
- Multiple interceptors can be defined using @Interceptors annotation or through deployment descriptor.
- CDI is used in enterprise applications for type-safe dependency injection.
- To configure a CDI application, the scope of the bean has to be appropriately defined for the bean class.

WRITE-UPS BY

EXPERTS AND LEARNERS

TO PROMOTE NEW AVENUES AND
ENHANCE THE LEARNING EXPERIENCE



FOR FURTHER READING, LOGIN TO

www.onlinevarsity.com



Welcome to the Session, **Transactions**.

This session explains the concept of transactions and its properties. This session discusses the usage of Java Transactions API (JTA) in enterprise applications. It also discusses transaction management through container-managed and bean-managed transactions in the Java enterprise applications.

In this Session, you will learn to:

- Explain the concept of transactions
- Define various properties of transactions
- Explain Java Transaction API (JTA)
- Describe the interfaces of JTA that can be used in the enterprise applications
- Explain programmatic versus Declarative demarcation
- Explain container-managed transaction
- Explain bean-managed transaction
- Explain how to manage transactions in messaging

8.1 Transaction Basics

A transaction can be defined as a group of operations which are supposed to be executed as a single unit. For instance, a purchase made through an e-commerce portal involves different operations such as choosing the product, adding it to the shopping cart, making payment for the product, and finalizing the transaction. All these operations are an integral part of a transaction and all of them have to execute and finish to make the transaction as complete.

8.1.1 Transaction and Data Integrity

An enterprise application stores critical information for business operations in databases. The data present in the databases should not only be accurate and reliable but also current. If multiple users access the same data at the same time then, the integrity of the data is lost. Again, if there is a system failure then, business transaction would partially update the data. Thus, transactions ensure data integrity of the data stored in the database.

Transactions on the database are managed by the transaction management mechanism provided by the DBMS. Apart from the transaction management provided in the database tier, enterprise applications require transaction management in the middle layer of the application. The transaction management mechanism in the middle layer is provided by the Java Transaction API (JTA).

Figure 8.1 illustrates the concept of transaction using the example of an e-commerce portal.

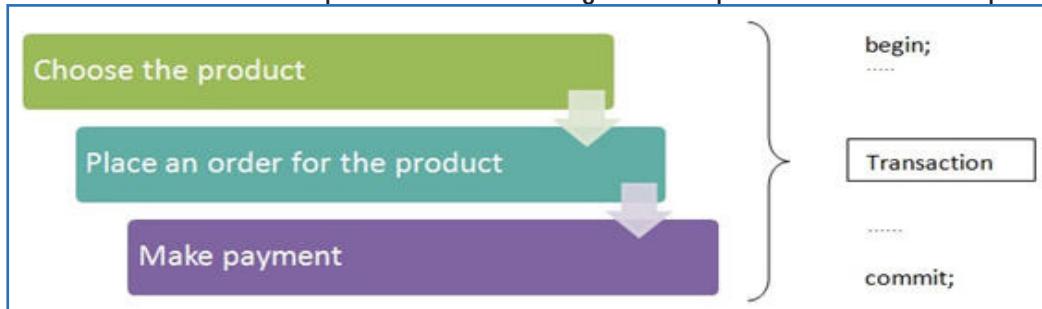


Figure 8.1: Concept of Transaction

As shown in figure 8.1, a transaction is an indivisible unit of work. It is an atomic unit of a process. It is always applied as a whole and if there is any failure, a partially completed transaction is rolled back. For instance, when a user clicks the **BUY** button on an online shopping site, the following processes take place on the server-side:

- User's credit card details are validated.
- Credit card is charged.
- Order is generated and sent to the Shipping department.
- Invoice is sent to the user's e-mail account.

8.1.2 Transaction Scope

A transaction scope extends from the begin operation of the transaction and ends at the commit operation of the transaction. All the operations between begin and commit have to successfully execute before the commit operation executes. If any of the operations fail then all the operations of the transaction must be undone. The system must be restored to a consistent state which existed before the begin operation. This process of restoring the system to a previous consistent state is known as rollback.

Applications use system log or application log to restore the system to a consistent state before the transaction begins.

8.1.3 Transaction Properties

The properties of a transaction are referred as ACID. ACID properties stand for Atomicity, Consistency, Integrity, and Durability. A transaction is said to be a valid transaction when all these properties hold good for reliable processing of the task.

The ACID properties are described as follows:

- **Atomicity** - Implies that either all the operations of the transaction execute or none of the operations execute. The transaction should execute as a single unit. If any one of the operations fail then, rest of the operations are rolled back resulting in a transaction roll back. An example of an atomic transaction is an account transfer transaction. In account transfer, the money is removed from account A and placed into account B. If the system fails after removing the money from account A, then the transaction processing system will put the money back into account A, thus, bringing the system to its original state.
- **Consistency** - Implies that the execution of the transaction should leave the application in a consistent state, where all the constraints of the application stay valid. All the validation constraints should hold good after the transaction execution. In account transfer system, the system is consistent, if the total of all accounts is constant. If an error occurs and the money is removed from account A and not added to account B, then the total in all accounts would have changed. This means that the system would no longer be consistent. Thus, to bring the system in a consistent state, the rolling back is done on the account A to bring the system back in a consistent state.
- **Isolation** - Implies that when multiple transactions are executing in the context of an application, then execution of one transaction should not affect the execution of another transaction. All the transactions should independently execute. Also, other components of the application should not see the intermediate state of the data when the transaction is executing. They should only see the committed data after the transaction execution completes.

- **Durability** - Implies that the changes made by a committed transaction are permanent and are persisted to a database or any storage. The changes made are not affected by system failures or errors in the system.

8.1.4 Transaction Demarcation

The process of determining where the transaction begins and ends is referred to as transaction demarcation. The enterprise bean developer must determine the transaction demarcation policy for each method of the enterprise bean class.

EJB supports two methods for transaction demarcation: programmatic and declarative.

In programmatic demarcation, the developer begins and ends a transaction explicitly in the methods by calling the transaction begin and transaction commit or abort methods. The developer is responsible for issuing the begin, commit, or abort statement.

In declarative demarcation, components are marked as transactional in the deployment descriptor or using annotations. The transactions are managed by the container, which decides when to invoke the begin and commit or abort methods. The container is also responsible to display consistent outcome to all the users.

The transaction demarcation ends after a transaction commits or rolls back. The commit request directs all the participating components to store the effects of the transaction operations permanently. The rollback request makes the components to undo the effects of all transaction operations.

8.1.5 Programmatic vs Declarative Demarcation

Programmatic demarcation is used only if you have a small number of transactional operations. For example, if you have a Web application, that require transactions only for certain update operations. On the other hand, if your application has numerous transactional operations, declarative demarcation is worthwhile. It keeps transaction code out of business logic, and is not difficult to configure.

In a programmatic demarcation, the programmer determines where the transaction begins and ends. However, in declarative demarcation container is not able to determine the boundaries within the code and places the entire method into transaction.

When designing an enterprise bean, the developer can determine how the boundaries need to be set, that is, by specifying whether the transaction is declarative or programmatic.

The EJB specifications refer the declarative transactions as Container-Managed Transaction (CMT) and programmatic transactions as Bean-Managed Transaction (BMT).

8.2**Transaction Processing**

Transaction processing in enterprise applications is done with the involvement of the following three components:

- Application Components** - These are the EJB business methods which invoke transactions.
- Resource Managers** - These components manage the persistent data storage. The resource managers are usually drivers with interfaces for communicating with databases.
- Transaction Manager** - This is the core component that creates and maintains transactions.

When the transaction manager receives a request from the application component to commit a transaction, the transaction manager performs the activity in two phases.

- In the first phase, the transaction manager requests all the involved resource managers to be prepared for the transaction. Each of the resource managers inform whether they are prepared or not. If a resource manager replies that it is not prepared, the transaction is rolled back immediately. If all the resource managers are ready, a transaction log with all the updates included in the transaction is created and the transaction manager gets ready for the next phase.
- In the second phase, the transaction manager sends a commit request to all the resource managers and the transaction is said to be committed.

Since, this complete process is achieved in two phases; it is referred as two-phase commit protocol. Figure 8.2 shows a two-phase commit protocol.

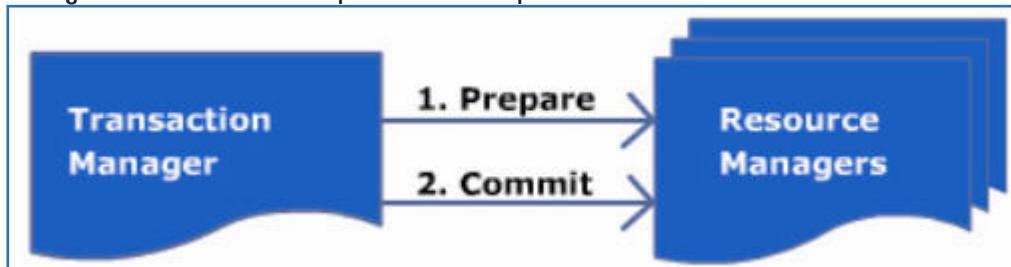


Figure 8.2: Two-Phase Commit Protocol

8.2.1**Transaction API**

The transactions in Java applications are managed through Java Transaction Service (JTS) and Java Transaction API (JTA). JTA is an interface between a transaction manager and the components involved in distributed transaction system. The transactions of an application are managed by the JTA API in the container.

JTS provides a specification for the implementation of a transaction manager in the application. JTS is based on the Object Transaction Service (OTS) which is a part of the Common Object Request Broker Architecture (CORBA) specification. The JTS service is implemented by the vendors on enterprise middleware to provide transaction processing infrastructure on the application servers.

JTA uses the services and implementation of the transaction manager provided by JTS to implement transaction managing functions.

Figure 8.3 demonstrates how JTS and JTA are interrelated.

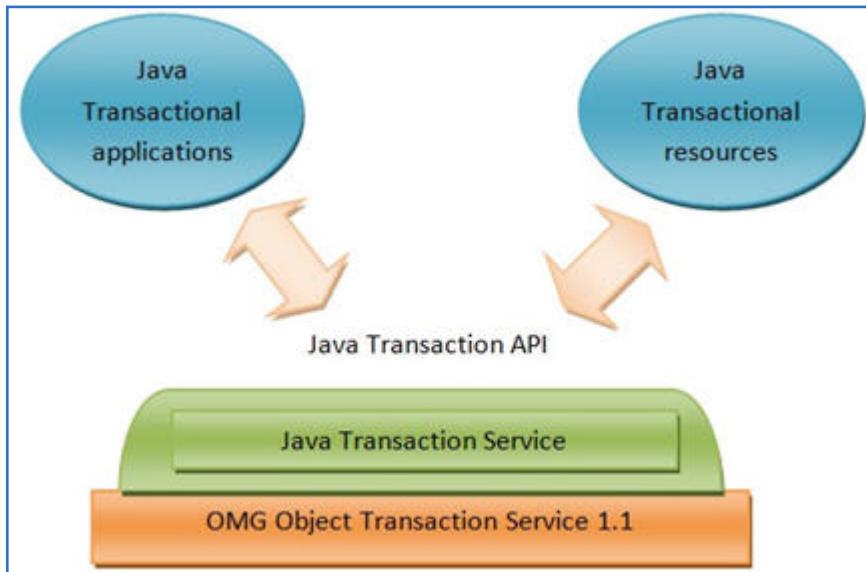


Figure 8.3: JTS and JTA API

As shown in figure 8.3, JTA uses the transaction manager interface provided by JTS to provide transaction support to the enterprise application components.

JTA API consists of the following three sets of interfaces:

- The `javax.Transaction.xa.XAResource` interface which is used by JTA to communicate with X/Open XA-enabled resource managers.
- The `javax.transaction.TransactionManager` interface which is used by JTA to communicate with the application servers.
- The `javax.transaction.UserTransaction` interface is used by the enterprise beans and other Java classes to work with the EJB transactions.

8.3

Container-Managed Transaction (CMT)

In case of CMT, the container is responsible for starting and managing the transaction. A container managed transaction can work with a Message-driven bean or a session bean. The limits of the transaction and aspects such as when the transaction should start and when it should end are not defined.

The container is the demarcation of the transaction. This category of transactions is said to implement declarative transaction demarcation where the transaction characteristics are specified in the deployment descriptor or metadata annotation through transaction attributes.

The transactions in an enterprise bean are associated with the method in the bean. A transaction begins as the method of the bean is invoked and the corresponding transaction is committed before the method of the bean ends. A method of the bean can be associated with only a single transaction. Multiple transactions or nested transactions are not allowed in bean methods.

Every method of the bean need not be associated with transactions.

When enterprise beans use container-managed transactions, then the bean does not use other transaction methods such as `commit`, `setAutoCommit`, and `rollback` which are part of the `java.sql.Connection` and `javax.jms.Session` packages.

Following are the methods which cannot be executed in container-managed transactions:

- `commit`, `setAutoCommit`, and `rollback` method of `java.sql.Connection`
- `getUserTransaction` method of `javax.ejb.EJBContext`
- All methods of `javax.transaction.UserTransaction`

8.3.1 Transaction Attributes

A transaction attribute controls the scope of a transaction. It conveys to the container the intended transactional behavior of the associated method.

To understand the transaction scope, consider a scenario illustrated in figure 8.4.

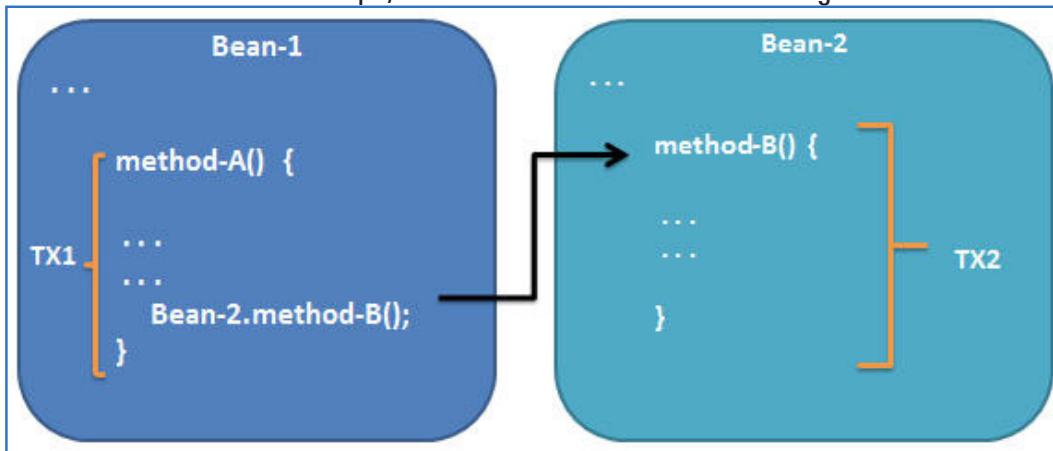


Figure 8.4: Transaction Scope

As shown in figure 8.4, `method-A` begins a transaction and then, invokes `method-B` of Bean-2. When `method-B` executes, a point which can be asked is, whether `method-B` runs within the scope of the transaction started by `method-A`? or Does it execute with a new transaction?

The answer to these questions, depends on the scope of transaction attribute selected by `method-B`.

A transaction attribute can have any one of the following values:

- Required
- RequiresNew
- Mandatory
- Never
- Supports
- Not Supported
- When the transaction attribute value is set to `Required`, then it implies that the bean method should be invoked within a transaction scope. If the calling method or application is not in a transaction scope then, a new transaction scope is created. The `Required`

attribute is the implicit transaction attribute for all enterprise bean methods running with container-managed transaction demarcation.

- The transaction attribute `RequiresNew` implies that the execution of the annotated bean method requires a new transaction scope to execute. Even, if the calling method is already in a transaction scope, a new transaction scope is created for the execution of the new method. The calling methods transaction is suspended, until the new transaction completes execution and returns. You should use the `RequiresNew` attribute when you want to ensure that the method always runs within a new transaction.
- When the transaction attribute is set to `Mandatory`, it implies that a transaction scope is required in order to execute the annotated method. The container need not create a new transaction scope. If the calling client is already in a transaction scope then the annotated bean method is executed in it otherwise a new transaction scope is created. You can use the `Mandatory` attribute, if the enterprise bean's method must use the transaction of the client.
- When the transaction attribute value is set to `Never`, then it implies that the annotated bean method should not be invoked in a transaction scope. If the calling client of the annotated method is already in a transaction scope then it throws an exception. If the client is not associated with a transaction, the container does not start a new transaction, before running the method.
- When the transaction attribute value is set to `Supports`, it implies that when an annotated bean method is invoked from a transaction scope then, the invoked bean method executes in the current transaction scope. The invoking method and the invoked method all become part of the original transaction. If the client is not associated with a transaction, the container does not start a new transaction before running the method.
- When the transaction attribute value is set to `Not Supported`, it implies that the annotated method cannot be executed in a transaction scope. If the annotated method is invoked from a transaction scope then the transaction is suspended until the invoked bean method returns.

Not all transaction attributes can be used with all kinds of beans. The list of transaction attributes permitted with the EJB type is listed in table 8.1.

Transaction Attribute	Stateless Session Bean	Stateful Session Bean	Entity Bean	Message-driven Bean
NotSupported	Yes	Yes	Yes	Yes
Required	Yes	Yes	Yes	No
Supports	Yes	Yes	Yes	No
RequiresNew	Yes	No	No	No
Mandatory	Yes	No	No	Yes
Never	Yes	No	No	No

Table 8.1: Session Bean Client Views of EJB 3.1Lite and EJB3.1

A transaction attribute can be set through the annotation `@javax.ejb.TransactionAttribute`. The `@javax.ejb.TransactionAttribute` can be defined as an enumerated value. A transaction attribute can be defined for an entire bean class or for a bean method. By default, the transaction attribute is applied for the entire bean class. The `@TransactionAttribute` annotation accepts the constant values provided by the `@javax.ejb.TransactionAttributeType` constants.

Table 8.2 lists the `TransactionAttributeType` constants that are provided to the `@TransactionAttribute` annotation.

Transaction Attribute	Constant
NotSupported	<code>TransactionAttributeType.REQUIRED</code>
Required	<code>TransactionAttributeType.REQUIRES_NEW</code>
Supports	<code>TransactionAttributeType.MANDATORY</code>
RequiresNew	<code>TransactionAttributeType.NOT_SUPPORTED</code>
Mandatory	<code>TransactionAttributeType.SUPPORTS</code>
Never	<code>TransactionAttributeType.NEVER</code>

Table 8.2: `TransactionAttributeType` Constants

8.3.2 Implementing CMT

CMT are those transactions which are initiated, managed, and closed by the application container. The application developer uses the services of the underlying container by declaratively specifying the application requirements. Developers can specify the requirements through annotations and deployment descriptors.

In order to demonstrate the implementation of a CMT, an EJB module is created and a bean class which will initiate a transaction.

Following are the steps involved in implementing the transaction:

1. Create an EJB module.
2. Add a bean class and define methods in the class.
3. Annotate the bean methods appropriately according to the application requirement. In this case, perform a transaction on the database where a row is inserted into the database table.
4. Create a client class which will initiate the transaction.

Code Snippet 1 demonstrates the CMT.

Code Snippet 1:

```
...
@TransactionManagement(TransactionManagementType.CONTAINER)
@Stateless
@LocalBean
public class CMT {

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public String add(Connection c) {
        String message;
        try{
            c.createStatement();
            CachedRowSet st = new CachedRowSetImpl();
            String query="insert into Customers values(1007,'Martha')";
            st.setCommand(query);
            st.execute(c);
            message="Row Inserted";
        }catch(SQLException e){      message=e.toString();  }
        return message;
    }

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public CachedRowSet display(Connection c) throws SQLException{
        Statement stmt=c.createStatement();
        CachedRowSet st = new CachedRowSetImpl();
        String quer="select * from Customers";
        st.setCommand(quer);
        st.execute(c);
        return st;
    }
}
```

Code Snippet 1 shows a bean class with two bean methods defined. The annotation in the code defines the transaction management type as Container. This implies that the transaction management is taken care of by the application container. Annotations define the transaction attributes for each of the bean methods.

The bean method add() takes a Connection object as a parameter. This object is used to connect the bean to the database of the application. The method then initiates a transaction on the database to add a row to the table.

The bean method display() also connects to the database through a Connection object and displays all the rows present in the Customers table. These bean methods are accessed by a client and initiate the transaction.

Code Snippet 2 shows the client code which accesses the bean methods.

Code Snippet 2:

```
...
public class CallCMT {
    public static void main(String[] args)
    try {
        String driver = "org.apache.derby.jdbc.ClientDriver";
        String url = "jdbc:derby://localhost:1527/sample";
        String username = "app";
        String password = "app";
        Class.forName(driver).newInstance();
        Connection conn = DriverManager.getConnection(url, username,
password);
        CMT c = new CMT();
        c.add(conn);
        c.display(conn);
    } catch (...) {
}
}
```

Code Snippet 2 is a main class which creates a connection with the database. It creates an object of the bean class and then, invokes the bean methods.

On executing the application, the state of the database must be checked by choosing the database from the Services tab in the NetBeans IDE as shown in figure 8.5.

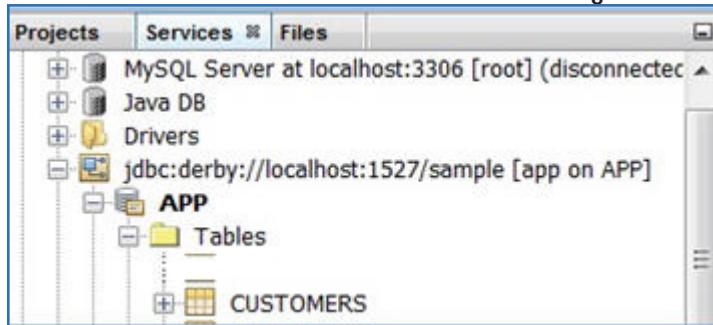


Figure 8.5: Database Configuration in NetBeans IDE

After executing the client, double-clicking the database table executes commands on it. The result of the transaction can be seen by executing commands as shown in figure 8.6.

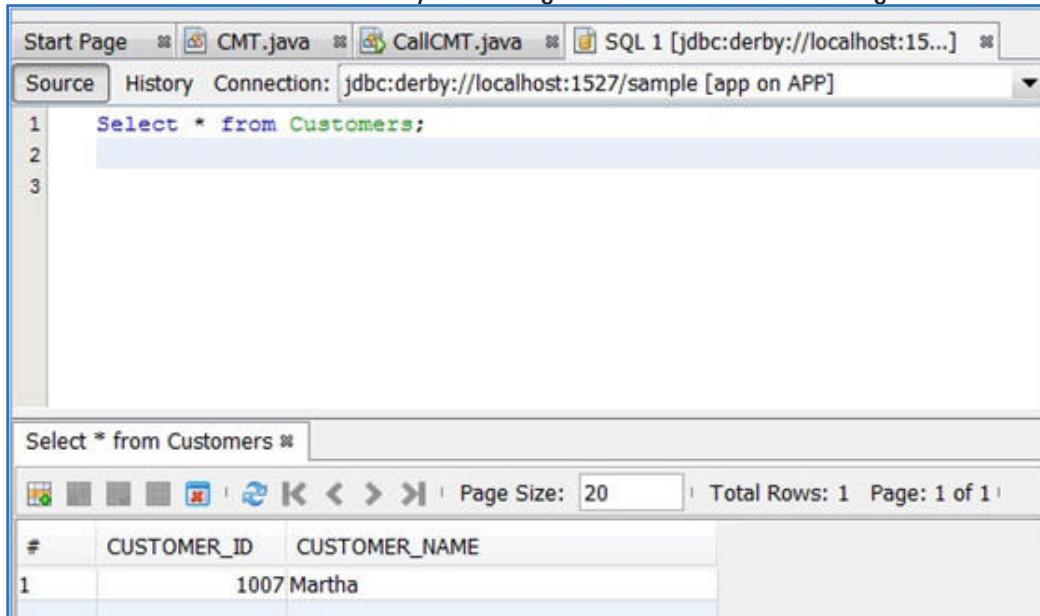


Figure 8.6: Transaction Execution on Database

Figure 8.6 demonstrates how the execution of the transaction can be checked on the database. Execute the SQL command 'Select * from Customers' in the editor provided on double-clicking the table 'Customers' to see the resultant.

A transaction committed by CMT can be rolled back in the following ways:

- If a system exception is raised, then the container automatically rolls back the transaction.
- If an application exception is raised, then the container will not rollback it automatically. The EJB must invoke the method `setRollbackOnly()` of the `EJBContext` interface to notify the container to roll back the transaction.

8.3.3 Developer Responsibilities

With CMT, setting up of the transaction attribute in the deployment descriptor is an important task for the developer. Besides that, the developer should also handle exceptions appropriately. If required, the developer can also make use of the `javax.ejb.EJBContext` interface's `setRollbackOnly()` method, if the transaction code throws any exception. The `setRollbackOnly()` method permanently marks the current transaction for rollback.

Figure 8.7 shows developer responsibilities.

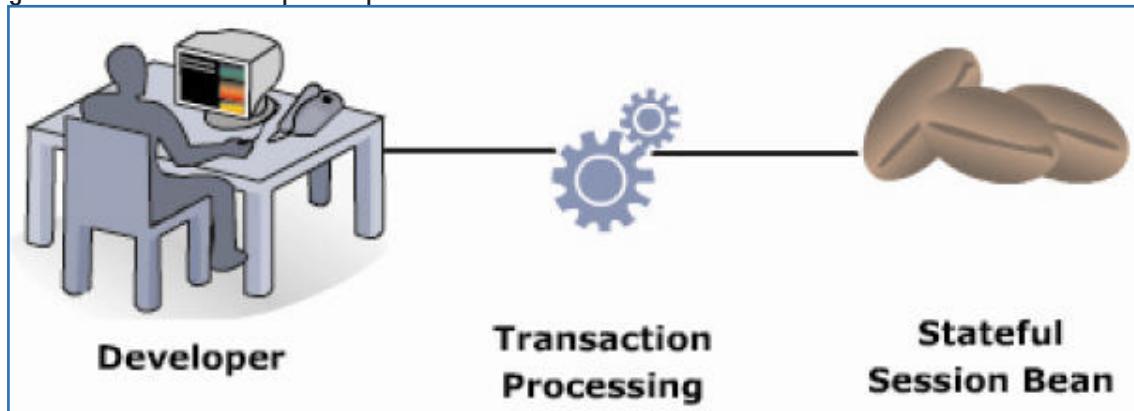


Figure 8.7: Developer Responsibilities

The developer responsibilities increase when handling transactions for stateful session beans. The stateful session bean's lifecycle is longer when compared to the stateless session bean. While stateless session bean can handle only a single client invocation, the stateful session bean handles multiple client invocations. During the creation of a stateful session bean, a developer can implement `javax.ejb.SessionSynchronization` interface, which provides methods for persisting the state at various stages.

The three main methods of the `javax.ejb.SessionSynchronization` interface that you can use for handling transactions in the stateful session beans are as follows:

□ **afterBegin()**

The `afterBegin()` method notifies the EJB instance that a new transaction has started. At this point, the method can save state information about the bean prior to the transaction commitment. The EJB instance can use this method to read data from the database and cache the data in the instance fields.

Syntax:

```
public void afterBegin() throws EJBException, java.rmi.  
RemoteException
```

where,

- `EJBException`: is raised to indicate a failure caused by a system-level error.
- `java.rmi.RemoteException`: is raised to indicate communication-related exceptions that occur during the execution of a remote method call.

- **beforeCompletion()**

The `beforeCompletion()` method notifies the EJB instance that a transaction is about to complete. The session bean can use this method to roll back the transaction, if required. The session bean can also use the `beforeCompletion()` method to update the database with the values of the instance variables.

Syntax:

```
public void beforeCompletion() throws EJBException, java.rmi.RemoteException
```

- **afterCompletion(boolean committed)**

The `afterCompletion()` method indicates that the container tried to commit the transaction, and whether it succeeded or failed. If a rollback occurred, the session bean can refresh its instance variables from the database in the `afterCompletion()` method. This method has a single boolean parameter whose value is true if the transaction was committed and false if it was rolled back.

Syntax:

```
public void afterCompletion(boolean committed) throws EJBException, java.rmi.RemoteException
```

Code Snippet 3 shows the example of a synchronized Stateful Session bean.

Code Snippet 3:

```
...
@Stateful
public class MyStatefulEJB implements SessionSynchronization {
    protected int total = 0; // actual state of the bean
    // value inside transaction, not yet committed
    protected int newtotal = 0; protected String clientUser = null;
    protected javax.ejb.SessionContext sessionContext = null;
    public void ejbCreate(String user)
    {
        total = 0;
        newtotal = total;
        clientUser = user;
    }
}
```

```
//Callback methods  
  
//Called just after the start of a transaction  
  
public void afterBegin() { newtotal = total; }  
  
//Called just before the end of a transaction  
  
public void beforeCompletion() { total = newtotal; }  
  
//Called just after the end of a transaction  
  
public void afterCompletion(boolean committed) {  
    if (committed) { total = newtotal; }  
    else { newtotal = total; }  
}  
  
public void buy(int s)  
  
{ newtotal = newtotal + s; return; }  
  
  
public int red() { return newtotal; }  
}
```

In Code Snippet 3, using the `afterBegin()`, `beforeCompletion()`, and `afterCompletion()` methods, the state of the bean is stored in `total` variable at various stages.

8.3.4 Container Responsibilities

The container's responsibilities, as far as the transactions are concerned, is based on the transactional state of the application and the transaction attribute of the method set in the deployment descriptor. The container can perform one of four things before executing the method:

- Continue the current transaction
- Suspend the current transaction and run the method without a transaction
- Suspend the current transaction and begin a new one
- Refuse to execute the method at all

At the end of a method call, the container will attempt to commit or roll back any transaction it started and resume any transaction it suspended.

8.4 Bean-Managed Transaction (BMT)

CMT have a limitation that a bean method executing can associate with a single transaction or no transaction. When the method has to associate with more than one transaction, then using bean managed transactions is appropriate. In bean managed transactions, the code in the session bean or Message-driven bean defines the scope of the transaction. Bean managed transactions provide better control on the execution of the application based on certain conditions in the application. The developer is responsible for starting the transaction and then committing or rolling back the transactions to end it.

Application managed transactions can be implemented through Java Database Connectivity (JDBC) or through JTA.

JTA allows demarcation of transactions that is independent of the transaction manager. JTA transaction allows the transaction to span across multiple databases and it is controlled through Java EE transaction manager. The Java EE transaction manager does not support nested transactions.

In order to use a bean managed transaction demarcation, an instance of the class `UserTransaction` is to be obtained. This object can be obtained through `getUserTransaction()` method. A `UserTransaction` object enables the developer to define the transaction boundaries.

When the transaction demarcation is programmed into the beans, the transactional behavior cannot be changed. However, this is possible in declarative transaction demarcation.

`UserTransaction` interface has various methods defined to manage transactions in case of bean managed transactions. Following are some of those methods:

- **begin** – begin method is used to start a transaction context.
- **setTransactionTimeout** – Transaction managers have to set a time limit on the transactions that are executing. The transaction must rollback if it does not complete before the timeout period expires. The timeout value is an integer measuring the time in seconds.
- **commit** – commit method is used to commit a transaction. If the method associated with the committed transaction fails for some reason, it will result in a **RollBackException**.
- **rollback** – this method is used to rollback the associated transaction.
- **setRollBackOnly** – this method modifies the method such that the transaction is finally rolled back.
- **getStatus** - this method returns the status of the transaction associated with the method.

8.4.1 Implementing BMT

In case of BMTed transaction, the bean client manages the transaction by explicitly initiating the transaction, managing it, and closing the transaction.

To convert the container managed transaction into a bean managed transaction, the annotation `@TransactionManagement(TransactionManagementType.CONTAINER)` needs to change to `@TransactionManagement(TransactionManagementType.BEAN)`.

The transaction must be accessed through a Web client such as a Servlet. Code Snippet 4 shows the servlet code, which explicitly initiates a transaction.

Code Snippet 4:

```
public class BMTServlet extends HttpServlet {  
    @EJB  
    Private BMT b;  
    @Resource  
    javax.transaction.UserTransaction utx;  
  
    protected void processRequest(HttpServletRequest request,  
        HttpServletResponse response)  
        throws ServletException, IOException {  
        response.setContentType("text/html; charset=UTF-8");  
        PrintWriter out = response.getWriter();  
        try{  
            /* TODO output your page here. You may use following sample code. */  
  
            out.println("<!DOCTYPE html>");  
            out.println("<html>");  
            out.println("<head>");  
            out.println("<title>Servlet BMTServlet</title>");  
            out.println("</head>");  
            out.println("<body>");  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
out.println("<h1>CUSTOMER DETAILS</h1>");

String driver = "org.apache.derby.jdbc.ClientDriver";
String url = "jdbc:derby://localhost:1527/sample";
String username = "app";
String password = "app";
Class.forName(driver).newInstance();

Connection conn = DriverManager.getConnection(url, username,
password);

System.out.println("Connection Done");

utx.begin();

out.println(BMT.add(conn));

utx.commit();

out.println("</body>");
out.println("</html>");
}

catch (Exception ex) {
    out.println("Error: Other exception - " +ex);
}
}
}
}

...
}
```

Code Snippet 4 shows a Web client initiating a transaction on the database. The Web client in this case is a servlet which uses the UserTransaction interface.

8.4.2 Client-Managed Transaction Demarcation

Besides BMT and CMT, the EJB specifications also make provision for transaction demarcations by the EJB clients. The clients may be JSPs, Servlets, or stand-alone clients. The process of implementing client-managed transaction demarcation is similar to BMT. The client gets a javax.transaction.UserTransaction object and calls the begin(), commit(), and rollback() methods as required. The container makes the UserTransaction available to the client using the JNDI lookup. Figure 8.8 shows client-managed transaction demarcation.

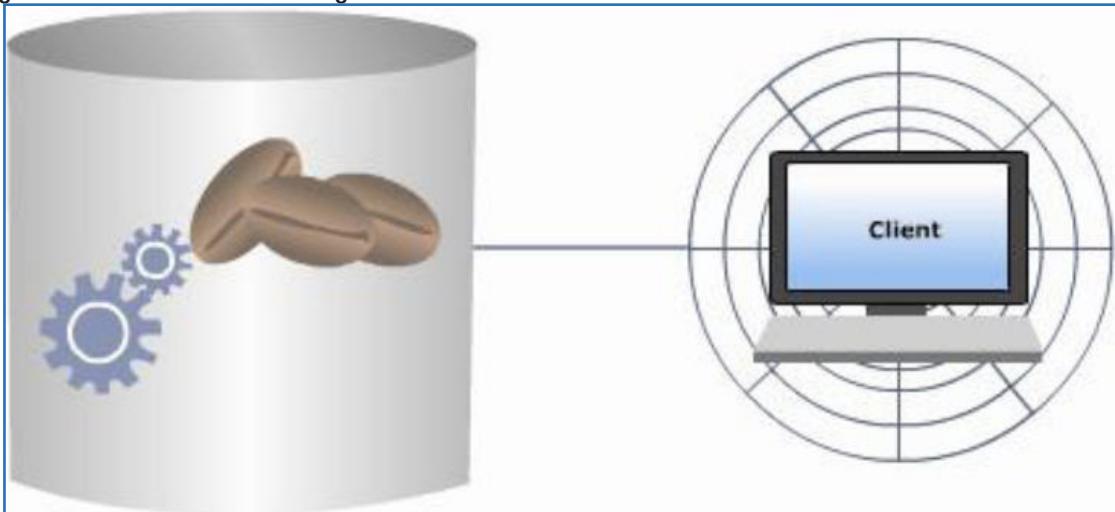


Figure 8.8: Client-Managed Transaction Demarcation

Code Snippet 5 demonstrates the JNDI lookup of javax.transaction.UserTransaction from the client.

Code Snippet 5:

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

8.5

Applying Transaction to Messaging

Transactions may also include operations such as sending and receiving messages in the application. Java applications use Message-driven beans and JMS services to send, receive, and process messages in the application. When the application has to send a message, the message is posted onto a message queue. These messages are not released to the consumer until the transaction commits.

Consider a scenario where an individual performs a cash withdrawal at an ATM. The application sends a message to the user's mobile and an e-mail informing about the transaction.

An ATM transaction includes the following operations:

- User logs in to the account using valid credentials
- Enters the amount to be withdrawn
- System performs a check in the account whether there is sufficient balance or not
- If there is sufficient balance the application directs the ATM to dispense the cash
- System sends a message to the user

There might be events such as the ATM machine does not have sufficient cash to dispense, in such scenario the transaction has to be rolled back. The message sent to the user has to be withdrawn from the message queue in such a case. This functionality in the application is implemented through JMS queues in Java applications.

Message-driven beans can work with both CMT and BMT. You can use both CMT and BMT to implement transaction on the `onMessage()` method. With CMT, you specify the transaction attribute in the deployment descriptor. Message-driven bean supports only two transaction attributes, Required and NotSupported. With BMT, you must specify transaction demarcation within the `onMessage()` method.

Figure 8.9 shows application of transactions to messaging.

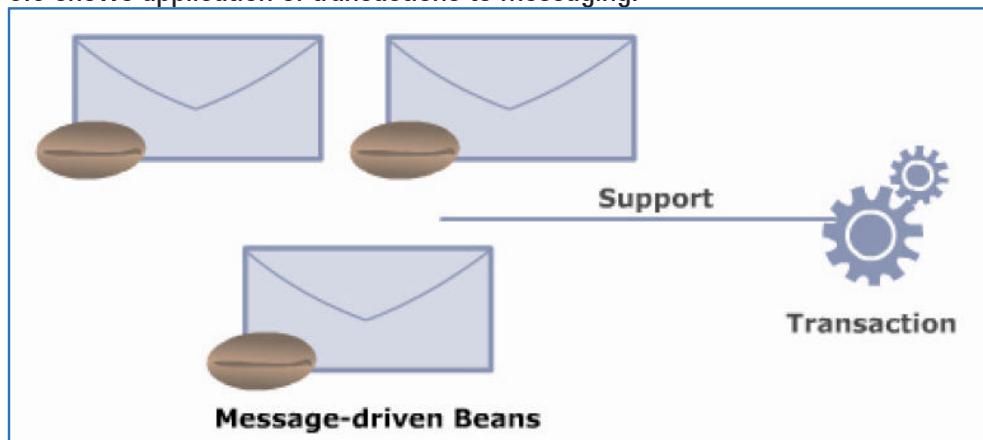


Figure 8.9: Applying Transactions to Messaging

8.6

Database Locking and Isolation

When multiple transactions are executing simultaneously, they may concurrently access application resources or data. The operations of different transactions are interleaved into a schedule. To ensure consistency of the application data, the outcome of the transaction schedule should be equivalent to the outcome when the transactions are executed serially one after the other. This property is known as transaction serializability, where the result of interleaved transactions is equivalent to the result of transaction execution in a serial order.

To ensure transaction isolation, locking mechanism is used. When a transaction acquires an exclusive lock on the data object, other transactions which are trying to access the data object are forced to wait.

These transactions are blocked. The transaction management module thus ensures transaction isolation. The blocked transactions can only read consistent data.

The locking mechanism acquires different types of locks on the application resources to achieve serializability.

There are two types of locks acquired by a transaction – a shared lock or read lock and an exclusive lock or write lock. When a transaction intends to access a resource for read only operations, it acquires a shared lock. Multiple transactions can hold a read lock simultaneously on a resource with a shared lock.

When a transaction intends to access a resource for modification then, it has to acquire a write lock on the resource. Only one transaction can modify a resource at any given time, therefore, only one transaction can hold the write lock on a resource for the acquired exclusive lock.

Figure 8.10 illustrates different types of locks acquired by transactions.

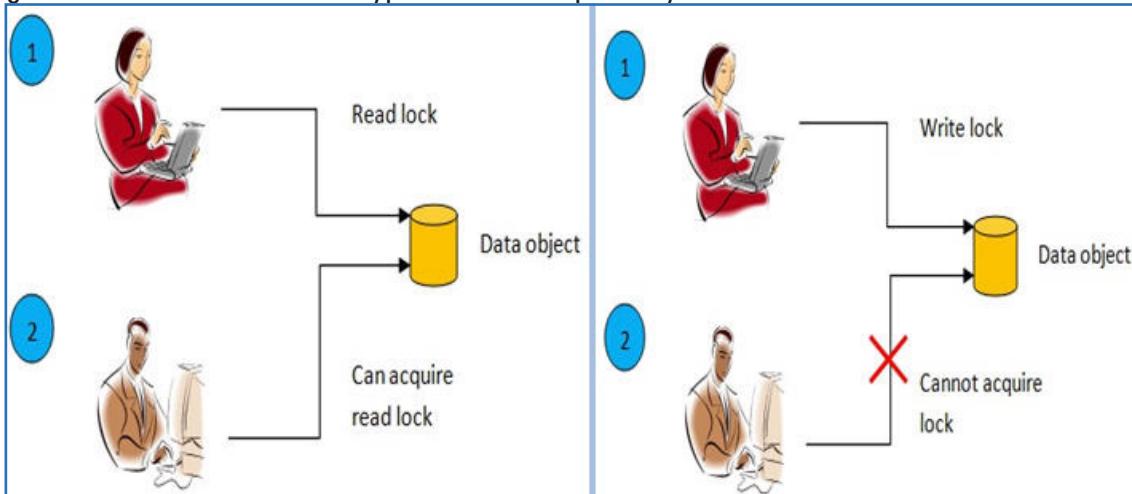


Figure 8.10: Shared and Exclusive Locks

Figure 8.10 illustrates how application clients can acquire read and write locks on data objects in the application. If a transaction is holding a read lock on a data object then, another transaction can also acquire a read lock on it. If a transaction is holding a write lock or exclusive lock on certain data object then other transactions cannot acquire a lock on the data object.

Transactions also use locking techniques such as optimistic locking and pessimistic locking according to the application requirement.

Optimistic locking mechanism does not acquire locks for executing a transaction, it accesses user data and performs the transaction and before committing the transaction it checks whether there is any change in data version since the data has been used. If there is a change in data, the transaction is rolled back else it is committed.

In case of pessimistic locking, the transaction acquires all the required locks, before starting the transaction and holds them until the transaction is committed. Pessimistic locking ensures that the transaction executes reliably without any conflicts.

Check Your Progress

1. Which of the following interfaces are used for bean managed transactions?

(A)	UserTransaction	(C)	TransactionManager
(B)	XAResource	(D)	None of these

2. Which of the following transaction attributes requires a new transaction scope in the absence of an existing transaction scope? **RequiresNew**

(A)	Required	(C)	NotSupported
(B)	Never	(D)	None of these

3. Container-managed transactions are defined through _____.

(A)	Deployment descriptor	(C)	Classes
(B)	Annotations	(D)	Both A and B

4. Which of the following provide the definition of a transaction manager?

(A)	Java Transaction API	(C)	Object Transaction Services
(B)	Java Transaction Services	(D)	All of these

5. Which of the following is not a method of UserTransaction interface?

(A)	commit	(C)	rollback
(B)	rollBackOnly	(D)	None of these

Answer

1.	A
2.	A
3.	D
4.	B
5.	B

Summary

- Transactions are a set of operations which have to execute as a single unit.
- Transactions should have properties such as atomicity, consistency, isolation, and durability.
- Transactions are supported in enterprise applications through Java Transaction Services and Java Transaction API.
- Transactions can be managed both declaratively and programmatically.
- Declarative transaction management makes use of annotations, deployment descriptors, and transaction attributes. They are also known as container-managed transactions.
- Explicit transaction management makes use of the JTA and its interfaces for transaction management, such transactions are also known as bean managed transactions.
- Transactions can execute along with Stateful, Stateless, and Message-driven beans.
- Messages in transactions can be handled through both container-managed transactions and bean-managed transactions.



Welcome to the Session, **Persistence of Entities**.

This session explains how enterprise application can access data from the database. It discusses the Java Connectivity Database (JDBC) API provided by the Java Standard Platform (Java SE). Further, the session discusses about the Object Relational Mapping (ORM) and its technologies to persist data in the database. The session introduces Java Persistence API (JPA). It explains the various interfaces provided by JPA and how are they used by application to communicate with the database.

In this Session, you will learn to:

- Understand how to use JDBC API for persisting data
- Explain Object Relational Mapping (ORM)
- Describe the ORM tools used for performing data persistence
- Explain Java Persistence API (JPA)
- Describe Entities and Entity Manager in JPA
- Understand how to manage entities using JPA
- Understand how persistent objects are mapped onto the database

9.1 Introduction

Enterprise applications whether big or small needs to persist the data in the relational database. A relational database system stores the data in the form of tables, these tables are known as relations. Each table has data organized in the form of rows and columns.

Java SE platform introduced JDBC API specification that allows the developer to persist the data of the object into the relational databases. JDBC API is a low-level API which runs as a service in the EJB container. Further, Java EE platform developed a component-based model for persistence of objects – Entity beans.

9.1.1 Persistence Using JDBC API

The Java SE platform uses JDBC API for persistence of data in the relational database. The JDBC API is a low-level API used by Java developers to store and retrieve data from the relational database through Structured Query Language (SQL). SQL is a command language used by databases to store and retrieve data to/from the relational databases.

Developers write SQL statements to perform operations on the relational database. JDBC API allows the developers of Java applications to establish a connection with the database using a driver. Then, the developer can write SQL statements to be executed in the relational databases. SQL statements on the database are executed with the help of the JDBC API.

The JDBC API defined on Java SE platform helps to access the data stored in relational database. The components provided by the JDBC are as follows:

- JDBC API** - Is a set of classes and interfaces which provide programmatic access to the relational databases from the Java application.

The two packages defined for JDBC API are as follows:

- **java.sql** - This package contains the API for storing and retrieving data from the relational database in the Java applications.

- **javax.sql** - This package contains the API for processing data from the relational database in the Java applications running on the middle-tier servers.

- JDBC Driver Manager** - The Driver Manager is a class defined in JDBC API. It is the backbone of JDBC architecture and connects the Java application to different types of JDBC drivers.

- JDBC Test Suite** - The JDBC driver test suite helps to check the connectivity of the JDBC drivers used in Java application.

- JDBC-ODBC Bridge** - The JDBC-ODBC is a Java software bridge driver which connects the JDBC API to the relational database through ODBC drivers. The ODBC binary code needs to be loaded on every machine using JDBC-ODBC bridge driver. This driver is useful in corporate networks and applications running in the middle-tier of the three-tier architecture.

There are different types of architectures supported by JDBC API to access the database. Figure 9.1 displays the two-tier architecture for accessing the database using JDBC API.

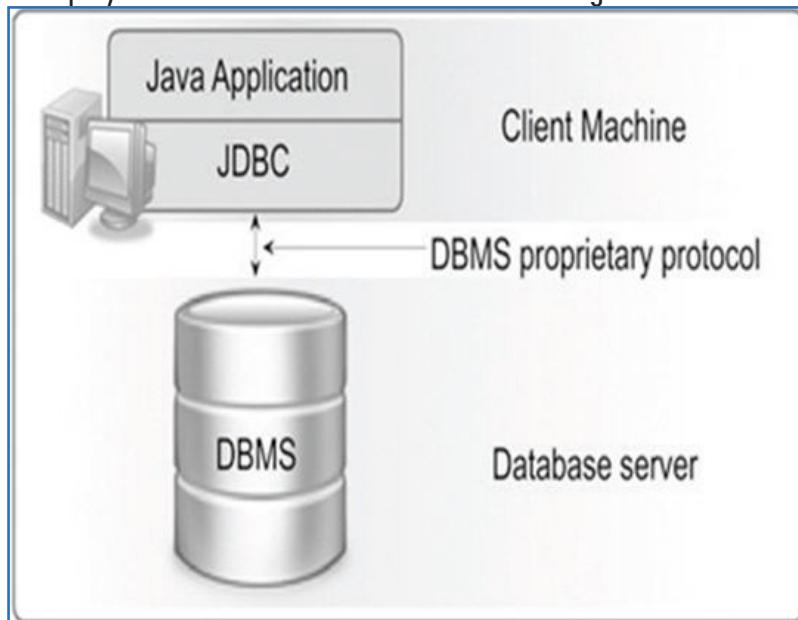


Figure 9.1: Two-tier Architecture

In two-tier architecture, a Java application can directly access the data source using JDBC API.

9.1.2 Implementing JDBC API

Following are the steps to process Structured Query Language (SQL) statements using JDBC API:

- **Load the Drivers** - A database can understand only SQL statements and not Java statements. So, a software component known as JDBC driver is used which helps the Java application to interact with the relational databases such as Oracle, Microsoft SQL Server, and so on. The JDBC driver provides connection to the database and helps in transferring queries and their results between Java application and relational databases. The different types of JDBC drivers are as follows:
 - JDBC-ODBC bridge driver
 - Native API party Java driver
 - NET-protocol All-Java driver
 - Vendor-specific driver
- **Establishing Connection** - In this a database connection is established with the data source which can be a Relational Database Management System (RDBMS), a legacy system or some other data source. The JDBC driver should be loaded and initialized before obtaining the connection with the database. To establish the connection, either the `DriverManager` class or `DataSource` class can be used. The `DriverManager`

is a fully implemented class which contains a method `getConnection()` to establish the connection with the database by accepting a connection Uniform Resource Locator (URL). The return type of the method is a `Connection` object. A `DataSource` object can also be used for establishing the connection by setting the properties related to a particular data source.

- **Creating statements** - There are different types of interfaces which can be used to represent SQL statement in the JDBC API. They are as follows:

- Statement Interface - It implements SQL statements as constants.
- PreparedStatement Interface - It implements pre-compiled statements with positional parameters to supply user specific input data.
- CallableStatement Interface - It executes stored procedures and function present in the database.

The `Connection` object creates the `Statement` object by invoking `createStatement()` method and `PreparedStatement` object by invoking the `prepareStatement()` method.

- **Executing the Query** - The different methods of `Statement` interface are used to execute different type of SQL statements such as Data Manipulation Language (DML) and Data Definition Language (DDL). The different methods are as follows:

- `execute()` - It returns more than one `ResultSet` object.
- `executeQuery()` - It returns a `ResultSet` object.
- `executeUpdate()` - It returns the number of rows affected by the SQL statement.

The return type of all the methods is a `ResultSet` object.

- **Processing ResultSet object** - The `ResultSet` object holds the data retrieved from the query executed in the database. It contains the column names and records retrieved from the query in tabular format. The obtained `ResultSet` object is further processed in the application using a cursor. Cursor is a pointer pointing before the first row of data in the `ResultSet` object.
- **Close the Connection** - The `close()` method in the `Statement` interface can be used to close the statement on completion. This method frees the resources and its corresponding `ResultSet` object is closed.

9.1.3 Limitations of JDBC API

The statements for establishing connection, performing updation, and executing queries are hard-coded in the programs using SQL statements. The biggest drawbacks of JDBC API are as follows:

- Developers have to manually map the object representation to the relational database model in the JDBC code.
- JDBC application contains large amount of code which are database specific.

Figure 9.2 displays the storage of object in the database using JDBC API.

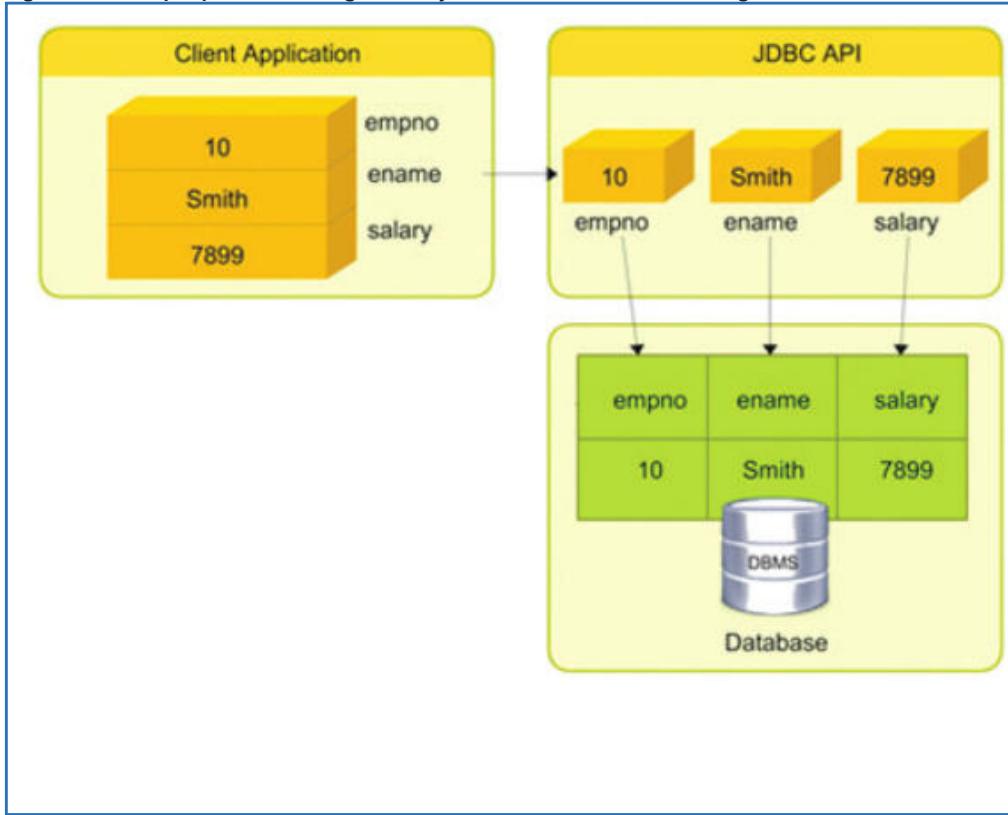


Figure 9.2: Storage of Object Using JDBC API

In figure 9.2, the data of the `Employee` object is persisted into the relational database table. The developer manually decomposes the object into parts such as `empno`, `ename`, and `salary`. These parts are then stored into each field of the relational table using SQL statements in the Java program.

9.1.4 Persistence with EJB 2.0

The EJB 2.0 version introduced Entity beans as Java objects that were basically designed to render themselves to persistent storage. In other words, an entity bean corresponds to a row in a relational database table. For each row in the table, an entity instance is created. In other words, an entity typically represents a row in the database table. Whenever store or retrieve operation is performed, the data from the database is loaded in the entity instance created in the memory. Any changes made by the developer on the entity bean instance gets reflected in the database.

Figure 9.3 shows the Entity bean model.

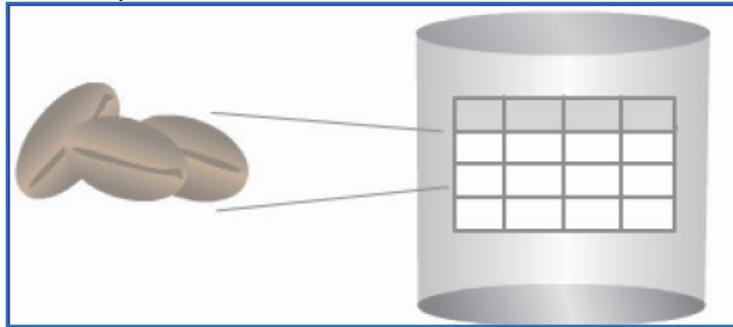


Figure 9.3: Entity Bean Model

The main features of entity beans are as follows:

Persistence

The entity bean stores the state in a data storage. Persistence refers to the fact that the entity bean's state exists beyond the lifetime of the application or the Application Server. The persistent data remains intact even after the storage shuts down.

Shared Access

Multiple users can use the same entity bean. To maintain data consistency, the entity beans work within a transaction, which is also a part of EJB package.

Primary Key

An entity bean supports a unique attribute. This unique attribute called a primary key in the database parlance enables the client to locate the exact entity bean.

Relationships

Like the data of a relational database, the entity beans are also related to each other. For instance, `EmployeeBean` and `SalaryBean` may be related by `employee_id` and one may contain the employee's personal details and the other may contain the employee's salary details.

9.1.5 Object-Relational Paradigm

To overcome the drawbacks of JDBC, a sophisticated method of persisting objects using Object-Relational Mapping (ORM) is used. ORM is an alternative technique used in the object-oriented world.

ORM technique automates the task of object persistence to the relational database without decomposing the object into parts. It decomposes the data from the relational database into Java objects.

To map objects to relation data, the object-relational mapping can be written manually in the mapping file. Alternatively, the object-relational mapping details can be generated through an ORM tool. There are different ORM tools available in the market such as Oracle TopLink, Hibernate, and so on that support object-relational mapping.

The ORM tool contains an automated mapper that generates the data definition of the target platform using DDL. These data definitions are generated either from the Java class or from a descriptor file.

Figure 9.4 displays the persistence of an object using an object-relational mapping tool.

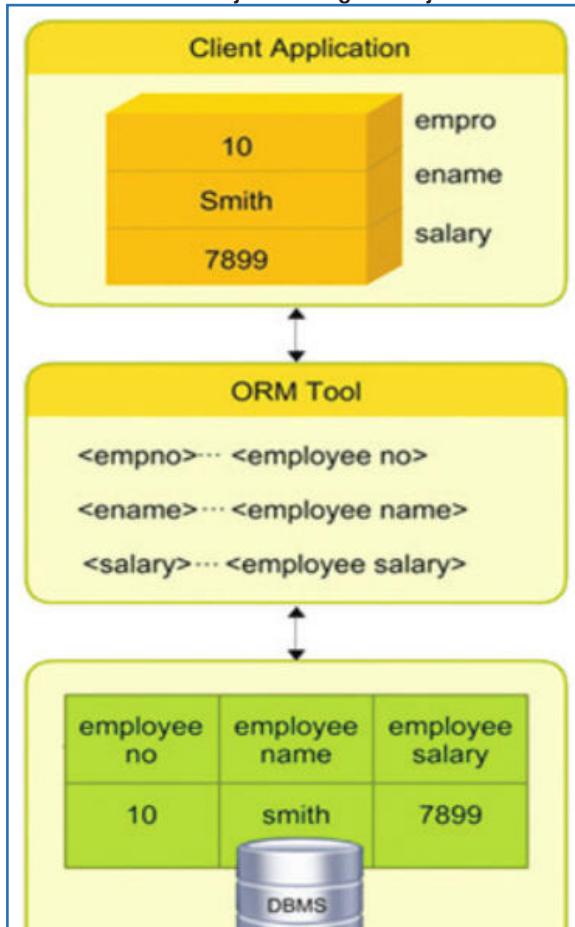


Figure 9.4: Object Persistence Using Object-Relational Mapping

Consider an instance of representing application data for HR domain. For example, the `Employee` class that is trying to represent the `Employee` objects to be persisted with the database can be designed as:

```
class Employee{
    int empno;
    String ename;
    float salary;
}
```

The class `Bank_Account` represents the properties associated with the object `Bank_Account` and methods associated with it to manipulate the bank account. When the same data is represented in relational model, it is presented as a table format as shown in figure 9.5.

Bank_Account		
account-number	account_holder_name	balance

Figure 9.5: Bank_Account Table

Relational model only represents the data of the application domain in a structured form known as relation. Relational model does not represent the operations associated with the data in the relation. All the operations associated with the database are performed through SQL. Any methods to be executed on the relational data are defined as procedures in the relational database.

Since there is a huge gap in data interpretation of both the data models, JDBC API uses different database drivers which are specifically written for certain types of database. The API then presents this data in a compatible format for Java applications.

Java objects are persisted onto permanent storage either through object serialization or by using a relational data model. The process of mapping the Java objects onto the relational database is known as object-relational mapping. Object-relational mapping converts the Java objects into relational data and data from the relational database into Java objects.

9.1.6 ORM Tools

There are various open source and commercial tools for object-relational mapping. Following are some of the ORM tools which can be used in Java applications:

- **Java Persistence API** – JPA is an API which manages persistence of Java objects by defining entity classes. It has an interface `EntityManager` which manages entities.
- **Java Data Objects** – Java data objects define object persistence through external XML meta files.
- **Hibernate** – It is an open source ORM framework which maps Java classes into database tables. It also enables database querying and retrieval facilities. It is capable of generating SQL calls and manages the result set of handling the queries.
- **EclipseLink** – It is also an open source ORM tool which enables the interaction between the application and various other data services such as databases, Web services, Object XML mapping, and Enterprise Information Systems.
- **Fast Java Object Relation Mapping** – It is a lightweight ORM tool. It does not support mapping of m:n relationships and n:1 relationships.
- **Java Object-Oriented Querying** – It is a database mapping software library which implements active record pattern.
- **ActiveJDBC** – It is an ORM tool base in active record pattern.



Active record pattern associates a database table with a class known as wrapper class, where each row of the table is associated with an object instance. The wrapper class implements all the methods required to access the database and properties of each row of the application.

9.2

Overview of Java Persistence API

JPA was introduced in Java EE 5 for the first time as part of EJB 3.0 specification. The basic function of this API is to map the relational data onto the Java objects, the storage of Java objects onto relational databases and retaining the state of the Java objects even after the application exits. It standardizes object-relational mapping in the applications. The current version of JPA in Java EE 7 is JPA 2.1.

JPA manages data by converting the Plain Old Java Objects (POJOs) into entities. These entities are mapped onto the relational database and managed through an entity manager in the application.

Java Persistence API (JPA) defines a standard mechanism of object-relational mapping. It provides specification compliant products known as ORM tools for object-relational mapping. While mapping the Java objects onto relational tables, each entity class is mapped onto the table and each property of the entity is mapped onto the column of the table.

The object-relational mapping should also define how the following tasks with respect to the application are performed:

- Inheritance among different entities of the application.
- Association of different entities, whether it is one-to-one relationship, one-to-many relationship, and so on.
- Manage applications' interactions with the database.



The concept of entities in JPA is different from Entity beans provided in EJB 2.0.

9.2.1

Entities

An enterprise application comprises two types of objects. Objects which implement the business logic of the application, these objects are known as enterprise beans. Objects whose state has to be stored onto the permanent storage are known as persistent objects.

These persistent objects are also known as entities according to Java Persistence specification. Entities store data as fields, methods are also associated with the entities. For instance, in case of banking application, `Bank_account` is an entity and it is associated with properties such as `account_number`, `account_holder_name`, and `account_balance`. These

properties are represented as data fields. Entities also have methods associated with it such as `getBalance()`, `getAccountHolderName()`, and so on.

An entity in the application domain represents a table in a relational database. Each row of data in the relation represents an entity instance in the application domain.

9.2.2 Requirements of Entity Class

While designating a class as an entity class, it has to satisfy following requirements:

- An entity class should have a default constructor whose access specifier is protected or public. Apart from a default constructor the entity class can have other constructors also.
- An entity class should be annotated with `javax.persistence.Entity`.
- Neither an entity class nor the methods and persistence variables of the entity class should be declared final.
- Whenever an entity instance is passed by value through business methods, then the entity class must implement `Serializable` interface.
- Like other Java classes, entity classes can also be inherited. An entity class can extend both entity and non-entity classes. A non-entity can also extend an entity class.
- Persistence instance variables can be prefixed with access modifiers such as `private`, `protected`, or `package private`.

Code Snippet 1 shows an entity class.

Code Snippet 1:

```
Package Manage;

import java.io.Serializable;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

public class Message implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id

    @GeneratedValue(strategy = GenerationType.AUTO)

    private Long message_id;

    private String text;
```

```
public Long getmessage_Id() {  
    return message_id;  
}  
  
public void setmessage_Id(Long id) {  
    this.message_id=id;  
}  
  
@Override  
public int hashCode () {  
    int hash=0;  
    hash+= (message_id != null ? message_id.hashCode () : 0);  
    return hash;  
}  
  
@Override  
public boolean equals (Object object) {  
    if (! (object instanceof Message)) {  
        return false;  
    }  
    Message other= (Message) object;  
    if ((this.message_id==null && other.message_id!=null) ||  
    (this.message_id !=null && !this.message_id.equals (other.  
    message_id))) {  
        return false;  
    }  
    return true;  
}  
  
@Override  
public String toString () {  
    return "Manage.Message [ id=" + message_id + " ]";  
}
```

```
public String getText () {  
    return text;  
}  
  
public void setText (String text) {  
    this.text = text;  
}  
}
```

9.2.3 Primary Keys in Entities

Every entity object in the application is identified through a unique identifier. This unique identifier is known as primary key, it is used to reference each entity of the application. The primary key for an entity can be a simple primary key or a composite primary key. A simple primary key uses only one property of the entity to identify it. In case of a composite primary key a combination of properties of the entity are used to uniquely identify an entity.

To represent a simple primary key, the primary key field is annotated with `javax.persistence.Id`.

Composite primary keys are defined in the primary key class. They are annotated with `javax.persistence.EmbeddedId` and `javax.persistence.IdClass`.

A primary key field (simple or composite) can have any one of the following data types:

- Primitive data types
- Java primitive wrapper types
- `java.lang.String`
- `java.util.Date`
- `java.sql.Date`
- `java.math.BigDecimal`
- `java.math.BigInteger`

A primary key class has the following requirements:

- The primary key class should be publicly accessed. It should be prefixed with a `public` key word.
- The properties of the primary key class should be `public` or `protected`.
- The primary key class must have a default constructor.
- The primary key class must implement `hashCode()` and `equals(Object other)` methods.
- The class must be serializable.
- A composite primary key must be mapped onto the properties of the entity class or created as an entity class in the embeddable class.
- When the primary key class is mapped onto the properties of the entity class then the corresponding name of the properties and type of the properties should be the same.

9.2.4 Comparison on Entity with Session Bean

Both entities and session beans are implemented through Java classes. They are instantiated by the application in the container.

Following are the differences between the two application components:

- Session beans are instantiated into the EJB container as per the application requirement.
The number of instances to be created is decided by the container into which these beans are deployed.
- In case of entities, the number of instances of the entity class is defined by the application domain. Every entity instance is identified through domain specific data such as primary key. This identity can be used to access the entity or pass it as a reference to other application components.
- Session beans can be accessed locally or remotely through local interface and remote interface respectively. Entities cannot be remotely accessed.
- Different entities can be compared with each other and can be referenced through their respective unique identities (primary key). This is not the case for session beans.
- The lifecycle of an entity is independent of the lifecycle of a session bean. Entities can exist even after the application is shut down, thus having a longer life than the session beans.
- Entities can survive critical application failures whereas session beans cannot survive.
The state of session bean is lost in case of system failure.

9.2.5 Packaging and Deploying Entity Classes

Entity classes are packaged as persistence units. A persistence unit refers to a set of logically connected entity classes and related configuration information on how the comprising entities are interrelated.

Persistence units are defined in a configuration file `persistence.xml`. This descriptor file is added to the `META-INF` directory of the application. A `persistence.xml` file is essential for creating persistence units in an application.

Code Snippet 2 shows the definition of a persistence unit in the `persistence.xml` file.

Code Snippet 2:

```
<persistence>
<persistence-unit name="AccountOperation">
<description>This unit manages accounts of customers when
the customer has multiple accounts this unit links these accounts
</description>
<jta-data-source>jdbc/MyAccounts</jta-data-source>
```

```
<jar-file>MyAccount.jar</jar-file>
<class>FixedDeposit</class>
<class>SavingsAccount</class>
<class>CurrentAccount</class>
</persistence-unit>
</persistence>
```

Code Snippet 2 shows the code of a sample persistence unit in `persistence.xml` file. The persistence unit manages multiple accounts of the customer. The relational database source for this persistence unit is located in `jdbc/MyAccounts` which is identified through the `<jta-data-source>` element. The `<description>` element is used to provide comments about different aspects of the persistence unit. The `<jar-file>` element is used to represent the archive file created for the Java application. The `<class>` element represents the class files which operate upon the persistence unit.

Apart from the elements shown in Code Snippet 2, the descriptor file can also have the following elements:

- `<provider>` which defines the persistence provider for the application. A persistence provider is responsible for managing permanent storage of data for the application.
- `<transaction-type>` defines the type of the transaction. This element can assume a value which is either JTA or RESOURCE _ LOCAL.
- `<mapping-file>` Object-relational mapping for the entity classes can be specified through annotations or through the `orm.xml` file. The object-relational mapping can be specified through more than one mapping file. These mapping files are specified through `<mapping-file>` element.

9.2.6 Persistence Provider

Persistence provider refers to a third party technology which can be used to manage application data objects on the disk. Usually a persistence provider is a database provider such as MySQL, Oracle, and so on. Other than databases, data objects can be persisted onto flat files also.

9.3 Managing Entities

Entities are managed through an instance of `EntityManager`. Each persistence unit is managed by an `EntityManager`. `EntityManager` is an instance of `javax.persistence.EntityManager`. An `EntityManager` is responsible for managing activities associated with a set of entities such as persisting the entity, removing an entity, and querying the entity. Every `EntityManager` instance is associated with a `PersistenceContext` instance.

9.3.1 Persistence Context

Persistence context is responsible for keeping track of the state changes associated with an entity object. When an `EntityManager` instance is created it is associated with a persistence context. The persistence context refers to the set of entities existing in the application environment when the `EntityManager` is instantiated.

Following are the two variants of the persistence context:

- Transaction scoped persistence context
- Extended persistence context

Transaction scoped persistence context refers to a set of entities associated with a transaction. The state of these entities may change during the course of transaction. If the transaction commits, the changes made to these entities are persisted. If the transaction rolls back, the changes made to the entities associated with the transaction are rolled back.

The lifetime of the persistence context is dependent on the lifetime of the transaction. Once the transaction commits and closes, any changes made to these entities are not persisted.

An extended persistence context is not dependent on the transaction. It is associated with the `EntityManager` instance and is managed manually by the application code.

9.3.2 EntityManager Interface

The `EntityManager` API is responsible for the following operations on the entities:

- Create and remove persistent entity instances.
- Find entities based on the primary key.
- Allow queries to run on entities.

Entity managers can be created and removed either through container or through application code.

9.3.3 Container Managed Entity Managers

When the `EntityManager` is instantiated by the container, it defines the `PersistenceContext` of the entity manager. The container shares this `PersistenceContext` with all the application components deployed in the container.

Any transaction in the application which involves different application components can therefore share the same persistence context.

The lifecycle of the `EntityManager` is also managed by the container.

9.3.4 Application Managed Entity Managers

In case of application managed entity managers, the instantiation of the entity manager, propagating the persistence context to the entity manager, and other tasks of the entity manager should be explicitly managed by the application code.

Application managed entity managers are used when the persistence context is not to be shared among all the application components. When an entity manager is created as application managed entity manager, it creates an isolated PersistenceContext.

Creating an EntityManager instance is not thread-safe. Therefore, applications create EntityManager instance from an EntityManagerFactory class. The createEntityManager() method of EntityManagerFactory class creates a thread-safe EntityManager instance.

An instance of the EntityManager can be obtained through the following statement:

```
@PersistenceUnit  
EntityManagerFactory E;  
EntityManager EM=E.createEntityManager();
```

Code Snippet 3 shows the creation of an EntityManager instance.

Code Snippet 3:

```
package Access;  
  
import Entity.Message;  
  
import javax.ejb.Stateless;  
  
import javax.persistence.EntityManager;  
  
import javax.persistence.EntityManagerFactory;  
  
import javax.persistence.Persistence;  
  
import javax.persistence.PersistenceContext;  
  
@Stateless  
  
public class MessageFacade extends AbstractFacade<Message>  
implements MessageFacadeLocal {  
  
    @PersistenceContext(unitName = "MessageStore-ejbPU")  
    private EntityManager em;  
  
    @Override  
  
    protected EntityManager getEntityManager () {  
  
        return em;
```

```
}

public MessageFacade() {
    super(Message.class);
}

public static void main() {
    EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("jdbc:derby://
localhost:1527/sample[app on APP]");
    EntityManager e = emf.createEntityManager();
}
}
```

In Code Snippet 3, the instance can further manage the entities created during application execution.

9.3.5 Obtaining a Persistence Context

`PersistenceContext` refers to a set of managed entity instances which are managed by an `EntityManager` object. In a relational database, persistence context refers to the set of rows which are actively accessed. Every `EntityManager` is associated with a persistence context to manage a set of entities.

There are two variants of persistence context – Transaction scoped persistence context and Extended persistence context.

In case of transaction scoped persistence context, the existence of the persistence context is dependent on the life of the transaction. A set of entities used in the transaction are bound with a `PersistenceContext` when the transaction is created. A new `PersistenceContext` is created when the `EntityManager` is invoked by an active transaction. The persistence context writes the changes made by the transaction to the permanent storage. When the transaction ends the set of entities are unbound from the `PersistenceContext`, therefore the changes made to the entities are not persisted to the permanent storage.

The `PersistenceContext` is defined through the annotation `@PersistenceContext`. The following statement defines the `PersistenceContext` as the set of entities in the persistence unit:

```
@PersistenceContext(name = "PersistenceUnitName")
```

Extended persistence context can be initiated in the scope of a Stateful Session bean. The `PersistenceContext` comes to existence whenever the application code defines a dependency on the entity manager. All the entities managed by the entity manager are bound with the persistence context. The dependency on the `EntityManager` is declared through the annotation `@PersistenceContext`.

This PersistenceContext is retained as long as the stateful session bean is in the container. When the Stateful Session bean is removed, the PersistenceContext is also removed. A PersistenceContext can be injected into the stateful session bean through @PersistenceContext annotation.

In Code Snippet 3, the PersistenceContext is injected as follows:

```
@PersistenceContext(unitName = "MessageStore-ejbPU")
```

All the entities on the MessageStorePU persistence unit are part of the PersistenceContext.

9.4

Managing the Lifecycle of an Entity Instance

The lifecycle of an entity instance is managed by the entity manager. EntityManager associates the entity with a PersistenceContext. An entity instance can exist in any one of the following states:

- New** – When the entity instance is in this state, it does not have any persistence identity. It is also not associated with a persistence context. The EntityManager associates the entity with the PersistenceContext.
- Managed** – When the entity instance is in a managed state then the entity is associated with a PersistenceContext and is associated with an identity.

A 'New' method can transform from 'new' state to 'managed' state when persist() method is invoked by the EntityManager.

- Detached** – An entity instance is said to be in a detached state if it is not associated with the PersistenceContext.
- Removed** – An entity instance is said to be in removed state when the entity is scheduled to be removed from the data store. The entity instance transforms to 'removed' state when the entity manager invokes a remove method on the managed entity.

9.4.1

Lifecycle Callback

Figure 9.6 shows the transition from one state to another state of the entity instance.

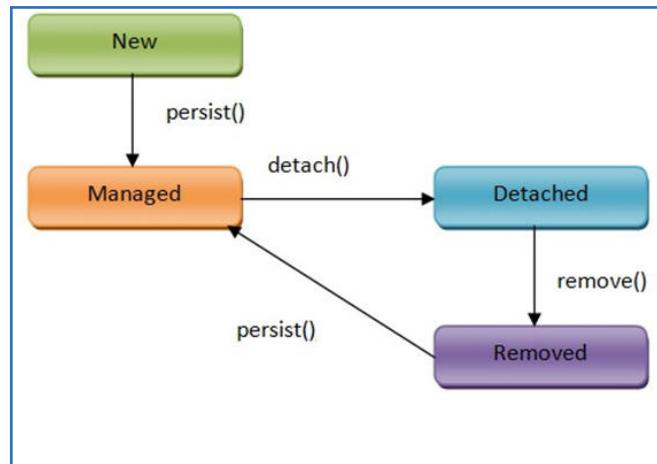


Figure 9.6: Lifecycle of an Entity Instance

Figure 9.6 demonstrates the lifecycle of an Entity instance. An Entity instance moves from one state to another state by executing lifecycle callback methods. The lifecycle callback methods used by an Entity instance are explained in detail.

9.4.2 `persist()` Method

- When `persist()` method is invoked on an entity in 'new' state, then the entity is assigned an identity. On invoking the `persist()` method the entity is stored onto the database based on the type of persistence context in use.
- If the `persist()` method is invoked on a method in 'managed' state then the method is ignored. If the current entity references other entities in the application then the `persist()` method is cascaded to the referenced entities. The option of how the persist operation has to be cascaded to the referenced entities is defined in the deployment descriptor of the application.
- If the entity instance is in 'removed' state and `persist()` method is invoked then the entity transforms to 'managed' state.
- If the entity instance is in detached state and `persist()` method is invoked then it results in an `EntityExistsException`.



Any lifecycle callback method can be cascaded in two ways. The options for cascade operation are `cascade = persist` and `cascade = ALL`. When the option is `persist`, only `persist` operation is cascaded among different entities. When the option is `ALL`, all operations on the entity are cascaded onto the referenced entities.

9.4.3 `remove()` Method

- When `remove()` method is invoked on an entity in 'new' state then it is ignored, however any other entities referenced by the current entity are removed. This removal is dependent on the option provided for the cascade operation.
- When `remove()` method is invoked on the entity instance when it is in 'managed' state then the entity transforms to removed state. This operation is cascaded onto all the referenced entities based on the options provided for the cascade operation in the deployment descriptor.
- When `remove()` method is invoked on an entity in 'detached' state then it throws an `IllegalArgumentException`.
- If `remove()` method is invoked on an entity in 'removed' state then the method invocation is ignored.

9.4.4 `detach()` Method

`detach()` method is used to remove an entity from the persistence context. If the entity is already detached from the persistence context, then it will throw an `IllegalArgumentException`.

9.5 Mapping the Persistent Objects

Enterprise applications store persistent data objects in relational databases. In the application the data objects are created as entity classes. The instances of the entity classes are termed as entities. These entities are persisted onto the relations of a relational database. Data objects are usually persisted onto relational databases because relational database model is the most widely used database and supports huge volumes of data as required by the enterprise applications.

The entity classes encapsulate various properties of the entity and behavior of the entity in the application domain. The entity classes are plain Java classes. The instances of the entity class are persisted onto the database by the entity manager. The entity manager creates a new row in the relation and then stores the data of the entity onto the relation. The entity manager should also manage the updates and modifications made to the entity during the application execution.

Code Snippet 4 shows the code for a `BankAccount` entity.

Code Snippet 4:

```
...
@Entity
public class BankAccount implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id=id;
    }
}
```

Code Snippet 4 consists of an entity class `BankAccount`. Whenever, an entity is created it is in combination with a unique identifier. This unique identifier can be used as the primary key of the entity or the developer can create another attribute which can act as the primary key.

Three annotations are used in this code and they are:

- `@Entity` – The entity annotation represents the entity class that has to be persisted on a relational database.
- `@Id` – The id annotation represents the attribute of the entity which can be used as a primary key.
- `@GeneratedValue` – `GeneratedValue` annotation is used in conjunction with the `Id` annotation or along with any column annotation. It defines the strategy according to which the values in the column can be generated. As in Code Snippet 4, if the generator value is set to be `AUTO`, it implies that the value is either set to a default value or to auto increment for the column in the table definition.

In this scenario, the entity class `BankAccount` is mapped onto a relation in the database. The `Id` attribute is mapped onto a column in the table. This column is defined as the primary key of the relation.

Java Persistence allows object relational mapping in both the directions. Applications can start from the Java object model and derive the database schema. Similarly, applications can start from database and derive the Java object model.

9.5.1 Mapping of Entities onto Database Through XML Files

Apart from the annotations used in the entity classes, the object-relational mapping can be carried out through XML mapping files. The persistence provider looks in the `META-INF` directory of the enterprise application for the '`orm.xml`' file. Entity mapping can be provided through other `.xml` files also. The mapping file should be defined through the `<mapping-file>` element in the `persistence.xml` deployment descriptor.

Code Snippet 5 shows entity mapping of the `BankAccount` entity.

Code Snippet 5:

```
<entity-mappings>
  ...
  <entity class="BankAccount" name="BankAccount">
    <table name="Account"/>
    <attributes>
      <id name="Account_number">
        <generated-value strategy="TABLE"/>
      </id>
      <basic name="account_holder_name">
        <column name="acc_hld_name" length="100"/>
      </basic>
    </attributes>
  </entity>
</entity-mappings>
```

```

</basic>
<basic name="acc_balance">
</basic>
</attributes>
</entity>
</entity-mappings>

```

Following are the XML elements used to define entity mappings onto the database:

- <entity-mappings> element is the top element within which all the entity mappings are defined.
- <entity> element is used to define the entity which has to be mapped onto a table in the relational database. The entity element has the name of the entity class in the application which will be mapped onto to a table in the database.
- <attributes> element has the properties of the entities which are supposed to be mapped onto the columns of the table of the database.
- <id> element can be used along with the <attributes> element to identify the column which has to be designated as the primary key in the table.
- <generated-value> strategy specifies the method of generating the values of the primary key column. The value can be defined explicitly while instantiating the entity or they can be auto incremented based on some seed value.
- <basic> element is used to define mapping of entity properties onto the columns of the table.

9.5.2

Mapping of Entities to Database Through Annotations

The developer can define every detail of the entity class mapping onto the database through annotations. Following are the annotations used to define the mapping of the entity classes onto the database:

- @Table

The @Table annotation is used by the EntityManager to identify the table of the database onto which the entity of the annotated class has to be mapped.

The annotation can be used as follows:

```

@Entity
@Table(name = "BankAccount")

```

The @Table annotation can have four attributes – name, catalog, relational schema, and uniqueconstraints.

The catalog attribute is used to identify the relational table catalog in which the table is located. Similarly, the schema attribute is used to identify the relational schema to which the current table belongs.

The `uniqueconstraints` attribute can be used to define the requirement of unique values in certain columns of the table. According to the application domain requirement, certain columns in the table should only have unique values. This requirement is defined through the `unique constraint` attribute.

`@Column`

`@Column` annotation is used to define how certain property of the entity class can be mapped onto the column of a table in the database.

The `column` annotation has following attributes, which provides the `EntityManager` with details of how to map the property onto the database.

- `name` attribute defines the name of the column onto which the current property of Java entity has to be mapped.
- `unique` attribute can assume a true/false value. If the attribute value is set to true then a unique constraint is applied on the column of the table.
- `nullable` attribute can assume a true/false value. When the `nullable` attribute is set to true then it implies that the column can have null values otherwise the column cannot have null values.
- The attributes `insertable` and `updatable` implies that the columns do not have an auto increment or preset values. The values into the column can be explicitly inserted or updated. The primary key columns are not generally updatable as changing the value of primary key can lead to inconsistent state of the database.
- The `columnDefinition` attribute can be used to define the type of column.
- The `length` attribute is used along with the `columnDefinition` attribute. If the column is defined to be of `VARCHAR` type then the length defines the length of the string that can be inserted into the column.
- `scale` and `precision` attributes can be used if the column is of `INTEGER` type.

9.5.3 Mapping Primary Keys onto the Database

Primary keys are used to uniquely identify the entities of the class. It is an essential element of any table or relational database as it plays a crucial role in data retrieval from the database. In object model of Java application, primary key can be implemented as one property of the entity class, a set of properties of the entity class or as a primary key class.

When the primary key is implemented as a single property of the entity class then the property is annotated with `@Id` annotation.

When the entity class requires persistence provider generated keys for the primary key value, then `@GeneratedValue` annotation should be used in addition to the `@Id` annotation.

Primary keys can be defined as a single attribute or through a group of attributes. Primary keys defined through a group of attributes are known as composite primary keys. Such primary keys are

implemented as primary key classes. Primary key classes can also be implemented as embedded classes.

Primary key classes are annotated with `@IdClass`, when implemented as embedded classes the primary key classes are annotated with `@EmbeddedId`.

When defining a primary key class for an entity, the properties of the entity which will be part of the primary key are annotated with `@Id`. The primary key class should be annotated with `@IdClass`. The type properties of the primary key class should be same as the type of the corresponding entity class properties. The number of properties in the primary key class should also match on the number of properties in the corresponding entity class.

Code Snippet 6 demonstrates the usage of primary key classes.

Code Snippet 6:

```
public class BankAccount_PK{  
    private long Account_number;  
    private String acc_hld_name;  
}  
  
@Entity  
@IdClass(BankAccount_PK.class)  
public class BankAccount{  
    @Id  
    private long acc_num;  
    @Id  
    private String account_holder;  
    ...  
}
```

Code Snippet 6 shows a primary key class named `BankAccount_PK`, which is a primary key class for the entity `BankAccount`.

The entity class `BankAccount` identifies the primary key class through annotation `IdClass` and defines the mapping attributes.

Primary key auto generation is not applicable when the primary key is implemented as a class.

Check Your Progress

1. Which of the following is a Persistence provider in enterprise applications?

(A)	Oracle	(C)	JDBC
(B)	JPA	(D)	EntityManager

2. Which of the following is an invalid access modifier to the Primary key class?

(A)	protected	(C)	private
(B)	public	(D)	None of these

3. Which of the following annotations are used in conjunction with the @Id annotation?

(A)	@Entity	(C)	@EmbeddedClass
(B)	@IdClass	(D)	@GeneratedValue

4. Which of the following is not an attribute of @Table annotation?

(A)	uniqueconstraints	(C)	relation schema
(B)	name	(D)	primary key

5. Which of the following are false about remove() method?

(A)	remove() method when invoked on a 'New' method then it is ignored.	(C)	a remove method invoked on an entity in detached state is ignored.
(B)	a remove() cannot be invoked on an entity in managed	(D)	None of these

6. Which of the following is a lifecycle callback method for an Entity?

(A)	remove()	(C)	persist()
(B)	detach()	(D)	All of these

Answer

1.	A
2.	C
3.	D
4.	D
5.	C
6.	D

Summary

- Java applications use Java Database Connectivity (JDBC) and Java Persistence API (JPA) to connect to the database.
- JPA defines a standard mechanism for object-relational mapping.
- JPA interprets the database objects in the application as entity classes.
- JPA defines an EntityManager to manage the entities in the application.
- The primary key for the entity objects can be implemented as an attribute in the entity class, as a primary key class or as an embedded class.
- orm.xml and persistence.xml files have the entity mapping information in the application.
- Persistent objects can be mapped using the declarative XML files.
- The persistent objects can be mapped onto the persistence unit by adding appropriate annotations to the Java classes.

GROWTH
RESEARCH
OBSERVATION
UPDATES
PARTICIPATION





Welcome to the Session, **Advanced Persistence Concepts**.

This session explains the techniques to model the objects in the object-oriented programming. It also explains the concepts of data modeling. Then, the session explains how to manage relationship in JPA. It explains different types of relationships managed in JPA and their directionality. The session explains how to adopt different strategies to map inheritance relationship to a relational database.

In this Session, you will learn to:

- Explain how relationships are managed in OOP
- Explain how relationships are managed in relational databases
- Explain cardinality and directionality in object relationships
- Explain how relationships are managed in JPA
- Describe annotations provided by JPA to create object relationships
- Describe the mapping of different aspects of database to the enterprise application
- Describe the different JPA strategies to map inheritance in relational databases
- Explain implementation of inheritance among the entities

10.1 Introduction

Entity beans in an enterprise application usually relate to one another. For instance, the Student entity bean is related to the Teacher entity bean in an enterprise application because the students are taught by the teacher.

Implementing the relationship between the student and teacher is handled differently by application developers and database designers. Today, most of the languages are based on object-oriented approach, which deals with objects and classes. The relational databases deal with tables and key constraints to establish the relation between the data stored in the tables. Hence, the concepts and techniques used for managing relationships will also differ.

10.2 Relationships in Object-oriented Programming

An object by itself cannot perform any function. It needs to interact with other objects to represent some concrete function. There are three kinds of relationships between objects: Association, Inheritance, and Aggregation.

- Association is when an object makes use of another object's operations. This is the simplest of all relationships.
- Inheritance is when one object is derived from another object. The derived object contains all the attributes and operations of the object it is derived from as well as some attributes and operations of its own. An object can be derived from a single object or multiple, if the implementation language supports.
- Aggregation is when one object contains another object. In this case, the contained object appears as an attribute of the container object.

Relationships describe how objects collaborate with one another, to contribute to the behavior of the system. For instance, a motorcar is composed of various components and the various components collaborate to move the car.

The features of a relationship include the operations that an object can perform on another, the result that an object expects from the relationship, and the nature of the relationship.

Nature of a relationship can be broadly classified as, link and aggregation. A link refers to a connection between two objects and aggregation refers to one object contained within another object.

Figure 10.1 depicts relationships between objects.

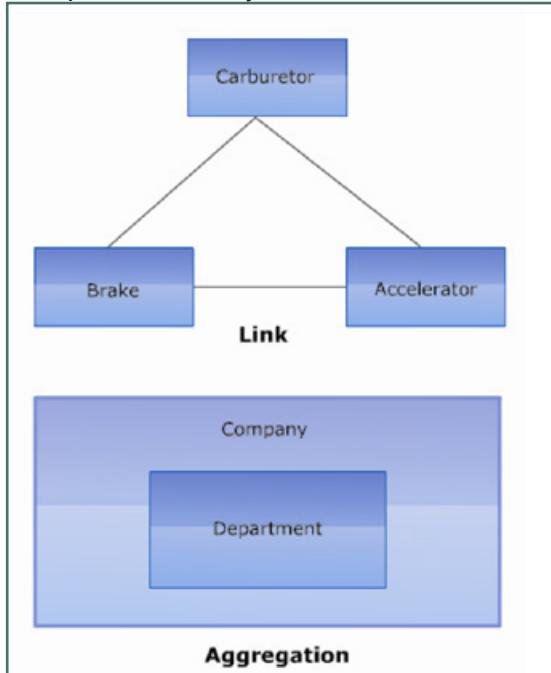


Figure 10.1: Relationships

In generalization relationship, one object or subtype is based on another object or super type. You can add Generalization relationships to reuse the attributes, operations, and relationships present in the super type with one or more subtypes. As the sub type inherits the attributes, operations, and relationships of the super type, you must only define those attributes, operations, and relationships in a subtype that are different from the super type.

The UML graphical representation of Generalization is a hollow triangle shape on the super type end of the line that connects it to one or more subtypes. The Generalization relationship is also known as inheritance or 'is-a' relationship.

Figure 10.2 depicts inheritance.

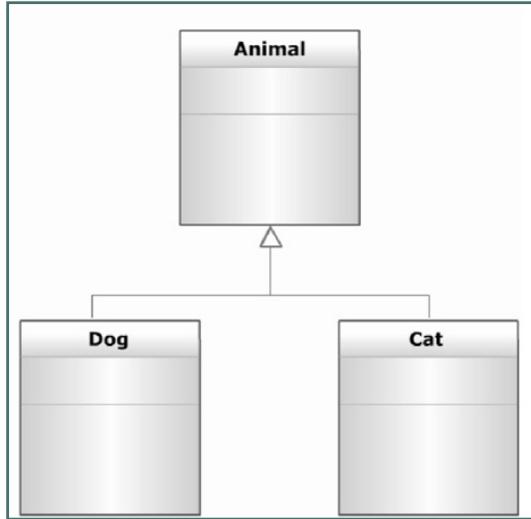


Figure 10.2: Inheritance Relationship

10.2.1 Object Modeling

Class diagrams are one of the most widely used diagrams amongst all the UML diagrams used for depicting objects and relationships in an object model. A class diagram follows a simple notation. A class is represented as a rectangle divided into three sections. The topmost section contains the name of the class. The middle section contains a list of attributes, and the bottom section contains a list of operations. Access modifiers can also be represented in a class diagram. The '+' symbol is used for public, '-' is used for private, and '#' is used for protected.

In many diagrams, the bottom two sections and their corresponding details are omitted. Even when they are present, they typically do not show every attribute and operations. The goal is to show only those attributes and operations that are useful for the design.

Figure 10.3 depicts a class diagram.

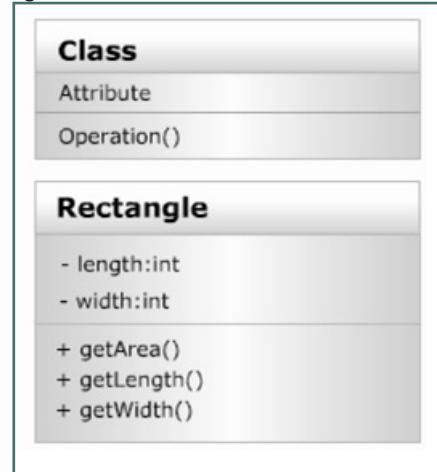


Figure 10.3: Class Diagram

10.2.2 Multiplicity in Relationships

In a class diagram, relationships between objects are shown using connectors. Connectors have different kinds of ends such as a diamond, an arrow, and others. The line ends depict the nature of the relationships. You can also depict multiplicity in relationships using the following notations at each ends of connectors:

- 0..1 No instances, or one instance
- 1 Exactly one instance
- 0..* or * Zero or more instances
- 1..* At least one instance

Figure 10.4 depicts multiplicity.

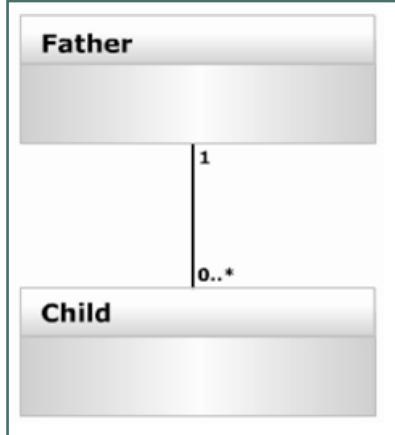


Figure 10.4: Multiplicity

10.3 Relationships in Database

A database schema describes the table structure, data types, and relations in a database. A schema can be a graphical layout of a database with all the tables, fields, primary keys, and the relations marked. A database schema can also be a series of Structured Query Language (SQL) statements. These SQL statements contain CREATE statements that help to create tables from scratch and populate them with data. A database schema is mostly used for recreating the data structure from scratch when the database has to be moved to a new server.

Figure 10.5 shows a database schema.

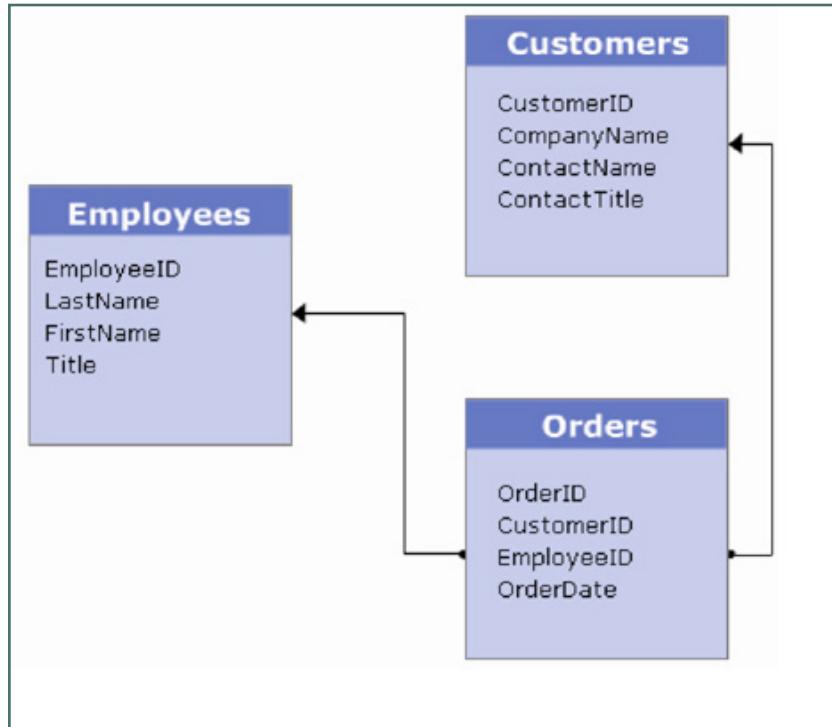


Figure 10.5: Database Schema

Some of the common terms with storing data in table are as follows:

Attributes

An entity will have one or more attributes. An attribute is any detail that identifies, qualifies, classifies, quantifies, or otherwise expresses the state of an entity instance. Attributes are specific pieces of information about an entity which needs to be known or stored.

For example, attributes for a Song entity are: title, date of release, singers, music directors, and album. From these attributes, only title and album are mandatory; others are optional. Figure 10.6 lists the attributes of an entity **Song**.



Figure 10.6: Attributes

Primary Key

An attribute which uniquely identifies the occurrence of an entity is called the primary key. A primary key is a single attribute or a combination of attributes that uniquely defines an entity instance. A primary key is always a mandatory attribute of an entity. An entity cannot exist without the primary key. If such an attribute does not exist naturally, a new attribute is defined for that purpose, for example, an ID number or code. For a Song entity, the song title cannot be the primary key as there may be more than one song with the same title, so it requires an ID attribute.

Figure 10.7 highlights the primary key of the entity Song.

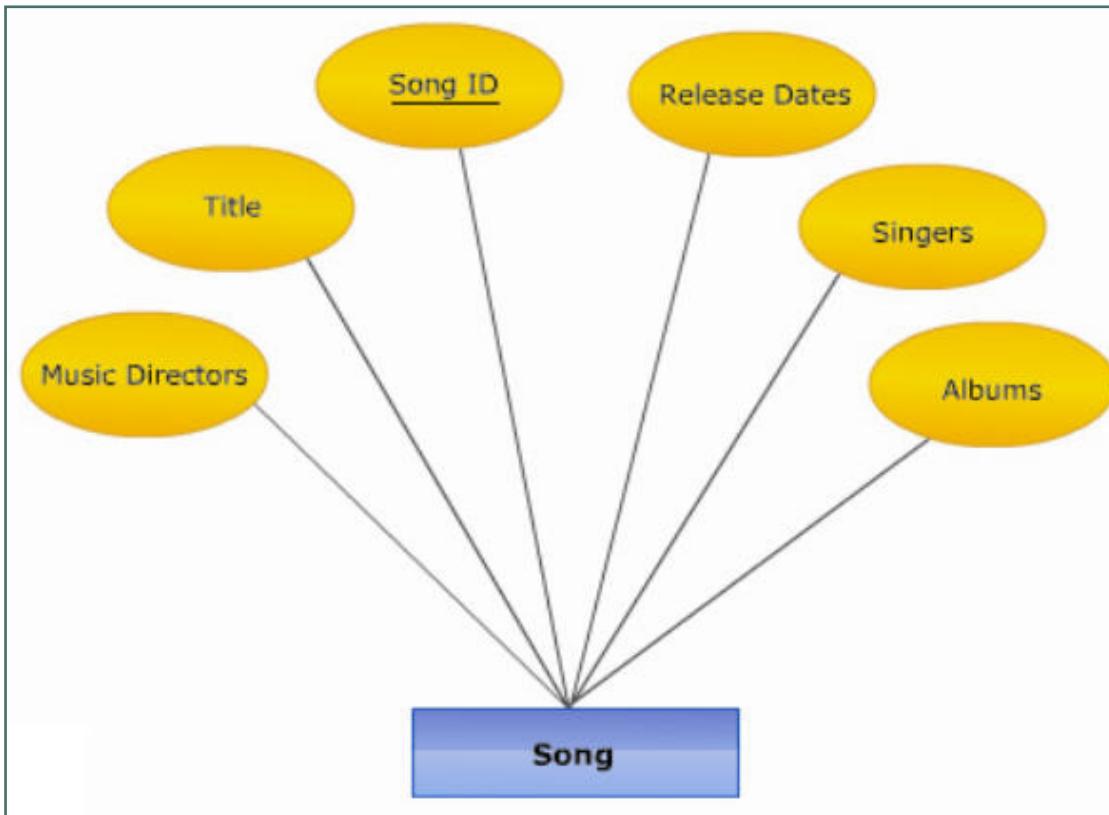


Figure 10.7: Primary Key

Foreign Key

The foreign key is applicable when two entities are being considered. If the primary key of one entity is available as an attribute of another entity, then it qualifies as a foreign key. When considering the two entities; Song and Broadcast, the Song ID primary key also finds a place as an attribute in the Broadcast entity, where it is the foreign key. Like primary key, the foreign key may consist of one or more attributes.

Figure 10.8 depicts a foreign key.

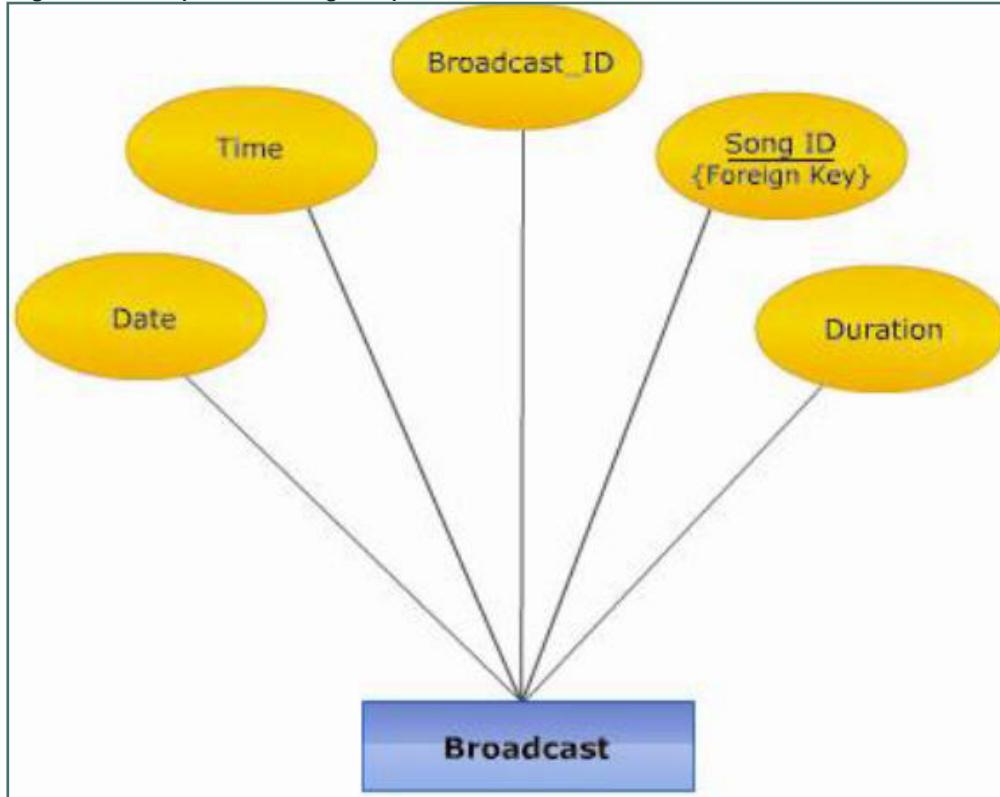


Figure 10.8: Foreign Key

10.3.1 Relationship

Foreign keys help in forming relationships between entities. A relationship may be defined as an association, linkage, or connection between the entities.

There are four types of relationships that can be created between the entities in the database:

- One-to-one relationship
- One-to-many relationship
- Many-to-one relationship
- Many-to-many relationship

For example, there is a relationship between Song and Broadcast. Each song may be broadcast once or many times. During a broadcast, only one song can be played at a time. This is a many-to-one relationship. The other kinds of relationships are many-to-many and one-to-one.

Figure 10.9 depicts a many-to-one relationship.



Figure 10.9: Many-to-One Relationship

10.3.2 Data Modeling

Data modeling is usually achieved with the help of an Entity-Relationship Diagram (ER Diagram). The data modeling concepts such as entities, attributes, and relations are represented in the ER diagram using symbols. ER Diagram maps the application to relational database.

To create an ER diagram, you should identify the entities, attributes, and relationships of an activity for which data has to be maintained, and only then go about drawing an ER diagram. In a hospital system for instance, an activity that needs to be logged is a doctor examining a patient. The two main entities here are Doctor and Patient. Examination is the relationship that exists between them. Doctor's attributes are Name, Qualification, and Specialization. Patient's attributes are Name, Age, and Gender.

Activity can be depicted in an ER diagram using tools such as Microsoft Visio, Rational Rose, and Gliffy. Once an ER diagram is drawn, it can be easily interpreted into a database design with the entities serving as tables and attributes serving as fields. Figure 10.10 shows an ER diagram.

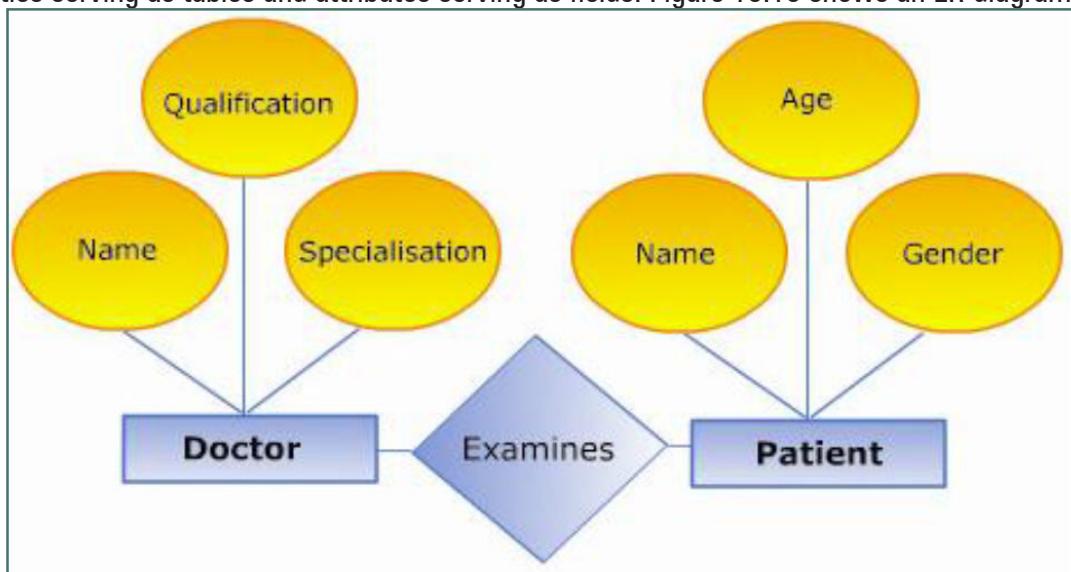


Figure 10.10: ER Diagram

An entity is represented as a rectangle that contains its name. It is connected to at least one relationship and some attributes. A relationship is represented as a diamond that contains its name. It connects two entities. An attribute is represented as a circle that contains its name. It is connected with an entity. When connected with the help of lines, they depict an activity.

10.4 Managing Entities Relationship in JPA

Entity beans represent objects in Object-Oriented Analysis and Design (OOAD). Data modeling and object modeling are the two most important aspects of OOAD, as they help in optimally designing your application. Object modeling is arranging a standardized set of symbols to represent the design of the system. Data modeling enables to structure and organize data so that it is implemented in a database.

In order to support entity relationships, the object-to-relational mapping engine must provide support for object-oriented features such as inheritance, polymorphism, and so on.

The Java Persistence specification supports the entity inheritance and polymorphism, managing relationships associations, polymorphic queries, and so on. The association among the entities is also termed as relationships among entities.

Entities in the enterprise applications can be associated based on the two things:

- Cardinality
- Directionality

10.4.1 Cardinality

Cardinality, in terms of relationships, refers to the number of instances of the base bean that relates to the number of instances of another bean.

JPA supports the following three kinds of cardinality relationships:

- One-to-One**

In one-to-one relationship, one bean instance relates to only one bean instance. Relationship between a student bean and score card bean is an example of one-to-one relationship, as one student can be related to only one score card.

- One-to-Many/Many-to-One**

In one-to-many relationship, one bean instance relates to multiple bean instances. For instance, one customer bean instance can relate to multiple invoice beans but one invoice cannot be related to multiple customers.

- Many-to-Many**

In many-to-many relationship, many bean instances relate to many instances of another bean. The fact that many actors work in many movies is an instance of many-to-many relationship.

Figure 10.11 depicts cardinality of relationships.

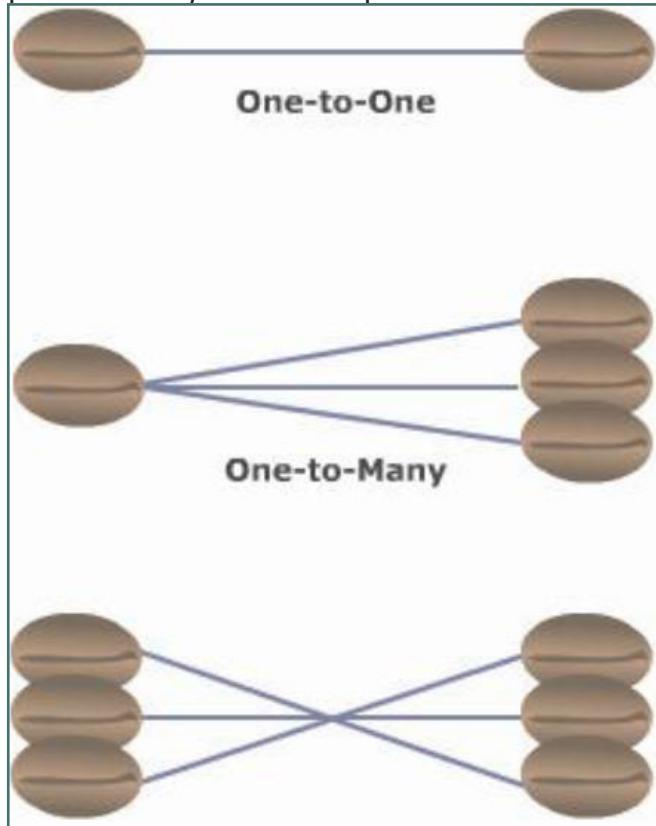


Figure 10.11: Cardinality of Relationships

10.4.2 Directionality

Besides cardinality, directionality is another aspect of relationship between two beans that container supports. Directionality defines the navigation pattern between two beans. Directionality can be either unidirectional or bidirectional.

10.4.3 One-to-One Relationship

In case of one-to-one unidirectional relationship, the state of one entity determines the state of another entity. The entity whose state defines the state of the other entity is said to be the owning side of the relationship. For instance consider two entities Customer and Address, the Customer entity will determine the corresponding address, but the Address entity cannot refer to the customer. Therefore, customer is the owning side of the relationship.

Figure 10.12 depicts one-to-one unidirectional relationship between Customer and Address entity.

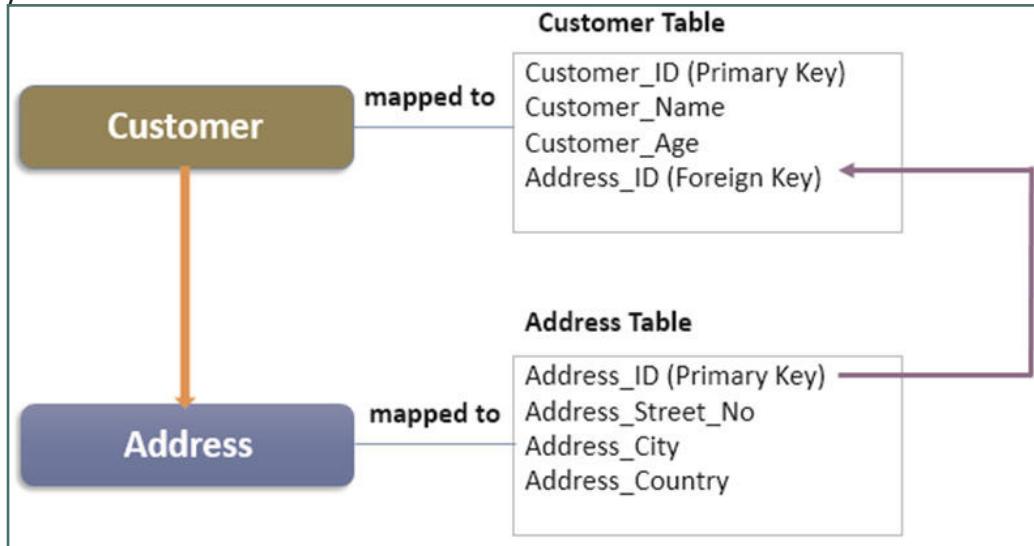


Figure 10.12: One-to-One Unidirectional Relationship

One-to-one relationships are mapped using primary key and foreign key associations. This creates a parent-child relationship between the entities. For example, in the described scenario, the Customer entity can be viewed as the Parent entity.

In JPA, entity classes associated with one-to-one relationships are annotated through `javax.persistence.OneToOne`. The annotation is prefixed to the owning entity class of the application. The developer can define an one-to-one mapping either at the field level or getter/setter methods defined for the property.

The `@OneToOne` annotation has attributes such as `targetEntity`, `cascade`, `fetch`, `mappedBy`, and `orphanRemoval`.

The description is as follows:

- `targetEntity` represents the entity class to which the Entity is associated.
- `cascade` attribute defines how certain operations can be percolated through the relationship to the associated entities.
- `fetch` attribute defines the pattern of loading the entities involved in the association. This fetching mechanism of the entity can be either lazy or eager. According to the lazy fetching mechanism, when an entity is fetched all the entities associated with the entity are not fetched. In case of eager fetching mechanism when an entity is fetched by the application all the associated entities are also fetched.
- `mappedBy` attribute is used to define a bidirectional relationship. It identifies the entity which will map the current entity through `mappedBy` attribute.
- `orphanRemoval` attribute is used to determine whether the removal of entity will result in the removal of the entities referred by it.

If the underlying table contains the foreign key column, referencing the table, then the `@JoinColumn` annotation is used to map the relationship in the OOP model. The attributes of `@JoinColumn` are `updatable`, `insertable`, `table`, and `unique`.

Code Snippet 1 demonstrates the one-to-one mapping for Customer and Address entities.

Code Snippet 1:

```
@Entity
@Table(name="Customer")
public class Customer {

    @Id
    @Column(name="Customer_ID")
    protected String Customer_ID;
    ...
    ...
    // Mapping ForeignKey
    @OneToOne
    @JoinColumn(name="Cus_address_id",
    referencedColumnName="Address_ID", updatable=false)
    protected Address address; // reference of Address
    // object
}

@Entity
@Table(name="Address")
public class Address {

    @Id
    @Column(name="Address_ID")
    protected String Address_ID;
    ...
    ...
}
```

The `@JoinColumn` annotation contains the attribute name which refers to the name of the foreign key in the `Customer` table. The attribute `updatable` is set to `false`, which means that the persistence provider would not update the foreign key, even if the address reference were changed.

In a one-to-one bidirectional relationship, entities on either side of the relationship can be used to refer the other entity.

Consider a scenario of a bank application where a customer is assigned to a customer id. The domain also defines a constraint that every customer can hold only one savings bank account with the bank. In such a scenario, with the help of customer id the savings bank account number can be determined. Similarly, through the savings bank account number, the customer id of the account holder can also be determined.

This kind of a relationship where the entity on either side of the relationship can determine the entity on the other side of the relationship is termed as a one-to-one bidirectional relationship.

Figure 10.13 depicts one-to-one bidirectional relationship in a banking domain.



Figure 10.13: One-to-One Bidirectional Relationship

To set the bidirectional relationship, the entry on the inverse side of the relationship can be mapped with one-to-one relationship. To achieve this, the `@OneToOne` annotation is provided with `mappedBy` element.

Code Snippet 2 shows the bidirectional mapping of relationship for `Customer` and `Address` entity.

Code Snippet 2:

```

@Entity
@Table(name="Address")
public class Address {

    @OneToOne(mappedBy="address")
    protected Customer customer;

    @Id
    @Column(name="Address_ID")
    protected String Address_ID;
    ...
    ...
}
  
```

The `mappedBy` element identifies the corresponding association field on the owing side of the relationship. The value specified in this attribute is the relationship field defined in the `Customer` table.

10.4.4 One-to-Many/Many-to-One Relationship

Consider a scenario where the bank application keeps track of user's one primary contact number and multiple secondary contact numbers. In this case, each customer is associated with multiple contact number entities. The reference to the contact number entity is always through the `Customer` entity. Therefore, the `Customer` entity is the owning entity of the relationship and it is a unidirectional one-to-many relationship.

Figure 10.14 shows the association of one `Customer` entity with many `Contact_Number` entities.

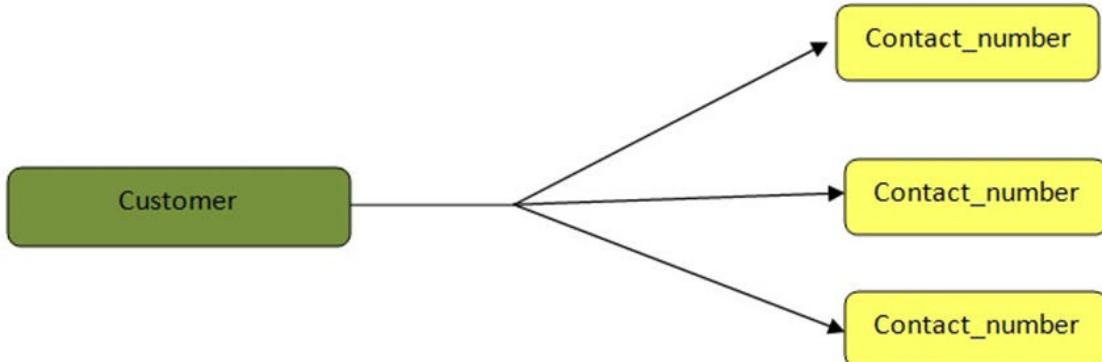


Figure 10.14: Unidirectional One-to-Many Relationship

The `javax.persistence.OnetoMany` annotation can be used to set the relationship between two entities. The attributes associated with `@OnetoMany` annotation are same as `OnetoOne` annotation; they are `targetEntity`, `cascade`, `fetch`, `mappedBy`, and `orphanRemoval`.

To map the relationship between the two entities in the inverse, then Many-to-One associations can be built. The annotation `javax.persistence.ManyToOne` is used. The attributes associated with this annotation are `targetEntity`, `cascade`, and `fetch`.

Note that both the relation types are implemented as primary key and foreign key association in the underlying database tables. To hold the multiple object retrieved for the 'Many' relationship, the collections types is used.

The `@OnetoMany` and `@ManyToOne` annotations can be defined on the field level or getter/setter methods of the property in the entity class.

Figure 10.15 shows the database schema of Item and Bid relationship in an auction.

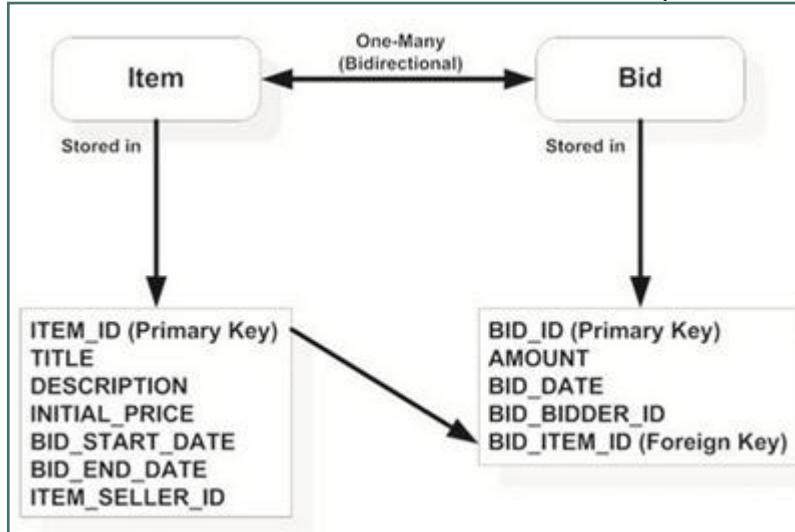


Figure 10.15: One-to-Many Relationship in an Auction

Figure 10.16 shows the bidirectional implementation of Item and Bid relationship.

```

@Entity
@Table(name="ITEMS")
public class Item {
    @Id
    @Column(name="ITEM_ID")
    protected Long itemId;
    ...
    @OneToMany(mappedBy="item")      ←① One-to-many
    protected Set<Bid> bids;
    ...
}

@Entity
@Table(name="BIDS")
public class Bid {
    @Id
    @Column(name="BID_ID")
    protected Long bidId;
    ...
    @ManyToOne
    @JoinColumn(name="BID_ITEM_ID",
                referencedColumnName="ITEM_ID")
    protected Item item;
    ...
}
    
```

The code shows the bidirectional implementation of the Item and Bid entities. The Item entity has a many-to-one relationship with the Bid entity, indicated by the @ManyToOne annotation and the @JoinColumn annotation with name "BID_ITEM_ID" and referencedColumnName "ITEM_ID". The Bid entity has a one-to-many relationship with the Item entity, indicated by the @OneToMany annotation and the mappedBy attribute set to "item".

Figure 10.16: One-to-Many/Many-to-One Relationship

As shown in figure 10.16, the @Many-to-one relationship is provided with @JoinColumn annotation. This is because the BID_ITEM_ID column is mapped as foreign key, so it is considered as the owing end of the relationship.

10.4.5 Many-to-Many Relationship

In a many-to-many relationship, each entity on either side of the relationship is associated with more than one entity on the other side. Many-to-many relationship can be a unidirectional or bidirectional relationship.

In a bank application, consider a scenario where multiple customers can provide the same landline phone number as a contact number and can also provide multiple contact numbers. Such an association is referred to as many-to-many relationship.

Many-to-many unidirectional relationship has an owning entity through which the associated entity can be referenced but it is not the case in reverse direction. Figure 10.17 shows a many-to-many relationship between the customer entity and the contact number entity.

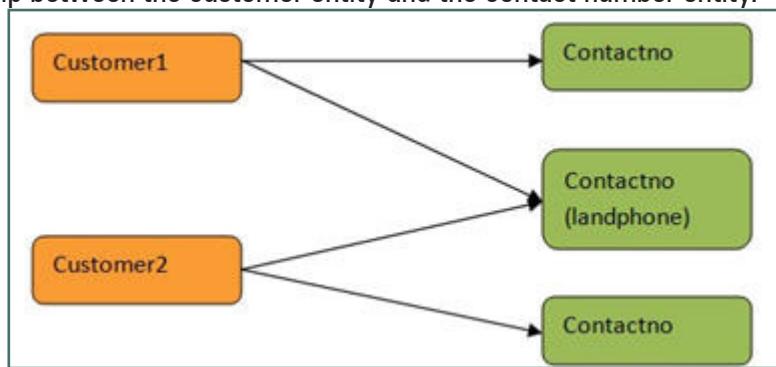


Figure 10.17: Many-to-Many Unidirectional Relationship

In figure 10.17, each customer entity is associated with more than one Contact_no entity and each Contact_no entity can be referenced by more than one customer entity. The customers can reference their corresponding contact numbers, however, the contact numbers cannot be used to refer their respective customers. Therefore, it is a unidirectional relationship where the customer entity is the owning entity.

In a many-to-many bidirectional relationship both the entities in the association can be referenced by more than one entity of the other type.

Both many-to-many unidirectional and bidirectional relationships are annotated with `javax.persistence.ManyToMany` annotation in the Java entity classes.

Figure 10.18 shows the many-to-many relationship for a Category containing multiple Items and an Item can belong to multiple Category entities.

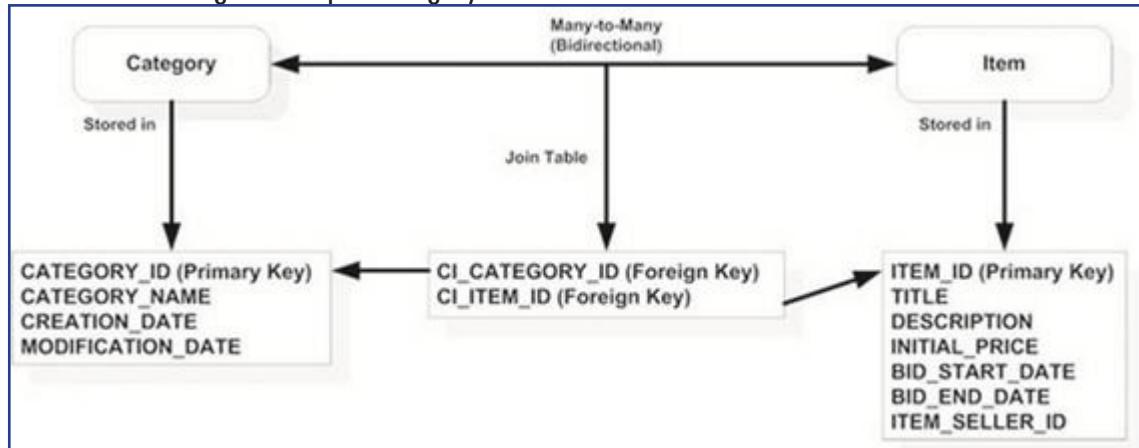


Figure 10.18: Many-to-Many Relationship

As shown in figure 10.18, the association or join table allows to indirectly match up primary keys from either side of the relationship by storing arbitrary pairs of foreign keys in a row.

10.5 Entity Inheritance

Enterprise applications use inheritance among entities for code reuse and to implement polymorphism. The entities of the application are defined on the object model of the application. When there is an inheritance existing in the object, it has to be translated onto the database of the application.

In order to map entity inheritance onto the database, JPA provides three different strategies:

- A single table per class hierarchy
- A table per concrete entity class
- A table per subclass

The default mapping strategy is single table per class hierarchy.

10.5.1 Single Table Per Class Hierarchy

All the entities in the class hierarchy are mapped onto a single table.

Consider figure 10.19 which shows a hierarchy in a banking application.

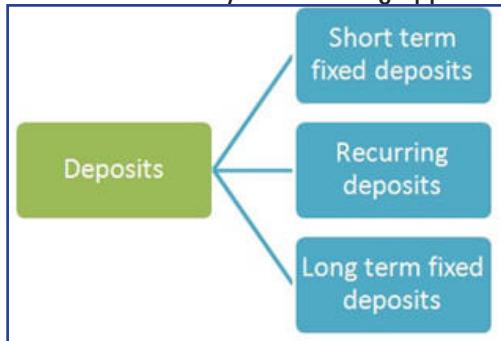


Figure 10.19: Class Hierarchy in a Banking Domain

Figure 10.19 shows a hierarchy where 'Deposits' is a super entity class. The entity class Short term fixed deposits, Recurring deposits and Long term fixed deposits are inherited from the Deposits entity.

When a 'single table per class hierarchy' strategy is used, all the entities shown in the hierarchy are mapped onto to a single table as shown in the given table definition:

Fixed deposit number	Linked savings Account number	Customer name	Amount
----------------------	-------------------------------	---------------	--------

This strategy corresponds to the default InheritanceType.SINGLE_TABLE.

However, the table definition requires a way for discriminating between the child classes of the Fixed Deposit class. Therefore, a discriminator column is added to the table. The table definition with the discriminator column is as shown in the given table definition:

Fixed deposit number	Linked savings account number	Customer name	Amount	Fixed deposit type
----------------------	-------------------------------	---------------	--------	--------------------

The discriminator column for this mapping strategy is defined through javax.persistence.DiscriminatorColumn annotation. This annotation has four attributes namely, name, Discriminator type, column definition, and length.

The brief description is as follows:

- The name attribute represents the name of the column in the mapped table.
- The DiscriminatorType represents the data type of the discriminator column. The default type of the discriminator column is String. This column can assume String, char, or integer type.
- The columnDefinition attribute defines the method of generating the value in the column. This is generated by the persistence provider.
- The length attribute is used when the column type is String. The default value of the string length is 31.

Code Snippet 3 demonstrates the single table per class hierarchy.

Code Snippet 3:

```
@Entity(name = "Account")  
 @DiscriminatorColumn(name = "DISCRIMINATOR", discriminatorType = DiscriminatorType.STRING)  
 @DiscriminatorValue("account_type")  
 @Inheritance(strategy = InheritanceType.SINGLE_TABLE)  
 class Account{  
     @Id  
     String account_type;  
     @Id  
     long account_number;  
     ....  
 }
```

Code Snippet 3 shows the mapping of the Account hierarchy onto a table in the database. The discriminator is of type String. The property account_type of the entity is used to discriminate different types of deposits in the table. All types of accounts are mapped onto a single table. The entity has a primary key which is composite. The properties account_type, and account_number together forms the primary key of the entity.

10.5.2 Table Per Concrete Entity Class

According to table per concrete entity class strategy, a table is created for every entity class in the hierarchy. This strategy corresponds to InheritanceType.TABLE_PER_CLASS. According to this strategy each concrete entity class is mapped to a corresponding table in the database. Every field and property of the entity class including the inherited classes and properties are translated to a column of the table.

For instance in the hierarchy shown in figure 10.19, there are three concrete entity classes – Short term fixed deposits, Recurring deposits and Long term fixed deposits. Three tables are created on the database. Each of these tables has columns which are properties of the subclass. This strategy does not require a discriminator column as in the case of single table per class hierarchy.

In order to extract data from individual classes, a separate query is written on the individual tables or SQL UNION queries are used. This strategy provides poor support for polymorphic relationships.

10.5.3 Table Per Subclass

This strategy is also known as Joined class strategy. This strategy corresponds to InheritanceType.JOINED. It is represented through the annotation `javax.persistence.Joined`. The super class or the root of the class hierarchy is represented by a single table. Each subclass of the hierarchy is mapped as a different table. The columns of these tables are the properties of the subclass.

Every subclass has a primary key. The subclass is related with other subclasses or super class through the primary key- foreign key relationship.

All the entities of the hierarchy can be aggregated by applying a `JOIN` operation on all the subclasses.

According to 'Table per Subclass' strategy, a new table is created for each subclass in the class hierarchy. The hierarchy as shown in figure 10.19 can therefore be implemented using the following tables:

FixedDeposit

Fixed Deposit number	LinkedAccount number	Amount
----------------------	----------------------	--------

Short Term Fixed Deposits

Short term FD number	Linked savings account number	Amount
----------------------	-------------------------------	--------

Recurring Deposits

Recurring Deposit number	Linked savings account number	Amount
--------------------------	-------------------------------	--------

Long Term Fixed Deposits

Long term FD number	Linked savings account number	Account
---------------------	-------------------------------	---------

This strategy provides good support for polymorphic relationships. However, one or more join operations need to be performed when instantiating entity subclasses which may result in poor performance in case of extensive class hierarchies.

10.5.4 Mixing Strategies

Java persistence also allows for mixing of these strategies to suit the requirement of the enterprise application.

10.5.5 Annotations used for Entity Inheritance Mapping

The mapping strategy of a hierarchy can be defined by annotating the root class of the hierarchy with `javax.persistence.Inheritance`. This annotation also defines the strategy of mapping the inheritance onto the database. The strategy of mapping the inheritance onto the database is defined through the annotation `javax.persistence.InheritanceType`. This annotation can assume any one of the three values – `SINGLE_TABLE`, `JOINED`, and `TABLE_PER_CLASS`. The default inheritance mapping strategy is `SINGLE_TABLE`.

The `SINGLE_TABLE` value corresponds to single table per class hierarchy, `TABLE_PER_CLASS` value corresponds to a table per concrete entity class strategy, and `JOINED` value corresponds to a table per subclass strategy.

10.6 Non-Entity Base Classes

Entities can also be inherited from non-entity base classes. Non-entity base classes refer to those classes which are not instantiated. They are abstract classes and mapped super classes.

- Abstract entity classes are similar to concrete entity classes, but they cannot be instantiated. Concrete entity classes can extend and define the functionality of these abstract entity classes. Whenever a query is executed on abstract entity classes, the query is executed on all the concrete subclasses of the abstract entity class.
Abstract entity classes are prefixed with 'abstract' keyword.
- Mapped Super classes are those classes in the enterprise application which are not persisted. The entities of the application may inherit the behavior and properties of such super class but the state of the mapped super classes are not persisted. In short, they are not annotated with `@Entity` annotation.

They are annotated with `@MappedSuperClass` annotation. The `EntityManager` therefore does not persist the state of the mapped super classes. The `EntityManager` cannot access the data of the mapped super class and it does not create corresponding tables on the database.

Check Your Progress

1. Which of the following annotations are used when a primary key of an entity is used as a class?

(A) @Id	(C) @Entity
(B) @IdClass	(D) None of these

2. Primary keys can be defined as:

(A) A property in the entity class	(C) An embedded class
(B) A primary key class	(D) All of these

3. Discriminator column is required in which of the following mapping strategies?

(A) Single table per class hierarchy	(C) A table per subclass
(B) A table per concrete entity class	(D) None of these

4. Which of the following attributes defines whether the given relationship is unidirectional or bidirectional?

(A) fetch	(C) mappedBy
(B) cascade	(D) None of these

5. Which of the following is the default inheritance mapping strategy?

(A) SINGLE_TABLE	(C) TABLE_PER_CLASS
(B) JOINED	(D) None of these

6. Which of the following classes are not persisted?

(A) Mapped super classes	(C) Primary key classes
(B) Entity classes	(D) None of these

Answer

1.	B
2.	D
3.	A
4.	C
5.	A
6.	A

Summary

- Relationships describe how objects collaborate with one another, to contribute to the behavior of the system.
- A database schema describes the table structure, data types, and relations in a database.
- Some of the terms common in relational database are attributes, primary key, and foreign key.
- There are four types of relationships that can be created between the entities in the database that includes one-to-one, one-to-many, many-to-one, and many-to-many.
- Entity beans represent objects in OOAD.
- In order to support entity relationships, the object-to-relational mapping engine must provide support for object-oriented features such as inheritance, polymorphism, and so on.
- Enterprise applications use inheritance among entities for code reuse and to implement polymorphism.
- Mapping of persistent application objects onto the database can be defined through annotations in JPA.
- The association among entities is defined through relationships that can be unidirectional or bidirectional.
- JPA defines three strategies for mapping the entity inheritance onto the database.
- javax.persistence.Inheritance and javax.persistence.InheritanceType are the annotations used to map inheritance onto the database.
- Abstract Entity classes and Mapped super classes are non-entity base classes.

B  g

Balanced Learner-Oriented Guide

for enriched learning available





Welcome to the Session, **Query and Criteria API**.

The session discusses Java Persistence Query Language (JPQL) and its characteristics. It introduces Query and TypedQuery API used to set parameters and retrieve results from the query. Further, the session explains Criteria API and its methods to manage the query results. It also explains the Metamodel API to create criteria queries.

In this Session, you will learn to:

- Explain Java Persistence Query Language (JPQL)
- List the characteristics of JPQL
- Explain Query and TypedQuery API
- Explain how to write the named queries in JPQL
- Explain how to execute the named queries in JPQL
- Explain the methods to tune the JPQL query results
- Explain polymorphic queries
- Explain Criteria API
- Explain how to develop queries using Criteria API
- Explain how to manage Criteria query results
- Explain Metamodel API

11.1 Introduction

Querying is the mechanism used to extract data from databases. Traditionally, relational databases used Structured Query Language (SQL) to extract data from the database. SQL queries and procedures were written to generate complex reports and results based on a data stored in the database tables.

JPA provides Java Persistence Query Language (JPQL) and Criteria API to write queries on the objects, rather than writing the queries in SQL to extract data from the databases. The JPQL is a string-based query language which allows the developers to write queries over entity objects created in the enterprise applications. In other words, it is designed to work with Java objects. The JPA queries interprets the persisted data of objects, their properties, and relationship with the other entity objects.

The main benefit of JPQL is that it allows the developers to write the queries in portable format. The JPQL queries follow object-oriented approach and makes the application independent of a particular database schema.

The entity manager is responsible for mapping queries onto the underlying tables of the relational database. The `EntityManager` object uses the metadata provided through annotations and deployment descriptors to map the entities onto the respective tables. It is responsible for managing the entities of a persistence unit and executing queries on the persistence unit. It uses `Query` interface to create and execute queries on the persistence unit.

11.1.1 Characteristics of JPQL

Some of the characteristics of JPQL are as follows:

- It is a query specification language for creating static and dynamic queries on persisted entities.
- It is defined by JPA specification.
- It is an object-based query language.
- It allows the developers to write queries over stored entities, their state, and relationships.
- It can be compiled to a target language, such as SQL understood by relational databases.
- It allows to write the queries on the abstract persistence schema which contains entities and their relationships. The queries written on abstract persistence schema are translated to the specific database schema by JPA.
- It uses SQL-like syntax to retrieve objects or values based on the abstract schema.
- It allows to define the queries in metadata annotations or in the XML descriptor.

11.2 Creating and Executing JPQL Queries

Following are the steps involved for creating and executing a JPQL Query:

1. Obtain an instance of `EntityManager` within a `PersistenceContext`. `PersistenceContext` in the application refers to the set of entities which are currently active in the application. When an instance of `EntityManager` is created, it manages all the active entities in the application which belongs to the `PersistenceContext`.
2. Create a `Query` instance through methods of `EntityManager`. `createQuery()` is one such method used to create queries.
3. Execute the query.
4. Retrieve the results using methods of `Query` interface.

11.3 JPQL Named Query

A named query is also referred to as static query. A named query defines a predefined static query string. In other words, they are predefined SQL statements which can be reused in the application. Generally, named queries are created in the application when certain query has to be executed repeatedly.

The named query is created by applying it with `javax.persistence.NamedQuery` annotation. The `@NamedQuery` annotation consists of four attributes, `name`, `query`, `hints`, and `lockMode`.

Table 11.1 lists the attributes and descriptions of the `@NamedQuery` annotation.

Description	Attribute
Refers to the query.	<code>Name</code>
Refers to the query string written in the JPQL text format.	<code>Query</code>
Refers to the hints and properties related with the query.	<code>Hints</code>
Used in query execution.	<code>lockMode</code>

Table 11.1: Attributes of `@NamedQuery` Annotation

Code Snippet 1 shows the named query.

Code Snippet 1:

```
@NamedQuery(name = "Account.All"
query = "Select a from Account a")
public class Account implements Serializable {
    ...
}
```

In Code Snippet 1, the `@NamedQuery` annotation is used to define a name for the query and associate it with the query to be executed. The other two elements, `lockMode` and `hints` are optional. Note that the `@NamedQuery` is attached with one entity class. The name assigned to the query assigned with `@NamedQuery` annotation must be unique through the persistence unit, as the scope of the named query is the entire persistence unit.

11.3.1 Multiple Named Queries

The developer can attach multiple named queries to the same entity class. This requires to wrap all the named queries within the `@NamedQueries` annotations.

Code Snippet 2 shows the `@NamedQueries` annotation.

Code Snippet 2:

```
@NamedQueries({  
    @NamedQuery(name="Account.findAll",  
        query="SELECT a FROM Account a"),  
    @NamedQuery(name="Account.findByName",  
        query="SELECT a FROM Account a WHERE a.name =  
            :name")  
})  
public class Account implements Serializable {  
    ...  
}
```

11.3.2 JPQL Dynamic Query

Dynamic queries are used when the application have to build queries at run time. They can be created based on certain conditions. The `createQuery()` method of the `EntityManager` interface is used to create the dynamic queries.

Code Snippet 3 shows how to create the dynamic queries within a Stateless Session bean.

Code Snippet 3:

```
@Stateless  
Public class AccountStatelessSessionBean implements AccountBean {  
    ...  
    @PersistenceContext
```

```

Public EntityManager e;
public List displayAccounts() {
    Query Q=e.createQuery("select a from Account a",
        Account.class);
    List accountsList=Q.getResultList();
    . . .
}
}

```

In Code Snippet 3, a query has been created with a criteria or a condition provided through 'where' clause. The query retrieves the Account objects for all the customers and returns as a list from the displayAccounts() method.

11.3.3 Query APIs

The `Query` interface is returned by the `EntityManager`. It is used to execute the queries. The `Query` interface provides various methods to set the parameters of the query, to execute the queries, and to retrieve results of the query. The `Query` interface also provides methods to set the pagination properties of the result set and control the flush mode. It is mainly used when the result type is unknown or when a query returns a polymorphic results.

JPA 2.0 introduced `TypedQuery` interface. It extends the `Query` interface and is used to control the execution of the query. It is usually used when you expect the query to return more specific result types. In other words, it processes the query result returned by `TypedQuery` is type safe.



Paging refers to the process of replacing old unused pages of memory with new data. This process is initiated when the system memory is full and the application requires more data from the storage disk into the memory. Pagination properties and flush mode determine what data has to be erased out of the memory when the memory is full.

Table 11.2 shows some of the methods present in the `Query` interface and their purpose in handling the query execution.

Description	Method
Used to extract the result set of a query.	<code>List getResultList()</code>
Used to extract one object from the result set.	<code>Object getSingleResult()</code>
Used to execute an update or delete statement on the database.	<code>int executeUpdate()</code>

Description	Method
Used to set the maximum number of results that can be received from a query.	Query setMaxResult()
Used to set the position of the first result the query is set to receive.	Query setFirstResult()
Is an overloaded method and is used to set parameters for query execution.	void setParameter()
Used to set the flush mode to execute the query.	Void setFlushMode()

Table 11.2: Methods of Query Interface

11.3.4 Executing the Named Query

The named queries can be bounded with the parameters. The parameters can be either named parameters or indexed parameters.

Code Snippet 4 shows how to invoke the query in a Stateless Session bean.

Code Snippet 4:

```
@Stateless
public class AccountStatelessSessionBean implement AccountBean
{
    @PersistenceContext(unitName="mypersistenceunit")
    private EntityManager e;
    public List findAccounts() {
        Query q=e.createNamedQuery("Account.findAll", Account.class);
        List accList=q.getResultList();
        return accList;
    }
}
```

In Code Snippet 4, an instance of EntityManager is injected in the AccountStatelessSessionBean class. The findAccounts() method invokes the createNamedQuery() method on the entity manager, which returns the Query object. The Query object contains the list of all Accounts retrieved from the Account entity result after execution of the query is retrieved in a list of strings. Then, the retrieved Account objects list is returned from the method to the calling environment.

11.4 Parameterized Queries

JPQL allows execution of parameterized queries, where the values required for the execution of the query are passed dynamically. These values are referred as parameters which are passed to the query through methods in the `Query` interface. The `setParameter()` method of the `Query` interface is used for setting parameters for the query. The parameters to the query can be passed in two different ways – based on the name used in the query or based on the position of the parameter in the query.

11.4.1 Named Parameters

Named Parameters are those variable names in the query which can assume any value during the query execution. They are prefixed with a colon (:). The named parameters for the queries are set through the `setParameter()` method with the following prototype:

Syntax:

```
Query.setParameter(String parameter_name, Object parameter_value);
```

Code Snippet 5 shows how to pass the parameters to execute the query.

Code Snippet 5:

```
Query Q = e.createQuery("Select a from Account a where accnum = :acno",  
Account.class);  
  
Q.setParameter("acno", 10234);
```

In Code Snippet 5, the query has only one parameter ':acno'. This parameter is used to pass the account number to the query. Named parameters are case-sensitive. The `setParameter()` method is called before the query execution to bind the parameter with the query.

11.4.2 Positional Parameters

Positional parameters refer to the parameters in a query based on their numeric position which is in accordance with the order in which they appear in the query. When there is more than one parameter in the query, they are sequentially numbered.

The syntax for assigning the positions is to prefix a '?' to the position. These numbers are used to pass values for each of the parameters in the query. The `setParameter()` method is used to set the positional parameters.

Code Snippet 6 shows the usage of the positional parameters.

Code Snippet 6:

```
...
Query Q = e.createQuery("Select a from Account a where accnum = ?1 and
acc_bal = ?2");
Q.setParameter(1, 10234);
Q.setParameter(2, 2000);
...
```

In Code Snippet 6, the query requires two parameters for execution. These parameters numbered 1 and 2 are prefixed with a '?'. While setting the parameter values with `setParameter()` method, these positions are used to set the values.

11.4.3 Temporal Parameters

Passing Date parameters to the query is different from passing basic data type parameters to the queries. Date parameters passed to the JPQL queries are objects of `Date`, `Time`, and `TimeStamp` classes. These parameters are collectively referred as temporal parameters. The `Query` interface provides various definitions of `setParameter()` methods to use temporal parameters in queries.

Code Snippet 7 shows the invocation of the query passed with a `Date` object.

Code Snippet 7:

```
query.setParameter("date", new java.util.Date(), TemporalType.
DATE);
```

As shown in Code Snippet 7, `TemporalType.Date` represents a pure date, the time part of the newly constructed `java.util.Date` instance is discarded. This is very useful in comparison against a specific date, when time part is not required to be processed.

11.5 Tuning the Queries

Each query in the application can return varying set of results. `Query` interface provides various tuning methods that are used to effect the result of the query execution. The methods such as `getResultSet` and `getSingleResult` can be invoked before the query execution to set the result.

11.5.1 Paging the Result

The `getMaxResults()` to limit the maximum number of results from query execution.

The method `setFirstResult()` is used for processing the output of an executed query. This method is used to set the initial value in the result set from where the output can be displayed. However, these methods cannot be used to limit the result set in accordance with the available memory space.

Certain queries such as 'Retrieve the details of all the customers holding a savings account or loan with the bank' may generate huge result set as output. Such queries may result in memory overflow situations in an application.

The `EntityManager` instance associated with the entities involved in the query needs to be able to manage such situations. The query executes on a block of rows, instead of entire table. Once the query is executed and a block of rows are generated, `EntityManager` invokes a `clear()` method to clear the generated rows and make space for new set of rows.

Code Snippet 8 shows the usage of `setFirstResult()` and `setMaxResult()` methods.

Code Snippet 8:

```
...
Query Q=e.createQuery("Select a from Account");
Q.setParameter("acno", 10234);

List results=Q.setFirstResult('10234').setMaxResult(1).
getResultSet();
...
```

Code Snippet 8 demonstrates the usage of `setFirstResult()` method. In the scenario, the first value of the result set is set to the value 'Alex'. The `setMaxResult()` method defines the maximum number of values that can be retrieved as a result of the query, in this case it is set to 1. Based on these settings the result for this query is retrieved by the `getResultSet()` method.

11.5.2 Query Hints

Hints are set along with the queries to define additional information which can be used while executing a query. This is done for varying purposes such as defining the timeout period for the query execution, defining the lock to be acquired during the query execution and so on.

The `Query` interface provides a method `setHint()` to optimize the query. The prototype of `setHint()` method is:

Syntax:

```
Query setHint (String hint_name, Object value);
```

Code Snippet 9 shows how to set the `setHint()` method for a specific query.

Code Snippet 9:

```
...
Query Q=e.createQuery("Select Customer_name from BankAccount where
accnum=:acno");
Q.setParameter("acno", 10234);
Q.setHint("timeout", 1000);....
```

In Code Snippet 9, the method sets the timeout value for the query to 1000 milliseconds.

11.5.3 Flush Mode

When an application performs an operation on the database, the changes are synchronized with the database only when the `EntityManager` invokes a `flush()` method. There are two flush modes in which the applications can invoke a `flush()` method – `AUTO` and `COMMIT`.

`AUTO` flush mode implies that `flush()` method is invoked before a correlated query is executed.

`COMMIT` flush mode implies that a `flush()` method is invoked only when a transaction is committed.

The flush mode of the query is usually set through the `EntityManager`. However, there may be instances where the query has to execute before the `flush()` method is invoked. The `Query` interface provides a `setFlushMode()` method to set the flush mode.

Following is the prototype for using the `setFlushMode()` method:

Syntax:

```
Query setFlushMode (FlushModeType flushmode);
```

The `flushmode` variable given here can be either `AUTO` or `COMMIT` values.

11.5.4 WHERE Clause

The ‘where’ clause is used to specify a condition on the result set. The condition on the result set identifies the entities that need to be part of the result set or the entities that need to be modified. The condition can be a single condition or multiple conditions which can be used along with ‘AND’ and ‘OR’ operators. Conditions on the entities can be defined using relational operators or clauses.

Following are the clauses which are used along with the **WHERE** clause to refine the result set of queries:

- **DISTINCT** clause in the query is used to eliminate duplicate values from the result set. It is used along with the **SELECT** clause while defining the properties of the entity which is to be chosen as a part of the result set.

Code Snippet 10 shows the use of distinct clause in the 'SELECT' statement.

Code Snippet 10:

```
...
TypedQuery<String> query = em.createQuery(
    "Select DISTINCT a.Customer_name from Account a", String.class);
List<String> results = query.getResultList();
...
```

In Code Snippet 10, the `createQuery()` method is used to create the query and `getResultSet()` method of the `Query` interface to retrieve the executed results. The query retrieves the set of unique customer names from the `Account` entities.

- **IN** clause is used along with **SELECT** statements to retrieve a subset of entities from a set of entities. 'IN' clause is used to write nested queries involving multiple entity sets. The equivalent JPQL query is shown in Code Snippet 11.

Code Snippet 11:

```
...
TypedQuery<String> query = em.createQuery(
    " SELECT b.name from Account b
    WHERE acc_no IN (SELECT a.acc_no from Account a
    WHERE city = :city)", String.class);
List<String> results = query.getResultList();
...
```

In Code Snippet 11, the nested query is passed as a String parameter to `createQuery()` method. The result of the query is retrieved through `getResultSet()` method.

In the query, the inner query retrieves account numbers of all the entities that hold an account in certain city's branch. The query then refines the result by selecting all the customer names with those account numbers returned by the inner query. To summarize, the query chooses the names of all the customers who hold an account in a certain city, where the city name is accepted as an input parameter.

- LIKE clause is used to match patterns while generating the result set in the query. Following query shows the usage of the 'LIKE' clause along with the 'SELECT' statement.

Code Snippet 12 shows the 'LIKE' operator.

Code Snippet 12:

```
...
TypedQuery<String> query = em.createQuery(
    " SELECT AVG(acc_bal) from Account a
    WHERE city LIKE "B%"
    GROUP BY city", String.class);
List<String> results = query.getResultList();
...
```

In Code Snippet 12, the select query creates and retrieves the result of the query through `createQuery()` and `getResultList()` methods, respectively. The query generates average account balance of all the account holders in a city, whose name starts with 'B'.

- BETWEEN clause is used along with the 'SELECT' statement. It is used to choose a set of entities which have certain property value within a range.

Code Snippet 13 shows the usage of 'BETWEEN' clause.

Code Snippet 13:

```
...
TypedQuery<String> query = em.createQuery("SELECT a.acc_no,
a.name FROM Account a
WHERE a.acc_bal BETWEEN :min_sal and :max_sal"
, String.class);
List<String> results = query.getResultList();
...
```

The select query in Code Snippet 13 has two parameters: `min_sal` and `max_sal`. The values of these parameters are to be provided at run time. The select statement retrieves all the account numbers and corresponding customer names whose account balance is in the range of values between `min_sal` and `max_sal`.

- IS NULL clause is used along with the 'SELECT' statement to check whether certain property has an associated value or not.

Code Snippet 14 shows the use of 'IS NULL' operator.

Code Snippet 14:

```
...
TypedQuery<String> query = em.createQuery(
    " SELECT a.name, a.address
FROM Account a WHERE a.email IS NULL", String.class);
List<String> results = query.getResultList();
...
```

In Code Snippet 14, the select query is executed through methods provided in the EntityManager and Query interface. In the query, the name and address of all the customers whose email is a NULL value in the database is being retrieved.

- IS EMPTY clause has a similar usage as 'IS NULL'. 'IS NULL' checks for a NULL value and 'IS EMPTY' checks for a non-existent value in the database.

Code Snippet 15 shows the use of 'IS EMPTY' clause.

Code Snippet 15:

```
...
TypedQuery<String> query = em.createQuery(
    " SELECT b.customer_name, b.customer_address
FROM BankAccount b WHERE b.cust_email IS EMPTY", String.class);
List<String> results = query.getResultList();
...
```

In Code Snippet 15, the JPQL equivalent of the SELECT query is passed as a String parameter to the createQuery() method. The result of the query is retrieved through getResultList() method. The query returns the values of the customer_name and customer_address whose value is empty.

11.5.5 Polymorphic Queries

When a JPQL query is executed on an entity class, then the query executed is applicable to all its sub classes. The entity in the JPQL query is placed along with the 'where' clause in the query. By default, all queries are polymorphic. This means that the instances returned by a query include instances of all the subclasses satisfying the condition.

For example, consider the query:

```
select customer_name from Account;
```

The `Account` entity has subclasses namely, `current_account` and `savings_account`. This query written on `Account` class is applicable to both the `current_account` and `savings_account` classes. The result of the query has instances from `Account` class, `current_account` class, and `savings_account` class.

Code Snippet 16 shows the query returning the average age of all the customers with the bank, which includes its subclass also.

Code Snippet 16:

```
select avg(c.age) from Customer c where c.age > 20
```

11.6 Overview of Criteria API

Criteria APIs are used to define dynamic queries through query-defining objects. Criteria queries are defined by instantiating Java objects. Criteria API offers better integration with Java language than JPQL queries as it operates on the database in terms of objects. The JPQL queries discussed earlier interpret the queries as string objects. The syntax validation of the JPQL queries is not done while application compilation. The application sends these queries to the database provider without validating them. Hence, when the database provider executes these queries it may result in runtime errors in the application, due to syntax errors in the query.

Criteria query accepts values for each clause of the query through different methods and validates them. After validating the values for each clause it builds the `Query` object to be executed by the database provider.

Criteria API queries are based on the abstract schema of persistent entities, their relationships, and embedded objects. Criteria API invokes Java Persistence API entity operations through the abstract name schema to enable developers to find, modify, and delete persistence entities.

The queries written using Criteria API are similar to the JPQL queries. The JPQL syntax can be converted into equivalent operations in the Criteria API. Criteria queries enable quick detection of errors in the application code as compared to the JPQL queries. Being string based, JPQL queries are easy to understand and use.

JPQL queries are preferred for simple static queries, whereas criteria API queries are used for dynamic queries.

11.6.1 Creating Criteria Query

Criteria API creates queries by obtaining an instance of `javax.persistence.criteria.CriteriaQuery`.

A criteria query is created by creating an instance of `CriteriaQuery` class. This instance is created by invoking a `createQuery()` method with the `CriteriaBuilder` object as shown in Code Snippet 17.

Code Snippet 17:

```
...
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery cq = cb.createQuery();
...
...
```

11.6.2 Query Root

The query root of a Criteria query refers to that entity of the query, where the navigation initiates. A query root is created by invoking the `from()` method of the `CriteriaQuery` interface. The query root is defined as follows:

```
Root c = cq.from(Customer.class);
```

Certain queries may require more than one query root, which implies that the query is retrieving data from more than one table. In such a case the number of times the `from()` method is invoked is equal to the number of entity classes required.

11.6.3 Using Criteria API

Following are the steps involved in creating Criteria API queries:

1. The entities in the application are managed through the `EntityManager`. The `EntityManager` in turn creates a `CriteriaBuilder` object. The `CriteriaBuilder` object is the main interface of the Criteria API; it is responsible for creating Criteria queries.
2. The `CriteriaBuilder` object further creates a `CriteriaQuery` object. This object defines a database select query; it can enable usage of all the clauses associated with the JPQL SELECT statement. The `CriteriaBuilder` and `CriteriaQuery` classes are specific to the Criteria API. They build the context to accept queries on entities, where the entities are interpreted as objects.
3. The `from()` method of the `CriteriaQuery` interface is used to set the query root. The function performed by the `from()` method is similar to that of the `FROM` clause in a SQL query. The query root implies the entity sets from which the result set of the query has to be generated. For instance, if the query has to be executed on `Account` entity set, then it forms the query root in the Criteria API.
4. The clauses used along with the 'SELECT' statement are defined as methods in the Criteria API such as `where()`, instead of `WHERE` clause. The `where()` method is used to apply conditions on the result set.

5. The `select()` method is used to define the return object type of the query. This method is used when the query has to return only one type of value. When there are values with multiple data types to be returned by the query, then the developers have to use `multiselect()` method.
6. The `multiselect()` method is used in instances where multiple properties of the entity are to be selected.
7. The query is executed by invoking the method `getResultSet()`.

Code Snippet 18 shows various Criteria queries.

Code Snippet 18:

```
...
EntityManager em=....;
CriteriaBuilder cb=em.getCriteriaBuilder();

// Query for a List of objects
CriteriaQuery cq=cb.createQuery();
Root c=cq.from(Customer.class);
cq.where(cb.greaterThan(c.get("salary"), 100000));
Query query=em.createQuery(cq);
List<Employee> result=query.getResultList();

// Query for a single object
CriteriaQuery cq=cb.createQuery();
Root c=cq.from(Customer.class);
cq.where(cb.equal(c.get("id"), cb.parameter(Long.class,
"id")));
Query query=em.createQuery(cq);
query.setParameter("id", id);
Customer result2=(Customer) query.getSingleResult();

// Query for a single data element
CriteriaQuery cq=cb.createQuery();
Root c=cq.from(Customer.class);
```

```
cq.select(cb.max(c.get("salary")));  
Query query=em.createQuery(cq);  
BigDecimal result3=(BigDecimal)query.getSingleResult();  
  
// Query for a List of data elements  
CriteriaQuery cq=cb.createQuery();  
Root c=cq.from(Customer.class);  
cq.select(c.get("firstName"));  
Query query=em.createQuery(cq);  
List<String> result4=query.getResultList();  
  
// Query for a List of element arrays  
CriteriaQuery cq=cb.createQuery();  
Root c=cq.from(Customer.class);  
cq.multiselect(c.get("firstName"), c.get("lastName"));  
Query query=em.createQuery(cq);  
List<Object[]> result5=query.getResultList();  
....
```

In Code Snippet 18, multiple queries are executed on the `Customer` entity. The root of the query is set through the `from()` method.

Once the root is set, the query is defined by using appropriate methods of the `CriteriaQuery` object `cq`. The query is created using the `createQuery()` method. The query result is retrieved through the `getResultList()` method.

11.6.4 Managing Criteria Query Results

The Criteria API provides methods such as `orderBy()` and `groupBy()` to implement the functionality of ORDER BY and GROUP BY clauses in SQL. The function of `orderBy()` method is similar to the function of an ORDER BY clause in a JPQL query. It sorts the result set based on the values of an attribute of the entity. The order of sorting can be in ascending or descending order which is defined by `asc()` and `desc()` methods, respectively.

Code Snippet 19 shows the usage of the `orderBy` method.

Code Snippet 19:

```
...
CriteriaQuery<BankAccount> cq = cb.createQuery(BankAccount.class);
Root<BankAccount> BA = cq.from(BankAccount.class);
cq.select(BA);
cq.orderBy(cb.desc(BA.get(BankAccount_.acc_bal)));
...
```

In Code Snippet 19, the query sorts the result set based on the value of the account balance.

The `groupBy()` method is used to perform operations on a group of entities. The group is defined by the attribute provided as parameter to the `groupBy()` method.

Code Snippet 20 shows the usage of `groupBy()` method.

Code Snippet 20:

```
...
CriteriaQuery<BankAccount> cq = cb.createQuery(BankAccount.class);
Root<BankAccount> BA = cq.from(BankAccount.class);
cq.groupBy(BA.get(BankAccount_.acc_type));
...
```

The query creates a query on `BankAccount` entity and creates groups of entities based on account type.

The `having()` method is used to apply conditions on the groups defined by the `groupBy()` method.

Code Snippet 21 shows the usage of `having()` method.

Code Snippet 21:

```
...
CriteriaQuery<BankAccount> cq = cb.createQuery(BankAccount.class);
Root<BankAccount> BA = cq.from(BankAccount.class);
cq.groupBy(BA.get(BankAccount_.acc_type));
cq.having(cb.in(BA.get(BankAccount_.acc_type)).value(savings));
...
```

In the query, group is defined based on the type of the account. The result of the query is a group whose account type is savings account.

11.6.5 Executing Criteria Query

Before executing the query, the `CriteriaBuilder` should prepare for the execution of the query. A `TypedQuery` object must be created with the type of the query result. The `TypedQuery` object must be defined through the `createQuery` method invoked by the `EntityManager`.

Following statement creates an instance of the `TypedQuery`.

```
TypedQuery<BankAccount> query = em.createQuery(cq);
```

The methods used for executing queries are `getSingleResult()` and `getResultList()`. The `getSingleResult()` method returns only one result as output of the query. The `getResultList()` method returns a collection of entities as output of the query.

Following statement is used to retrieve the result from the `TypedQuery`:

```
List<BankAccount> results = query.getResultList();
```

11.6.6 Querying Relationships Using Joins

Join operation is applied on relations of a relational model to merge the data of two relations according to the semantics of the database. The join operation implies comparing the values of one attribute of an entity with a similar attribute of another entity. The join condition in `SELECT` queries is specified along with the `WHERE` clause. In Criteria queries, the join condition on the queries is applied through the `join()` method.

When a join method is applied on two different entities of type **X** and **Y**, then the return type of the join method is `<X,Y>`, where **X** is the source entity from where the join operation is initiated and **Y** is the target entity.

Code Snippet 22 shows the usage of `join()` method in the `Criteria` interface.

Code Snippet 22:

```
....  
CriteriaQuery<BankAccount> cq = cb.createQuery(BankAccount.  
class);  
Root<BankAccount> BA = cq.from(BankAccount.class);  
Join<BankAccount, Customer> result = cq.join(BankAccount_.  
Customer);  
....
```

In Code Snippet 22, a join operation is applied on two entities of the bank database—`BankAccount` and `Customer`. The join operation is applied based on the common attribute among the two participating entities.

11.7 Using MetaModel API

The MetaModel API works along with the Criteria API to model persistence entity classes. These persistence entity classes are used by the Criteria queries.

JPA Metamodel API enables the developers to examine the persistent model of the objects in the application, which is the database schema of the application data. It enables retrieval of details on managed classes and persistent fields and their relationships in the Criteria Queries. The Metamodel comprises information about the mapped attributes for an entity class, their corresponding data types, and so on.

The Metamodel classes corresponding to the entity classes are of the form:

```
javax.persistence.metamodel.EntityType<T>
```

Metamodel classes are typically generated by the annotation processors during compile time or at run time. The developers can use the `getModel()` method in the Criteria query on the query root to access the entity from it.

The developer can use Metamodel API to create the metamodel of the entities managed in the persistence unit of the application. For each entity class in the application a corresponding Metamodel class can be created. This Metamodel class has the same class name as entity class followed by an underscore. In addition, the Metamodel class has attributes which correspond to the properties of the entity class.

Code Snippet 23 shows an entity class and its corresponding Metamodel class.

Code Snippet 23:

```
...
@Entity
public class BankAccount{
    @Id
    protected Long acc_no;
    protected String account_holder_name;
    protected String acc_type;
    protected Long acc_bal
    ...
}
// The corresponding metamodel class generated for BankAccount
...
```

```
@StaticMetamodel(BankAccount.class)
public class BankAccount_ {
    public static volatile SingularAttribute<BankAccount, Long> acc_no;
    public static volatile SingularAttribute<BankAccount, String> account_holder_name;
    public static volatile SingularAttribute<BankAccount, String> acc_type;
    public static volatile SingularAttribute<BankAccount, Long> acc_bal;
    ...
}
```

Code Snippet 23 shows the entity class `BankAccount` and the corresponding Metamodel class. The Metamodel class can be generated using `getModel()` method. The Metamodel class is a static class and all the variables in it are also static. These variables can be used to retrieve meta information from the database provider.

Code Snippet 24 shows how to obtain the `BankAccount` entity using the `getModel()` method.

Code Snippet 24:

```
CriteriaQuery cq = cb.createQuery(BankAccount.class);
Root<BankAccount> c = cq.from(BankAccount.class);
EntityType<BankAccount> account_ = c.getModel();
```

The method `Root<T>.getModel()` obtains the entity, `BankAccount` model in the Criteria query.

Check Your Progress

1. Which of the following persistence queries are object-based?

(A)	Criteria queries	(C)	Both a and b
(B)	JPQL queries	(D)	SQL queries

2. _____ flush mode ensures that the changes are written to the disk before executing correlated queries.

(A)	AUTO	(C)	CONDITIONAL_COMMIT
(B)	COMMIT	(D)	None of these

3. Which of the following components of the Criteria queries are equivalent to the 'FROM' clause of the JPQL query?

(A)	Query strings	(C)	Typed queries
(B)	Query Roots	(D)	Metamodel classes

4. Which of the following categories of queries are not object based queries?

(A)	JPQL queries	(C)	Metamodel API
(B)	Criteria queries	(D)	None of these

5. Which of the following methods is not used to retrieve the output from query execution?

(A)	getSingleResult()	(C)	getMultipleResult()
(B)	getResultList()	(D)	None of these

6. Which of the following is not a method of Query interface?

(A)	executeUpdate()	(C)	createQuery()
(B)	getResultList()	(D)	None of these

Answer

1.	A
2.	A
3.	B
4.	A
5.	C
6.	C

Summary

- Querying of databases is implemented through Java Persistence Query Language (JPQL) and Criteria API.
- EntityManager instance is responsible for executing both named queries and dynamic queries.
- Query API executes the queries created by the EntityManager and extracts the results.
- JPQL can be used to write queries equivalent to SQL queries. The developer can attach multiple named queries to the same entity class.
- JPA 2.0 introduced TypedQuery interface. It extends the Query interface and is used to control the execution of the query.
- Criteria queries are used to write object-based queries.
- Criteria API queries are based on the abstract schema of persistent entities, their relationships, and embedded objects.
- JPQL queries are preferred for simple static queries, whereas criteria API queries are used for dynamic queries.
- The TypedQuery object must be defined through the createQuery method invoked by the EntityManager.
- The MetaModel API works along with Criteria API to model persistence entity classes. These persistence entity classes are used by the Criteria queries.



Visit
Frequently Asked Questions
@

Get
WORD WISE



Visit
Glossary @

www.onlinevarsity.com



Welcome to the Session, **Concurrency, Listeners, and Caching**.

This session explains the concept of concurrent access and locking mechanism. The session explains different mechanisms to achieve locking in the enterprise applications. Further, the session explains the handling of entity lifecycle callback events through lifecycle callback methods and event listeners. The session also discusses the caching techniques provided by Java EE to support caching in enterprise applications.

In this Session, you will learn to:

- Describe concurrency utilities provided by Java EE
- Explain locking techniques used to enable concurrency in entities
- Describe different locking modes
- Explain entity lifecycle callback events
- Explain how to inject external listeners to handle lifecycle callback events
- Explain caching techniques
- Explain how to specify caching modes in applications

12.1 Introduction

Multiple applications can simultaneously access entity data from the database. When multiple applications access data, there is concurrent access to the data stored in the database. Thus, locking is essential to avoid simultaneous updates to the same data by concurrent users.

Concurrent access to the database is always protected with the transaction isolation techniques. Thus, the developer applying transaction boundaries to the data access may not have to provide any additional concurrency control to protect the integrity of the data stored in the database.

Database providers provide locking techniques to maintain data integrity. In addition to that, persistence providers also provide delay database writes, until the end of the transaction.

By default, the entities are provided with the isolation levels provided by the EJB's container-managed transactions service. However, the developer can provide the concurrency controls on the applications. To achieve this, the JPA specification have defined two important mechanisms that can be used to tune the concurrent accessed entities in the enterprise applications.

The two mechanisms that can be used to manage concurrent access to the entities are as follows:

- Optimistic locking
- Explicit read and write locks

Optimistic locking is the most commonly used technique in applications. Applications do not acquire any locks in this case. It checks that no other applications have modified the data used by the current application, before performing a commit operation.

Persistence providers use pessimistic locking when data integrity is crucial for the application and the data being accessed is frequently accessed by multiple applications. It acquires locks on all the data, before starting the database operations.

12.2 Optimistic Locking

Optimistic locking technique does not acquire locks for providing concurrency control, while performing any operation on certain data. However, conflicts may arise due to concurrent modifications to the data. In such case, the first completed transaction is committed and subsequent transactions are rolled back.

This mechanism provides high degree of concurrence access to the data which is required in large enterprise applications. However, the application should handle the occurrences which can led to conflicts due to concurrent access.

By default, JPA specification configures transaction isolation level to READ COMMITTED. The isolation level means that from the point of view of each concurrent transaction, it appears that no other transaction is in progress. However, 'read' data is not guaranteed to be consistent. The 'write' operations to the database are deferred, till the transaction is not committed.



READ COMMITTED means that reading transaction will not block other transactions from accessing the database row.

Before the transaction commits, optimistic locking technique ensures that update to the database row corresponding to the state of an entity are made without other transaction updating the database row since, the entity state was read. In other words, if an update operation is performed on an old version of the entity instance corresponding to the database table row whose data is already being updated by an entity instance running in other transaction boundary. Then, an exception is thrown. The exception named `OptimisticLockException` is thrown by the persistence provider when such a conflict occurs.

Optimistic locking technique is not a part of EJB specification, so, every application server would support some mechanism to check write conflicts for optimistic locking. The solution adopted for optimized locking should be independent of the application server.

One solution to make the application independent to enable optimistic locking for entities is to specify `Version` attribute to the entities. This can be done either by annotating the entity field with `javax.persistence.Version` annotation or specify the `version` attribute in the XML descriptor file.

`javax.persistence.Version` annotation to identify the version of the persistent field or property being accessed by the application.

12.2.1 Version Attribute

The `Version` attribute specified on the entity field or property is used by the persistence provider. Only the persistence provider can modify or update the `Version` attribute, whenever the entity state data is accessed and modified by the application.

Following are the requirements of `@Version` annotation:

- Every entity class is associated with only one `version` attribute.
- The persistent provider is allowed to set or update the value of the `version` attribute.
- The `version` attribute of the entity should be present in the primary table of the mapped database.
- The `version` attribute in the entity tables can be of type – `int`, `Integer`, `long`, `Long`, `short`, `Short`, or `java.sql.Timestamp`.

Code Snippet 1 shows the optimistic locking provided in the entity class.

Code Snippet 1:

```
public class Employee {  
  
    @ID  
    int id;  
  
    @Version  
    int version;  
  
    ...  
}
```

Code Snippet 1 shows the inclusion of `version` attribute marked with the annotation `@Version`. The annotation instructs the persistence provider to check the entity instance for any concurrent modifications and increments of the `version` attribute in each update operation. Thus, the `version` attribute is incremented with a successful commit. The result SQL query generated for incrementing the `version` attribute is as follows:

```
"UPDATE Employee SET ..., version=version + 1  
WHERE id = ? AND version = readVersion"
```

Figure 12.1 shows the example of two transactions which are concurrently updating Employee e1.

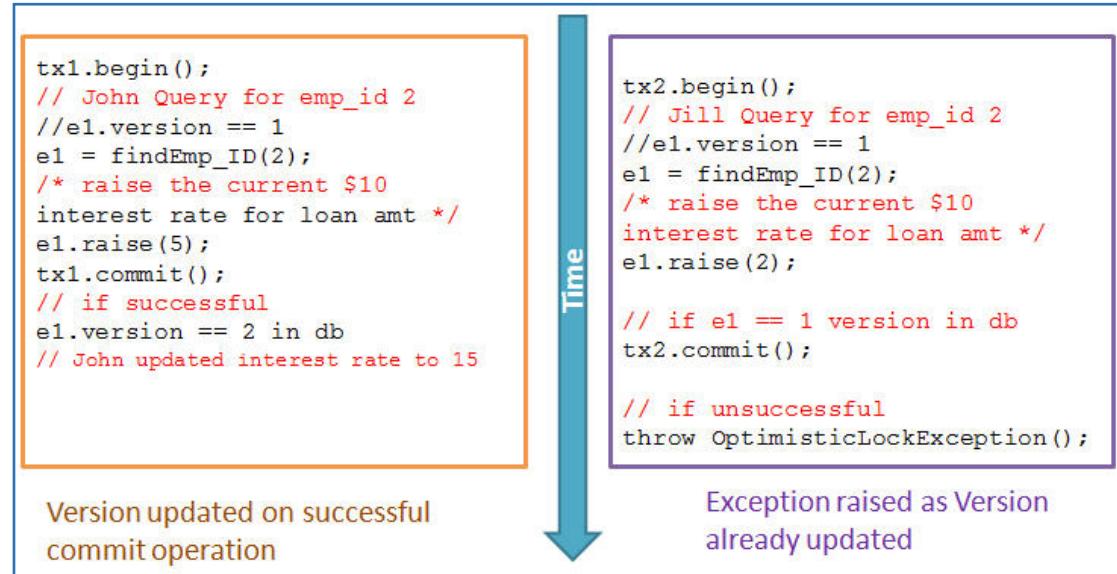


Figure 12.1: Concurrent Update Transactions

As shown in figure 12.1, the transaction T1 on the left-side accesses the Employee `e1` object and invokes the method `raise()` to update the object. This transaction is performed successfully in the database row and get committed. Thus, causing the `version` of `e1` to be incremented to 2 with the update.

While the transaction was getting committed, another transaction T2 access the e1 object with the version number 1. The transaction also performs the `raise()` on the e1 object, however, when the transaction is committed, the persistence provider throws an exception `OptimisticLockException` and roll backs the transaction. This is because the e1 version attribute is incremented in the previous transaction and is higher than the version number of the entity instance in T2 transaction.

12.3 Pessimistic Locking

The advantage of optimistic locking is that it does not locks the database, however, checks the entity version number before committing the transaction and synchronizing the entity state with the database.

Sometimes, it is desirable to acquire the database locks for a long-terms on entities. Such immediate obtained database locks are referred to as 'pessimistic' locks.

A pessimistic locking technique ensures that:

- No other transaction updates or deletes the entity instance, till the transaction locks are not removed from the entity.
- In case, if the pessimistic lock is applied as an exclusive lock, then, the same transaction boundary can update or delete it.

A pessimistic lock can be acquired on entity data by setting the lock mode of the entity instance to `PESSIMISTIC_READ`, `PESSIMISTIC_WRITE`, or `PESSIMISTIC_FORCE_INCREMENT`.

When a versioned entity is locked in `PESSIMISTIC_WRITE` lock mode then, the version attribute is incremented on successful commit of the transaction. If the application is unable to acquire a pessimistic lock, then a `PessimisticLockException` is thrown.

If the transaction is not able to apply a pessimistic lock, however, it does not result in a transaction roll back, then a `LockTimeoutException` occurs.

When the application tries to lock a non-versioned entity in `PESSIMISTIC_FORCE_INCREMENT` it throws a `PersistenceException`.

An application cannot wait indefinitely to acquire a pessimistic lock. The persistence provider has to explicitly define a time out period for which a transaction can wait to acquire a lock. The timeout period can be set through `javax.persistence.lock.timeout` property. When a transaction waits to acquire a lock on entity data and the time period for which it waits exceeds the value set through the timeout property then `LockTimeoutException` is thrown.

The lock timeout property can be set through the `EntityManager.Query.setLockMode` and `TypedQuery.setLockMode` methods can be used to set the lock timeout period. The timeout period can also be set through the deployment descriptor and as an attribute to the `@NamedQuery` annotation.

When the timeout period is set at multiple places, then the timeout period is determined as per the following order:

1. If the timeout period is set using the instances of `EntityManager` or `Query` methods then, this value is considered above all other methods.
2. If the timeout is not set by `EntityManager` or `Query` instances and it is specified through `@NamedQuery` annotation then this value is considered by the persistence provider.
3. If none of the earlier methods are used and the timeout period is set through the `createEntityManagerFactory()` method then this value of timeout is considered.
4. If the timeout period is not defined anywhere in the application code and defined in the deployment descriptor then this value is the final timeout value.

12.4 Lock Modes

Every application has different data integrity requirements. Certain applications such as bank applications require higher data integrity to be maintained against the social networking applications where certain data loss is tolerable. Various lock modes are defined with a combination of optimistic and pessimistic locking.

Using optimistic locking involves checking the version of the entity data being accessed. Using pessimistic locking involves acquiring long term write locks for the entity data before performing operations on the entity data.

JPA provides lock modes which are layered on the top of the `@Version` annotation provided in the optimistic and pessimistic locking.

Table 12.1 shows various locking modes that can be applied to entities.

Description	Locking mode
This mode acquires optimistic read locks on all the entities used by the application with corresponding version attributes.	OPTIMISTIC
This mode acquires optimistic read locks on all the entities required by the application and version attributes. It force increments on the version attribute value.	OPTIMISTIC_FORCE_INCREMENT

Description	Locking mode
This mode acquires a long term read lock on the data accessed by the application. This lock is acquired to prevent data to be accessed or modified by other applications. Other applications can read this data when the pessimistic read lock is held on the entity data. The pessimistic read lock can be upgraded to a pessimistic write lock.	PESSIMISTIC_READ
Pessimistic write lock is a long term write lock acquired on entity data. This lock prevents other applications from both reading and writing the locked entity data.	PESSIMISTIC_WRITE
This mode of lock obtains a long term lock on the entity data and increments the value of the version attribute. This lock prevents the data from being modified by other applications.	PESSIMISTIC_FORCE_INCREMENT
This mode represents an acquired optimistic read lock.	READ
This mode represents an optimistic write lock acquired on the entity data. It forces to increment the version attribute.	WRITE
No locking mode is used on the database.	NONE

Table 12.1: Lock Modes

The locking can be specified at multiple levels that are as follows:

- Entity Manager methods such as `find()`, `refresh()`, and `lock()`.
- Query methods such as `setLockMode()`.
- `NamedQuery` annotation attribute named `lockMode`.

The lock mode is set on the entities by the entity manager. Following are the various methods through which lock mode is set on the entity data:

1. The `EntityManager` can acquire a lock by invoking a `lock()` method on the application. The lock mode according to which the lock has to be acquired is passed as a parameter to the `lock()` method.

```
EntityManager em = ...;
Customer c = ...;
em.lock(c, LockModeType.OPTIMISTIC);
```

2. The `EntityManager` can invoke a `find()` method to retrieve entity data from the application. It accepts the attribute through which a search operation can be performed, the entity class and the lock mode to be acquired as parameters.

3. Following is the usage of the find method:

```
EntityManager em = ...;  
String customerPK = ...;  
Customer C = em.find(Customer.class, customerPK, LockModeType.  
PESSIMISTIC_WRITE);
```

4. The EntityManager can invoke a refresh() method to change the currently existing lock on entity data to another lock. Following is the usage of this method:

```
EntityManager em = ...;  
String customerPK = ...;  
Customer C = em.find(Customer.class, customerPK);  
...  
em.refresh(C, LockModeType.OPTIMISTIC_FORCE_INCREMENT);
```

5. A lock can also be acquired through setLockMode() method of Query interface or TypedQuery interface. Following is the usage of setLockMode() method of Query interface:

```
Query q = em.createQuery(...);  
q.setLockMode(LockModeType.OPTIMISTIC);
```

6. A lock mode element can be added to @NamedQuery annotation.

```
@NamedQuery(name="lockCustomerQuery",  
query="SELECT c FROM Customer c WHERE c.name LIKE :name",  
lockMode=OPTIMISTIC_FORCE_INCREMENT);
```

12.5 Entity Callbacks and Listeners

An entity contains a predefined set of lifecycle events that are triggered on the execution of methods called from EntityManager or Query API. For example, the persist() method triggers database events. While doing so, it may be required to log the interactions performed on the database rows.

JPA allows you to set up callback methods on the entity classes so that the entity instances are notified about the events. The developer can also register separate listener classes that are intercepted with the events. This can be achieved by writing entity listeners. An entity listener class is a class whose methods are invoked in response to lifecycle events on an entity.

12.5.1 Entity Callback Events

Callback methods are defined within the entity class. They are annotated with the lifecycle callback annotations.

Code Snippet 2 shows the implementation of the callback methods in an entity class.

Code Snippet 2:

```
@Entity  
public static class MyEntity {  
    @PrePersist void onPrePersist() {}  
    @PostPersist void onPostPersist() {}  
    @PostLoad void onPostLoad() {}  
    @PreUpdate void onPreUpdate() {}  
    @PostUpdate void onPostUpdate() {}  
    @PreRemove void onPreRemove() {}  
    @PostRemove void onPostRemove() {}  
}
```

Code Snippet 2 shows the declaration of callback methods. Each callback method takes no argument and returns void. The `@PrePersist` and `@PostPersist` events are triggered on the insertion of an entity instance data into the database row. The `@PrePersist` event is triggered when the `persist()` method is called. The `@PostPersist()` event is triggered after the data is inserted in the database row.

The `@PostLoad` is triggered after an entity has been retrieved from the database. The `@PreUpdate` is triggered when an entity is identified for modification. The `@PostUpdate` event is triggered after an entity state is updated in the database row. The `@PreRemove` annotated method marks the entity for removal and the `@PostRemove` event is triggered after deleting an entity from the database.

Code Snippet 3 shows the usage of lifecycle callback methods.

Code Snippet 3:

```
@Entity  
@EntityListeners(com.bank.AlertMonitor.class)  
public class Account {  
  
    Long accountId;  
  
    Integer balance;  
  
    boolean preferred;  
  
  
    @Id  
    public Long getAccountId() { ... }  
  
    ...  
    public Integer getBalance() { ... }  
  
    ...  
    public void deposit(Integer amount) { ... }  
  
    public Integer withdraw(Integer amount) throws NegativeException  
    { ... }  
  
    @PrePersist  
    protected void validateCreate()  
    {  
        if (getBalance() < MIN_REQUIRED_BALANCE)  
            throw new AccountException("Insufficient balance to open an  
account");  
    }  
  
    @PostLoad  
    protected void adjustPreferredStatus()  
    {  
        preferred = (getBalance() >= AccountManager.  
                    getPreferredStatusLevel());  
    }  
}
```

```
public class AlertMonitor {  
    @PostPersist  
    public void newAccountAlert(Account acct) {  
        Alerts.sendMarketingInfo(acct.getAccountId(), acct.getBalance());  
    }  
}
```

12.5.2 Entity Listeners

The entity class can be injected with the entity listeners through CDI. The entity listeners are classes that are intercept entity callback events. They are not entity class, rather external classes that are attached to an entity class through annotations or XML configuration. The developer can design multiple entity listener classes to intercept lifecycle events.

The external callback methods in the listener class should accept the object as a parameter that represents an entity instance on which events are triggered. The methods should return void. The listener class should be stateless and should have a public no-argument constructor.

Code Snippet 4 shows the listener class annotated with the lifecycle callback methods.

Code Snippet 4:

```
public class Auditor {  
    @PostPersist  
    void postInsert(final Object entity) {  
        System.out.println("Inserted entity: " + entity.getClass().getName());  
    }  
    @PostLoad  
    void postLoad(final Object entity) {  
        System.out.println("Loaded entity: " + entity.getClass().getName());  
    }  
}
```

Code Snippet 4 declares the class Auditor that defines two methods namely, PostInsert() and PostLoad(). Both the methods are annotated with the lifecycle callback event annotations.

Then, the listener class can be applied on the entity class by using the @javax.persistence.EntityListeners annotation as shown in Code Snippet 5.

Code Snippet 5:

```
@Entity  
 @EntityListeners ({Auditor.class})  
 public class EntityListenerEmployee  
 {  
     ...  
 }
```

By using the @EntityListeners annotation on the EntityListenerEmployee entity class, any callback methods within those entity listener classes will be invoked, whenever EntityListenerEmployee entity instances interact with a persistence context.

12.5.3 Default Entity Listeners

You can specify a set of default entity listeners that are applied to every entity class in the persistence unit by using the <entity-listeners> element under the top-level <entity-mappings> element in the ORM mapping file.

Code Snippet 6 shows how to map Auditor listener for all entity classes in the application.

Code Snippet 6:

```
<entity-mappings>  
     <entity-listeners>  
         <entity-listener class="com.listener.Auditor">  
             <post-persist name="postInsert"/>  
             <post-load name="postLoad"/>  
         </entity-listener>  
     </entity-listeners>  
</entity-mappings>
```

12.5.4 Invocation Order of Callback Methods

If more than one callback method has to be invoked for a lifecycle event, then the invocation order is based on the following rules:

- All the external callback methods (which are defined in listeners) are invoked, before the internal callback methods (which are defined in entity classes).
- Default listeners are handled first, then listeners of the top level entity class, and then down the hierarchy until listeners of the actual entity class. If there is more than one default listener or more than one listener at the same level in the hierarchy, the invocation order follows the definition order.
- Internal callback methods are invoked starting at the top level entity class and then down the hierarchy until the callback methods in the actual entity class are invoked.

12.6 Using Second Level Cache in Persistence Applications

Caching is a technique used by enterprise applications to improve the application performance. This is primarily used to optimize the execution of expensive database calls. This cache is transparent to the application. The cache is managed by the persistence provider. The application reads and commits the data using the usual entity operations.

Caching can be implemented in applications at two levels – first level cache and second level cache.

First level caching stores the data in the cache for the duration of the transaction or application request. The first level cache is also known as `EntityManager` cache. This form of caching is essential for implementing proper semantics in JPA. Hence, `EntityManager` is responsible for this function.

Second level caching can span across multiple transactions and `EntityManagers` to improve application performance. Second level cache is available for the entire application. It enables the persistence provider to manage a local store of entity data which in turn improves performance. It also improves performance by reducing the number of database calls. Applications use `javax.persistence.Cache` interface to manage the second level cache.

There are different cache modes which define the pattern according to which the data is stored in the cache. These modes are set according to the application's performance requirements. Table 12.2 shows the cache modes defined in JPA.

Description	Cache Mode Setting
All the data accessed from the persistence provider is stored in the second level cache.	ALL
None of the data accessed from the persistence provider is stored in the second level cache.	NONE
This enables caching of the entities which are explicitly annotated with <code>@Cacheable</code> annotation.	ENABLE_SELECTIVE

Description	Cache Mode Setting
This enables caching of all the entities except those entities which are explicitly annotated with <code>@Cacheable(false)</code> annotation.	DISABLE_SELECTIVE
When the cache mode is not specified then the caching behavior is set to the default behavior of the persistence provider.	UNSPECIFIED

Table 12.2: Cache Mode Settings

Applications can define whether the entity data can be cached or not through the `javax.persistence.Cacheable` annotation. The cache mode can be set to `ENABLE_SELECTIVE` or `DISABLE_SELECTIVE` to avoid caching only a subset of entities in the database.

Following is the usage of `cacheable` annotation:

```

@Cacheable(true)

@Entity

public class Customer{ ... }

.....

@Cacheable(false)

@Entity

public class Loan_Application{ ... }

```

When the cache mode is set to `ENABLE_SPECIFIC`, all the entities which are marked with `@Cacheable(true)` are cached.

When the cache mode is set to `DISABLE_SPECIFIC`, all the entities which are marked with `@Cacheable(false)` are not cached. Entities other than the marked entities are cached.

When the cache mode is set to `UNSPECIFIED` all the `@Cacheable` annotations are ignored.

12.6.1 Specifying the Cache Mode Settings

The cache mode settings to a persistence unit can be set through the deployment descriptor. The element `<shared-cache-mode>` can be set in `persistence.xml` file.

Code Snippet 7 shows how to define the cache mode in the deployment descriptor.

Code Snippet 7:

```
...
<persistence-unit name="BankPU" transaction-type="JTA">
<provider>org.eclipse.persistence.jpa.PersistenceProvider</
provider>
<jta-data-source>java:comp/DefaultDataSource</jta-data-source>
<shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
</persistence-unit>
...
```

12.6.2 Specifying the Cache Retrieval and Store Mode

The behavior of the application with respect to the secondary cache should be defined both during retrieving data from the database and while storing data on the database. Therefore, the developer has to set `javax.persistence.cache.retrieveMode` and `javax.persistence.cache.storeMode`.

These properties can be set through the `EntityManager` instance, using the `setProperty()` method.

Cache Retrieval mode can be set to either `USE` or `BYPASS`. This cache retrieval mode is an enumerated type. When the mode is set to `USE` it implies that the data should be retrieved from the database by storing a copy in the cache. When the retrieval mode is set to `BYPASS` then a copy of the data retrieved from the database is not written to the cache.

Cache Store mode defines how the data is stored on the database. This mode can have any one of the three values – `USE`, `BYPASS`, and `REFRESH`.

When the mode is set to `USE` then the data is created or updated when the data is committed to the database. When the mode value is set to `BYPASS` then the value is not written to the cache when it is written to the database. When the mode value is set to `REFRESH` then the data is read from the database, the value in the cache is refreshed.

The cache retrieval or store mode can be set through the `EntityManager` as follows:

```
EntityManager em=...;
em.setProperty("javax.persistence.cache.storeMode", "BYPASS");
```

The cache mode can also be set through the `Query` and `TypedQuery` objects through `setHint()` method as follows:

```
EntityManager em=...;
CriteriaQuery<Customer> cq=...;
TypedQuery<Customer> q=em.createQuery(cq);
q.setHint("javax.persistence.cache.storeMode", "REFRESH");
```

12.6.3 Controlling the Cache Settings Programmatically

Developers can use methods defined in `javax.persistence.Cache` to define the caching behavior of the application. The `Cache` interface has methods defined for the following tasks:

- To check if a given entity has cached data or not.
- To remove a particular entity from the cache.
- To remove instances of a given entity class from the cache.
- To clear cache and remove all entity data from it.

Following are the methods defined in `Cache` interface:

- `contains()` – Accepts the entity class and search attribute as input parameters and returns a boolean value. The returned value signifies whether the given entity is present in the cache or not.
- `evict()` - This method is an overloaded method which accepts an entity class as parameter and removes all the instances of the given entity class from the cache. The overloaded method accepts the entity class name and search attribute as parameters and removes the entity from the cache.
- `evictAll()` – Removes all the entities in the cache and returns a null value.

Code Snippet 8 shows the entity class which is accessed through cache.

Code Snippet 8:

```
package data;

...
@Entity
@Table(name = "EMPLOYEE")
@Cacheable
@XmlElement
public class Employee implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @NotNull
    @Column(name = "EMP_ID")
    private Integer empId;
    @Size(max = 15)
    @Column(name = "ENAME")
    private String ename;
```

```
// @Max(value=?) @Min(value=?) //if you know range of your decimal  
fields consider using these annotations to enforce field validation  
  
@Column(name = "SALARY")  
  
private Double salary;  
  
  
public Employee() {  
}  
  
  
public Employee(Integer empId, String name, Double sal) {  
  
    this.empId=empId;  
  
    this.ename=name;  
  
    this.salary=sal;  
}  
  
  
public Integer getEmpId() {  
  
    return empId;  
}  
  
  
public void setEmpId(Integer empId) {  
  
    this.empId=empId;  
}  
  
  
public String getEname() {  
  
    return ename;  
}  
  
public void setEname(String ename) {  
  
    this.ename=ename;  
}  
  
public Double getSalary() {
```

```
        return salary;
    }

    public void setSalary(Double salary) {
        this.salary = salary;
    }

    @Override
    public int hashCode() {
        int hash = 0;
        hash += (empId != null ? empId.hashCode() : 0);
        return hash;
    }

    @Override
    public boolean equals(Object object) {
        // TODO: Warning - this method won't work in the case the id fields are
        // not set
        if (!(object instanceof Employee)) {
            return false;
        }
        Employee other = (Employee) object;
        if ((this.empId == null && other.empId != null) || (this.empId != null && !this.empId.equals(other.empId))) {
            return false;
        }
        return true;
    }

    @Override
    public String toString() {
        return "data.Employee[ empId=" + empId + " ]";
    }
}
```

Code Snippet 9 shows the Employee entity class. In code snippet 6, this entity class is accessed through cache.

Code Snippet 9:

```
package access;

. . .

import javax.persistence.Persistence;
import javax.persistence.PersistenceContext;
@Stateful
@LocalBean
@Cacheable

public class AccessEmployee {

    @PersistenceContext (unitName = "CacheAccessPU")

    public EntityManagerFactory emf = Persistence.
    createEntityManagerFactory ("CacheAccessPU");

    public EntityManager em;

    Employee E = new Employee(101, "Alex", 25000D);

    Cache C = emf.getCache();

    public void store() {

        String out = E.getEname();

        System.out.println("Here "+out);

        if (C.contains(Employee.class, out))

            {

                System.out.println("In cache");

            }

    }

    public static void main(String args[]) {

        AccessEmployee A = new AccessEmployee();

        A.store();

    }

}
```

In Code Snippet 9, an EntityManager instance is created which is used to manage the `Employee` entity class. An entity of type Employee is created through the Employee constructor. The `getCache()` returns the cache allocated for the given entity class.

This entity is accessed through the cache. In order to check whether the data is present in the cache or not `contains()` method of the Cache object is used. If the data is present in the cache then the method `contains()` returns true and executes the corresponding if block.

Check Your Progress

1. Which of the following is not a valid lock mode?

(A)	OPTIMISTIC_FORCE_INCREMENT	(C)	PESSIMISTIC_WRITE
(B)	OPTIMISTIC_WRITE	(D)	PESSIMISTIC_FORCE_INCREMENT

2. Which of the following data types cannot be assumed by a version attribute?

(A)	Integer	(C)	Long
(B)	Short	(D)	Float

3. Which of the following exception is thrown when the application tries to set a PESSIMISTIC_FORCE_INCREMENT mode on a non-versioned entity?

(A)	PersistenceException	(C)	LockTimeoutException
(B)	OptimisticLockException	(D)	None of these

4. Which of the following is not type of cache retrieval mode?

(A)	USE	(C)	REFRESH
(B)	BYPASS	(D)	None of these

5. Which of the following is an invalid way of setting timeout period for acquiring pessimistic lock?

(A)	Through EntityManager instance	(C)	Through deployment descriptor
(B)	Through @NamedQuery annotation	(D)	Through shared-lock-mode element

6. Which of the following is an invalid type for version attribute in entity tables?

(A)	File	(C)	Integer
(B)	Timestamp	(D)	None of these

Answer

1.	A
2.	D
3.	A
4.	C
5.	D
6.	A

Summary

- Concurrent access to the database is always protected with the transaction isolation techniques.
- Database providers provide locking techniques to maintain data integrity. In addition to that, persistence providers also provide delay database writes, until the end of the transaction.
- Optimistic locking technique does not acquire locks for providing concurrency control, while performing any operation on certain data.
- Optimistic locking can be done either by annotating the entity field with javax.persistence.Version annotation or specify the version attribute in the XML descriptor file.
- Sometimes, it is desirable to acquire the database locks for a long-terms on entities. Such immediate obtained database locks are referred to as 'pessimistic' locks.
- JPA provides lock modes which are layered on the top of the @Version annotation provided in the optimistic and pessimistic locking.
- An entity contains a predefined set of lifecycle events that are triggered on the execution of methods called from EntityManager or Query API.
- The entity listeners are classes that are intercept entity callback events.
- You can specify a set of default entity listeners that are applied to every entity class in the persistence unit.
- Caching is a technique used by enterprise applications to improve the application performance.
- First level caching stores the data in the cache for the duration of the transaction or application request.
- Second level caching can span across multiple transactions and EntityManager to improve application performance.
- Developers can use methods defined in javax.persistence.Cache to define the caching behavior of the application.



Login to www.onlinevarsity.com



Welcome to the Session, **Security**.

This session discusses securing various aspects of the enterprise application. It explains how to assign access rights to different categories of users accessing the application. It also discusses the security mechanisms provided by Java EE 7 for authentication and authorization of the users accessing the application. Further, the session explains the architecture of Java Authentication and Authorization Service (JAAS) and how it can be used to secure application components and application clients.

In this Session, you will learn to:

- Describe enterprise application security
- Explain how to implement security at various levels in an application
- Explain how roles, users, and user groups are defined in an application
- Define authorization and authentication mechanisms used in enterprise applications
- Explain JASS architecture and its services
- Explain how to secure application clients

13.1 Overview of Java EE Security

Enterprise applications have various components that need to be protected from unwanted access. When an enterprise application is accessed through the Internet or any other open network, the users accessing the application components must be appropriately authenticated and authorized, before they can access the services from the application.

All the application components are deployed on the application server and are logically managed through the container. The container is responsible for providing security services for the components deployed in it.

The security requirements of the application are defined by the application domain. Based on the requirements, application developers need to define the security policy using mechanisms such as user authentication, password protection of resources, data encryption, and so on.

In Java EE, the security policy is implemented within the application code through annotations and deployment descriptors. As the application components are accessed through the container, the container implements the security policy defined in the application code. Java EE also provides classes and interfaces to implement the security policy programmatically.

13.1.1 Implementing Security at Various Levels in Enterprise Applications

Enterprise application security can be implemented at three different levels in the application. This security is in addition to the operating system level security provided to the application data. Following are the three levels in which enterprise application security can be implemented:

- Application Layer Security
- Transport Layer Security
- Message Layer Security

13.1.2 Application Layer Security

All the EJB components are deployed in a container on the application layer. Application layer security is implemented by the application container. Firewalls can be used at the application layer level to implement the security requirements.

Application layer security is defined for containers either declaratively or programmatically.

While defining declaratively the application uses annotations and deployment descriptors. Annotations are specified as part of the application code and deployment descriptor is an XML file.

Annotations can be used to specify security information within class files. The application server uses this information while deploying the application. Annotations can specify security information pertaining only to the class in which it is present. Security information with a wider scope should be defined in the deployment descriptor.

Deployment descriptors are XML files which can be used to specify security information for the application. Every application has a default deployment descriptor with a specific name. While deploying the application the server looks up this specific file for deployment information.

Enterprise bean applications use `ejb-jar.xml` as deployment descriptor and Web applications use `web.xml`. Security information can be specified in these files for the application.

Programmatic definition of application security involves using methods of `EJBContext` and other interfaces provided by Java EE.

13.1.3 Transport Layer Security

Transport layer security refers to the security mechanisms implemented while the application data is transmitted through the network. Transport layer uses HyperText Transfer Protocol (HTTP) using Secure Sockets Layer. This is a point to point security mechanism. It implements message integrity, authentication, and confidentiality of the data transmitted.

The transport layer security uses cryptographic techniques for security. Following are the steps involved in implementing the transport layer security:

1. The client and server entity agree upon a cryptographic algorithm.
2. The secret key through which the communication will happen is exchanged through public key cryptography and certificate based authentication.
3. The agreed upon key is used to generate symmetric cipher which is then exchanged on the network link.

Transport layer security is unaware of the contents of message being transmitted. It applies the security mechanism either to the entire content of the message or none. It cannot be applied to a portion of the message content.

13.1.4 Message Layer Security

The security information is bundled along with the Simple Object Access Protocol (SOAP) message. The security information travels to the destination along with the message. Message layer security is also known as end-to-end security. When the message with encrypted information is transmitted from the sender, it passes through several intermediate nodes and reaches the destination. The encrypted SOAP message is only decrypted by the receiver.

Unlike transport layer security, message layer security can be selectively applied on a part of the message. It can be implemented independent of application environment.

Figure 13.1 shows security implementation at various layers.

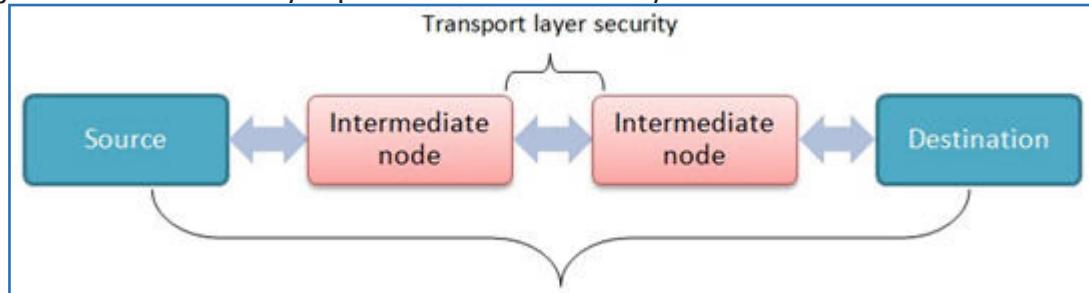


Figure 13.1: Security Implementation across Layers

13.1.5 Characteristics of Security Mechanisms

A security mechanism can be implemented at any of the three layers of the application based on the application requirement. Wherever implemented, the security mechanism should have certain essential characteristics.

Following are the essential characteristics a security mechanism used in an application should have:

- Prevent unauthorised access to the application data and application components or resources.
- When an application user performs certain operations on the enterprise application, then the identity of the user should be associated with the operations performed. The user cannot deny the performed operations. This characteristic is essential to trace the users who have performed malicious operations in the application.
- Protect the application from service failures such as server crash, network failure, and other interruptions.

13.1.6 Features of Application Security

All the resources in the application do not require user authentication to be accessed. Certain resources can be accessed by all the users accessing the application. Such resources are called unprotected resources. These resources can be anonymously accessed.

The resources which cannot be accessed without user authentication are known as protected resources. These resources are protected by the security mechanism. Proper implementation of the following features is required in a security mechanism to reduce the risk of security threats to the application:

- **Authentication** – Authentication is a process by which one entity in an interaction determines the identity of the other. In an EJB application, clients of EJBs may be applications or other EJBs. The EJB server determines the identity of all types of clients so, that it can determine the level of access to be granted. On the other hand, the client may also want to authenticate the server, to ensure that it is interacting with the correct server.

The most common form of authentication involves the use of username and password. There are different variations of this scheme, which involves different methods for encrypting and transmitting the password. One main concern with password authentication is that if someone else fraudulently gets your username and password, they can assume your identity.

The use of digital certificates offers a stronger form of authentication. Digital certificates can be used to identify end-users, servers, and other software components. A digital certificate is an electronic identity card that establishes your credentials. It is issued by a third-party Certification Authority (CA) and contains a serial number, expiration dates, the certificate holder's public key, and the digital signature of the CA. The certificate holder's public key is used for encrypting messages and the digital signature of the certification authority verifies that the certificate is real.

Digital certificate-based authentication can be used only on the server, the client, or both, depending on the needs of the application. In most of the cases, mutual digital certificate-based authentication is used so that both the client and the server are confident of each other's identity.

Figure 13.2 depicts authentication.

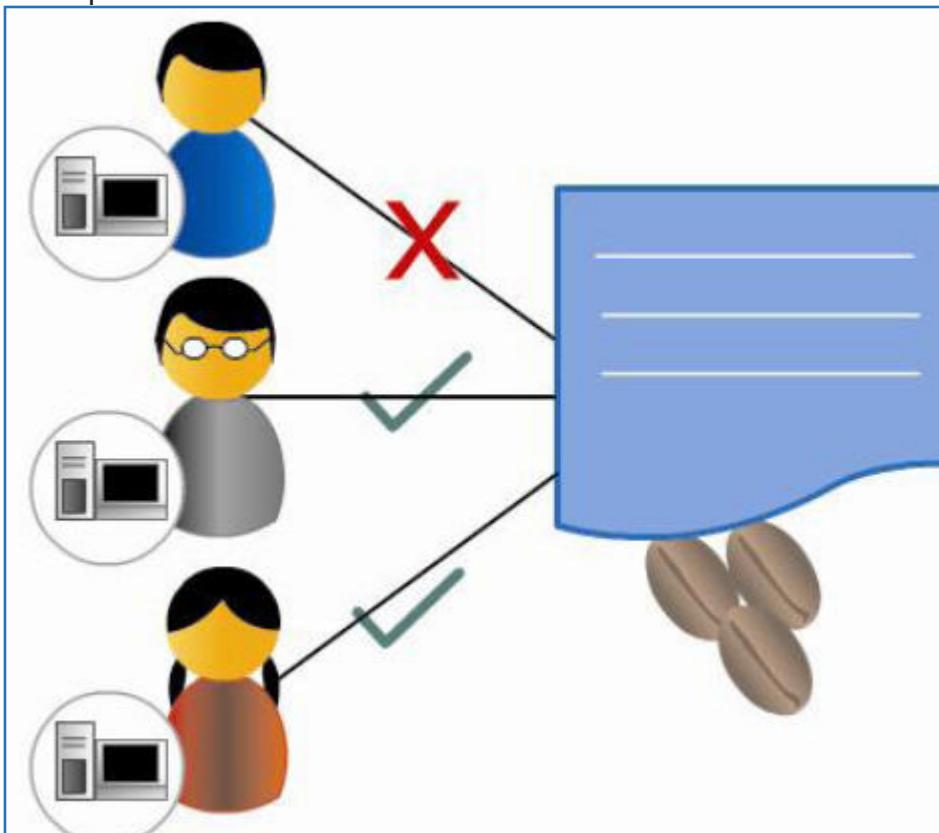


Figure 13.2: Authentication

- **Authorization** – In an enterprise application, client authentication is usually followed by authorization. While authentication ensures only valid users get access to the application. Authorization controls what the authenticated user is allowed to do after he/she is granted access.

For some simple EJB applications, you may only need to use authentication. However, in many enterprise applications, there are some administrative functions that a normal user should not have access. One of the ways that this can be prevented is by creating a list of permissions for actions that a user of each type can perform and then allowing the users to perform only those tasks that are permitted by users of that type.

Figure 13.3 depicts authorization.

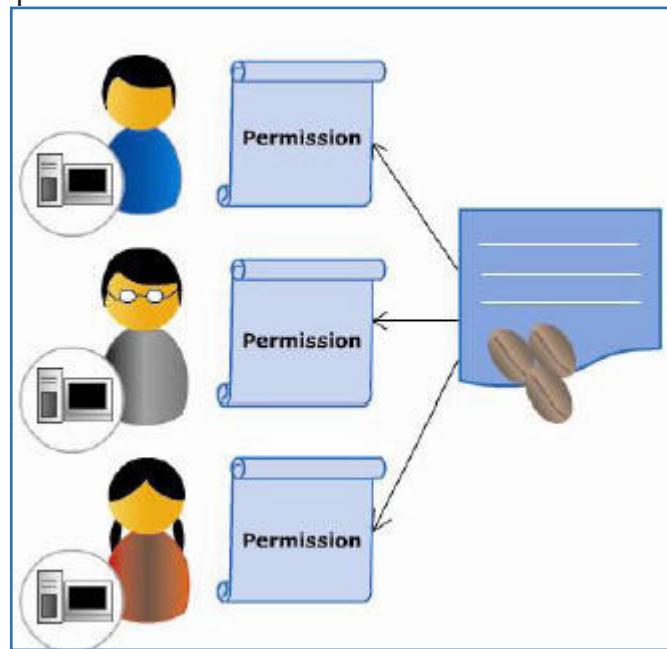


Figure 13.3: Authorization

- **Data integrity** - Data integrity is a characteristic which requires that the information is not modified by unwarranted users. Applications implement various checks on the data such as Cyclic Redundancy Checksum (CRC) codes and so on to detect whether the information is modified by any third party users.
- **Confidentiality** - Confidentiality implies secrecy, where by the security system allows access of data only to authorized users of the application.
- **Non-repudiation** – This security mechanism associates the identity of the user with actions performed by them on the application. If a user performs a malicious operation on the application, the security mechanism ensures that the user does not deny the operations performed.
- **Quality of Service** – The security mechanism implemented in the application increases the application execution time. The increase in the application execution time is because of executing the cryptographic algorithms, initiating the authentication process, looking up the authorized user list, and so on. For instance, when access to a resource requires username and password the application execution cannot proceed until the user provides the appropriate information. These delays add to the execution time of the application. However, this additional overhead should not affect the application performance.
- **Auditing** – All the operations performed by the application system are recorded in a tamper proof location. This is termed as an application log. Auditing of an application log is done to ensure that the application is performing as expected.

13.2 Simple Application Security Implementation

To understand the security implementation in an enterprise application, let us take the example of a Bank domain.

Following are some of the requirements of a Bank application which have to be implemented:

- Every customer of the bank is provided with online access to the account.
- The customer can only access his/her account. The customer can check balance in his/her account, transfer funds from the account to another account through the online portal.

In order to implement these simple requirements, the application has to perform the following steps:

1. Every user is provided with a unique username. This username is linked with the account held by the user.
2. Whenever a user tries to access an account, the application server should prompt for a user name and a corresponding password.
3. The application server checks the username and password against the customer database it holds and verifies the information. This process is known as authentication.
4. The customer of a bank can perform operations such as check the account balance and transfer funds to other accounts. Therefore, the username of the customer is associated with the operations the user can perform.
5. This process of allowing a user to perform certain operations is known as authorization. Usually the authentication process is followed by the authorization process.
6. Once the user authentication and authorization is complete, the customer can perform the necessary operations such as funds transfer and so on. The user need not enter the username and password for every operation performed, the user authorization is valid for a certain time period.
7. To perform the operations initiated by the customer the application has to invoke enterprise beans on the application server.

These steps can be generalized for any application domain. An application server is responsible for user authentication and authorization of a user who is trying to access the enterprise application.

Following are the steps involved in a typical application execution with a security mechanism in place:

- Request** - The application client or end user initiates an application request, this application request may have to access EJB components deployed in the container.
- Authentication** - The application server authenticates the application clients by prompting for the username and password. The application server may also use any other security mechanism to validate the user.
- URL authorization** – The credentials provided are used to determine whether the given user is authorized to access resources or not. Access to appropriate resources is provided according to the security policy of the application.
- Fulfilling original requests** - The application server sends requests to the security policy defined for the application to determine the resources to be accessed. The application request which in turn has initiated the authentication process is then fulfilled.
- Invoking Enterprise Bean methods** – The application request from the client is satisfied by invoking appropriate enterprise bean methods. The EJB container is responsible for securing the enterprise bean methods. The container refers to the security policy of the application and invokes the bean methods to which the user has access rights.

As the container is responsible for providing access to the enterprise components, the application server sends the outcome of the authentication process to the container. Based on the result, the container grants or rejects the requested access from the application client.

13.2.1 Access Control Lists (ACLs)

Permissions represent a right to access a particular resource or to perform some action on an application. An administrator usually protects resources by creating lists of users and groups that have the permission to access a particular resource. These lists are referred as Access Control Lists (ACLs). For example, a user with admin permissions on a chat site may create, modify, or close a chat room, but a user with normal permissions may be allowed only to enter a chat room and participate in a chat session.

An ACL file is made up of entries, which contain a set of permissions for a particular resource and a set of users who can access those resources.

Figure 13.4 shows access control list.



	Read	Write	Execute
Admin	✓	✓	✓
Professor	✓	✓	✓
Staff	✓		
Student	✓		

Figure 13.4: Access Control Lists

13.3 Users, Groups, Roles, and Realms

An application domain has various end users. For instance, in a bank application there are users such as customers, bank employees, and so on. Every end user of the application is assigned a unique identity in the context of the application as mentioned.

The users are logically grouped into user groups based on some common characteristics.

Every end user is termed as a user of the application; however, different users may have different roles to play in the application. For instance, an employee's role in a bank application is different from the customer's role in the same domain.

A realm is a single authentication policy that controls a set of users or user groups. An admin realm in an application therefore, grants administrative rights required by the application to the admin group.

Figure 13.5 shows the interaction between users, groups, and roles.

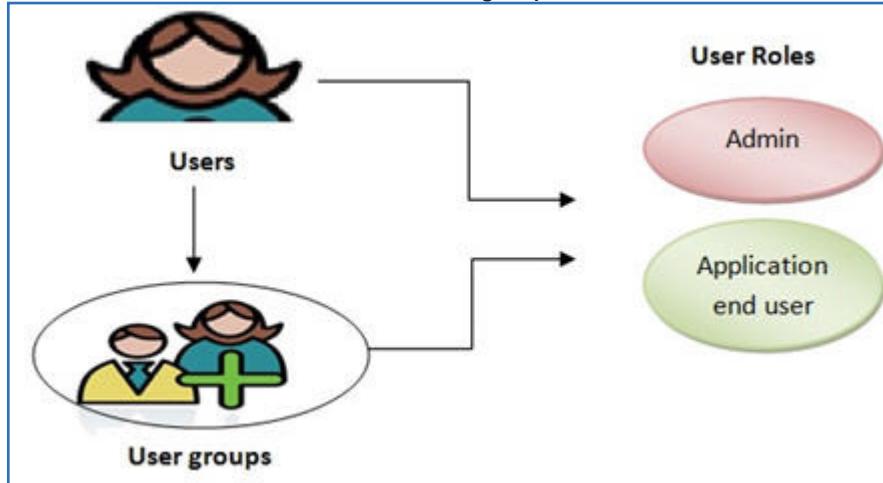


Figure 13.5: Users, Groups, and Roles

Figure 13.4 illustrates that each user in the application domain can be mapped onto an user group or a role. User groups are a set of users in the application domain with similar requirements. The users of the group are provided with similar access rights.

User roles represent different sets of access rights. The tasks that can be performed by a user who is in the role of an admin are different from the tasks that can be performed by an application end user. The access rights given to each of these users are different.

An application role can be assumed by a single user or a group of users in the application domain.

13.3.1 Users

A user is an end user or application program which accesses the enterprise application. The identity of the user is defined on the application server such as GlassFish.

The user presents his/her identity which is the username to the application server while trying to access the application resources. Based on the user name, the application server identifies the role assumed by the user and grants appropriate access rights to the user. The server maps it to the resources according to the definition of its role. There are no two identical users in the application domain. Each user has its own identity with credentials.

13.3.2 Groups

Group refers to a set of authenticated users with similar roles and access rights in the application. These groups can also be defined according to application semantics. For instance, in the context of a bank application, there can be two categories of users – the employees of the bank and the customers of the bank. These two user groups have different requirements and different access privileges. Both of them can access the bank application, however, operations performed by each of the user group are different. A customer may apply for loan and an employee can approve the loan.

13.3.3 Roles

Role in the application domain reflects the set of access rights held by a certain user. For instance, all employees in the bank application do not have the right to credit interest into the accounts of customers. This privilege lies with only a subset of the users. The application has to define a role with the required privileges assigned to that role so that an employee can perform the task of crediting interest into the accounts of the customers. For example, manager, administrators, and buyers are roles that are supported in an e-shopping Web site. Each role contains a particular set of permissions. While creating an EJB, you can specify which role can perform what action in your application.

Figure 13.6 shows the groups and roles.

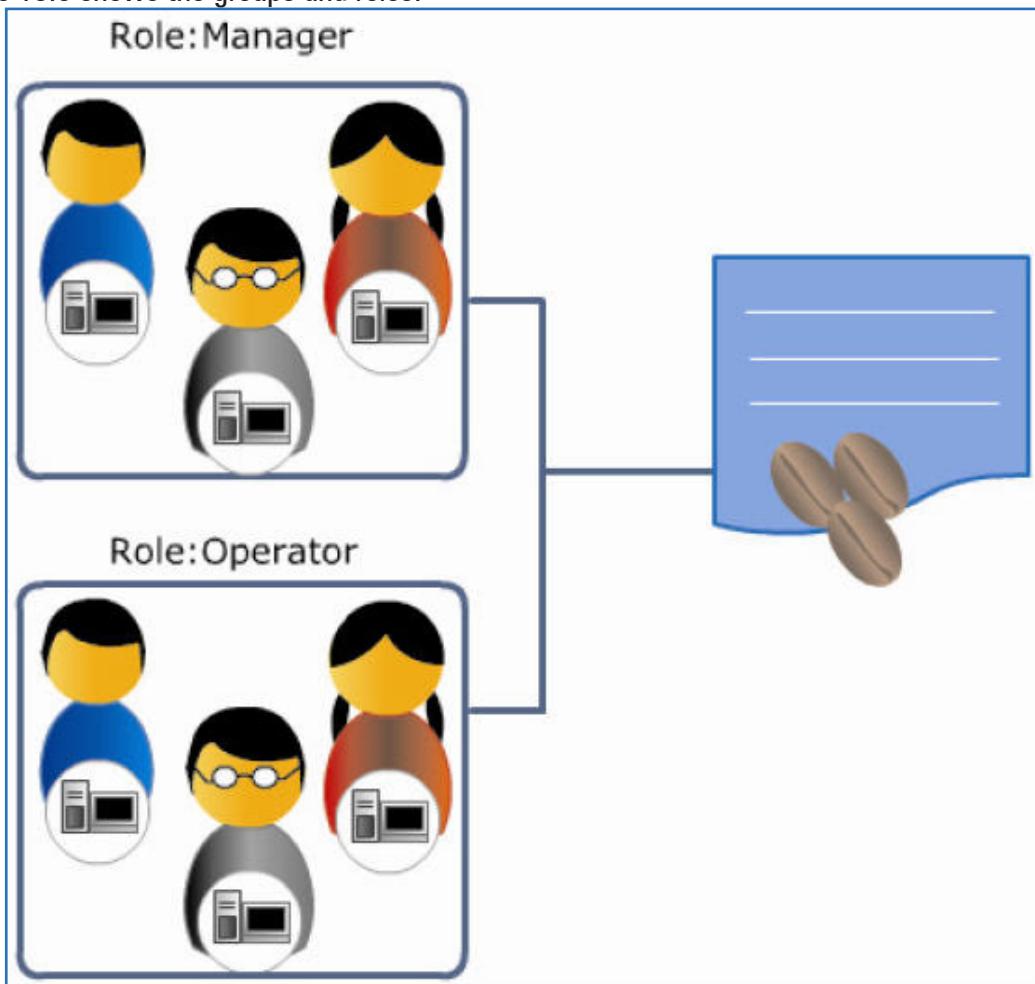


Figure 13.6: Groups and Roles

13.3.4 Realms

Realm is the protection policy implemented by the application. The protected resources on the application server are partitioned into a set of protection spaces. Each protection space is associated with an authentication scheme and an authorized database of users.

The realm refers to the database of authorized users controlled through the same authentication policy.

There are three default realms in Java EE server authentication service through which users are governed – admin realm, certificate realm, and file realm.

admin-realm stores all the user credential data in the admin-keyfile. The users in the admin realm are authenticated with the help of admin-keyfile. The authentication information in this case is locally stored.

certificate-realm stores all the user credential data in a certificate database. This mechanism uses X.509 digital certificates for user authentication. While using certificate realm, the application server uses HTTPs to authenticate the Web client. This mechanism is primarily used for Web applications.

When the application uses file realm, user credential data is stored in a file called keyfile. The user is authenticated using the data in the keyfile. The server verifies the user by referring to the keyfile. File realm can be used for all the enterprise clients but not for Web browser clients and those using HTTPs.

13.3.5 Managing Users and Groups on GlassFish Server

GlassFish server is one of the most commonly used application servers for Java enterprise applications. The users, roles, and user groups can be managed on this server through the Domain Admin Console. The Domain Admin Console can be accessed by right-clicking the GlassFish server. The GlassFish server should be in running state to define the users and user groups.

Figure 13.7 shows the Domain Admin Console of the GlassFish server in NetBeans IDE.

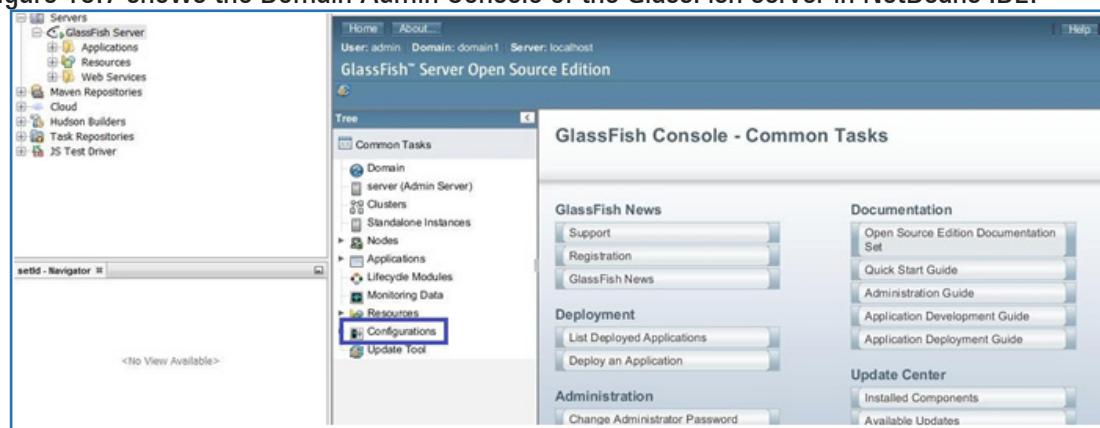


Figure 13.7: Domain Admin Console of GlassFish Server

In figure 13.7, the **Configurations** option in the **Domain Admin Console** can be used to configure the security features of the application.

Figure 13.8 shows the hierarchy through which the user can choose the realms to be used for the current application.



Figure 13.8: Choosing Security Realms

There are three types of realms from which the developer can choose the appropriate realm. Figure 13.9 shows the realms available with the Java application.

Select	Name	Class Name
<input type="checkbox"/>	admin-realm	com.sun.enterprise.security.auth.realm.file.FileRealm
<input type="checkbox"/>	certificate	com.sun.enterprise.security.auth.realm.certificate.CertificateRealm
<input type="checkbox"/>	file	com.sun.enterprise.security.auth.realm.file.FileRealm

Figure 13.9: Realms

The developer can choose any one of these realms and assign or remove users from a realm. For demonstration purposes, the files realm has been selected here. On selecting the files realm option, the IDE leads to the screen as shown in figure 13.10.

To manage users in a realm, in Netbeans IDE the 'Manage users' option is selected as shown in Figure 13.10.



Figure 13.10: Managing Users in the Realm

Instead of choosing individual users if the administrator intends to map a user group the administrator can use the 'Assign Groups' field. In the 'Assign Groups' text box the administrator can provide a comma separated list of the user groups to which the file realm is applicable.

On clicking 'Manage Users', it leads to the screen as shown in Figure 13.11.

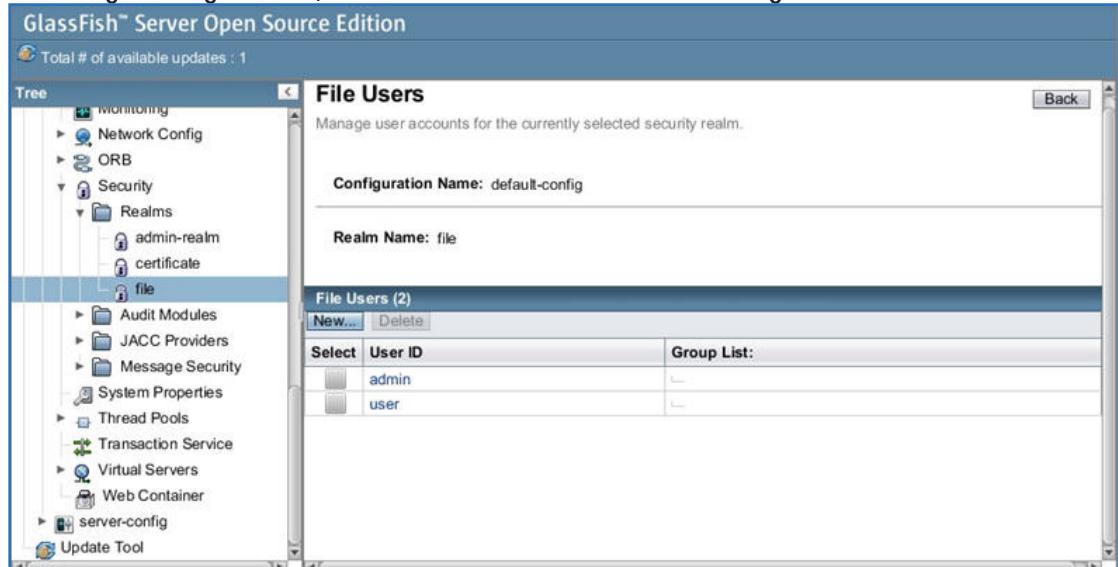


Figure 13.11: Creating and Managing Users through the Console

The screen in Figure 13.11 enables the administrator to assign the existing users to different realms. New users can be created through the console by clicking 'New user' option.

Figure 13.12 shows how new users can be created.

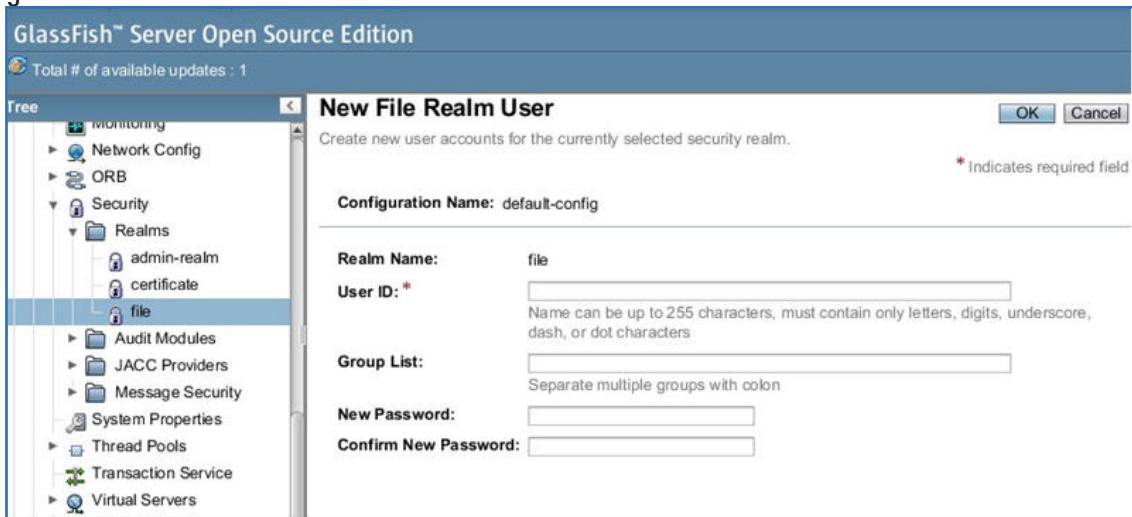


Figure 13.12: Creating New Users

The new users are assigned to different roles and user groups.

13.3.6 Creating and Mapping Roles to Users in the Application

Roles and their corresponding access rights in an application can be defined using annotations in the application. Following are the annotations which can be used to define the roles and their access rights:

- `@DeclareRoles` annotation is used to declare roles in the application domain which have the right to access the application resources or business methods.
- `@RolesAllowed` annotation is used to specify the application roles that are allowed to access the application resource.
- `@PermitAll` annotation specifies that all the security roles can access the given bean method. This does not invoke any authentication mechanism. This annotation can be used for both bean classes and bean methods.
- `@DenyAll` annotation specifies that no security role can access the current bean class or bean method. Such methods are excluded from execution in Java EE container.

Code Snippet 1 shows the usage of the `@DeclareRoles` and `@RolesAllowed` annotations.

Code Snippet 1:

```
import javax.annotation.security.DeclareRoles;  
import javax.annotation.security.RolesAllowed;  
  
...  
@DeclareRoles({"VALUATOR", "MANAGER"})  
@Stateless public class LoanApprovalBean {  
    @Resource SessionContext ctx;  
    @RolesAllowed("VALUATOR")  
    public void reviewPropertyValue(PropertyInfo info) {  
        .....  
    }  
    @RolesAllowed("MANAGER")  
    public void ApproveLoan(PropertyInfo info) {  
        ...  
    }  
    ...  
}
```

Code Snippet 1 shows two operations that can be performed in the context of a bank application. The application has two roles defined in the context of loan approval – valuator and manager. Each of these roles can access the `reviewPropertyValue()` and `ApproveLoan()` methods.

13.4 Establishing a Secure SSL Connection

Secure Sockets Layer is the technology used in the transport layer for secure communication. This is only applied between two adjacent network components. It uses cryptographic techniques to implement point-to-point security.

It implements three essential characteristics of a security mechanism – authentication, confidentiality, and integrity.

Two communicating entities agree upon a shared secret key using public key cryptography. The communicating entities then use the shared secret key to encrypt the data at the source and decrypt the data at the destination. Though SSL is efficiently designed, the cryptographic process adds an overhead on application performance.

SSL is used in scenarios where the application clients communicate with the application server through an open network.

13.5 Security Tasks in Enterprise Applications

Responsibility of implementing application security is shared among system administrators, application developers, bean providers, and deployers.

- System administrators are responsible for creating roles and users. Administrators are also responsible for mapping the users onto appropriate roles and user groups.
- Developers can also create roles and provide access rights to different roles using annotations or deployment descriptors.
- Deployers are responsible for deploying the application on the server according to the security specifications provided in the deployment descriptor.
- The bean provider supports the security mechanisms required by the application.

Following are the tasks which must be performed as part of security implementation for the application:

1. Creating a database of users who will be accessing the application.
2. Defining relevant user groups according to the application context.
3. Assigning the users to appropriate groups.
4. When the user provides authentication information, propagating the data across all the application components. This process is known as identity propagation, where the authenticated user id is propagated across all relevant application components.
5. Configuring the application server with appropriate user and role mappings.
6. Annotating the classes appropriately to declare roles and defining the access to be granted to different roles.

13.6 Securing Enterprise Beans

Enterprise beans use EJB technology to implement application logic. As they provide the core application functionality they need to be secured from unauthorized access. The security of the enterprise beans can therefore, be implemented either declaratively or programmatically.

13.6.1 Securing an Enterprise Bean Declaratively

While defining declaratively, the application deployer defines the security features on the application server based on the deployment descriptor and annotations. The deployer defines the users, user groups and their respective roles on the application server. The security specification of the application is also known as security view of the application.

13.6.2 Securing an Enterprise Bean Method Programmatically

In this method of specifying the security mechanism, the developer uses the security APIs and methods to define the security mechanisms.

13.6.3 Accessing an Enterprise Bean Caller's Security Context

Defining the security mechanisms declaratively is a preferred way of defining a security mechanism. `javax.ejb.EJBContext` provides methods to access security information about the user or entity who is invoking the enterprise bean method.

Following are the methods provided by the `EJBContext` interface:

- `getCallerPrincipal()` – This method is used to retrieve the name of the invoking entity or the user. The security methods may then use the name of the entities to check the user database to determine the role of the current entity.
- `isCallerInRole()` – This method is used to check whether the current user has been assigned with certain role or not. The application developer defines the roles for the users. If an application context object 'X' is trying to access methods which are meant for administrator, then the security mechanism can check whether the object has administrative rights or not through `isCallerInRole()` method.

Code Snippet 2 shows the usage of `isCallerRole()` method.

Code Snippet 2:

```
...
@Resource Session Context X
if (X.isCallerRole(admin)==true)
{
    System.out.println("Admin right assigned");
}
else
    System.out.println(" No Admin rights");
...
```

13.7 Security Mechanisms Provided by Java EE

While defining the security policy of the application, the designer should first understand the required security characteristics of the application. Based on the security requirements and available mechanisms the security policy and its implementation can be defined. Following are the security mechanisms provided by Java EE:

- Java Generic Security Services (Java GSS-API) - It is used for implementing security mechanisms during communication over the network. This is a token based API used for exchanging messages securely.
- Java Cryptography Extension (JCE) – JCE is used to implement cryptographic structures within the application. Developers can define implementations of Message Authentication Code (MAC) algorithms, key generation, and so on. The API also enables generating cipher text in the application. It allows incorporation of cryptographic structures in the application at various levels.
- Java Secure Sockets Extension (JSSE) – It provides a Java version of implementation of Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols. It provides the functionality of encryption, server authentication, message integrity, and so on.
- Simple Authentication and Security Layer (SASL) - It is an Internet standard which specifies a protocol for authentication and exchange of authentication data between the client and server applications.
- Java Authentication and Authorization Service (JAAS) – It is a set of APIs used to define authentication and authorization mechanisms of the application. It provides a pluggable and extensible framework for developers.

13.8 Java Authentication and Authorisation Service (JAAS)

JAAS implements a Java version of Pluggable Authentication Module (PAM) framework. Therefore, JAAS enables independent development of the authentication modules and the underlying authentication technologies.

JAAS provides the following classes and interfaces to implement security mechanisms in the application:

- LoginModule – LoginModule is an interface which is implemented by the application to define the login process. Application developer writes the code for the authentication and authorization process.
- LoginContext – LoginContext initiates the authentication process. It creates a Subject in the application execution environment.
- Subject – An instance of Subject is used to represent the user who is trying to access the protected application resources.
- Principal – Principal refers to the properties associated with the Subject which are used in the authentication process.

These are part of the core class library of JAAS. Apart from these there are other classes such as Credentials, CallBackHandler, and so on which can be used in the authentication and authorization processes. Applications enable authentication process by creating instances of LoginContext. This in turn references a Configuration or LoginModule for performing the authentication.

Note that Pluggable Authentication Module (PAM) is a mechanism which integrates various low-level authentication schemes into a high-level API. It allows independent development of the code that uses the authentication scheme. These independent components can be later plugged to implement the authentication mechanism.

13.8.1 JAAS Authentication

Authentication process through JAAS involves the following steps:

1. Create a `LoginContext`, the client application accesses the authentication mechanism through an instance of `LoginContext`.
2. The `LoginContext` module accesses the `LoginModule`, which is defined in the configuration file.
3. The authentication is performed through the `LoginModule`.
4. In the authentication process, a `CallBackHandler` is used to communicate with the client and acquire authentication information such as username, password, and so on.
5. If the authentication process fails or login process was unsuccessful, a `LoginException` is thrown.
6. `LoginContext` is used to logout from the session.

Figure 13.13 shows the flow of control during the JAAS authentication process.

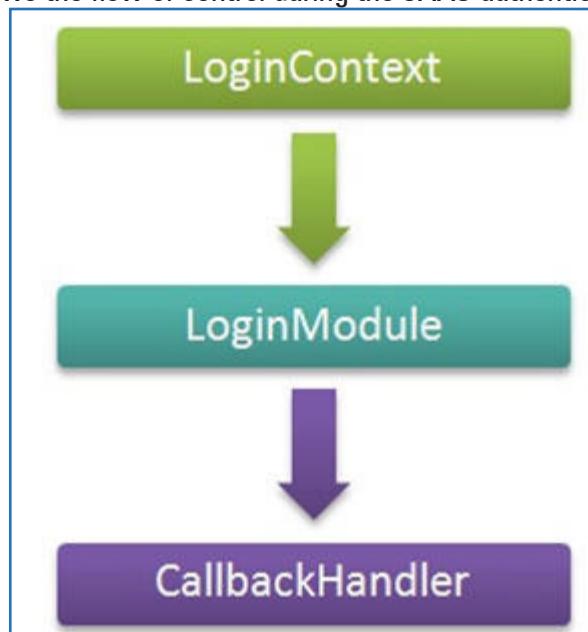


Figure 13.13: JAAS Authentication

13.8.2 JAAS Authorization

Authorization is the process of determining the operations that can be performed by an authenticated user. Authorization process is dependent on the security policy defined by the application. It uses a policy configuration file defined for the application. The authenticated user who is trying to access a protected resource is an instance of `Subject`. Each `Subject` is associated with a `Permission` instance indicating the access rights of the user. The subjects are managed by a `SecurityManager` instance in the application.

The Authorization process also involves instances of AccessController and AccessControlContext through which the permissions of the authenticated user are checked and granted access.

Following are the steps involved in the authorization process:

1. The doAs() method of Subject class is invoked to associate a role to the authenticated user.
2. The SecurityManager checks the permissions associated with the Subject using checkPermission() method. It in turn invokes the AccessController.
3. AccessController performs the required check and updates the AccessControlContext with the Subject and its associated permissions.

13.9 Propagating a Security Identity

When the application server grants access of an application resource to a user, the user can access the protected resource. While accessing the resource the user may also require access to other protected resources in the application. In this scenario, instead of initiating the authentication process again, the identity of the authenticated user can be propagated to the second access request initiated by the user.

The user identity can be propagated through any one of the following options:

- The identity of the entity through which the user accessed the first entity can be propagated to the second entity by default. This technique is used when the intermediate entity is a trusted entity.
- When the target enterprise bean expects a specific identity, then the expected identity is forwarded to the target enterprise bean. In order to propagate an identity to the target enterprise bean, a run-as identity is configured for the bean. The 'run as' identity is one which is used by the enterprise bean when it makes calls. The 'run as' identity is bound to a user whose role is determined after authentication. The RunAs annotation can be used to configure the 'run as' identity or the propagated identity of an entity.

13.10 Securing Application Clients

The security requirements of application clients are similar to that of EJB components. Therefore, the mechanisms used for EJB components can be used for application clients as well. The application client makes use of the authentication service provided by the underlying platform to authenticate its users. The application client authenticates the users accessing it either when the application client starts or when the user is trying to access a protected resource in the application.

The application client can use a LoginModule object to gather the user information. The CallBackHandler instances can further carry out the authentication process.

Check Your Progress

1. Which of the following security mechanism characteristics require application log?

(A)	Authentication	(C)	Auditing
(B)	Authorization	(D)	Confidentiality

2. Which of the following APIs can be used to implement Transport Layer Security?

(A)	JAAS	(C)	Java GSS
(B)	JCE	(D)	JSSE

3. Which of the following is an interface in JAAS?

(A)	LoginContext	(C)	Subject
(B)	LoginModule	(D)	Credentials

4. Which of the following annotations are used to define the roles which can access the current method?

(A)	DenyAll	(C)	DeclareRoles
(B)	PermitAll	(D)	RolesAllowed

5. Which of the following is a method of Subject class?

(A)	doAs()	(C)	Both a and b
(B)	checkPermission()	(D)	authenticate()

Answer

1.	D
2.	C
3.	B
4.	D
5.	A

Summary

- Security mechanisms can be defined at three levels – application layer level, message layer level, and transport layer level.
- Security mechanisms in applications can be defined declaratively and programmatically.
- Security mechanisms are declaratively defined through annotations and deployment descriptors.
- Programmatically security mechanism is defined using EJBContext interface.
- Application users are logically defined as roles according to the application semantics.
- Users are categorized into logical groups known as user groups.
- Both users and user groups can be assigned to different roles in the application.
- JAAS provides various classes and interfaces for implementation of authentication and authorization process.

ASK to LEARN

Questions
in your
mind?



are here to HELP

Post your queries in **ASK to LEARN** @

www.onlinevarsity.com

Technowise



*Are you a
TECHNO GEEK
looking for updates?*

Login to

www.onlinevarsity.com



Welcome to the Session, **EJB Timer Service**.

This session discusses using timers in enterprise applications. Enterprise applications require scheduling of certain operations such as generating performance reports and so on. EJB provides a timer service which can be used to execute these scheduled operations. This session explains the different types of timers provided by EJB. It also discusses the interfaces provided by EJB to implement timers in an application.

In this Session, you will learn to:

- Describe how to use a timer in an enterprise application
- Explain the different types of timers in EJB
- Describe how to handle timers in an application
- Describe annotations associated with timers
- Describe operations performed on Timer objects

14.1 Introduction

Enterprise applications use scheduling systems to perform various tasks. These systems can be used for executing tasks such as generating reports, auditing the application performance, and so on. The tasks are scheduled to be executed at regular intervals or at a prescribed time according to the application requirement. Scheduling systems perform certain operations at prescribed times. This type of scheduling systems were earlier defined through the operating system.

Java EE provides EJB timer service to enable scheduling of tasks in the application code.

EJB timer service was first introduced in EJB 2.1. The timer service allows the EJB container to register certain EJB methods to be invoked at a scheduled time. The timer service is used to implement long running processes therefore, the timer is expected to survive server crashes and shutdowns.

14.1.1 Timer Service

The EJB Timer Service is a container-managed service. The container allows the developer to create Timers are created and implemented through `Timer` objects. They are created by interacting with `TimerService` of the container and through annotations.

Following are the functions which can be implemented by the developer:

1. **Create a timer callback function** - When a developer creates a `Timer` object, it is created with either a prescribed calendar time or number of seconds in which the timer can expire. When the calendar time is reached or when the timer expires, the operation to be performed is defined in the timer callback function.
2. **Process a timer callback function** – The timer callback function has to be invoked and executed once the timer expires or the scheduled time is reached.
3. **Check and interact with the timer callback notification object** - When an object generates a timer callback notification, the `TimerService` enables interacting with the notification generating object.
4. **Obtain a list of outstanding timer notifications** – When there are multiple timer callback notifications, the `TimerService` enables listing outstanding timer notifications.
5. **Cancelling a timer notification** – The `TimerService` enables the application to cancel the timer notification according to the application requirement.

14.1.2 Timer Notification

Timer notification can be defined at the timer expiry event. The event is handled through respective callback methods.

To designate a method as a timer notification callback method, the developer can use any one of the following methods:

- Annotate the method with `@Timeout` annotation.
- A timer callback notification method can be created by implementing the `javax.ejb.TimedObject` interface. The class can invoke `ejbTimeOut` method.
- The method can also be annotated with `@Schedule` annotation.

14.1.3 Creating a Timer Callback Notification

Java EE provides various classes such as `TimerTask`, `TimerHandler`, `TimerService`, `TimedObject`, and so on to handle the timers in the application.

Code Snippet 1 shows an illustration of creating and using a Timer callback notification.

Code Snippet 1:

```
import java.util.Timer;  
import java.util.TimerTask;  
  
public class TimerDemo {  
    Timer timer;  
  
    public TimerDemo(int seconds) {  
        timer = new Timer();  
        timer.schedule(new Reminder(), seconds * 1000);  
    }  
  
    class Reminder extends TimerTask {  
        public void run() {  
            System.out.println("You have a meeting!");  
            System.exit(0);  
        }  
    }  
  
    public static void main(String args[]) {  
        System.out.println("About to schedule a reminder.");  
    }  
}
```

```
TimerDemo T = new TimerDemo(10);  
System.out.println("Reminder scheduled after 10 seconds");  
}  
}
```

In Code Snippet 1, a `Timer` object is created which expires in 10 seconds. The program sends a reminder after the timer expires. The `Timer` object is instantiated in the parameterized constructor. The timer notification callback method that is an internal class `Reminder` is invoked when the timer expires.

Figure 14.1 shows the output of the program.

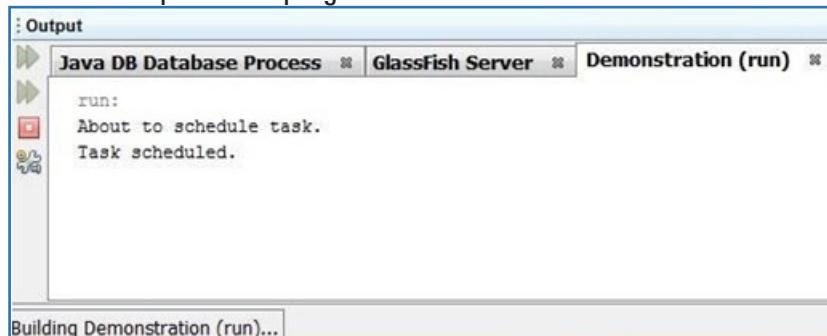


Figure 14.1: Timer Begins on Execution

Figure 14.2 shows the execution after the timer of 10 seconds elapses and the timer notification callback method is invoked.

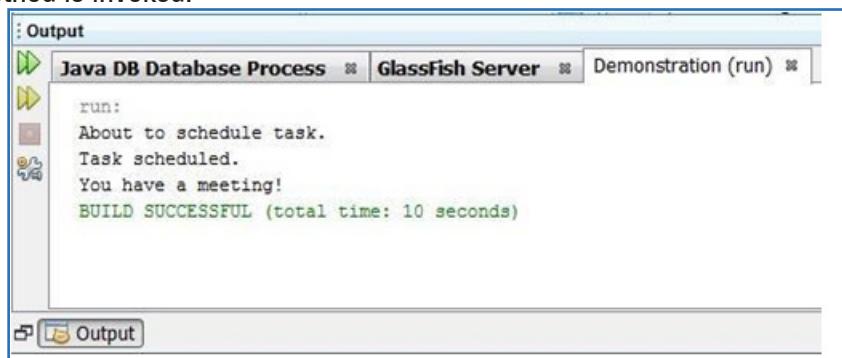


Figure 14.2: Execution after the Timer Elapses

Timers can be created by Stateless Session beans, Singleton Session beans, and Message-driven beans. Stateful sessions cannot use the scheduling and timer service.

14.2 Types of Timers

There are three variants of generating notifications in the applications. These are as follows:

- Non-interval notification timer
- Interval notification timer
- Calendar-based scheduled notification timer

14.2.1 Non-interval Notification Timers

Non-interval notification timers create a notification based on the absolute time or delay interval. These notifications are generated based on a `Date` object or delay interval specified in terms of seconds.

A non-interval notification timer occurs only once, it does not repeat itself.

There are two variants of non-interval notification timers:

- Absolute Time Single Event Timer
- Relative Time Single Event Timer

Absolute Time Single Event Timer invokes a timer callback method only once and at a specific time in the future. Following is the prototype of the `createTimer()` method used for creating an absolute time single event timer:

Syntax:

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

The method returns a `Timer` object, which expires at the date specified as the `Date` parameter. The `info` parameter represents the application information to be delivered along with the notification.

Relative Time Single Event Timer is used to define a timer callback method which is invoked after a specific time in the future. Following is the prototype of the `createTimer()` method used for creating a timer which is invoked relative to a specific time:

Syntax:

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

The `duration` parameter is a milliseconds unit. Therefore, an appropriate long value has to be specified as the parameter.

14.2.2 Interval Notification Timers

Interval notification timers generate recurring notifications which occur at regular intervals based on the parameters passed to the `Timer` object. These timers are used when certain task in the application has to be scheduled at regular intervals of time.

There are two variants of the interval notification timers - Interval timer with initial absolute timer and Interval timer with initial relative timer.

□ Interval timer with initial absolute timer

It has the first timer notification specified by an absolute value through a Date parameter and the duration of consecutive intervals are specified.

Following is the prototype of the `createTimer()` method used to create an interval timer with initial absolute timer.

Syntax:

```
createTimer (Date initialExpiration, long interval, Serializable info)
```

□ Interval timer with initial relative timer

It has the first timer notification specified by a relative value, which is relative to the time when the timer is scheduled. The consecutive timer notifications are specified through the long value.

Following is the prototype of the `createTimer()` method used to create an interval timer with initial relative timer.

Syntax:

```
createTimer(long initialDuration, long interval, Serializable info)
```

14.2.3 Calendar Based Notification Timers

Calendar-based notification timers generate notifications based on a schedule created on the basis of absolute calendar dates. Developers can define the schedule through calendar based expressions.

These calendar-based expressions can be used along with annotations such as `@Schedule` and classes such as `ScheduleClass` to define the schedule of timer notifications in the application.

Here is an example of calendar based scheduling expression:

```
year = "2008,2012,2016" dayOfWeek = "Sat,Sun" hour = "10" minute = "0-10,30,40"
```

This scheduling expression specifies to the `TimerService` that it must send out a timer notification on every Saturday and Sunday at 10:10, 10:30, and 10:40 in the years 2008, 2012, and 2016 respectively. The values `year`, `dayOfWeek`, and `minute` are the attributes of the calendar expression.

There are seven attributes which can be used in the calendar expressions. Table 14.1 shows the attributes which can be used in calendar expressions.

Attribute	Default Value	Possible Values
second	0	[0,59]
minute	0	[0,59]
hour	0	[0,23]
dayofWeek	*	[0-7] or { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" }. Both 0 and 7 refer to Sunday
month		[1-12], {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"}
dayofMonth	*	[1-31] or {"1st", "2nd", "3rd", ..., "Last"} which implies the last day of the month
year	*	Four digit value
timezone		Timezone according to tz database

Table 14.1: Calendar Expression Attributes

The calendar expressions can accept data in different forms:

- As a single value, the calendar expression can be specified as:

year = "2015" month = "May"

- A wildcard expression can be used to specify a set of values in the calendar expression. Here is an instance of using a wild card:

month = "*"

This expression implies any of the possible values of the attribute.

- List is a comma separated set of values which can be assigned to the attributes of the application. It can be used as follows:

month = "May, Apr, Dec, Nov"

The list specifies that the month attribute can have any one of the four values given in the list.

- Range expression specifies a set of continuous values between two values. A range expression can be specified as follows:

dayofMonth = [10-20]

- Increments expression is used in specifying interval timers where the starting point of the timer and their intervals are specified by separating it with a slash as follows:

second = "*/15"

This expression implies that the timer may be initiated at any instance, after initiating the timer it has to be incremented by 15 seconds.

Consider expressing the fact that the timer should send a notification on every Thursday at 10:00 am. The calendar based expression is given as follows:

```
second = "0", minute = "0", hour = "10", dayofMonth = "*",
month = "*", dayofWeek = "Thu", year = "*"
```

14.3 Creating a Timer Object

In order to create a Timer object in an enterprise bean, the enterprise bean has to create an object reference of TimerService through getTimerService() method.

```
private SessionContext SC ;  
.....  
TimerService TS = SC.getTimerService();  
timerService.createTimer(1000*60, text);  
....
```

Figure 14.3 illustrates how a Timer object is created using the TimerService.

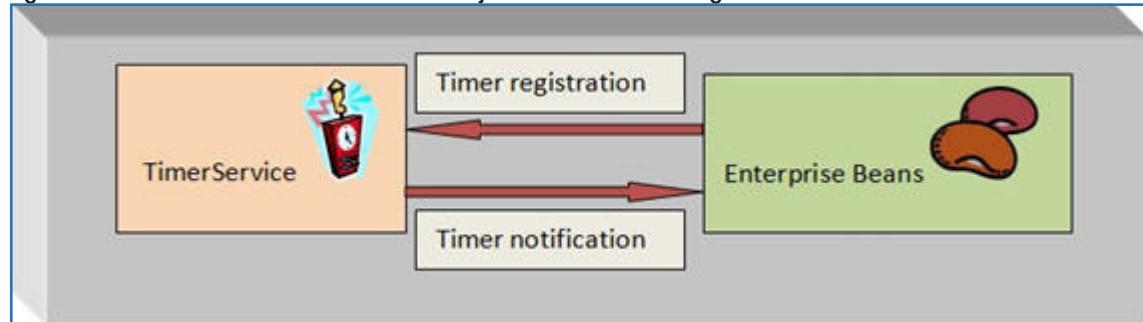


Figure 14.3: Using TimerService to create Timer Objects

The `createTimer()` method creates the required Timer object.

The Timer objects can also be created declaratively through `@Schedule` annotation. Here is an example of creating Timer objects declaratively.

```
@Schedule(minute = "/30", hour = "*")  
public void generateReport()  
{  
    ...  
}
```

14.4 Processing a Timer Callback Notification

The timer callback notification can be processed in anyone of the following ways:

- The timer callback notification can be implemented through an enterprise bean which implements `TimedObject` interface.
- A method prefixed `@Timeout` annotation can be used to process the timer callback notification. The `Timer` object is passed as an argument to such a method.
- A method annotated with `@Schedule` can also be used to process the timer callback notification.

14.4.1 Using the TimedObject Interface

The `TimedObject` interface has only one method `ejbTimeOut()`. This method accepts the `Timer` object which it has to handle as a parameter. The prototype of the `ejbTimeOut()` method is as follows:

Syntax:

```
public void ejbTimeOut(Timer timer);
```

14.4.2 Using @Timeout Annotation

A timeout callback method in the bean class is annotated with `@Timeout` annotation. The method annotated with `@Timeout` annotation that receives timer notifications.

14.4.3 Using @Schedule Annotation

This annotation allows the container to create timers. It is responsible for automatic timer creation according to the schedule specified as attribute to the annotation. The method annotated with `@Schedule` annotation receives notifications from the `TimerService`.

14.5 Guidelines for Coding Timer Handler Method

While creating the timer handler method, the developer should consider the following factors:

- Identifying the timer which has sent a notification.
- Handling application shutdowns.
- Transaction handling for timer notification callback methods.
- Propagating the security context to the timer callback notification.
- Handling non-persistent timers.

When there are multiple timers defined on the bean class, there should be a mechanism in place to identify the timer which has expired. The `info` parameter of the `createTimer()` method can be used for the purpose. The timer handler method can identify the method with the help of the `info` parameter received along with the timer notification.

A timer has to survive application shutdown. The `TimerService` holds the timer notifications of the timers that expired when the application was shutdown. These timers are delivered when the application restarts.

If the `ejbTimeout()` method has to execute in a transaction context, then the callback method should have code which can handle transaction rollback. In case of transaction rollback the `ejbTimeOut()` method is invoked again.

The timeout notification callback methods do not have the security context of the application client. The bean provider uses the run-as deployment descriptor to specify the security identity of the client and execute the timeout callback methods.

Persistent timers are functional even on application shut down or server crash. Applications can also create non-persistent timers, these timers are not active when the application shuts down or when the application crashes. Non persistent timers can be created either programmatically or declaratively.

When created programmatically, non-persistent timers can be specified through `setPersistent()` method of the `TimerConfig` class. Non-persistent timers can be created declaratively through `@Schedule` annotation by choosing appropriate attributes.

14.6 Managing Timer Objects

The `Timer` interface provides various methods to manage timer objects in the applications. Developers may have to perform the following tasks while managing timers:

- Interrogate a timer callback notification
- Obtain a list of outstanding timers
- Cancel timer notification

14.6.1 Interrogating a Timer Callback Notification

The application may require information such as what is the time left for the next timer notification, `info` parameter that the bean class would receive when the timer expires, and so on. The `Timer` class provides various methods for performing these operations.

Following are the methods provided by the `Timer` class for interrogating the `Timer` object:

- `getHandle()` – This method returns a `Serializable` handle to the `Timer` object. This handle can be used to reference the `Timer` object later in the application.
- `getInfo()` – It returns the `info` associated with the `Timer` object. The `info` field is defined when the `Timer` object is created and can contain any type of data which is `Serializable`.
- `getNextTimeout()` – This method returns the object of `Date` type which indicates the date when the next timer would expire.

- `getSchedule()` - This method returns an object of type `ScheduleExpression`, which signifies the schedule of the timer.
- `getTimeRemaining()` – This method returns the number of milliseconds remaining for the time to elapse before the bean receives a timer notification.

14.6.2 Obtaining a List of Outstanding Timers

In order to obtain the list of outstanding timers, bean class can execute the `getTimers()` method of the `TimerService` class. The prototype of the `getTimers()` method is:

```
Collection<Timer> getTimers();
```

This method returns a collection of timers associated with the bean class.

To obtain a reference to the `TimerService` object, the enterprise bean can invoke a `getTimerService()` method on the associated `EJBContext`.

14.6.3 Cancelling the Timer

Developers can use `cancel()` method on the `Timer` objects to cancel the timers. Invoking this method removes the timer from the container. If there are no timers existing then the bean would encounter a `NoSuchObjectException`.

Check Your Progress

1. Which of the following classes enable creating and cancelling timers in applications?

(A)	Timer	(C)	TimedObject
(B)	TimerService	(D)	TimerHandler

2. Which of the following beans cannot use TimerService?

(A)	Stateless session beans	(C)	Message-driven beans
(B)	Singleton session beans	(D)	Stateful session beans

3. Which category of timers are recurrent timers?

(A)	Non interval notification timers	(C)	Interval notification timers
(B)	Calendar based notification timers	(D)	All of these

4. Which of the following methods is not a timer interrogating method?

(A)	getTimers ()	(C)	getNextTimeout ()
(B)	getSchedule ()	(D)	getInfo ()

5. Which of the following methods return a Collection object?

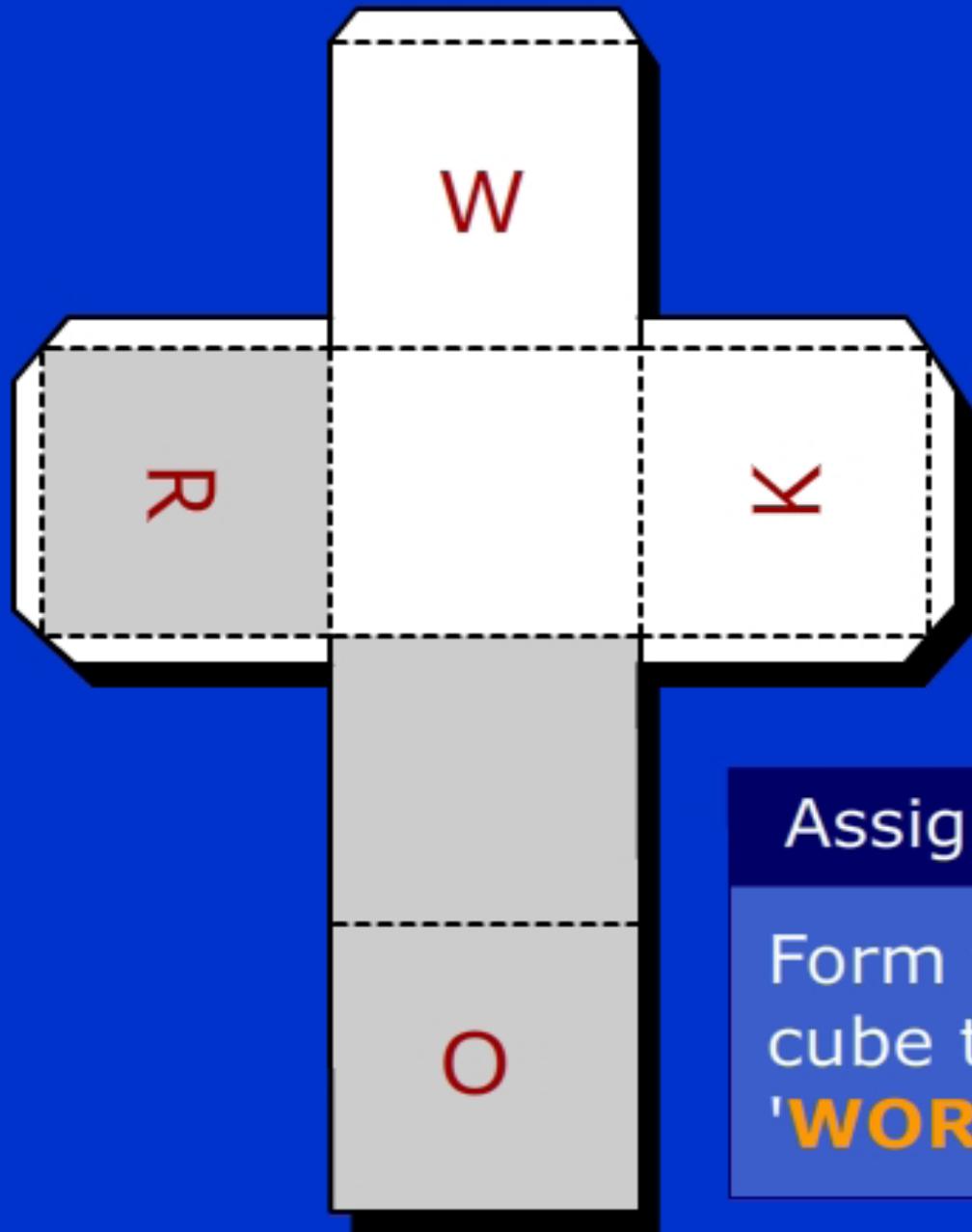
(A)	getNextTimeout ()	(C)	getTimers ()
(B)	getInfo ()	(D)	None of these

Answer

1.	B
2.	D
3.	C
4.	A
5.	C

Summary

- Enterprise applications use Timer objects to schedule tasks in the application.
- Java EE provides TimerService class to handle, create, and manage timers in the application.
- Timers can be created declaratively through @Timeout and @Schedule annotations.
- Timers are created and managed programmatically using TimedService, Timer, and other classes available in Java EE.
- Timers can be defined as absolute timers, recurrent timers, and calendar based timers.



Assignment

Form the
cube to read
'WORK'.

"Practice does not make perfect. Only perfect practice makes perfect."
- Vince Lombardi

For perfection, solve assignments @

www.onlinevarsity.com



Big

Balanced Learner-Oriented Guide

for enriched learning available



www.onlinevarsity.com



Welcome to the Session, **EJB Design Patterns**.

This session discusses how design patterns are used in enterprise application design. The session explains Gang-Of-Four (GOF) and lists the different categories of design patterns provided in GOF. Then, the session explains Java EE design patterns and also explains how they are used in designing enterprise solution. Finally, it lists the best practices followed when using EJB as business components in the enterprise applications.

In this Session, you will learn to:

- Describe design patterns
- Explain different types of design patterns
- Explain mostly commonly used Java EE design patterns
- Describe best practices in selecting EJB component in the enterprise applications

15.1 Introduction

In day to day life, whenever a problem is encountered, most people tend to refer to a similar problem faced in the past. The problem had been successfully solved using some technique. That technique can be used as a model and can be tried out on the new problem by making relevant changes as per the current circumstances. Similarly, in the world of programming, a previously used solution to a problem can be applied to a new problem by making relevant changes. This solution which has been used effectively in an earlier problem is described as a pattern.

15.1.1 Design Patterns

A pattern in real world is a shape, model, or structure appearing again and again to create a design. Take an example of the printed tiles used for flooring. The tiles are printed in such a way that they need to be arranged correctly so that the print falls in the right place and a pattern is formed. Once a pattern is formed out of the combination of first set of tiles, the rest of the flooring is completed by copying the same pattern all over. The first set of tiles became a pattern or a solution for the rest of the floor. This is shown in figure 15.1.

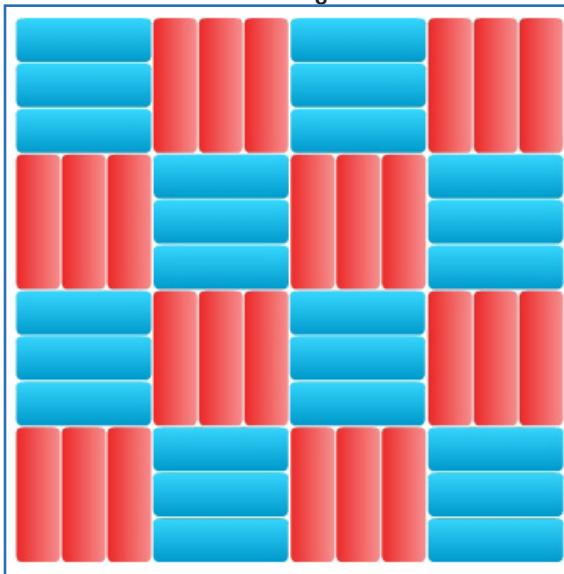


Figure 15.1: Tiles Arranged in a Pattern

Similarly, design patterns in software are strategies which are language-independent and used for solving commonly encountered object-oriented design problems. Design patterns are proven techniques for implementing robust, reusable, and extensible object-oriented software. In more technical terms, a pattern is a proven solution to a problem which has been documented so that it can be used to easily handle similar problems in the future.

Patterns have become an important concept in the development of object-oriented programming. They help in describing structures and relationships at a higher and different level of abstraction than classes, components, or instances. Documenting patterns serves two important purposes:

- It speeds up the process of finding a solution when another problem with similar characteristics is encountered. That is, it eliminates the need to 'reinvent the wheel'.

- Giving names to patterns allows programmers to use a common vocabulary to discuss design alternatives. This vocabulary is also known as pattern language.

Design patterns reduce the time required to find the solution to a problem as they suggest tried and tested solutions. As design patterns provide reliable solutions to the problem, they considerably reduce the resources consumed while developing large enterprise applications.

The concept of design patterns was proposed by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. They are popularly known as the '**Gang-of-Four**'. They describe design patterns as '**a solution to a problem in a context**'. There are three important aspects of a design pattern – problem, solution, and context.

15.2 Types of Design Patterns

In many different object-oriented applications, certain types of relationships appear over and over again. A pattern attempts to filter the important features of these associations. Based on the problems encountered, different patterns have been created.

There are three primary categories of design patterns defined by GOF. They are as follows:

- **Creational patterns** – Define the object creation process in an application.
- **Structural patterns** – Define the organization of classes and objects in an application.
- **Behavioral patterns** – Define the interaction between various objects in an application.

15.2.1 Creational Patterns

Creational patterns are useful in scenarios where the object composition is crucial for the application. They enable creating objects, deciding the properties, and defining the methods of the objects based on the situation. They provide flexibility in terms of deciding what properties of the class are used in the application and what should be the behavior of the instantiated object.

Creational patterns allow the developer to configure a system with objects which vary widely in structure and functionality.

Table 15.1 shows the creational patterns and their description.

Pattern	Description
Singleton	Singleton pattern restricts the number of instances of a class to only one. It is used to implement tasks such as application logging and so on.
Factory	Factory pattern is used to create multiple instances of a class. This is used in a scenario where there is a super class with multiple subclasses. Based on the user input an instance of appropriate sub class is created.
Abstract Factory	Abstract Factory pattern is similar to factory pattern. It is used to provide an interface for creating instances of sub classes in an inheritance structure without defining concrete classes.

Pattern	Description
Builder	Builder pattern is used when the instances of the class have large number of optional parameters.
Prototype pattern	Prototype pattern is used when creating an instance of the class is expensive and resource intensive.

Table 15.1: Creational Patterns

15.2.2 Structural Patterns

Structural patterns are meant for enabling multiple classes and objects to function together. These patterns use mechanisms such as inheritance to allow various classes, interfaces, and their implementations to work together. They define how objects and classes in the application combine together to form a large structure.

This pattern is essential for allowing independently developed classes and interfaces to work together. The focus of this pattern is how the classes are inherited from each other and how they are aggregated from other classes. It is used to realise the relationships among the classes of the application through object-oriented mechanisms such as abstraction and inheritance.

Table 15.2 lists the structural patterns and their description.

Pattern	Description
Adapter	Adapter pattern is used to bridge the structural gap between two interfaces.
Composite	Composite patterns are used when composite objects should be treated like primitive objects. For instance a geometrical shape can be composed of multiple primitive shapes such as triangle, square, and so on. The composite object should be treated in a way similar to the primitive object.
Proxy	Proxy pattern is used when the application wants to provide a controlled access to certain functionality of the application. This pattern allows an object to provide a place holder for another object so that it can access it in a controlled manner.
Flyweight	Flyweight pattern is used when the application has to create large number of objects of a particular class. This is applied when the application runs on low memory devices as it allows sharing of memory.
Facade	Facade pattern is used for defining the interaction of client systems with the application.
Bridge	Bridge pattern separates the interfaces from implementations. It is a structural pattern which aims at hiding the implementation details from the clients.

Pattern	Description
Decorator	Decorator design pattern enables modifying the functionality of an object at runtime. It also isolates other instances of the same class from being modified.

Table 15.2: Structural Patterns with their Descriptions

15.2.3 Behavioral Patterns

Behavioral patterns are those that determine the interaction of objects. These patterns are used when the behavior is complex. They simplify the complex behavior by specifying the responsibilities of the objects and their communication method.

Most common behavioral pattern is observer pattern, which is used when an application is being implemented through a Model-View-Controller architecture. This separates the presentation of data from the actual data store.

Java API provides `Observable` class which can be extended by objects that need to be observed. The objects that need to be observed can be components such as speedometer in an application where if the speed exceeds a certain limit, the driver should be warned. The `Observable` class has various methods that notice the changes in the object being observed.

Table 15.3 lists the behavioral patterns and their description.

Pattern	Description
Template method pattern	Template method pattern is a behavioral design pattern which defers the implementation of behavior of objects to sub classes in the hierarchy.
Mediator pattern	Mediator pattern is used to establish a communication mechanism among different objects in the application. It implements loose coupling among the objects in the application.
Chain of responsibility pattern	Chain of responsibility pattern defines the method of handling the client request in the application. The client request is passed through a chain of objects while processing it.
Observer pattern	Observer pattern is used to track the state of an object in the application. The object which keeps track of the state is known as an Observer.
Strategy pattern	Strategy pattern enables the application to have multiple algorithms which can be used on certain event. The algorithm to be used is decided at runtime.
Command pattern	Command pattern is a behavioral pattern which implements loose coupling of the components. The application components execute in a request response model.

Pattern	Description
State pattern	State pattern is used when the object changes its behavior based on its internal state.
Visitor pattern	Visitor pattern enables separating the operational logic to a separate class. It is used when an operation has to be performed on a group of similar objects.
Interpreter pattern	Interpreter pattern is used when the application has to analyze the syntax and semantics of input.
Iterator pattern	Iterator pattern is used when the application has to provide a standard way to traverse through a group of objects.
Memento pattern	Memento design pattern is used when the application has to save the state of the object and restore it for later use.

Table 15.3: Behavioral Patterns with their Descriptions

15.3 Java EE Design Patterns

There are number of patterns identified by Sun Java center for simplifying the development of enterprise applications.

Some of the useful patterns commonly used in Java EE application development are as follows:

- Session Façade
- Singleton Instance
- Value Object/Data Transfer Object
- Message Facade

15.3.1 Session Façade

Enterprise applications encapsulate business logic and business data in the enterprise beans and expose their interface to the clients. A multitiered application contains several server-side enterprise beans that are collectively referred to as business objects.

Clients interact with the business objects through their exposed interfaces. However, the client has direct interaction with the business objects which leads to a tight coupling between them. This also results in dependence on the implementation of the business objects.

Secondly, when invoking enterprise beans, every client invocation to bean methods is a remote method call. Thus, as the number of clients increases, the number of remote method calls also increases, resulting in network performance reduction of the application.

Thirdly, there is no uniform access strategy adopted by the clients to access business objects, as they are directly exposed to the clients.

All these problems, lead to the development of EJB pattern named session façade. The session façade pattern abstracts the interaction of business objects by providing a service layer. The service layer exposes only the required interfaces to the client. In other words, it defines a higher-level business components which in turn comprises the interactions among lower level components.

The Session Facade manages the interactions between the business data and business service objects that participate in the workflow of the application. This is achieved by using the session bean in the design pattern. The session bean provides a uniform interface to all the application clients by handling the relationship between the business objects. Session façade enables decoupling of all the lower-level application components.

The session bean manages the lifecycle of the business objects by creating, locating, modifying, and deleting them as per the requirements of the application.

Figure 15.2 shows the class diagram of the session façade pattern.

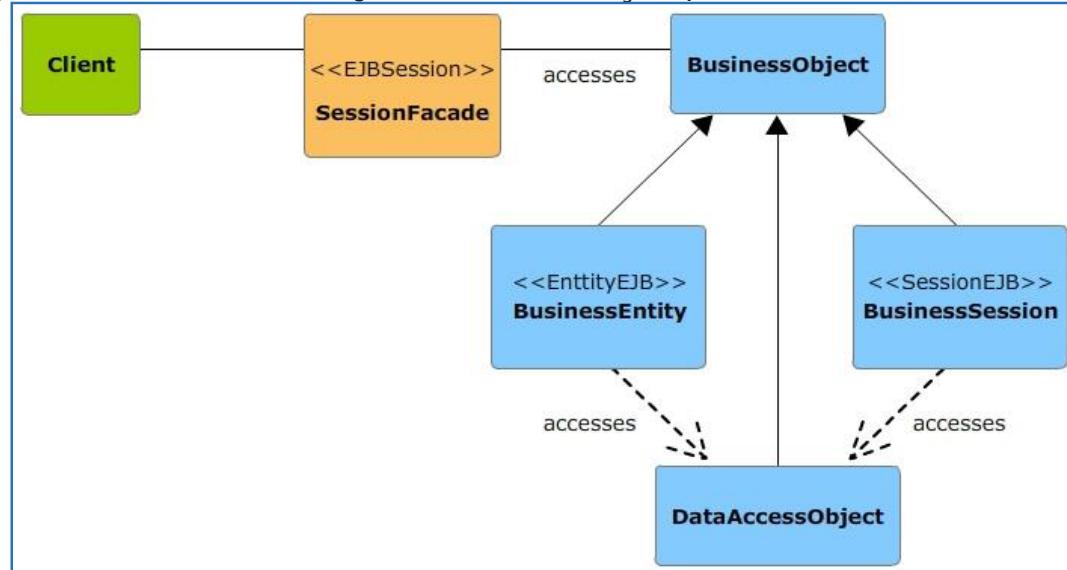


Figure 15.2: Class Diagram of Session Façade Pattern

As shown in figure 15.2, the client of the session façade pattern access the business services through session bean. The session façade manages the relationships between various business objects and provides a higher-level abstraction to the client. The business objects can be other session beans, entity beans, Data Access Objects (DAO) that provide data or business services to the application.

In complex applications, there can be various session façade service objects that can sit between clients and business objects to provide interactions between them.

Consider a scenario of bank application; the application has to implement the following tasks:

- Check balance
- Transfer money
- Withdraw money

The session façade pattern provides a common interface to implement all these functions in the application. Java application can define an enterprise bean such as Bank Teller bean which will implement these functionalities.

Similarly for all the services pertaining to the Loan services offered by the Bank application, an enterprise bean can be created with the following methods:

- Apply for loan
- Track status
- Calculate loan eligibility

Figure 15.3 shows the implementation of Session façade in enterprise applications.

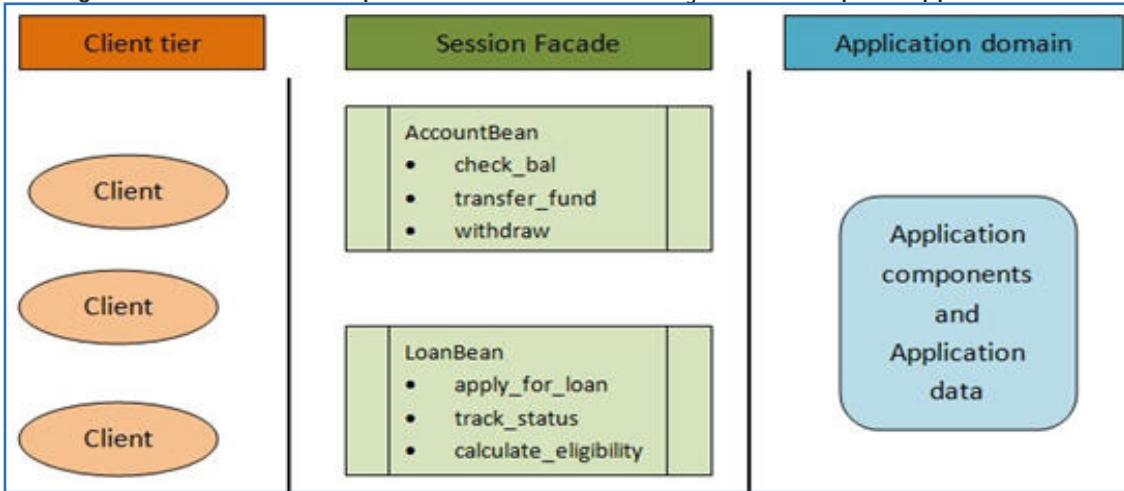


Figure 15.3: Implementation of Session Façade Pattern

Code Snippet 1 shows an entity class Message.

Code Snippet 1:

```
@Entity
public class Message implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String message;
    // Getter and Setter methods
    ...
    @Override
    public int hashCode() {
        int hash = 0;
        hash += (id != null ? id.hashCode() : 0);
        return hash;
    }
}
```

```
@Override  
public boolean equals(Object object) {  
    if (!(object instanceof Message)) {  
        return false;  
    }  
  
    Message other = (Message) object;  
  
    if ((this.id == null && other.id != null) || (this.id != null && !this.  
        id.equals(other.id))) {  
        return false;  
    }  
  
    return true;  
}  
  
@Override  
public String toString() {  
    return "entities.Message[ id=" + id + "]";  
}  
}
```

Code Snippet 1 creates the attributes 'id' and 'message' respectively. The getter and setter methods are developed for the attributes. The id attribute is used as the primary key for accessing the entity in the database. There are other methods such as equals(),toString(), and hashCode() which are generated by API to manage the entity created by the entity class.

After creating the entity class, define the access to this entity through a session bean by creating a session bean for the entity class. As per the EJB 3.1 specification, business interfaces for session beans are optional. In this example, the client accessing the bean will be a local client and hence, you can use a local interface or a no-interface view to expose the bean.

The steps to create a session facade (session bean) for the Message entity are as follows:

1. To create a session bean, right-click the project and select **New → Other → Enterprise JavaBeans → Session Beans For Entity Classes** as shown in figure 15.4.

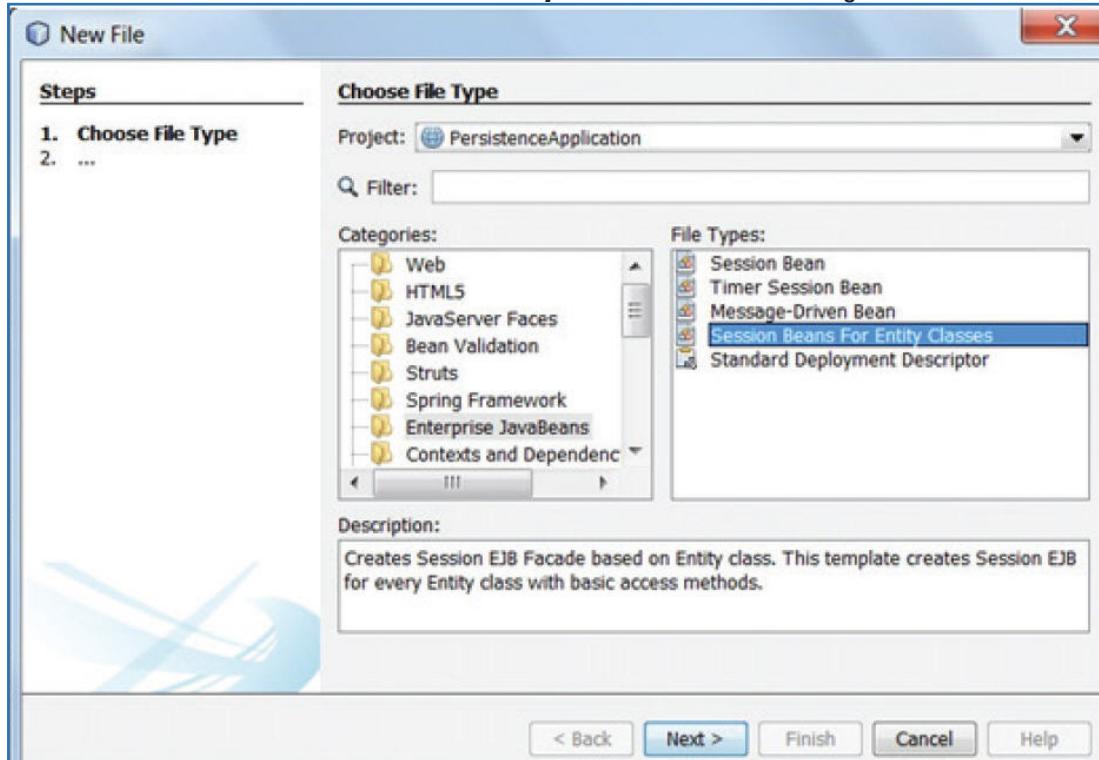


Figure 15.4: NetBeans IDE - Session Facade

2. Click **Next**. Select **entities.Message** from the **Available Entity Classes** and click **Add**. The Message entity will be added to the **Selected Entity Classes**.
3. Click **Next**. Create the session bean in a package '**façade**', this package name is based on the developer's choice.
4. Click **Finish**. This creates a session bean which will be exposed to a local managed bean using a no-interface view.

The IDE generates the session facade class named **MessageFacade.java** and **AbstractFacade.java**.

Code Snippet 2 shows the shows the code in AbstractFacade.java, this code is generated by the IDE.

Code Snippet 2:

```
...
public abstract class AbstractFacade<T> {
    private Class<T> entityClass; @GeneratedValue(strategy=GenerationType.AUTO)
    public AbstractFacade(Class<T> entityClass) {
        this.entityClass = entityClass;
    }
    protected abstract EntityManager getEntityManager();
    public void create(T entity) {
        getEntityManager().persist(entity);
    }
    public void edit(T entity) {
        getEntityManager().merge(entity);
    }
    public void remove(T entity) {
        getEntityManager().remove(getEntityManager().merge(entity));
    }
    public T find(Object id) {
        return getEntityManager().find(entityClass, id);
    }
    public List<T> findAll() {
        javax.persistence.criteria.CriteriaQuery cq = getEntityManager().getCriteriaBuilder().createQuery();
        cq.select(cq.from(entityClass));
        return getEntityManager().createQuery(cq).getResultList();
    }
}
```

```
public List<T> findRange(int[] range) {  
    javax.persistence.criteria.CriteriaQuery cq = getEntityManager().  
        getCriteriaBuilder().createQuery();  
    cq.select(cq.from(entityClass));  
    javax.persistence.Query q = getEntityManager().createQuery(cq);  
    q.setMaxResults(range[1] - range[0] + 1);  
    q.setFirstResult(range[0]);  
    return q.getResultList();  
}  
  
public int count() {  
    javax.persistence.criteria.CriteriaQuery cq = getEntityManager().  
        getCriteriaBuilder().createQuery();  
    javax.persistence.criteria.Root<T> rt = cq.from(entityClass);  
    cq.select(getEntityManager().getCriteriaBuilder().count(rt));  
    javax.persistence.Query q = getEntityManager().createQuery(cq);  
    return ((Long) q.getSingleResult()).intValue();  
}
```

Code Snippet 2 shows the code generated by the IDE. It has various methods which will connect to the database and retrieve data from it. EntityManager is responsible for carrying out these tasks. The methods `create()`, `remove()`, and `find()` are used to create an entity in the database, remove an entity from the database, and find an entity in the database respectively.

The `findAll()` method retrieves all the entities in the database and `count()` method is used to count the number of entities in the database. The `findRange()` method finds the entities within a range of values of the primary key, where the range is provided as an argument to the method.

The `MessageFacade` class extends the `AbstractFacade` class and uses `EntityManager` to manage the entities in the database.

Code Snippet 3 shows the code of MessageFacade class.

Code Snippet 3:

```
...
@Stateless
public class MessageFacade extends AbstractFacade<Message> {
    @PersistenceContext(unitName = "PersistenceApplicationPU")
    private EntityManager em;
    @Override
    protected EntityManager getEntityManager() {
        return em;
    }
    public MessageFacade() {
        super(Message.class);
    }
}
```

In Code Snippet 3, the code injects the resource of the Persistence unit through an annotation and then invokes an entity manager for this context.

The client applications can invoke the method `getEntityManager()` that returns an object of type `EntityManager` and access the entity to work with data persistence layer.

15.3.2 Singleton Design Pattern

A Singleton design pattern ensures that there is only one instance of a class in the application. A singleton pattern is used to implement tasks such as application logging and so on.

Figure 15.5 shows the singleton session bean instance.

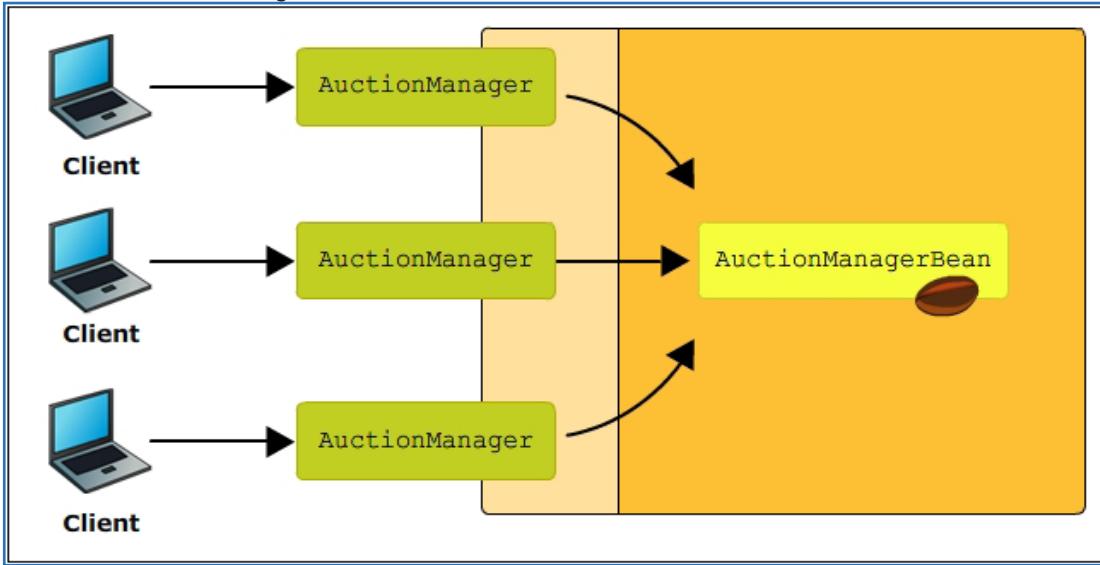


Figure 15.5: Singleton Session Bean

A Singleton pattern can be implemented in Java applications as follows:

- The constructor in the Java class can be declared private, this will restrict the instantiation of a class from other classes in the application.
- The singleton instance can also be implemented as a static instance of the class.

A singleton instance can be created through any one of the following strategies:

- Eager initialization** – The singleton instance is created while loading the class.
- Static block initialization** – The singleton instance is created while loading the class in a static block of code. This initialization also provides for exception handling while creating the singleton instance.
- Lazy initialization** – When the singleton is instantiated through lazy initialization, then the instance is created in the global access method.
- Thread safe singleton** – A thread safe singleton is created by instantiating the singleton in a synchronized block.

Code Snippet 4 shows the instantiation of an object according to singleton pattern.

Code Snippet 4:

```
public class SingleInstance {  
  
    // Declare an instance of the class  
    private static SingleInstance inst=null;  
    // set constructor as private  
    private SingleInstance() {}  
    // Define a synchronized method for creating the instance  
    public static synchronized SingleInstance getInstance() {  
        // check for existence of instance  
        if (inst==null) {  
            // create instance if it does not exist  
            inst=new SingleInstance();  
        }  
        return inst; // return the instance  
    }  
}
```

In Code Snippet 4, the class `SingleInstance` is a singleton class which can instantiate only one object at any instance of time. The constructor is declared as `private`, therefore external classes cannot create an instance of the class, `SingleInstance`. The `getInstance()` method creates a new instance of -`SingleInstance` class only when the instance value is 'null' which implies that there is no other instance of `SingleInstance` class.

It is always important to have the `getInstance()` method synchronized, so that it can remain thread safe. However, when a singleton instance is serialized and deserialized more than once, there is a possibility of having multiple instances of the class which needs to be checked.

A singleton class cannot be further inherited as its constructor is `private`.

15.3.3 Value Object/Transfer Object Pattern

The value object pattern is used while implementing communication among different entities in the application. Some attributes of an entity are always accessed together in the application; such entities are combined together into an individual class. Instances of this class are termed as Data Transfer Objects (DTO). This class is a Serializable class and can be used as a return type to the business methods. Data Transfer Objects are defined to minimize the number of method calls in a distributed system. It simplifies manipulation of request and response data in the application and decouples message structures from domain layer entities.

Figure 15.6 shows the data transfer object pattern.

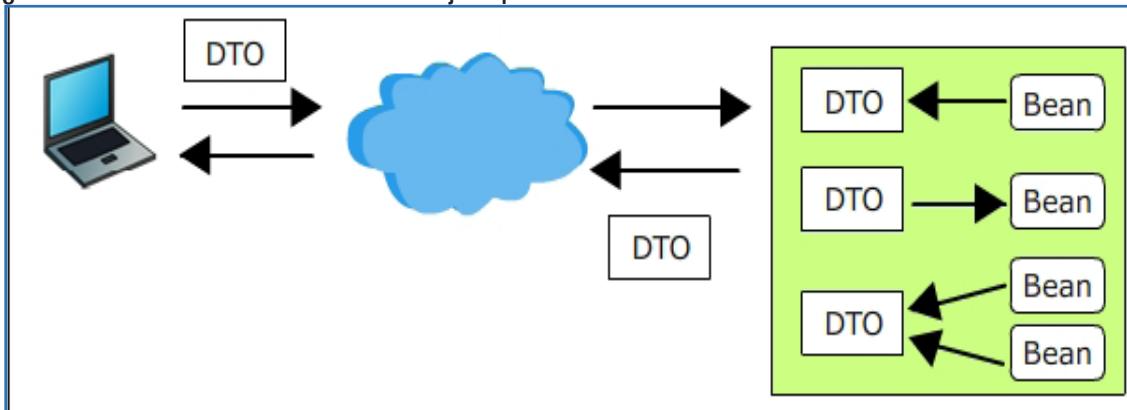


Figure 15.6: Data Transfer Object Pattern

For instance, when a session bean is trying to set the values of an entity Customer, the values are obtained through the application client. In order to set the values of customer name, age, and city the application can make three method calls:

- getName()
- getAge()
- getCity()

Multiple method calls may lead to network latency, thus dropping the performance of the application. The Data Transfer Object pattern provides a solution by replacing these method calls by a single method call. It defines DTO in the application which replaces interfaces as the medium of communication in the application.

15.3.4 Message Façade Pattern

Session Façade pattern defines an interface for communication with external clients. The communication is through method calls. However, the method calls are synchronously invoked. This may lead to performance bottle neck in the applications.

In order to avoid this situation, the Message Façade pattern proposes asynchronous handling of method calls. The Message Façade pattern introduces an additional component such as a JMS server to asynchronously handle the messages among the application components. The client sends a message to the JMS server and receives a response from the JMS server asynchronously.

This mechanism avoids blocking clients as the messages are asynchronously handled by the JMS server.

Figure 15.7 shows the message façade pattern.

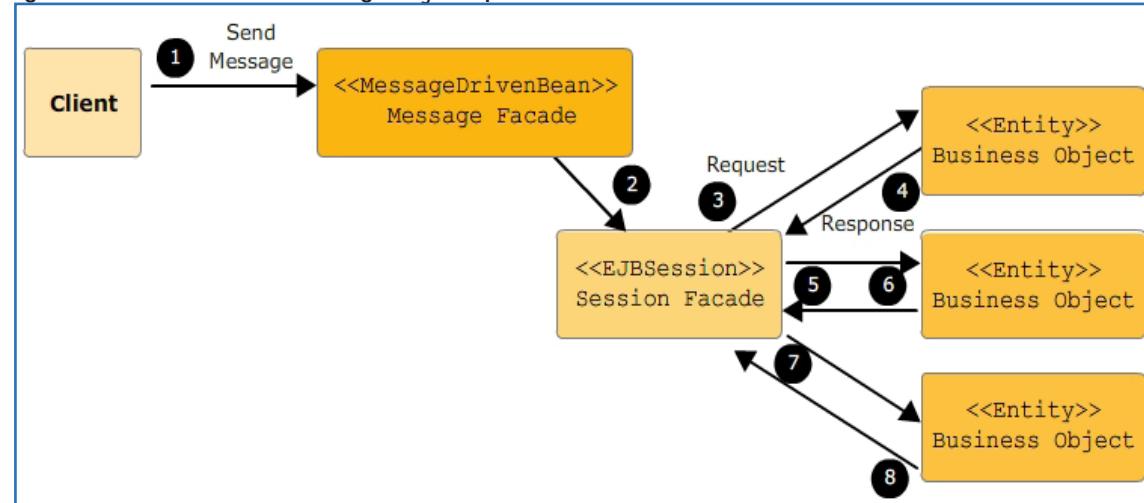


Figure 15.7: Message Façade Pattern

15.4 EJB Best Practices

Enterprise applications developed with EJBs should have the following characteristics:

- Remote access to the application components
- Distributed transactions in the application domain
- Security implementation to different components
- Data persistence
- Application scalability

Developers and designers should know how to use EJB in applications. Following are some of the best practices which should be incorporated while implementing EJB applications:

- Every EJB application should have at least four files home interface, remote interface, EJB implementation and deployment descriptor. It is a good practice to write the application using JSPs and Servlets. These Web components can then be refactored and expanded to include the EJB components in the application.
- Design patterns must be used whenever possible in the applications, as they provide a reliable, tried, and tested solution for the problem.
- The complete design of the application should be available before actually implementing the application.
- Container Managed Persistence should be used whenever possible in the application as the J2EE container is capable of efficiently managing the application persistence.
- An interface must be defined through which an EJB can be accessed. Even if the EJB is accessed locally, a local interface must be defined to access the EJB.
- Appropriate exception handling code must be written for all the exceptions that might occur in the EJB code.
- Lazy loading of database tables should be used for optimized performance of the application.

Check Your Progress

1. Which of the following is not a behavioral pattern?

(A)	Mediator pattern	(C)	Chain of responsibility
(B)	Template method	(D)	Adapter pattern

2. Which of the following design pattern optimizes the communication among the application components?

(A)	Value object	(C)	Adapter
(B)	Message Façade	(D)	Singleton

3. Which of the following patterns are used to track the state of an object in the application?

(A)	Adapter pattern	(C)	Command pattern
(B)	Observer pattern	(D)	Strategy pattern

4. Which of the following is not an element of design pattern template?

(A)	Structure	(C)	Applicability
(B)	Motivation	(D)	None of these

5. Which of the following is a creational pattern?

(A)	Session Façade pattern	(C)	Singleton pattern
(B)	Message Façade pattern	(D)	Value object pattern

6. _____ pattern is used to track the state of the object in the application.

(A)	Mediator pattern	(C)	Observer pattern
(B)	Memento pattern	(D)	All of these

Answer

1.	D
2.	A
3.	B
4.	D
5.	C
6.	C

Summary

- Design patterns in software are strategies which are language-independent and used for solving commonly encountered object-oriented design problems.
- The concept of design patterns was proposed by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. They are popularly known as the 'Gang-of-Four'.
- There are three primary categories of design patterns defined by GOF. They are namely creational, structural, and behavioral patterns.
- Creational patterns enable creating objects, deciding the properties, and defining the methods of the objects based on the situation.
- Structural patterns are meant for enabling multiple classes and objects to function together. Behavioral patterns are those that determine the interaction of objects.
- There are number of patterns identified by Sun Java center for simplifying the development of enterprise applications.
- Some of the Java EE design patterns are namely session facade, Singleton Instance, Value Object/Data Transfer Object, and Message Facade.
- The session façade manages the relationships between various business objects and provides a higher-level abstraction to the client.
- A Singleton design pattern ensures that there is only one instance of a class in the application.
- The value object pattern is used while implementing communication among different entities in the application.
- The Message Façade pattern proposes asynchronous handling of method calls.

**Get
WORD WISE**



**Visit
Glossary@**

www.onlinevarsity.com