

Fundamentals of Java Enterprise Components

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**

Fundamentals of Java Enterprise Components

© 2014 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

Edition 1 - 2014



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

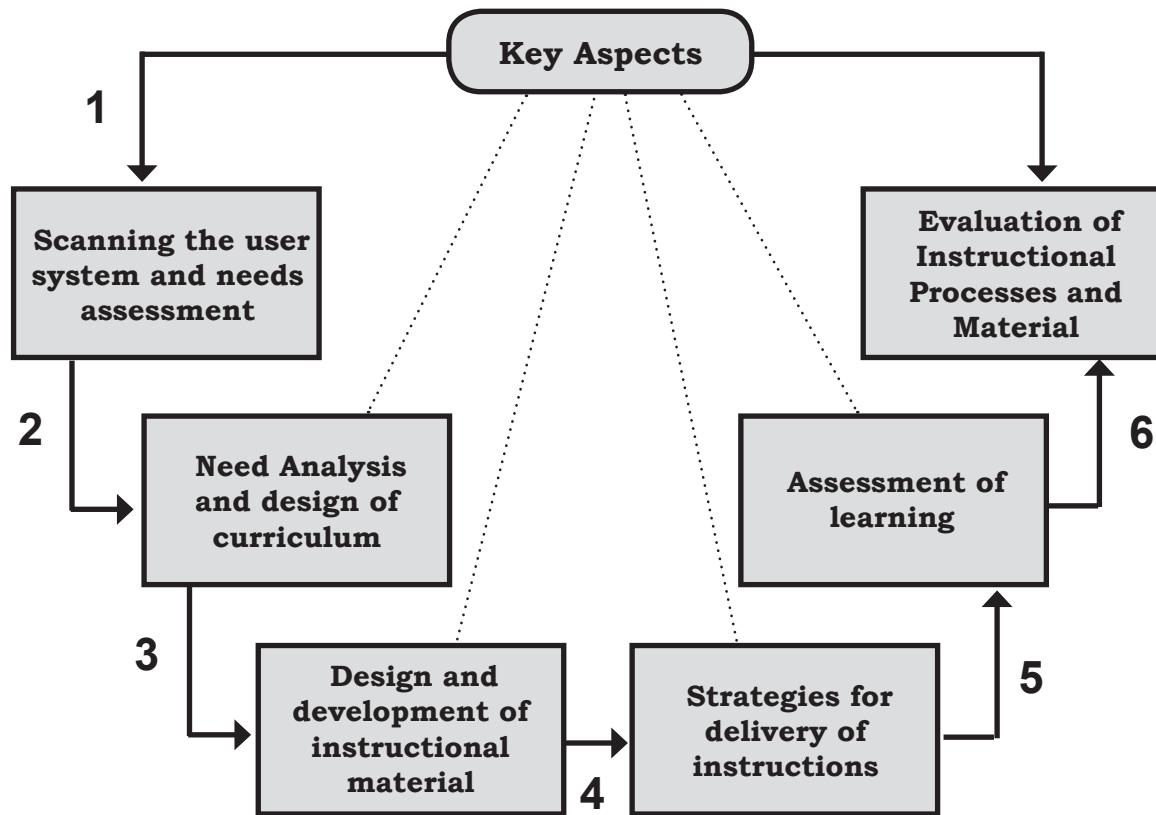
The instructional process is backed by an elaborate monitoring system to evaluate - on-time delivery, understanding of a subject module, ability of the instructor to impart learning. As an integral part of this process, we request you to kindly send us your feedback in the reply pre-paid form appended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

*TAG – Technology & Academics Group comprises of 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

Web and enterprise applications have become very popular today due to their efficiency and distributed nature. They can be used for different types of transactions and online activities. The use of Enterprise applications allows distribution of components at different levels that helps better management and troubleshooting in case of application errors.

This book has been designed to equip you with the knowledge required to develop distributed and efficient Web and enterprise applications. After reading this book, you will be able to identify and create components of Web and enterprise applications. It also introduces Web Services and security features available in Java EE applications.

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.

Design Team



B
I
g

Balanced Learner-Oriented Guide

for enriched learning available

@

www.onlinevarsity.com

Table of Contents

Sessions

1. Introduction to Java EE 7
2. Enterprise Application Architecture
3. Introduction to Web Application Development
4. Application Resources
5. Java Servlets
6. JavaServer Pages
7. Introduction to JavaServer Faces
8. JavaServer Faces as Web Pages
9. Facelets
10. Enterprise JavaBeans
11. Java Persistence API
12. Transactions
13. Java Message Service Components
14. Building Web Services with JAX-WS and JAX-RS
15. Java Security

ASK to LEARN

Questions
in your
mind?



are here to **HELP**

Post your questions in the **ASK to LEARN** section
for solutions.

Session - 1

Introduction to Java EE 7

Welcome to the Session, **Introduction to Java EE 7.**

This session provides an introduction to the Java enterprise model. It explains about multitier applications, application servers, and Web services. Further, it explains about deployment of enterprise applications on Web containers. Lastly, it explains the various APIs used in Java EE7.

In this Session, you will learn to:

- Explain Java Application model
- Explain multitier applications
- Describe various components of multitier applications
- Describe containers and services provided
- Describe Web services
- Describe various APIs used in Java EE 7 for enterprise and Web applications



1.1 Java Enterprise Application Model

The Java enterprise application model defines how various components of an enterprise application are organized. This organization is responsible for creating a portable, secure application with high developer productivity. The portability and security of the application are due to the Java programming language and Java Virtual Machine.

Java EE enables the creation of an enterprise application which caters to the needs of the customers, employees, suppliers, partners, and so on. These applications are very complex, where data is accessed from a variety of sources and has to be distributed among various targets. In order to manage this complex flow of data, a middle tier of the application is created.

Middle tier is run on a different server which has access to every component of the enterprise application. This can also be termed as the business logic of an application where it knows all about the organization of the data, processes of the enterprise (business logic), and client requirements.

Figure 1.1 shows an abstract representation of the Java enterprise application model.

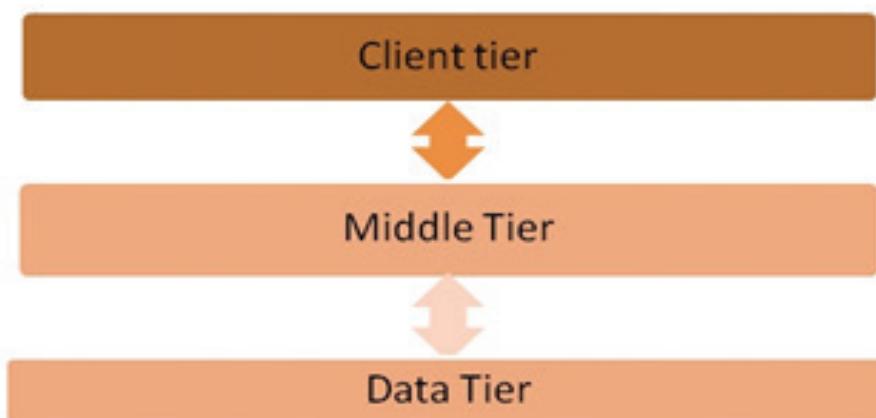


Figure 1.1: Application Model of Java EE

The client tier represents various users of the enterprise application such as customers, employees, suppliers, and so on.

The middle tier handles the enterprise business logic. The enterprise business logic refers to the processes in the enterprise and this layer is aware of enterprise data as well as the client requirements. The middle tier processes the data received from the client with the help of the Data tier. That is, developers implement the business logic in this layer. The processed data is then sent back to the client and presented using the presentation logic on the client tier.

The enterprise application model divides the work required to implement the multitier service into two categories:

1. The business logic and presentation logic which are supposed to be implemented by the developer.
2. The standard system services which are generic for any application such as connecting to the database, establishing a session, and so on.

The Standard system services are provided by Java EE 7 platform. There can be multiple tiers in an application based on requirement; in which case, the architecture is termed as n-tier architecture.

1.2 Multitier Applications

A multitier application has the application logic divided into multiple components based on their functionality. These multitier applications are distributed where each of the application components may be installed on the same or on different machines depending on the tier to which it belongs. In a typical enterprise application there are three tiers. Figure 1.2 illustrates the multitier architecture and its respective components.

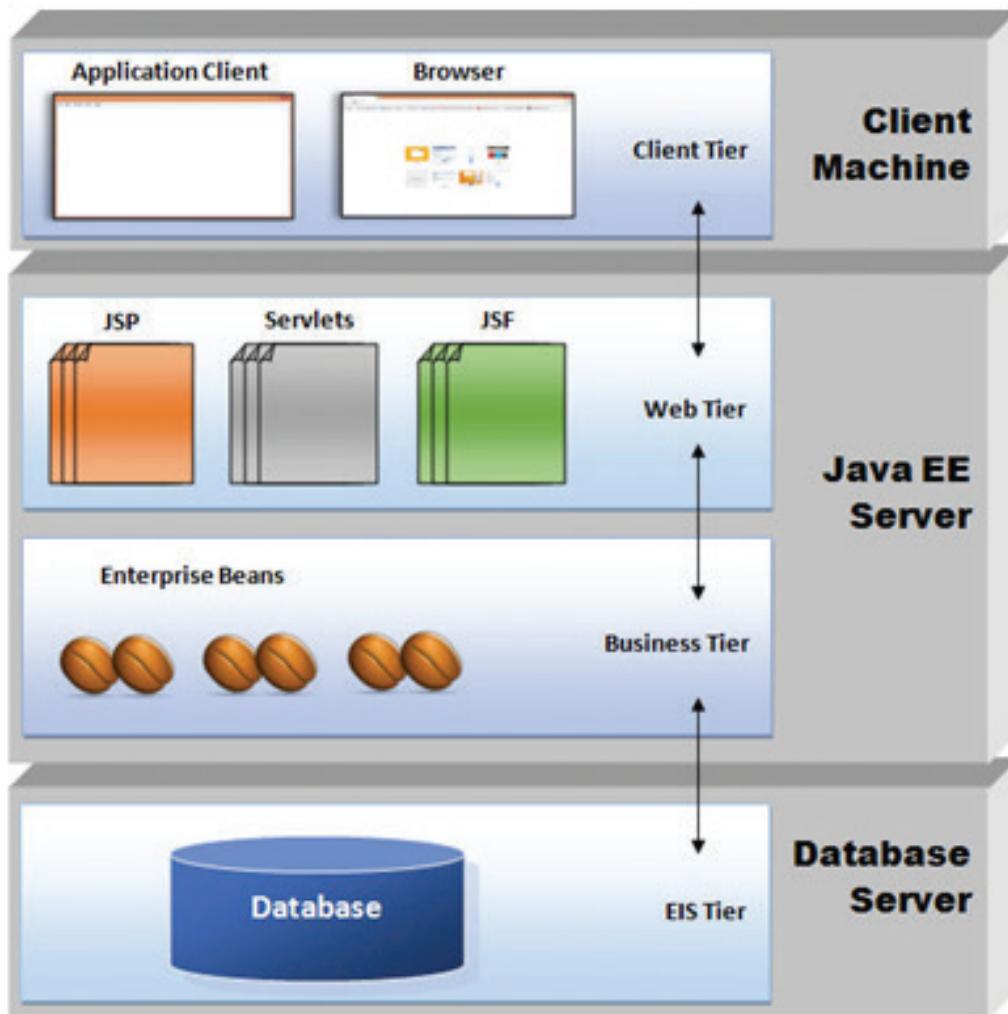


Figure 1.2: Multitier Applications

Various components of Java EE multitier application are as follows:

- ➔ Java EE clients
- ➔ Web components

- Java EE components
- Database components
- Security components

A Java EE application comprises various components that function together. A component can be defined as a fully functional software unit which can communicate with other components.

There are three essential components that are widely used in applications:

1. Application clients
2. Web components
3. Enterprise components

Application clients form the interface for the usage of the application. Examples for application clients are Web browsers for Web applications, user forms for enterprise applications, and so on.

Web components refer to client side scripts such as JavaScript, server side scripts such as Servlets, Java Server Faces, and so on.

Enterprise components refer to the business components such as Enterprise Java Beans (EJB) which run on the server.

1.2.1 Java EE Clients

Java EE clients can be:

- Web clients
- Application clients
- Applets
- Java Bean components

A Web client consists of two parts – dynamic Web pages and Web browsers. Dynamic Web pages are generated by the Web components in the Web tier through markup languages such as HTML and XHTML. The Web browser is responsible for rendering the Web pages.

An application client runs on the client machine and has a user interface such as forms developed through Abstract Window Toolkit (AWT) or Swing. These application clients communicate with the business logic which in turn is implemented through EJB.

Applets can be defined as Web clients that are written in Java language. These are stand alone applications executed through the Web browser.

The multitier application model is an extension of the basic client server application model with additional middle tier and other tiers such as security tiers. Java bean components can be present in both client and server tiers. These bean components manage the data flow between the application client and components running on the Java EE server and also, the data flow between the server components and a database.

1.2.2 Web Components

Web components in an application can either be a Web page at the client end or a script running at the server end to service client requests. Servlets are the server side scripts and on the client side there are Web pages generated through Java Server Pages (JSP) and Java Server Faces (JSF).

Servlets are invoked in response to a user request and construct a response page after processing the input. JSP pages are text based documents which execute as servlets. JSFs are built on top of both JSP and Servlets, primarily used to create the user interface.

Static HTML pages and applets as well as server side utility classes are not considered as Web components according to the Java EE specification.

1.2.3 Java EE Components

Java EE Components are business components that implement the logic of application domains such as banking, health care, mobile services, and so on. These components are essentially enterprise beans running in the application tier and client tier.

The enterprise bean is capable of receiving data from both the client tier and the data tier. The bean receives the data from the client tier processes it and stores it in the storage. Similarly, it can access data from the data storage, process it, and send it to the client.

1.2.4 Enterprise Information System Tier

This is the data repository of the application. This tier includes management of enterprise infrastructure systems such as resource planning, mainframe transaction processing, database systems, and so on.

1.2.5 Security

Java EE enables definition of the security constraints at the time of deployment. The application developers are shielded from the complexity of implementing these security features. The developer only defines the security features through a set of declarative access control rules. The implementation of these rules is taken care by the Java EE platform. Java EE also provides various login mechanisms and therefore, the developer does not require making efforts in implementing these login mechanisms.

1.3 Web and Application Servers

The Java EE components must be packaged and deployed on a server to make them available on the Web.

Figure 1.3 shows the Web and Application servers.

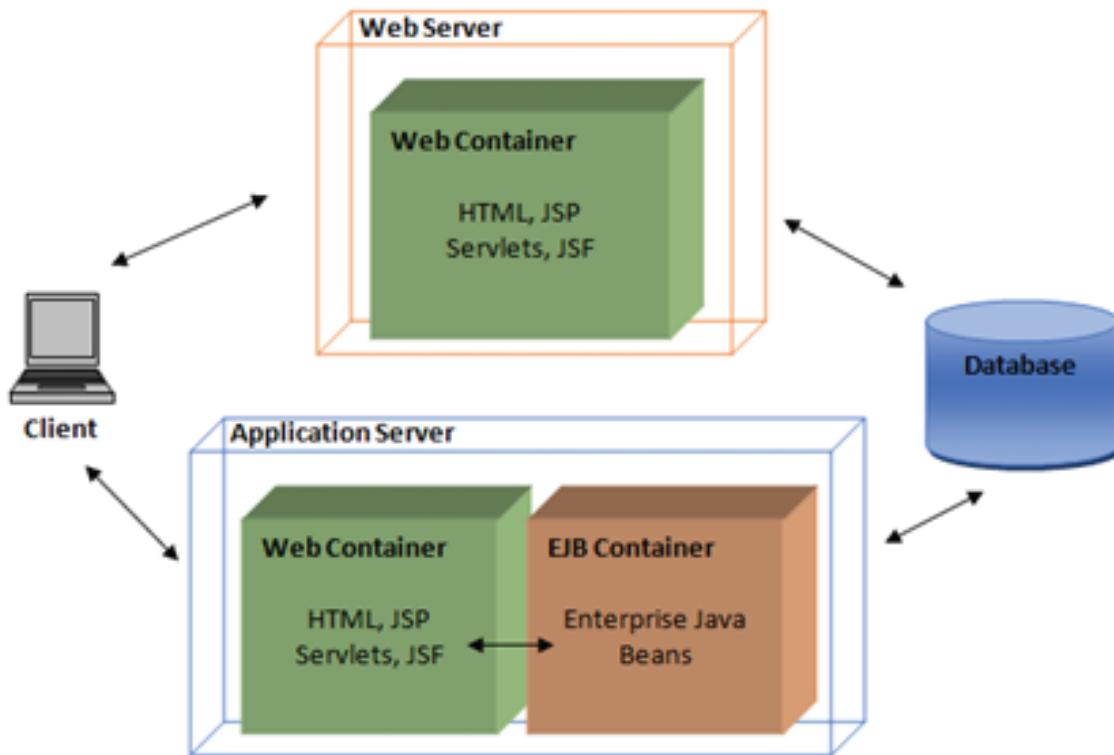


Figure 1.3 Web and Application Server

Figure 1.3 Web and Application Server

The Web server consists of a Web container that holds all the Web components. The application server consists of Web containers as well as an EJB container that holds all the enterprise Java bean components. Based on the requirement of the application, a Web server or an application server can be used for deployment.

1.3.1 Web Servers

Web servers accept requests from clients through Web pages and respond to these requests with the appropriate Web page. Web servers can be implemented through hardware or software and are primarily responsible for delivering Web content to the clients.

The communication between the client and Web server takes place through Hyper Text Transfer Protocol (HTTP) and the response to the client request is built in the form of HTML pages. The request from the client is interpreted through server side scripts such as Servlets.

1.3.2 Application Servers

An application server is a software component which is responsible for all the operations of an enterprise and for implementing the business logic of the enterprise.

For example, consider an enterprise application for banking domain, the application server should implement the functions required by the employees of the bank where employees interact with the database and carry out various transactions.

The application server of a bank should also function along with a Web server which provides the facilities of Internet banking to its customers. The application server will be aware of the business logic of the bank. The business logic in this case can be the interest rate applicable to certain accounts, the penalty for less than minimum balance situation, and so on. The application server is responsible for implementing all these domain specific issues.

1.4 Containers

Java EE simplifies the application development process by providing certain services to the developer through containers and components.

A container contains various components which together provide some services to the application. The services include transaction management, state management, resource pooling, thread pooling, security, and so on, which are utilized by the developers in majority of the applications. These services help in reducing the effort of developing these functionalities from scratch. These services such as transaction management, resource pooling, and so on can be implemented through different components and put together in a single container according to the requirement of the application. The support and services provided by containers and components enable the developer to concentrate on implementing the actual business logic. Figure 1.4 displays containers in Java EE.

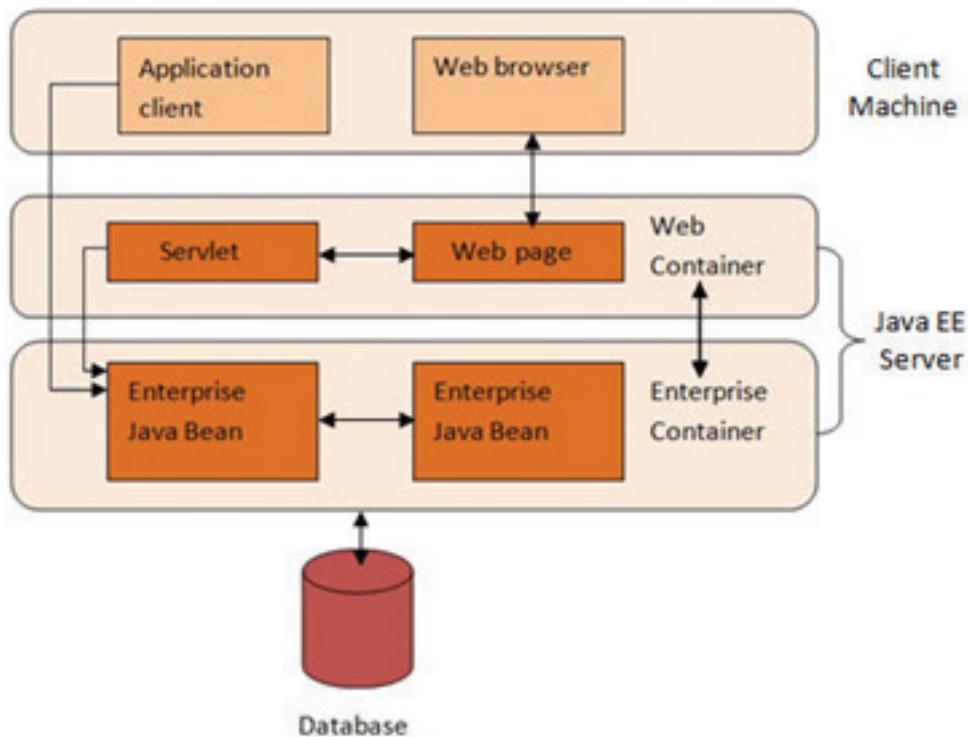


Figure 1.4: Containers in Java EE

1.4.1 Container Services

In order to use the container according to the requirement of the application, it is essential to configure the container settings. Following is a list of services provided by containers in a Java enterprise application:

- When an enterprise or Web application requires security mechanisms, Java EE enables you to configure the components to define access to various resources of the applications.
- Various methods can be put together to form a Java transaction in which case, Java EE transaction model enables you to establish the link between various methods.
- In scenarios where the server and client are located at different geographical locations, Java EE remote connectivity manages communication between the client and the server. Java EE client ensures that the client is unaware of the actual location of the server.
- Java Naming and Directory Interface (JNDI) provides naming service to access different objects and components in the enterprise application domain.

The container allows different configurations of the same application component based on the context of the application. The container manages non-configurable services such as enterprise beans, servlet lifecycles, database access, and so on. The different containers and their interaction are shown in figure 1.4.

1.4.2 Container Types

Various components of the enterprise application are installed in different containers during deployment. They are:

- Web Container
- Enterprise Java Beans Container
- Applet Container
- Application Client Container

The Web container is installed on the Java EE server, which comprises all the components of the Web applications. The Web container is responsible for handling HTTP communication, execution of servlets, managing the life-cycle of Web components, and so on.

The EJB container is also present on the Java EE server, which consists of the EJB components implementing the business logic of the application.

The Applet container manages the execution of applets. This container comprises the Web browser and plug-ins.

The Application Client container manages various application clients.

1.5 Development and Deployment

Developing a Java enterprise application involves various stages as follows:

- Procuring Java platform and tools for development
- Development of components by application component providers
- The components generated by the application providers are assembled by the assemblers and further deployed by the deployers

Product providers are the vendors who enable purchase of Java EE APIs and other features defined in the Java EE specification.

Tool providers are those vendors who provide application development tools such as Eclipse, NetBeans, and other tools for assembly, deployment, and packaging.

Application components include the bean component for enterprise applications, Web component for the Web applications and application clients. Following are the application component providers:

1. A Bean developer writes and compiles the source code, creates the deployment descriptor, and packages the .class file as an EJB JAR file.
2. A Web component developer will write source codes for servlets, JSPs, JSFs, and HTML files. The .class files, .jsp files, and .html files created are packaged as WAR files. As in case of Bean, the source code of the application client is written by an application client developer and compiled to generate a .class file along with the deployment descriptor for the class file. The .class files are then packaged as a JAR file.
3. An Application assembler receives all the components from the application component providers and assembles the respective JAR and WAR files to EAR (Enterprise Archive) file. The assembler may or may not specify the deployment descriptor by adding appropriate XML tags.
4. An Application deployer is responsible for deploying the application in the operational environment. The deployer configures the operational environment based on the specifications provided by the application component provider to resolve external dependencies, specify security settings, and assign transaction attributes.

A Java enterprise application is packaged as one or more standard units for deployment. Each unit has a functional component/components and deployment descriptor. A single developer can play all the mentioned roles or these roles can be divided among individuals and teams. This kind of division is possible because each stage outputs a portable file which is an input to the subsequent stage.

1.5.1 APIs in Java EE7

Java EE 7 provides a set of APIs that can be used for development of Web and enterprise applications.

Figure 1.5 displays the APIs used in EJB containers. The Concurrency utilities, Batch processing, and JSON-P APIS are new inclusions in Java EE7.

EJB Container		Java EE 7
EJB	Concurrency utilities	
	Batch	
	JSON-P	
	CDI	
	Dependency Injection	
	JavaMail	
	Java Persistence	
	JTA	
	Connectors	
	JMS	
	Management	
	WS Metadata	
	Web Services	
	JACC	
	JASPI	
	Bean Validation	
	JAX-RS	
	JAX-WS	

Figure 1.5: APIs Used in EJB Container

Figure 1.6 displays the APIs used in Web containers.

Web Container		Java SE 7
Servlet	WebSocket	
	Concurrency Utilities	
	Batch	
	JSON-P	
	Bean Validation	
	EJB Lite	
	EL	
	JavaMail	
	JSP	
	Connectors	
	Java Persistence	
	JMS	
	Management	
	WS Metadata	
	Web Services	
	JACC	
	JASPI	
	JAX-RS	
	JAX-WS	
	JSTL	
	JTA	
	CDI	
	Dependency Injection	

Figure 1.6: APIs Used in a Web Container

Apart from components such as EJB, Servlets, Java Server Pages, and Java Server Faces, various APIs and libraries are required in Java EE application development. Following is a brief description of the APIs and libraries that are widely used in application development.

1.5.2 Java Server Pages Standard Tag Library

JSTL is a set of JSP tags that are widely used in JSP applications. The tags in JSP have varying functionalities such as tags used for formatting the Web page, tags for iterating and managing the flow control, tags for accessing database, and so on. Java EE 7 uses JSTL 1.2.

1.5.3 Java Persistence API

Java Persistence API is used for bridging the gap between object-oriented interpretation of data and relational data in databases. A persistence entity is a lightweight class representing a table in the relational database. The instances of the class represent rows of the table.

The Persistence API has a query language for the entities in the database. Java EE7 uses Java Persistence API 2.1.

1.5.4 Java Transaction API

Java Transaction API is responsible for transaction management in the application database. It is responsible for commit and rollback operations on tables of the database and maintains database integrity. It ensures that every transaction reads appropriate and accurate data.

1.5.5 Java API for RESTful Web Services

REST stands for Representational State Transfer, which is an architectural style to develop Web services. REST defines certain architectural properties for the application such as performance, scalability of the application, simplicity of the interface, ability to modify the components, portability of component deployment, and so on. The API provides means to develop Web services with the given qualities. Java EE 7 requires JAX-RS2.0.

1.5.6 Managed Beans

Managed Beans are light weight objects running on the Java Virtual Machine, responsible for management related tasks of the application. They are responsible for reporting events such as faults, state changes, performance of the application, and so on. These managed beans can be used in both Web applications and enterprise applications. They are part of Java EE 7 specification that uses Managed Beans 1.0.

1.5.7 Contexts and Dependency Injection for Java EE

CDI is a framework which is an integral part of Java EE platform. It provides an architecture that enables various components of an application such as servlets, beans, and so on to exist only within the life cycle of the application. It defines the context of each component. It allows integration of components in a loosely-coupled but type safe manner.

The components are injected (activated) into the context of the application life cycle based on certain events. The components when added to context, CDI ensures that it is done in a type safe manner holding all the dependencies intact.

1.5.8 Bean Validation

Bean Validation process is responsible for validating the data in beans in various tiers of application. Instead of distributing the task of validation across multiple layers an API is defined along with the metadata model for validation of beans. The API enables defining the validation constraints in one location which is further used across layers. Java EE7 requires Bean Validation 1.1.

1.5.9 Java Messaging Service API

The Java Messaging Service API is used by the components to create, send, receive, and read messages. This messaging standard is used to enable distributed communication which is reliable and asynchronous.

1.5.10 Java EE Connector Architecture

The Connector Architecture is responsible for creating resource adapters that will access Enterprise Information Systems. These resource adapters are used by Java EE applications. A resource adapter is a specific resource manager. One of the common resources in the EIS layer is database.

The connector architecture also integrates the Web services with the existing EISs and ensures that the integration is performance oriented, secure, and scalable.

1.5.11 JavaMail API

Java Mail API enables sending mail notifications. The API has an interface used by application components that are used to send mail and a service provider interface, where service provider refers to the network carrier of the mail.

1.5.12 Java Authorization Contract for Containers

The JACC defines a contract between the Java EE components of the application and authorization policy providers. Based on the authorization policy `java.Permission.security` classes are defined by the JACC. This is in the case of third party policy providers.

The JACC provides a set of standard permission types relevant for all enterprise and Web applications. It also provides a standard model for role-based authorization, grouping certain permissions along with the roles defined.

1.5.13 Java API for Web Socket

Web Socket refers to an application protocol which provides full duplex communication between two communicating entities. The Web Socket API helps in creating the communication end point and allows specifying the configuration parameters. This is a new API introduced in Java EE 7.

1.5.14 Java API for JSON Processing

JSON refers to a text based data exchange format, which is used in Web services and other applications. The JSON data exchange format is derived from JavaScript. The API enables Java EE applications to parse, transform, and query data from JSON using the object model or streaming model.

1.5.15 Concurrency Utilities for Java EE

The Concurrency Utilities of Java are responsible for providing asynchronous capabilities to enterprise application components. These utilities include managed executor service, managed scheduled executor service, managed thread factory, and so on.

1.5.16 Batch Applications for the Java Platform

Batch applications refer to the set of applications which do not have extensive input and output. The Batch Applications framework for Java enables creating and executing batch jobs. The batch framework has a batch runtime, job specification language based XML, an API to interact with the Java runtime, and an API to implement batch artifacts.

1.6 Check Your Progress

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

a.	Components of various layers in an application have to run on different machines.		
b.	Components of various layers in an application may or may not run on different machines.		
c.	The client tier represents various users of the enterprise application.		
d.	Middle tier communicates with both application client layer and the data layer.		
(A)	a, b, d	(C)	b, c, d
(B)	a, c, d	(D)	a, b, c

2. _____ can be a Web client of a Java enterprise application.

(A)	Web browser	(C)	HTML page
(B)	Applet	(D)	All of these

3. Which of the following containers are responsible for HTTP communication and execution of servlets?

(A)	Web container	(C)	Application client container
(B)	Applet container	(D)	EJB container

4. Match the following Java EE APIs with the corresponding description.

	API		Description
a.	Java Message Service API	1.	Helps in creating the communication end point
b.	Java Mail API	2.	Used by the components to create, send, receive, and read messages
c.	Java API for Web Socket	3.	Responsible for commit and rollback operations on tables of the database
d.	Java Transaction API	4.	Enables sending mail notifications

(A)	a-4, b-3, c-1, d-2	(C)	a-3, b-1, c-4, d-2
(B)	a-2, b-4, c-1, d-3	(D)	a-2, b-3, c-4, d-1

5. In three tier architecture, the business logic of the enterprise is implemented in the _____ layer.

(A)	EIS	(C)	Client
(B)	Middle	(D)	Any of these

1.6.1 Answers

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

```
String pack( String str ) {  
    String p = str; // local variable  
    if (str.length() > 1) {  
        p = str.substring(1);  
    }  
    return p;  
}
```

Summary

- The Java enterprise application model defines how various components of an enterprise application are organized.
 - A component can be defined as a fully functional software unit which can communicate with other components.
 - A Web client consists of two parts – dynamic Web pages and Web browsers.
 - A container contains various components which together provide some services to the application. The services include transaction management, state management, resource pooling, thread pooling, security, and so on.
 - The EJB container is also present on the Java EE server, which consists of the EJB components implementing the business logic of the application.
 - Application assembler receives all the components from the application component providers and assembles the respective JAR and WAR files to EAR (Enterprise Archive) file.
 - Java Persistence API is used for bridging the gap between object-oriented interpretation of data and relational data in databases.



Visit the
Frequently Asked Questions
section @

Session - 2

Enterprise Application Architecture

Welcome to the Session, **Enterprise Application Architecture**.

This session discusses various concepts of enterprise application architecture which are useful in designing Java enterprise applications such as design patterns, layered architecture, and so on.

In this Session, you will learn to:

- ➔ Explain enterprise application design
- ➔ Use various design patterns in application design
- ➔ Describe Model-View-Controller design
- ➔ Explain communication among application components
- ➔ Describe network topologies and clustering
- ➔ Describe the layered architecture of application design



2.1 Introduction

Application architecture has evolved from client-server architecture to N-tier architecture. In case of client-server architecture, the client layer would request services and the server would respond by providing the services requested by the client. As the applications scaled among multiple clients and multiple servers, thereby increasing the complexity of applications, a middle tier was added between the client and the server layer resulting in three-tier architecture. The middle layer is responsible for servicing the data requests from the client and managing the data appropriately on the data servers.

The three-tier architecture evolved to N-tier architecture with the advent of Internet and better network infrastructure. Cloud applications and data centres enable locating various components of the application such as data servers, Web servers, applications, and client components in different geographical locations. N-tier architecture of Java provides the infrastructure to manage these components. Distribution of processing into various layers results in better resource utilization and efficient functioning.

When an application is developed as independent components, all of them are integrated through a uniform interface and also, require various other services to function properly. These services include the security mechanisms for data transmission among the components and compliance with Internet protocols to communicate over the Internet.

2.2 Designing an Enterprise Application

Application design in Java is object-oriented, which implies that the designer of the application has to identify the objects in the context and generalize a group of objects with common behavior and properties into classes. Once the objects in the context of the application are identified, the designer has to identify the interaction between various objects and determine appropriate methods for these classes. This is modelled through Unified Modelling Language (UML) diagrams.

The design in Java being multi-layered, these objects have to be placed in appropriate layers in the application components. The objects in different layers are assigned responsibilities and interfaces are laid out for interaction between layers. This design of distributed objects is more complicated than simple object-oriented design, as the designer has to consider the aspects of scalability, performance, transactions, and so on.

Object-oriented programming encourages reuse of code, classes, and objects. While designing a solution for a problem, it is considered a good practice to reuse an existing solution than redesigning the solution from scratch. This will reduce the development time and required effort to reach the final solution. Design patterns enable this reuse in Java applications.

While designing solution for a problem, if a similar problem has been solved earlier, it can be inferred that the solution for the current problem will also be on similar lines. Design patterns suggest a pattern of resolution for certain problems based on solutions for similar problems resolved earlier. Design patterns allow reuse of such designs and architecture that were successfully applied in solving a similar problem.

2.3 Design Patterns

A design pattern identifies a recurrently occurring problem in a domain and offers a solution to the problem. The solution offered can be used to solve similar problems occurring in the future. The design pattern identifies the classes and instances, their roles and collaborations, and distribution of responsibilities. Based on these aspects of the problem, an appropriate design pattern can be identified and used to design a solution for the problem. Each design pattern is described based on a consistent format and is divided into sections based on a template. Following are the sections of the template of design patterns:

- **Pattern name and classification** – This is a conceptual category for the pattern
- **Intent** – Describes the type of problems, this particular pattern can solve
- **Also known as** - Implies other common names of the pattern
- **Motivation** – A scenario which illustrates the problem
- **Applicability** – Describes the situations where the current pattern can be used
- **Structure** – Diagram using an object modelling technique such as UML
- **Participants** – Implies the classes and objects in the design
- **Collaborations** – Defines how the classes and objects in the domain, collaborate with each other
- **Consequences** – Describes the outcome of using this pattern and also describes any kind of side effects it may lead to
- **Implementation** – Describes the implementation specific or language specific issues that might arise on, using this design pattern
- **Sample code** – Provides an example of the design pattern created in languages such as Java or any other object-oriented languages and enables the developer to translate the code with similar classes and objects
- **Known uses** – Illustrates some real world scenarios where this design pattern has been successfully put to use earlier
- **Related patterns** – Provides comparison with similar patterns, so that the developer can choose the right pattern for their application

While designing an application, if the developer intends to use a design pattern, then the developer should first, understand the context and intent of the pattern. Identifying the classes and objects in the given domain, is the most critical aspect for an object-oriented application which can be done with the help of design patterns.

There are four primary categories of design patterns:

1. **Creational patterns** – Define the object creation process in the application
2. **Structural patterns** – Define the organization of classes and objects in an application
3. **Behavioral patterns** – Define the interaction between various objects in the application
4. **Concurrency patterns** – Define the access of resources by objects of the application

2.3.1 Creational Patterns

Creational patterns are useful in scenarios where the object composition is crucial for the application. It provides 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 the product objects which vary widely in structure and functionality.

Singleton pattern and abstract factory pattern are examples of creational patterns.

→ Singleton Pattern

Singleton pattern restricts on the number of objects that will be created during each object instantiation. It allows instantiation of only one object for the class. It is implemented by creating a class with a method which creates a new instance of the class, only when there is no other instance of the class existing. If another instance of the class is present, then the methods return a reference to the existing object instead of creating a new instance. The constructor of such a class is made private so that the object of the class cannot be instantiated in any other manner.

Code Snippet 1 shows the instantiation of an object according to singleton pattern.

Code Snippet 1:

```
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 1, 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 will create 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, then 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.

Other creational patterns are factory pattern, abstract factory pattern, builder pattern, and prototype pattern.

2.3.2 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 interfaces and their implementations to work together. 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.

Adapter, proxy, and decorator patterns are commonly used structural patterns.

→ Adapter Patterns

Adapter design pattern is used when two classes with two different interfaces have to function together. This is done by converting the interface of one class into an interface acceptable by the clients.

Figure 2.1 depicts a Class diagram of an adapter pattern.

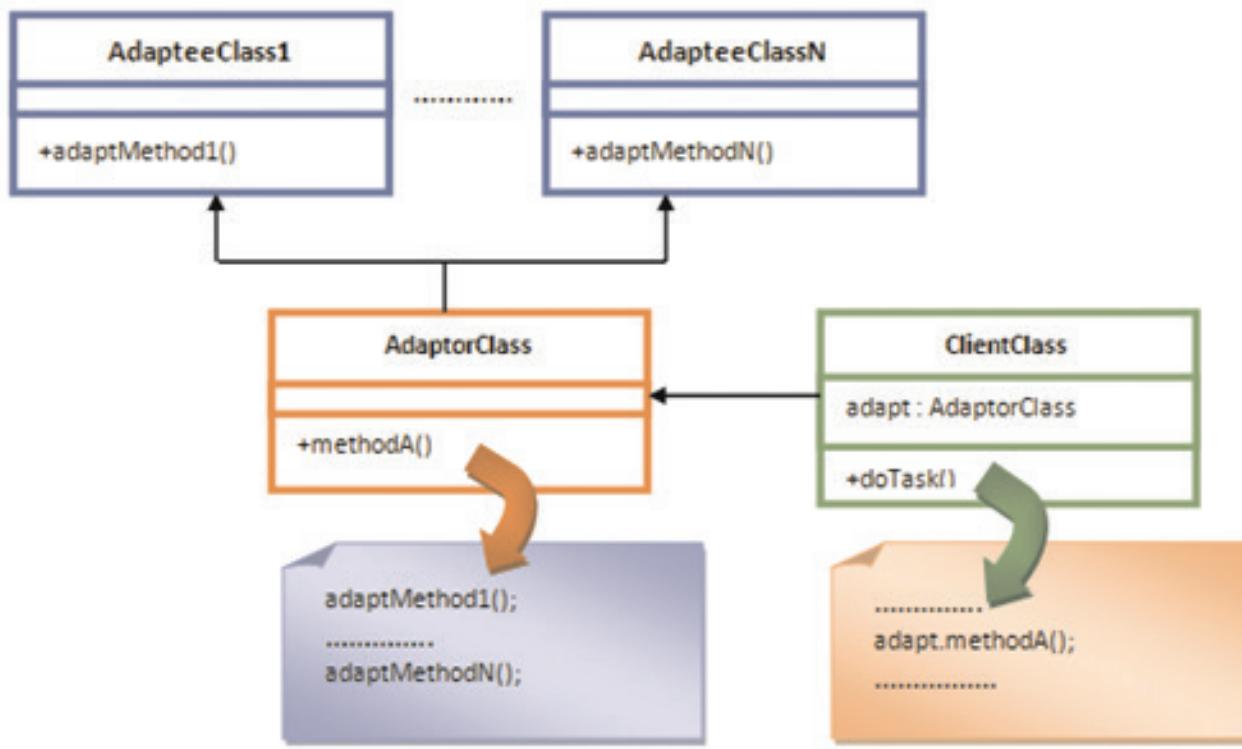


Figure 2.1: Adapter Pattern

An adapter pattern is used in the following situations:

- When the developer needs to use an existing class and its interface, does not exactly match what is required by the developer
- When the developer has to create a reusable class which has to function along with other incompatible classes

Proxy and decorator patterns are other structural patterns.

2.3.3 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.

2.3.4 Concurrency Patterns

Concurrency patterns are applied in cases where there are certain shared resources. These patterns ensure that the access to these resources is consistent and coordinated.

Single thread execution is the most common concurrency pattern. This pattern ensures that only one thread is accessing a certain resource at any instance of time. In Java, this is implemented through synchronized blocks of statements. The section of code which has to be accessed by only one thread is termed as a critical section.

Active object, double-checked locking pattern, and reactor pattern are other concurrency patterns.

2.4 Model-View-Controller

Model-View-Controller is a methodology of designing user interfaces for applications. It divides the components of the application primarily into three parts, model, view, and controller. With respect to the user interface, the view is most important. When there are multiple users for the application, each user has a different perspective for the application data. Each perspective of the end user is a view of the application.

The model is the logical part of the application which includes the business logic, Web logic, and database which provides data to the user interface and accepts data from the user interface to process it. The controller accepts data from the user interface and converts it into a format understandable by the model.

Apart from the functions of the components, MVC architecture also defines the interactions between these three components. Figure 2.2 shows the interactions among various components of MVC architecture.

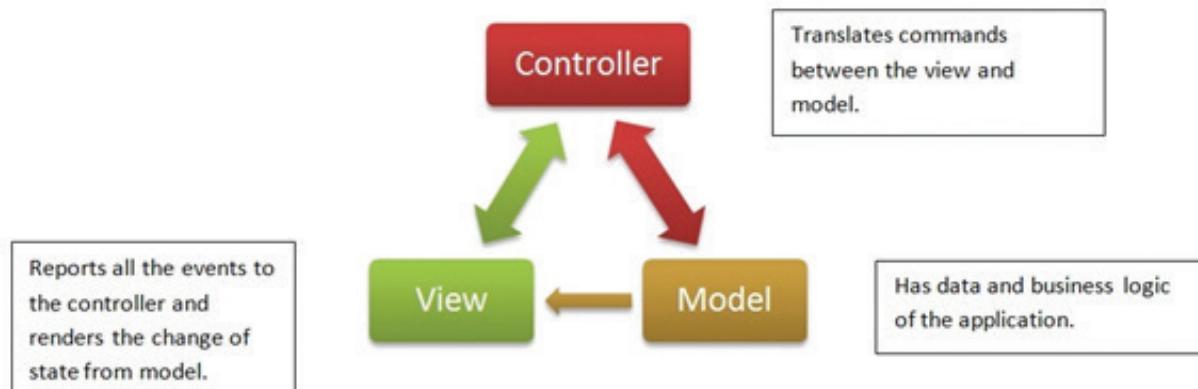


Figure 2.2: Model-View-Controller Architecture

According to MVC, the data flow between the user interfaces, that is, the view and the application, or model is mediated by a controller.

View refers to that part of the application which the end user actually sees. This can be a Web page or a data format. The user response is captured in the view. The data captured in the view is sent to the business components such as servlets (in Web applications), beans (in enterprise applications) through a controller.

The controller may receive data from the user interface, in the form of HTTP request or any other interface specific form. The controller translates this data into a form understandable to the business components, for instance the controller would invoke the correct servlet based on the data received from the view and return the response from the servlet to the view.

Model refers to the context specific processing done by the application. Consider an example of a banking application where the user logs in with a login ID and password. The controller receives this data and passes it to appropriate servlet and fetches the user data from the database. Here, model refers to the user data in the database.

Consider a Java application which uses Swing as user interface. In this case, Swing is the view component in the MVC architecture, listener is the controller, and model will be the Java class that uses the listener or database.

The MVC architecture aims at separating the presentation and implementation of the program logic. This separation enables independent modifications to the user interface and program implementation without affecting other components.

Code Snippet 2 shows a code with a UI component and controller.

Code Snippet 2:

```
package mvcpkg;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MVCdemo {

    // initialize the GUI controls
    private JFrame mainFrame;
    private JLabel headerLabel;
    private JLabel statusBar;
    private JPanel controlPanel;

    public MVCdemo() {
        prepareGUI(); // invoke prepareGUI method
    }

    public static void main(String[] args) {

        // instantiate the class
        MVCdemo objMVC = new MVCdemo();
        objMVC.clickOK();
    }

    // creates the GUI
```

```
private void prepareGUI() {  
  
    // design the GUI  
    mainFrame = new JFrame("To Demonstrate MVC");  
    mainFrame.setSize(400,400);  
    mainFrame.setLayout(new GridLayout(3, 1));  
  
    headerLabel = new JLabel("",JLabel.CENTER );  
    statusLabel = new JLabel("",JLabel.CENTER);  
    statusLabel.setSize(350,100);  
  
    // add listener to the frame  
    mainFrame.addWindowListener(new WindowAdapter() {  
        @Override  
        public void windowClosing(WindowEvent windowEvent) {  
            System.exit(0);  
        }  
    });  
    controlPanel = new JPanel();  
    controlPanel.setLayout(new FlowLayout());  
  
    mainFrame.add(headerLabel);  
    mainFrame.add(controlPanel);  
    mainFrame.add(statusLabel);  
    mainFrame.setVisible(true);  
}  
  
// handles the click of OK button  
private void clickOK(){  
  
    JButton okButton = new JButton("OK");  
    okButton.setActionCommand("OK");  
  
    // add listener to the OK button  
    okButton.addActionListener(new ButtonClickListener());  
    controlPanel.add(okButton);  
  
    mainFrame.setVisible(true);  
}  
  
// handler class for click event of the OK button  
private class ButtonClickListener implements ActionListener{  
    @Override  
    public void actionPerformed(ActionEvent e) {
```

```
String command = e.getActionCommand();
if( command.equals( "OK" ) ) {
    statusLabel.setText("ButtonClickListener(Controller) invoked");
}
}

}
```

Code Snippet 2 shows only two components of the MVC architecture, view and controller. The view is defined by the method `prepareGUI()` and `ActionListener` is the controller which is implemented by the `ButtonClickListener` class and generates the response to the input from the view. Further, this UI can be coupled with a bean component and/or a database to add the model component of the MVC architecture.

Figure 2.3 shows the output when the code is executed.

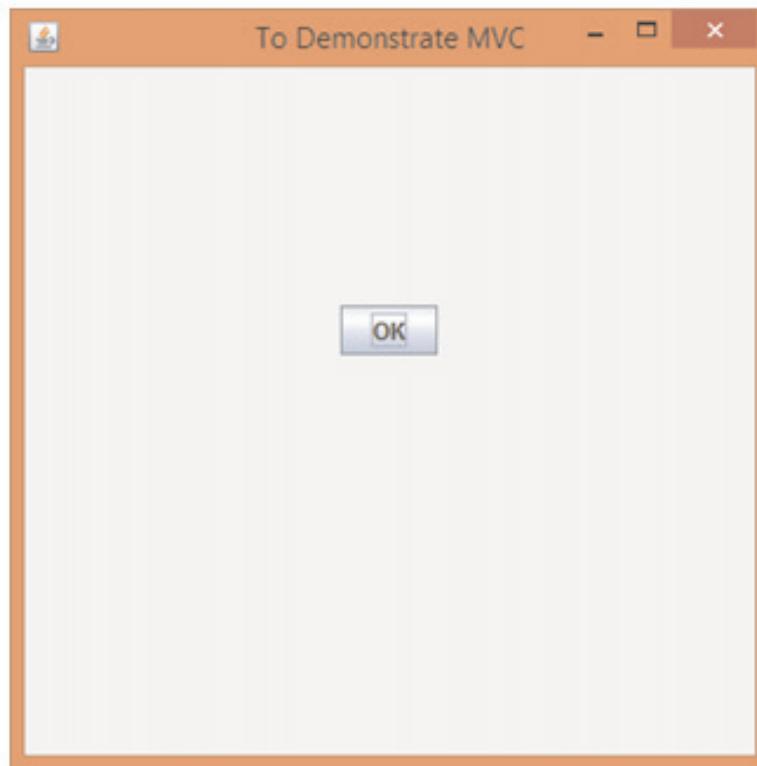


Figure 2.3: Output When Code is Executed

Figure 2.4 shows the output after an event occurs in the GUI, in this case, it is the button click.

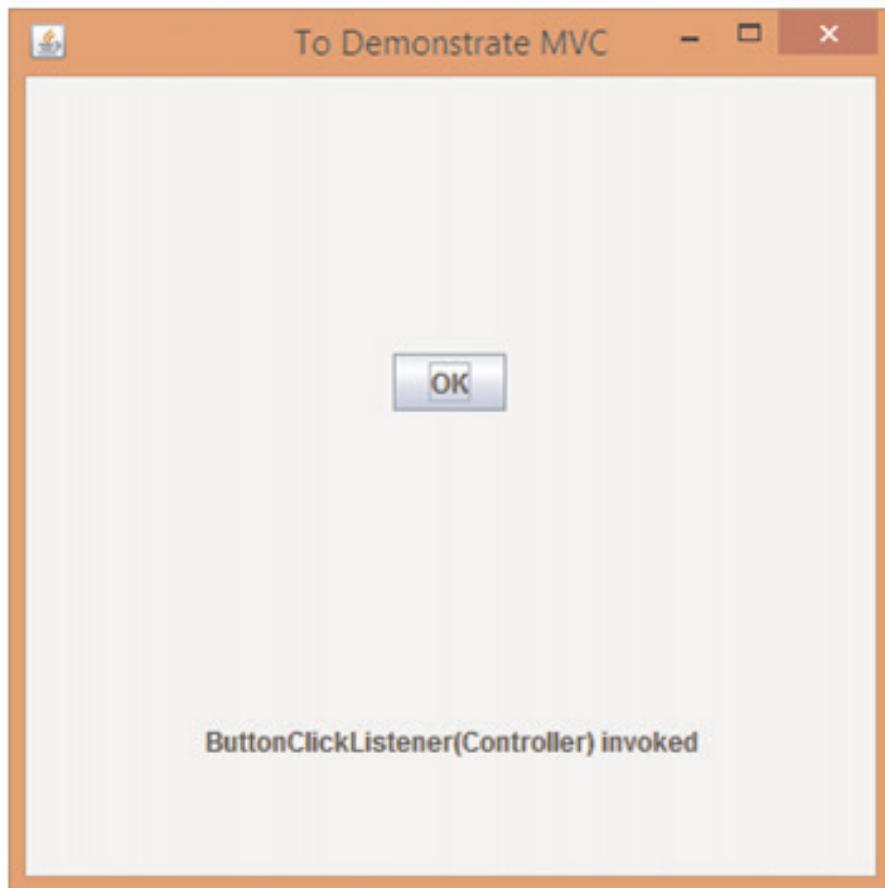


Figure 2.4: Output After the Button Click Event

2.5 Synchronous and Asynchronous Communication

Java applications are developed as independent components according to multi-layered architecture. These independent components that may reside at geographically distinct locations are integrated together and work in collaboration.

In order to make these components collaborate efficiently, the communication between them plays an important role. The communication among components of an application can be synchronous or asynchronous.

Synchronous communication is one where a request is sent and the sender waits for the response from the receiver before sending the next request.

Asynchronous communication is one where the request is sent but the sender does not wait for response from the receiver.

Remote Method Invocation (RMI)/Remote Procedure Call (RPC) in Java is an implementation of synchronous communication and Java Messaging Service (JMS) is an implementation of asynchronous communication.

Java implements both synchronous and asynchronous communication through Java Messaging Service (JMS) API.

2.6 Clustering and Network Topologies

Clustering refers to a technique where an application is deployed over a group of servers which is termed as a cluster of servers. A cluster of servers is a group of servers that are interconnected and run an application in a way that the end user is unaware of multiple entities running as servers. According to the user, the server is only a single entity.

Clustering technique is important to scale the application across multiple users and make the application highly available. There are certain essential services and components such as JNDI, EJB, JSP, and so on, that are used for Java applications to function. Clustering aims at ensuring that these services and components are always available.

Following are the various purposes of clusters:

- Load balancing
- Fail over

In case of Web applications:

- Web load balancing
- HTTP fail over

In generic terms, load balancing is a process wherein if a server is overwhelmed with client requests, then certain client requests can be serviced by another server which will respond in an identical manner. This is applicable for both enterprise and Web applications. A load balancer manages the client request and maps it onto appropriate server.

Fail over of server refers to a situation of server crash. When a server crashes all the requests being serviced by that server, are transferred to another server which would respond in an identical manner. In case of Web applications, it is termed as a HTTP fail over. When a HTTP session fails, a new HTTP session is initiated with respect to a different server. A dispatcher is responsible for redirecting the requests to another server.

In order to achieve this, the cluster of application servers should have an efficient communication mechanism and application processing. Figure 2.5 shows the topology of servers in clustering

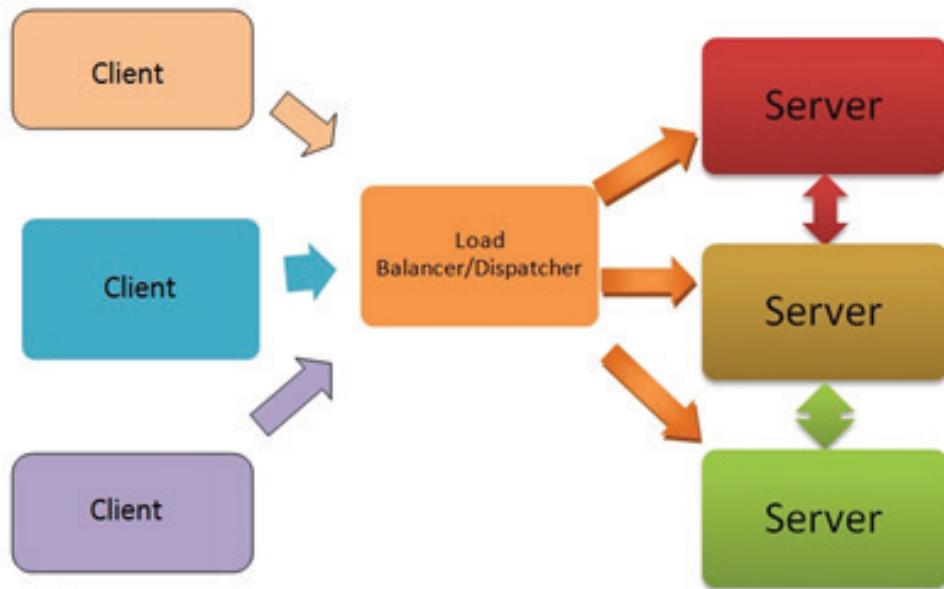


Figure 2.5: Clustering

Figure 2.5 shows that there is a load balancer/dispatcher which receives requests from clients. The load balancer has information about the kind of workload each server in the cluster has and based on this data, the load balancer sends the request to a server such that the workload among different servers is balanced.

Dispatcher is a component which handles the situation of fail over of a server. The fail over can be a HTTP fail over as in case of Web applications or an application server fail over as in case of enterprise applications. In both the cases, the dispatcher sends the service request to another server which is running, receives the response from the server, and sends it to the client that has raised the request. The role of both load balancer and dispatcher is similar in a way that both the components accept client requests and redirect the request to the appropriate server based on certain criteria, receive the response from the server, and return it to the appropriate client.

Databases are also distributed over a cluster of servers to ensure high availability of the database. Techniques such as RAID (Redundant Array of Independent Disks) are used in the case of databases.

→ Network Topologies

The network topology according to which the servers in the cluster are interconnected, has an impact on the accessibility of the server, resulting in performance trade-offs.

Following are some popular topologies of connecting various components of the network:

- Star Topology
- Ring Topology
- Bus Topology
- Tree Topology

→ **Star Topology**

In star topology, every node is connected to a central node called a hub or switch which is the server and the peripherals are the clients. In a network cluster, the central server/router connects to all other servers in the network cluster. Figure 2.6 shows the arrangement of servers according to star topology.

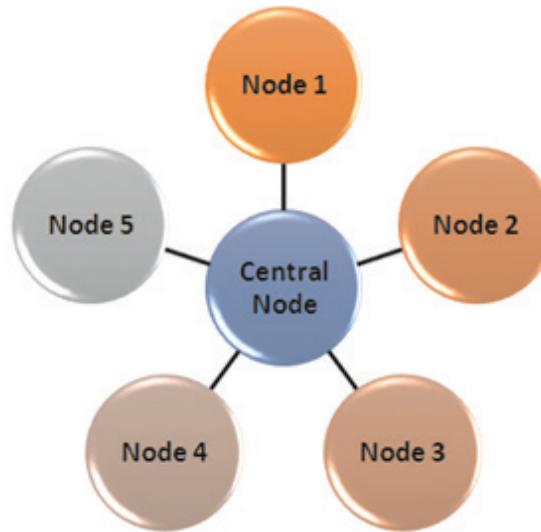


Figure 2.6: Star Topology

→ **Ring Topology**

In ring topology, each node is connected to exactly two other nodes, with the last node connecting to the first node forming a circular ring. The topology leads to formation of a single continuous pathway for signals through each node. Data travels from node to node and each node along the way handles every packet. As there is only one pathway between any two nodes, ring networks may disrupt it even a single link fails. Figure 2.7 depicts the Ring topology.

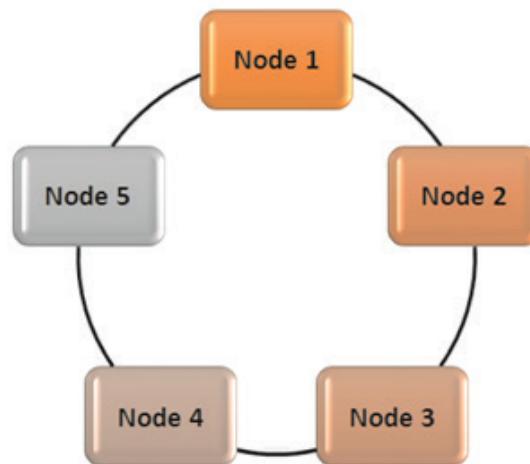


Figure 2.7: Ring Topology

→ Bus Topology

In bus topology, all the nodes are connected in a daisy chain in a linear sequence of buses. The host on a bus network is known as a workstation. All the servers are accessible through the shared bus only. The bus can transmit data only in one direction. If any network segment gets disconnected, all network transmission ceases. Here, each network segment has equal priority and therefore, becomes a collision domain. Figure 2.8 shows the arrangement of servers in bus topology.

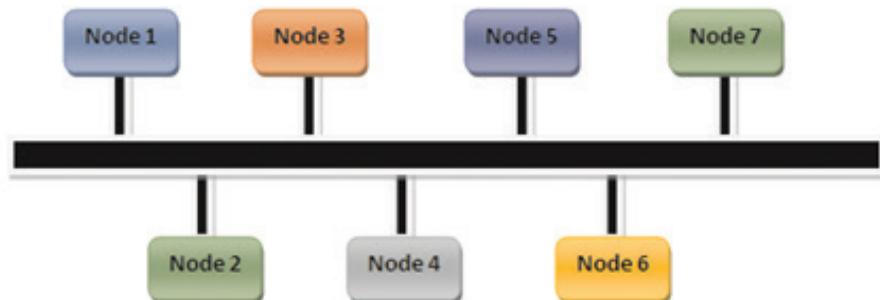


Figure 2.8: Bus Topology

→ Tree Topology

A tree topology follows a hybrid approach by combining the star and bus topologies to improve network scalability. In tree topology, all the servers in the cluster are connected like a tree, which can be a hierarchical or a non-hierarchical tree.

Figure 2.9 shows the arrangement of servers in a hierarchical tree structure.

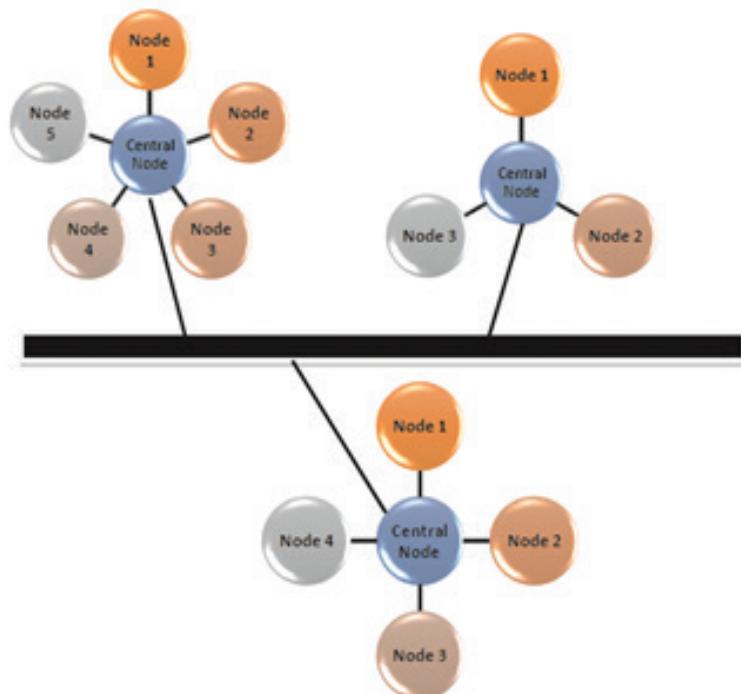


Figure 2.9: Tree Topology

Figure 2.9 indicates that if the main cable or link between the different star topology networks fails, the networks would not be able to communicate with each other. However, nodes on the same star topology, would still be able to communicate with each other.

2.7 Layering in Java Applications

Layers are different from tiers. Tiers refer to hardware components which can be located on different servers or machines for the functioning of application. Layering refers to logical separation of the components, components belonging to different layers may be implemented through different APIs, but may or may not reside on the same machine.

Java applications can have the following layers:

- Client Layer
- Presentation Layer
- Business Logic Layer
- Data Access Layer (Persistence)
- Integration Layer
- Services Layer

2.7.1 Client Layer

Client layer comprises all those entities that access the application. The clients can be browser clients, application container clients, CORBA (Common Object Request Broker Architecture) clients, JMS clients, and Web service clients.

2.7.2 Presentation Layer

Presentation layer generates the user interface for the client. In order to generate dynamic content for the user, the application developer can use components such as servlets, JSPs (Java Server Pages), HTML, CGI, and so on. Apart from the dynamic content, the user interface may also have static content.

Servlets provide session management, simple input validation, and also relates the business logic of the application with the user interface. JSPs handle the display tasks of the servlets. They define the screen navigation and other presentation logic. JSPs can also invoke JDBC objects.

Apart from these, there are HTML, XHTML, and CGI for server scripting, which are responsible for defining the presentation logic of the application.

2.7.3 Business Logic Layer

This layer is responsible for the implementation of business logic of the application. The components that are used for defining the business logic are session beans, entity beans, and message driven beans. Session beans are used in applications, where the client has to establish a session with the server and retrieve specific information. For instance, when a user tries to access his/her personal e-mail, then the client has to establish a session with the mail server and retrieve data. Based on the requirement, one can use stateless session beans or stateful session beans.

Entity beans also implement the business logic where they use persistent objects. This functionality can further be categorized as persistence layer.

2.7.4 Persistence Layer

Persistence layer deals with persistent data objects such as database rows. A persistent object is one which has dedicated storage location in the application database. In order to access them, the application has to instantiate Data Access Objects (DAOs) or JDBC interfaces. The Data Access Objects and the JDBC interfaces together form the persistence layer of the application.

2.7.5 Integration Layer

Java application development is modularized with well-defined components that can be independently developed. The integration layer is responsible for enabling these independent components to function together. The functions of integration layer are enabling communication among the components, message processing, transformation of data formats, transformation of protocol formats, and so on.

2.7.6 Services Layer

The services layer comprises services supporting business functionality. These services include definition of data access policy, security services through policy management tools, and cryptographic services for the application.

The implementation of these layers can be independent or combined based on the application requirement.

2.8 Check Your Progress

1. _____ is a creational design pattern.

(A)	Adapter pattern	(C)	Decorator pattern
(B)	Proxy pattern	(D)	Singleton pattern

2. Which of the following patterns can be used when there is resource sharing among the objects of the application?

(A)	Prototype pattern	(C)	Single thread execution pattern
(B)	Abstract factory pattern	(D)	Singleton pattern

3. _____ is an implementation of asynchronous communication.

(A)	CORBA	(C)	RMI
(B)	RPC	(D)	JMS

4. Which of the following statements are true?

a.	Load balancer comes into action, only when a server crashes in the cluster.
b.	The end user is unaware of the operations carried out by load balancer and dispatcher.
c.	Dispatcher maps the jobs of a crashed server to alternative servers.

(A)	a, b	(C)	a, c
(B)	b, c	(D)	None of these

5. _____ functions in the presentation layer of the application.

(A)	HTML	(C)	Servlet
(B)	CORBA client	(D)	Session bean

2.8.1 Answers

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

Summary

- Application design process in Java is object-oriented and uses design patterns based on the requirements and other attributes of the application.
- The various design patterns such as creational, behavioral, structural, and concurrency are based on the context of the application, components, and so on.
- Model-View-Controller architecture separates the presentation and the logic of the application which enables independent implementation of these components during application development.
- Asynchronous and synchronous communication are two different communication methodologies used to allow independently developed components to work together.
- In Star topology, there is a central server/router which connects to all other servers in the network cluster.
- Layering of the Java applications is essential to logically divide the functions of Java components. The implementation of these layers can be independent or combined based on the application requirement.
- The services layer comprises services supporting business functionality such as data access policy, security services through policy management tools, and cryptographic services for the application.

Get
WORD WISE



Visit
the **Glossary** section

@

www.onlinevarsity.com

Session - 3

Introduction to Web Application Development

Welcome to the Session, **Introduction to Web Application Development.**

This session describes Web applications and the process of developing Web applications using the Java EE7 platform and Netbeans IDE. It also describes the life cycle of the Web application, different Web components, and the protocols used by the Web application to communicate with different components of the application.

In this Session, you will learn to:

- Explain the basics of Web application development
- Describe the different components of a Web application and how they interact with each other
- Describe the protocols and communication standards associated with Web applications
- Explain the basic process of creating a Web application through the Netbeans IDE
- Describe the technologies used to develop Web applications
- Describe Web services and standards associated with Web services



3.1 Introduction to Web Applications

Web applications can be defined as a software entity that comprises Servlets, JSPs, static HTML pages, classes, and resources such as databases working in collaboration with each other to implement certain functionality of a particular domain. Web applications are accessed by the clients through Web browsers or other Web clients. Various components of the application interact through Hyper Text Transfer Protocol (HTML).

In order to understand the function of each component of the Web application, the flow of data from the client to server and response from server to client must be summarized first and the steps for doing so are as follows:

1. A Web application consists of various components. On the server, it can be deployed as a Web component which handles the function of communication over the Internet and a business component which performs the functions specific to the domain.
2. The data flow starts from the client when it sends a request to the server. The client request is converted to an XMLHttpRequest so that it can traverse through the Internet.
3. The XMLHttpRequest from the client is received by the Web server and the appropriate servlet is invoked. The Web server receives the client request and identifies the servlet to which this request has to be forwarded. The Web server does the required processing, if any, to forward the request to the servlet. It converts the XMLHttpRequest object into an HttpServletRequest object, so that it can invoke the appropriate servlet.
4. The servlets form the Web components of the application which generate dynamic Web pages as response to a client request. These servlets interact with the business components of the application and construct the response to the request.

JavaServer Pages (JSP), Servlets, and JavaServer Faces (JSF) can be used as the Web components. The difference among these three components is that, in JSPs, the Java code is embedded in HTML code. The JSPs generate the Web pages after receiving the response from Servlets. Servlets usually do not focus on the presentation, but on processing the client request and give appropriate response to the request. Servlets can also perform data formatting by embedding HTML code within Java code for formatting the Web page.

JavaServer Faces (JSF) are used to generate response Web pages. JSFs are components for UI design where the design is based on XML files known as view templates or facelet views.

5. The response to the client request is sent back to the client in the HTTP format. The client which is usually a browser interprets the response as a Web page and presents it to the client.

All these Web components are supported by a Web container. Web container provides multithreading, resource pooling, and other resource management functions to the Web components.

Figure 3.1 shows how client requests are handled in Web applications.

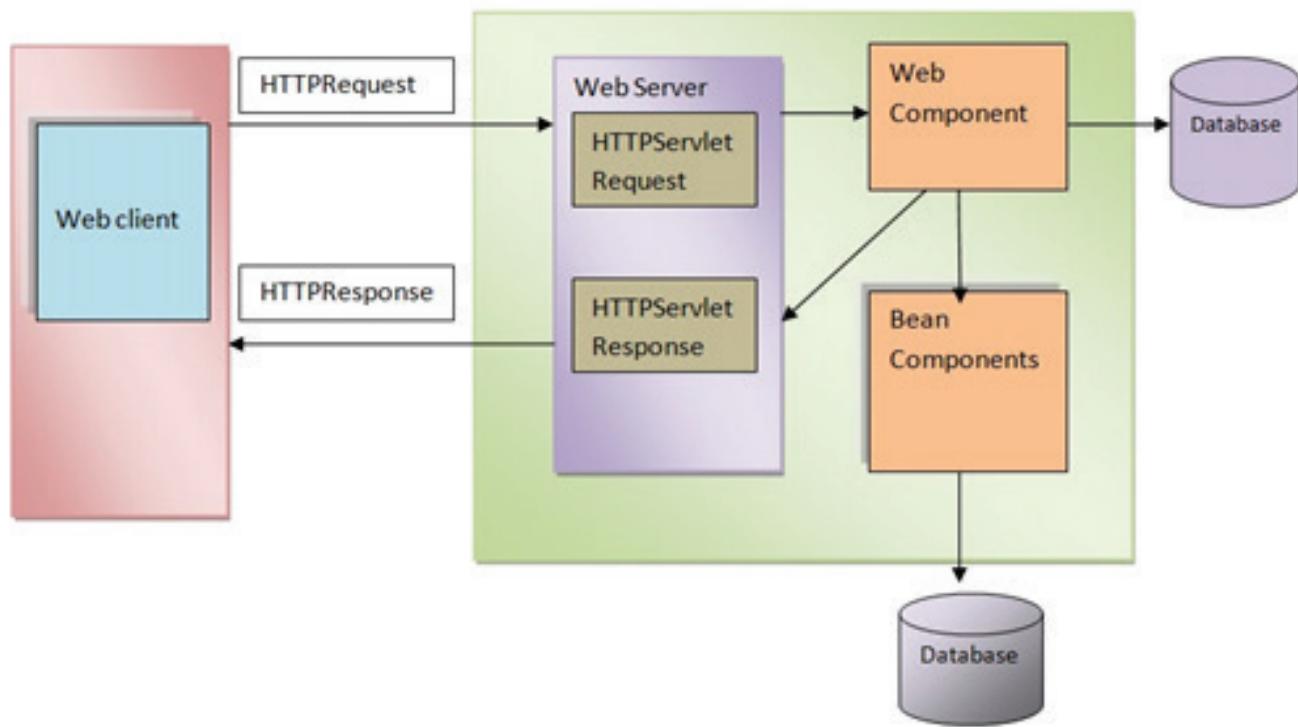


Figure 3.1: Handling Client Requests

The behavior of the Web application can also be configured during deployment through deployment descriptors. The configuration specifications can be given through annotations and maintained in an XML file. This XML file is known as deployment descriptor.

3.2 Web Application Life Cycle

While developing Web applications, the various components of the application need to be developed independently and should be integrated to work together as a single application. The steps for developing a Web application can be summarized as follows:

1. Decide the scope and requirements of the Web application scenario.
2. Decide on the architecture of the applications whether it will be a Two-tier (Client-Server) architecture, 3-tier, or n-tier.
3. Identify the layers and Web components required for developing the application including the database.
4. Create Unified Modelling Diagrams (UMLs) to identify the users, classes, and other components required for the application.
5. Develop the code for different components of the Web application.

Components of a Web application can be Web pages, CSS style sheets for the Web pages, Servlets, JSPs, JSFs, and so on. The developer has to write code for all these components. In order to decide upon the components required for an application, a design phase of the application should precede the coding phase.

6. Develop a Web application deployment descriptor.

A Web application deployment descriptor is an XML file, which has instructions about the deployment of the application. This file specifies the security settings required by the application, the configuration settings, and the container options. The deployment descriptor of a Web application is always a file named, web.xml which is located in WEB-INF subdirectory of the application. Any deployment tool will access the deployment descriptor and deploy the application according to the instructions in the configuration file.

7. Compile the Web application along with the helper classes required by the application.

Helper classes are also Java classes. While creating an application, the developer may create classes for independent functionalities. For instance, consider an example of a shopping portal where Shopping_Portal is the main class. In this application, processing the payment is an independent process, therefore, this task can be delegated to an independent object. The developer may create another class Payment_Process and create an object of this type and delegate the task of processing the payment to this object. In this case, the class Payment_Process is a helper class to the Shopping_Portal class.

Once all the classes are coded, they are compiled to generate respective class files.

8. After compilation is done, the application can be packaged into an archive file.

The Web applications can however be installed as open file structures also. In order to package it into an archive file, creating the deployment descriptor file becomes essential.

9. Deploy the application into a Web container.

The general interpretation of deployment is to locate all the components in their relative positions, ready to run. In case of Java Web applications, all the components are encapsulated into a container where the container provides the required platform services to the application.

10. Once the application is deployed, the application is accessible through the URL.

3.3 HTTP Request and Response

Hyper Text Transfer Protocol (HTTP) is a standard communication protocol for the Internet. It defines the process of communication between two communicating entities on the Internet. In Java Web applications, different components of the application communicate using HTTP.

The protocol functions according to request-response model. Initially, it was defined for client-server architecture where the client sends a request to the server to which, the server responds with appropriate HTML content. It uses the services of the other protocols such as Transmission Control Protocol (TCP), User Datagram Protocol (UDP), and so on.

HTTP identifies all the resources through Universal Resource Locators (URLs). Whenever a client requests for a resource or service, an HTTP session is established between the client and the server. HTTP session can be defined as a sequence of client requests and responses from the server between a client and server pair.

HTTP defines various methods for performing operations. Table 3.1 summarizes the methods used by HTTP.

Method	Description
GET	Retrieves data from a server, where the server is identified through an URL. Data sent using GET is visible in the URL.
PUT	Creates a resource or modifies an existing resource on the server. In this method, the URL is also required to identify the server.
POST	Submits certain values to the server, such as values from a Web form after submission of the form and so on. Data sent using POST is not visible in the URL.
DELETE	As the name suggests, DELETE method is used to delete resources.
TRACE	This method is used to identify the servers between the HTTP client and server. It is generally used to obtain routing statistics.
OPTIONS	This method can be used to check the functionalities that can be extended by the Web server for certain URLs.
CONNECT	This method is used to connect to the server. The connection may or may not be secure. It may be a TCP connection or UDP connection. All these properties are defined while creating a connection.
PATCH	Patch generally means a partial modification. For example, partial modifications to the HTTP server.

Table 3.1: HTTP Methods

The Web components of a Java application communicate through HTTP. Therefore, Java provides various classes for HTTP communication. Following are the classes used for HTTP communication in Java:

- **HttpURLConnection** - Handles all the HTTP connectivity related tasks in Java EE7
- **HttpServletRequest** – Creates an object of servlet request which uses HTTP
- **HttpServletResponse** – Creates an object of servlet response which also uses HTTP
- **HttpsURLConnection** – Used to make secure URL connections
- **HTTP.HttpRequest** – Used to create HTTP request
- **HttpResponse** - Used to create HTTP response

3.4 Introduction to Web Components

As mentioned earlier, a Web application has various components such as JavaServer Pages, JavaServer Faces, Servlets, and so on. The components are explained briefly in the following sections:

→ JavaServer Pages

JavaServer Pages enable the developers to create dynamic, interactive Web pages. It has a standard tag library, where the tags defined in the library carry out general functions required in most of the Web applications.

According to the MVC architecture, JSP can be used to define the view of the application along with Servlets and Java beans. Java beans can implement the model of the application and Servlets implement the controller.

→ Servlets

Servlets are server-side programs which extend the capabilities of the server to support processing of data. `javax.servlet` and `javax.servlet.http` are packages in Java which support servlet implementation. Servlets can access database, JSPs, and also user interface components. They communicate with various components of the Web application using HTTP as the communication protocol.

→ JavaBeans

Java beans are reusable software components which are coded according to JavaBeans API specification. Java beans should be serializable implementing the `Serializable` interface. They define getter and setter methods for various properties of the class. Java beans are designed to interoperate with legacy components of the Internet and provide components to the Web applications.

→ JavaServer Faces

JavaServer Faces are used to develop a component based user interface for Web applications which is compatible with XML files. It makes use of view templates written in XML. These view templates are known as Facelets. The specification is provided as an API with the Java EE. JSF can be seen as a combination of Struts and Swing.

The features provided by JSF for developing Web application user interface can be summarized as follows:

- Allows the developer to create user interface components from templates
- Supports JSP tags to access the user interface components
- Saves the state of the UI components transparently
- Capable of event handling and component rendering mechanisms, which makes it compatible with other markup languages, such as HTML

3.5 Introduction to Web Services

Web services are Web based programs through which a specific service is implemented. These programs can be accessed by end users through client programs, such as browsers or by other machines as part of a larger application. Web services also use Internet protocols similar to Web applications. However, the difference is that the Web services can be accessed by both end users and machines whereas Web applications are accessed through end users only.

Web services are Web based programs through which a specific service is implemented. They are similar to Web applications but not the same. Web applications are primarily accessed by users through browsers. Apart from users, other machines on the Internet can also access the application.

Consider an application of a shopping portal, where end users access the product list and place order for a product. Once the order is placed and card details are provided to complete the payment process, the interaction with the end user suspends and another thread of interaction with the payment server starts. Here, the shopping portal server accesses the payment gateway with credit card/debit card details. Since the shopping portal server is accessing the services of the payment gateway, the payment gateway is termed as a Web service.

Various standards and technologies are used for implementation of Web services such as SOAP, REST, XML, and so on.

Based on the standard used, there are two types of Web services:

1. **Simple Object Access Protocol (SOAP) Web services**

JAX-WS is the Java specification for implementing SOAP Web services.

2. **Representational State Transfer (REST) Web services**

JAX-RS is the Java specification for implementing REST Web services.

3.6 Creating a Simple Web Application Using Netbeans IDE

As mentioned earlier, a Web application has several components. The following demonstration shows how these different components work together.

In order to create the Web application in Netbeans IDE that accepts user input in a textbox and displays it on another page, perform the following steps:

1. Select File → New Project. The New Project dialog box is displayed as shown in figure 3.2.

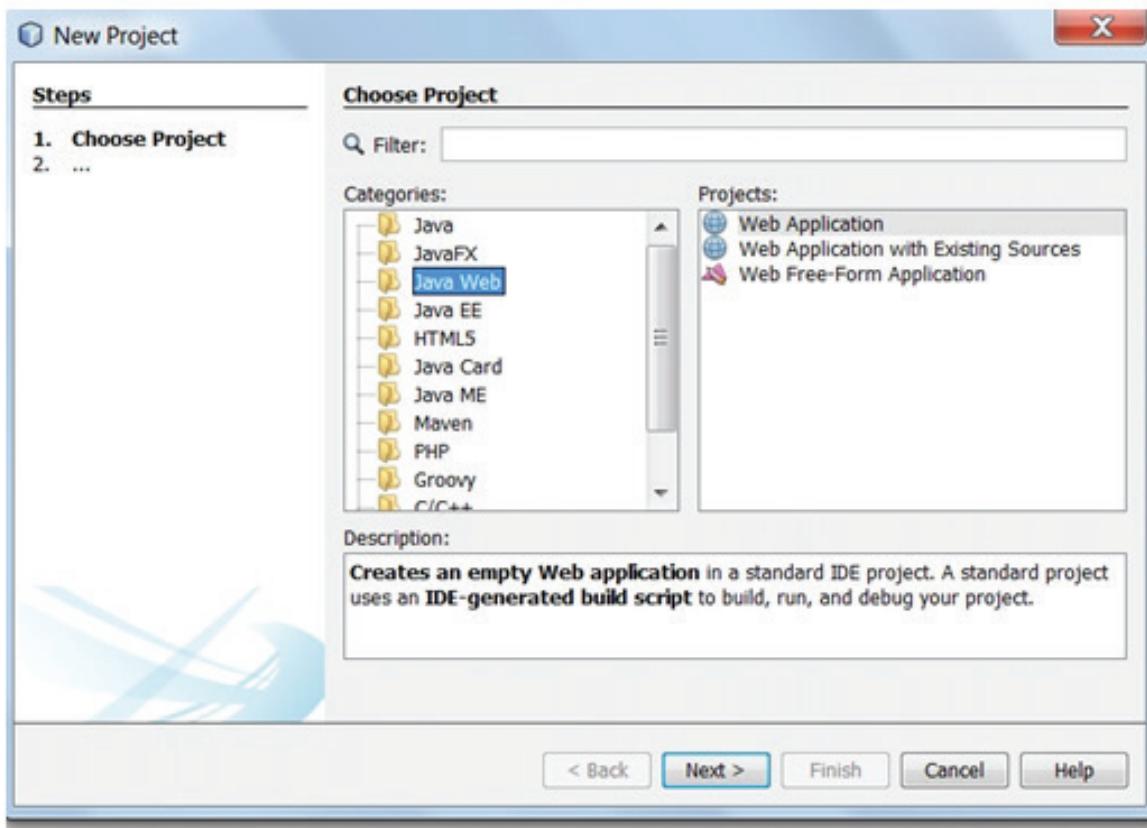


Figure 3.2: Creating a New Application

2. Select Java Web → Web Application and click Next. The New Web Application screen will be displayed prompting for a Web application name as shown in figure 3.3.
3. Specify an appropriate Web application name and click Next. The target folders for saving the project are specified by default as shown in figure 3.3.

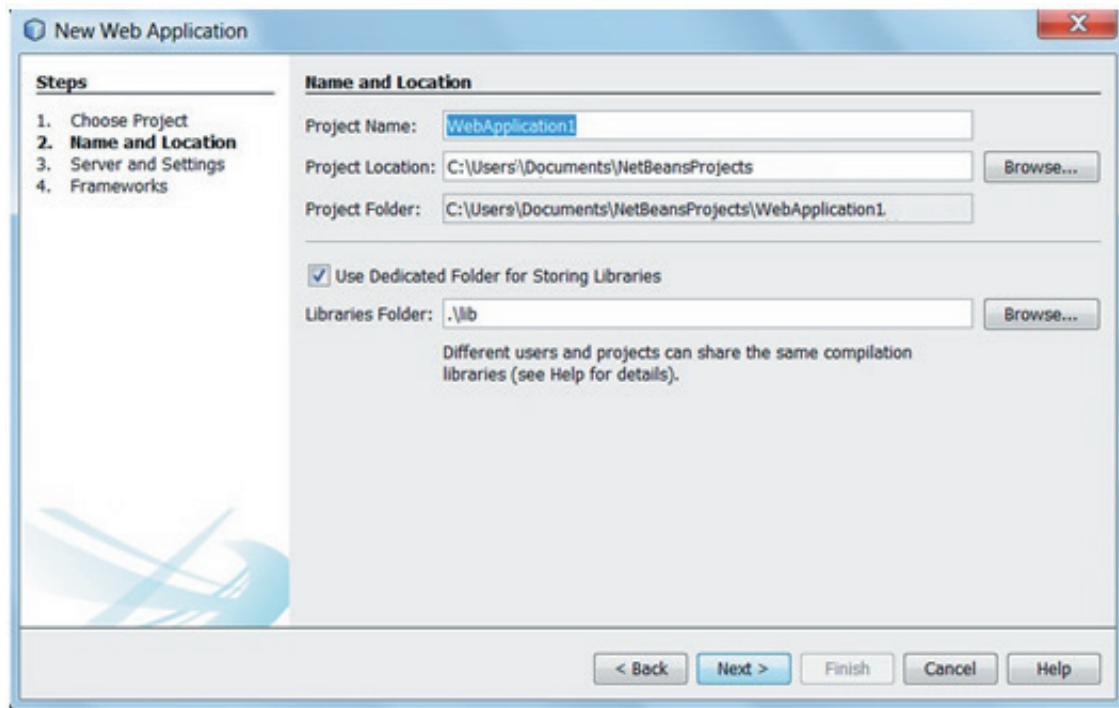


Figure 3.3: Naming the Web Application

It is recommended to select the Use Dedicated Folder for Storing Libraries checkbox in this window.

4. In the Server and Settings screen, select the Web server on which the application is supposed to be deployed and click Next. Figure 3.4 shows the list of servers registered with Netbeans displayed.

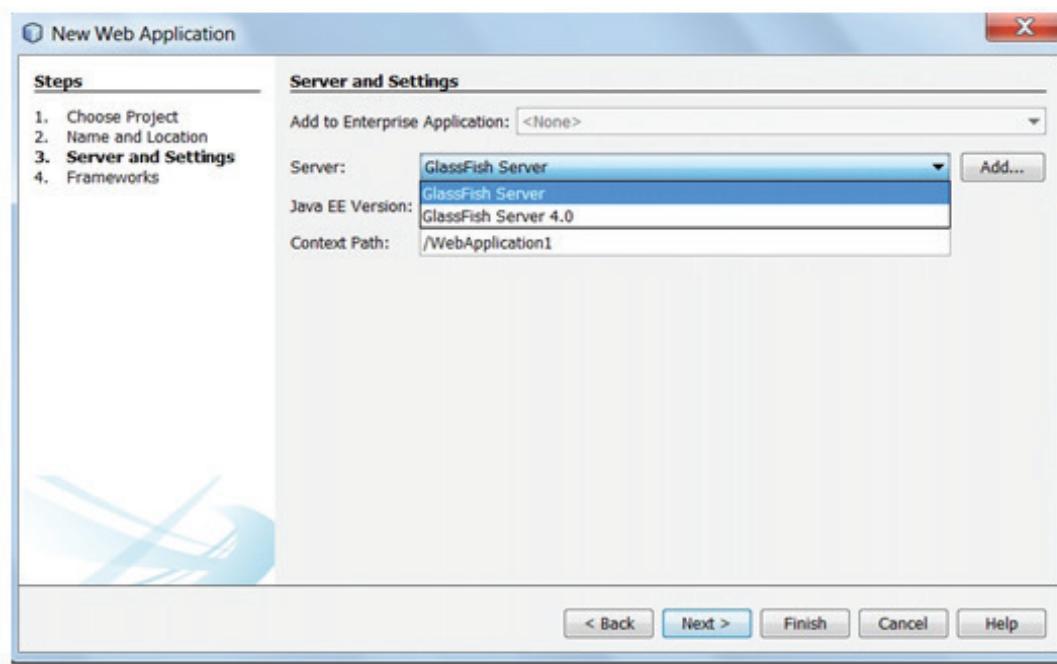


Figure 3.4: Selecting the Web Server

5. After selecting the Web server, the IDE prompts for any frameworks that will be used in the application. This application does not use any frameworks hence, click Next.
6. Click Finish, the Web application is created with folders pertaining to the project as shown in figure 3.5.

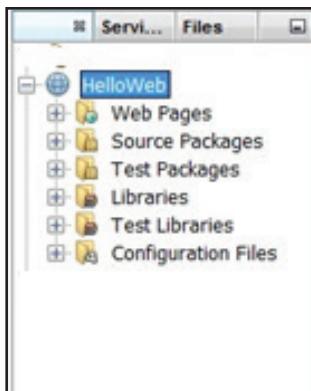


Figure 3.5: Project Folder

Among these folders, currently three folders namely, Source Packages, Web Pages, and Libraries are being used. The folder Source Packages comprises all the Java classes and beans pertaining to the application. The Web Pages folder comprises the JSPs, html pages, xml pages, and so on. The Libraries folder comprises the JDK and Glassfish server (or any other server used by the application).

The current 'HelloWeb' application has a bean, a user interface, and a jsp page. The bean of the application is created first. The bean defines the actual function of the application. In this case, the bean is substituting the name accepted from the user.

7. Right-click Source Packages and add a new package named, hello by selecting New Package from the menu. Add a new Java class to the package. Right-click the hello package and select New → JavaClass. Name the class as TextInput.java.
8. Add a string variable to the class as follows:

```
String name;
```

9. Right-click the variable name and select Refactor → Encapsulate Fields from the menu. This selection will open a window as shown in figure 3.6.

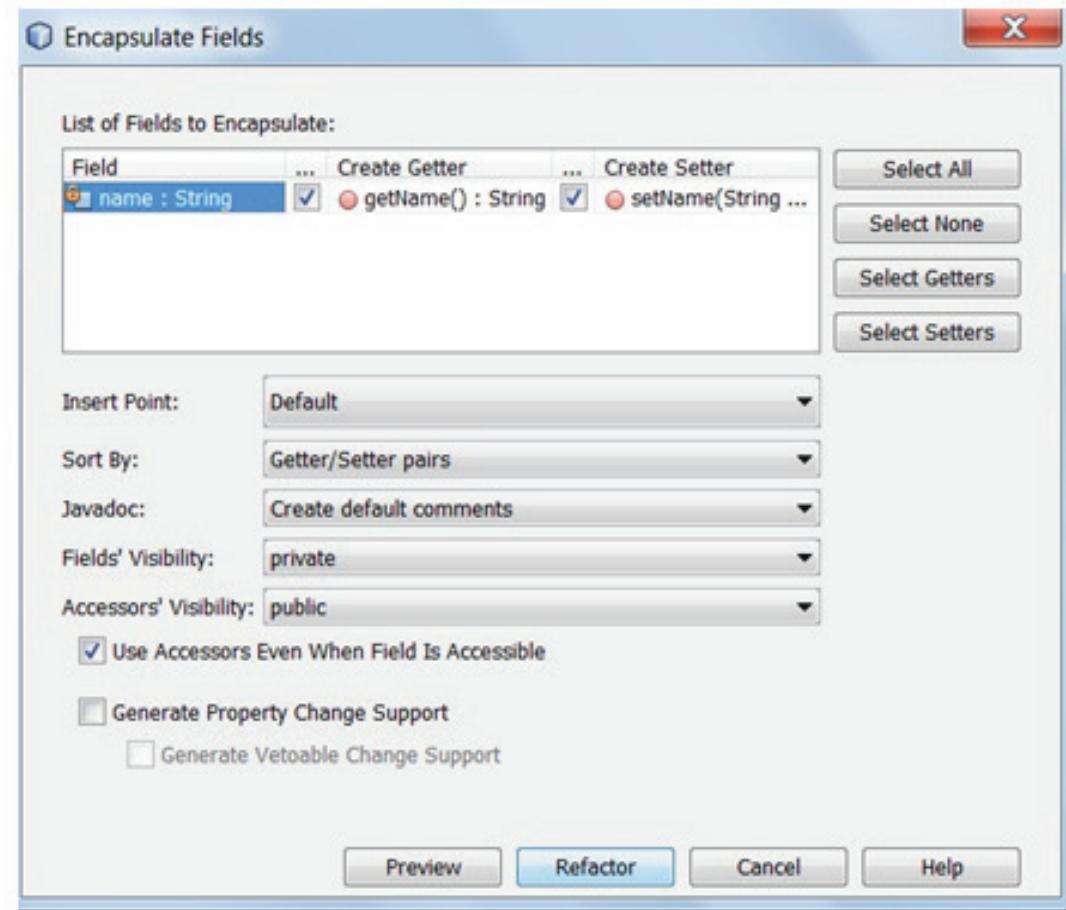


Figure 3.6: Creating Getter and Setter Methods for the Variable

10. Select the required checkboxes and click Refactor. The selected getter and setter methods will be created. After creating the getter and setter methods for the name variable, the code in the bean for the application will be as shown in Code Snippet 1.

Code Snippet 1:

```
public class TextInput {
    private String name;
    public TextInput() {
        name = null;
    }
    public String getName() {
        return username;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

This completes the bean creation process of the Web application. Next, the required Web pages need to be created.

The first Web page which is loaded when a Web application is loaded, is index.jsp.

11. Edit the index.jsp file which is present in the Web pages folder. Add GUI components to the Web page. NetBeans IDE provides a drag and drop mechanism for creating the user interface using a Palette as shown in figure 3.7. To open the Palette, click Window → Palette.

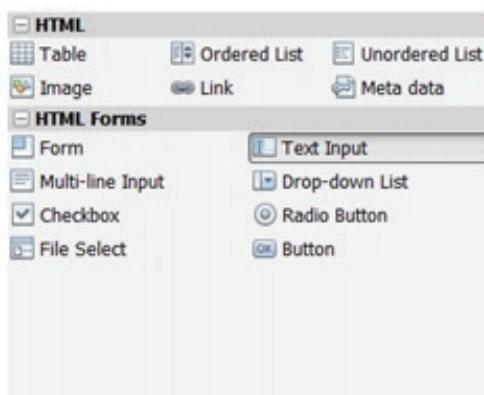


Figure 3.7: Creating the index.html File

12. Drag the Form component from the HTML Forms section of the Palette onto the body section of the html code. This will open a dialog box as shown in figure 3.8.

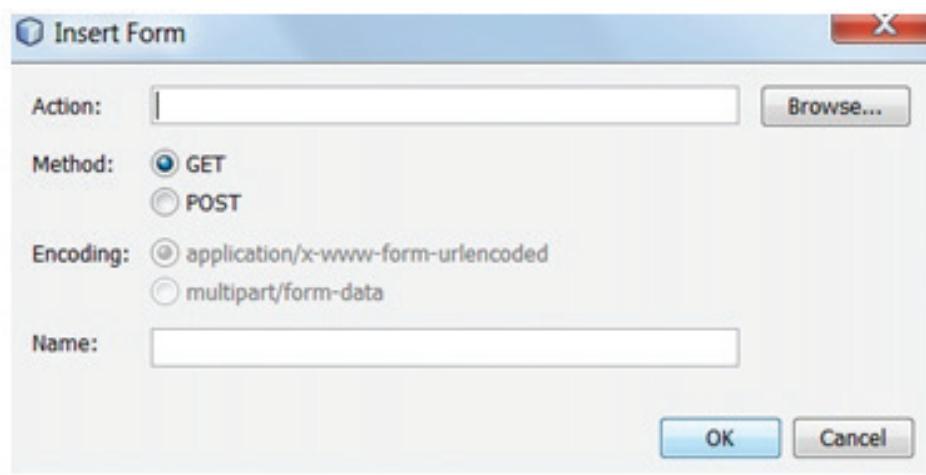


Figure 3.8: Form Component of User Interface

In the dialog box, populate the fields as follows:

Action Specify the name, response.jsp. This refers to the JavaServer page that will act in response to the actions taken in the Web page.

Method - Should be 'GET' in this case.

Name - Should refer to the name of the form, here it is named 'Input Name'.

13. Add components to the Web page by dragging and placing the UI components in the Web page. For the given example, add a text box and a button control.

Code Snippet 2 shows the code inside the index.jsp file after these actions are performed.

Code Snippet 2:

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>

    <body>
        <form name="Input Name Form" action="http://localhost:8080/HelloWeb/response.jsp">
            Enter your name:
            <input type="text" name="name" value="" />
            <input type="submit" value="ok" />
        </form>
    </body>
</html>
```

Next, the jsp file is created which acts as a response to the action taken in the index page. Since the action value specified in the index.jsp page was response.jsp, the response.jsp file needs to be created.

14. Right-click the Web Pages folder and create a new JSP file by selecting New → JSP from the menu. Name of the JSP file should be same as the one mentioned in the index.jsp file. Code Snippet 3 shows the code which is automatically generated by the IDE on creating the JSP file.

Code Snippet 3:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <h1>Hello World!</h1>
    </body>
</html>
```

Drag and drop the Use Bean component from the JSP section of the Palette onto the body section of the html code. This component specifies the bean to be used by the Web page. The Insert Use Bean dialog box is displayed as shown in figure 3.9.

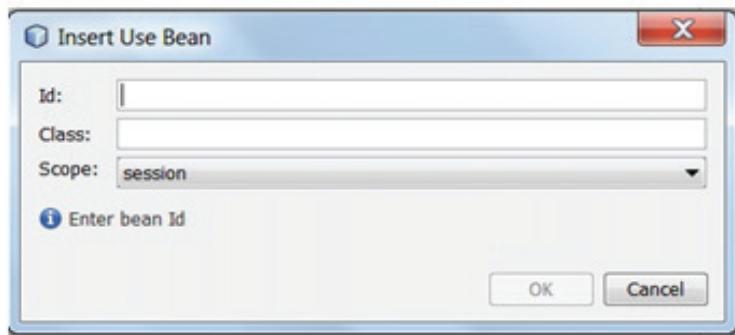


Figure 3.9: Use Bean Component

Specify a name for the Use Bean in the Id field. Here, the name mybean is given to the bean. In the Class field, specify the location of the bean as hello.TextInput.

Select the Scope of the bean as session.

With the Use Bean component, you have now specified which bean is to be used by the application. The variables of the bean need to be set next, for this the Set Bean Property is used. Drag and drop Set Bean Property from the JSP section of the Palette and place it after the Use Bean statement.

Use Get Bean Property to retrieve the value of the variable. Complete the code such that the final code is as shown in Code Snippet 4.

Code Snippet 4:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <jsp:useBean id="mybean" scope="session" class="hello.TextInput" />
        <jsp:setProperty name="mybean" property="name" value="<%=
request.getParameter("name")%>" />
        <h1>Hello, <jsp:getProperty name="mybean" property="name" /> !</h1>
    </body>
</html>
```

Here, the code within `<%= ... %>` braces is a scriptlet used to embed Java code in a JSP page.

Right-click the project and select Clean and Build. This will generate a .war file in the dist folder of the application. The Web application can be run after deployment. To run the application, click the Run icon on the standard toolbar.

The final output of the application is a Web page as shown in figure 3.11. Enter the text Good Morning in the text box and click ok. Figure 3.10 displays the output of the index.jsp page.

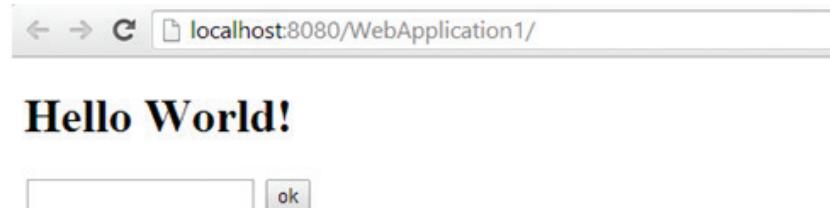


Figure 3.10: Output

Figure 3.10 shows the index.jsp page with the textbox and the ok button.

Specify the username as John in the textbox and click ok. Figure 3.11 shows the response.jsp page with the username displayed.

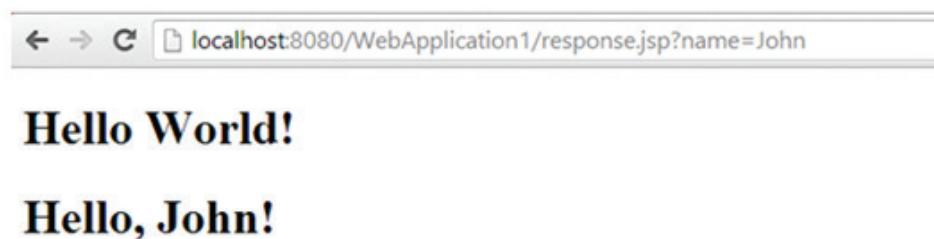


Figure 3.11: Username Displayed on the response.jsp Page

As can be seen, the username, John is displayed on the response.jsp page. Also, the value John is visible in the URL because the method used in the form tag was GET.

3.7 Check Your Progress

1. Which of the following components will initiate data flow in a Web application?

(A)	Deployment descriptor	(C)	Form data validator
(B)	Index.html	(D)	Master servlet

2. Which of the following statements about Web applications accessing a database are true?

a.	A Web application can access only one database.
b.	A Web application can access any number of databases.
c.	A Web application must access a database.
d.	A Web application and the database can be on the same tier.

(A)	a, b	(C)	c, d
(B)	b, c	(D)	a, d

3. Match the following components with the corresponding functionality.

	Component		Functionality
a.	JSP	1.	Component based user interface
b.	JSF	2.	Extend server capability
c.	JavaBeans	3.	Dynamic Web pages
d.	Servlets	4.	Reusable software components

(A)	a-1, b-4, c-3, d-2	(C)	a-3, b-1, c-4, d-2
(B)	a-2, b-3, c-4, d-2	(D)	a-4, b-2, c-2, d-1

4. Which of the following is not a valid HTTP method?

(A)	PUT	(C)	PROMPT
(B)	POST	(D)	PATCH

5. Which of the following statements about HTTP are true?

a.	HTTP uses services of only UDP.
b.	All the resources in HTTP are identified through URLs.
c.	OPTIONS method is used for routing statistics.
d.	All of these.

(A)	c	(C)	a
(B)	b	(D)	d

3.7.1 Answers

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

Summary

- Web applications comprise components such as Servlets, JSPs, static HTML pages, classes, and resources such as databases working in collaboration with each other to implement certain functionality of a particular domain.
- HTTP protocol is used by the Web application components to communicate over the Internet. It has a predefined set of messages used for communication over the Internet.
- Deployment descriptor is an XML file which comprises annotations. It is used for application deployment on the Web server.
- Helper classes are also Java classes created for independent functionalities while developing an application.
- JavaServer Pages enable the developers to create dynamic, interactive Web pages.
- Servlets are server-side programs which extend the capabilities of the server to support processing of data
- JavaServer Faces is a component based UI development technology which makes use of view templates written in XML. These view templates are known as Facelets.
- Web services are developed based on two standards – Simple Object Access Protocol (SOAP) and Representable State Transfer (REST).

GROWTH
RESEARCH
OBSERVATION
UPDATES
PARTICIPATION



Session - 4

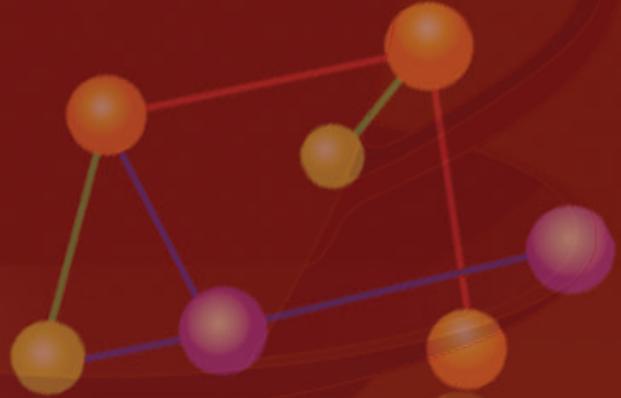
Application Resources

Welcome to the Session, **Application Resources**.

A Java application can have various resources or components in an application such as databases, mail service providers, images, and so on. This session describes how the resources are created and introduced in the context of the application. It also describes the APIs provided by Java EE 7 to support resource related functions.

In this Session, you will learn to:

- ➔ Describe the different types of resources
- ➔ Describe the different methods through which these resources are introduced in the application
- ➔ Describe Java Naming Directory Interface (JNDI)
- ➔ Describe the usage of dependency injection and resource injection
- ➔ Explain packaging of enterprise applications
- ➔ Explain packaging of Web applications
- ➔ Describe Context Dependency Injection (CDI) in Java EE 7



4.1 Resource Creation

Resources are those components of the application which can be accessed by the core application code. Applications can have resources such as databases, mail services, and so on. A Java Web or enterprise application, being a distributed environment, these components may be placed at different locations.

In order to locate these components, Java provides a naming and directory service called Java Naming and Directory Interface (JNDI) which is used to identify and locate objects. JNDI binds each resource object with a name.

4.1.1 Java Naming Directory Interface (JNDI)

JNDI is an integral part of the Java platform and it binds the resources in the applications with certain names as defined in the application. JNDI organizes the name of an object into a hierarchy, for example, com.mypackage.hello.Calc.JNDI binds the name of an object with an object or a reference of an object in the directory.

JNDI defines an initial context which specifies where to look up for objects. This context is set using the class Context and InitialContext. Code Snippet 1 shows a skeletal method of defining the context of an application.

Code Snippet 1:

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
.....
//In main function
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.fscontext.RefFSContextFactory");

Context c = new InitialContext(env);
.....
```

In Code Snippet 1, a Hash table object is created. A hash table object stores a pair of values in the form of (key, value). In the context of naming service, the initial context of the naming service is being stored.

Initial context implies the location where the object and their reference mapping are stored. env.put() method is used for creating the initial context by obtaining it from the naming service, that is, com.sun.jndi.fscontext.RefFSContextFactory. Once the initial context is set, all the objects pertaining to the application and relative to the context are stored.

Context class has the following methods to be used by the naming services:

- ➔ **bind()** – which binds a string or name to the object
- ➔ **lookup()** – which locates the object based on a string parameter

- **unbind()** – which removes the binding between the name and the object

Resources can be added to the application either by creating a binding in the JNDI namespace or through resource injection as specified in annotations.

Note - Annotations are metadata which are included at various locations in the code. It is not part of the code, but it is useful for providing information to the compiler during runtime and deployment processing.

4.1.2 Databases as Resources in Applications

Database is the most commonly used resource in Java applications. Relational databases are one of the widely used databases where data is stored in a tabular format. There is a huge difference in the way data is handled in Java applications and databases. Java provides APIs such as Java Data Objects (JDO) and Java Database Connectivity (JDBC) to access and manipulate databases.

In JDBC, a database is accessed through DataSource objects. Each DataSource object has various properties such as location of the database, user credentials for accessing the database, protocol used to communicate with the database, and so on. It accesses the database through a Connection object. A DataSource object can also read data from a file other than the database.

In order to deploy a DataSource object in the application, it has to be registered with naming service such as JNDI or a service which is used by the application.

Following is the summary of creating and binding a DataSource object with a database:

1. An instance of DataSource class is created. The DataSource object is created along with the respective driver of the database. For example, to use the SQLServer database, write the following code:

```
com.microsoft.sqlserver.BasicDataSource ds =
new com.microsoft.sqlserver.BasicDataSource();
```

In the given statement, com.microsoft.sqlserver refers to the database driver pertaining to the SQL server and DataSource object is created to access SQL server database.

2. The DataSource object ds is set with properties such as database server name, database name, and other similar properties specific to the application as follows:

```
ds.setServerName("Europa");
ds.setDatabaseName("Student");
ds.setDescription("Student information for university");
```

3. This DataSource object is bound with a JNDI name using the following statements:

```
Context c = new InitialContext();
c.bind("jndiTestDB", ds);
```

First, the context is created and an object of class Context is instantiated. The bind() method invoked along with the DataSource object which binds the JNDI name with the corresponding database object.

4. The DataSource object is deployed with the application.

4.1.3 Connections as Resources in Applications

DataSource objects are accessed by establishing a connection between the application and the database. Connection objects are created for this purpose. There is a variant to the Connection object known as PooledConnection.

A Connection object is created by instantiating it and then, destroyed after the connection is closed with the database. PooledConnection object is not destroyed but it is returned to a pool of connections. Whenever a database connection instance is required, it is retrieved from the existing objects in this pool of Connection objects. This improves the efficiency of handling database connections.

Apart from the DataSource objects, database connectivity and access to database can also be accomplished through Persistence API used in enterprise applications. Persistence API provides the object-relational mapping for data in a Java application.

4.2 Injection

The components in a Java application are distributed and the application code uses various resources while implementing the business logic of the application. All the resources required by the application are not instantiated at compile time but an annotation is added to the code which indicates the requirement of a resource for application execution. These resource requirements and dependencies are not hard coded in the application leading to a loosely coupled system. Based on the annotations, the references and instances are generated only during runtime. This process of injection of resources or dependencies into the application is managed by an API such as Context Dependency Injection (CDI) which is responsible to map the resources and obtain appropriate objects for execution of the application.

4.2.1 Resource Injection

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

A Resource annotation has five elements: name, type, authenticationType, shareable, mappedName, and description.

- name field stores 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 to 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, that is, the method name should be prefixed with 'set'.

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

4.2.2 Dependency Injection

An application is designed with different classes whose objects are instantiated as per requirement and collaborate together. There are dependencies among various objects while executing the application. Dependency injection resolves these dependencies at runtime. It determines the lifecycle of the objects in the application.

Dependency injection is implemented with @Inject annotation. Dependency injection is type safe as opposed to resource injection.

→ **Using Web Services with Dependency Injection**

The javax.xml.ws.WebServiceRef annotation is used to define dependency injection in Web services. Apart from the five elements defined for @Resource annotation, the WebServiceRef annotation has an additional element called Wsdllocation(where WSDL is Web Services Description Language).

4.3 Context Dependency Injection

Context Dependency Injection (CDI) is a service which is provided by Java platform to provide dependency injection. CDI is used by both enterprise and Web applications.

CDI can be termed as an integration library which connects the JavaServer Faces (JSF) components and managed beans of the application.

It provides service to bind contexts to the components in the application. Based on the context, the objects satisfying various dependencies are injected into the application. Annotations prompt the inject points in the application. @Inject is an annotation which identifies the inject point in the application.

Following are the features of CDI:

- Dependency injection is type safe as it does not resolve the dependencies based on the name of the object

- CDI is integrated with Expression Language (EL) which enables direct access of components from JavaServer Faces and JavaServer Pages
- It enables integration of third party service with the Java environment
- It allows loosely coupled components in the application
- CDI provides for an event-notification model
- A Web conversation scope is defined in CDI apart from session, request, and application scope

Note - A Web conversation scope refers to the scope where state of the session is maintained and can be retrieved again when the conversation restarts with the JSF application.

4.4 Packaging Applications

Java applications are packaged as archive files. Standalone Java applications are packaged as Java Archive (JAR), Web applications as Web Archive (WAR), and Enterprise applications are packaged as Enterprise Archive (EAR) files. They are deployable units of the application which contain the application components and deployment descriptors. The deployment descriptors may not be present as separate files; instead they can be included as annotations in the code.

The deployment descriptor is an .xml file. As deployment descriptor is a declarative file, it can be modified without changing the components of the application. The deployment descriptor is read at runtime and used to deploy the application. When an application has both annotations in the source and deployment descriptor, then the deployment descriptor has higher priority. The deployment descriptor overrides the annotations.

Figure 4.1 shows the structure of an EAR file.

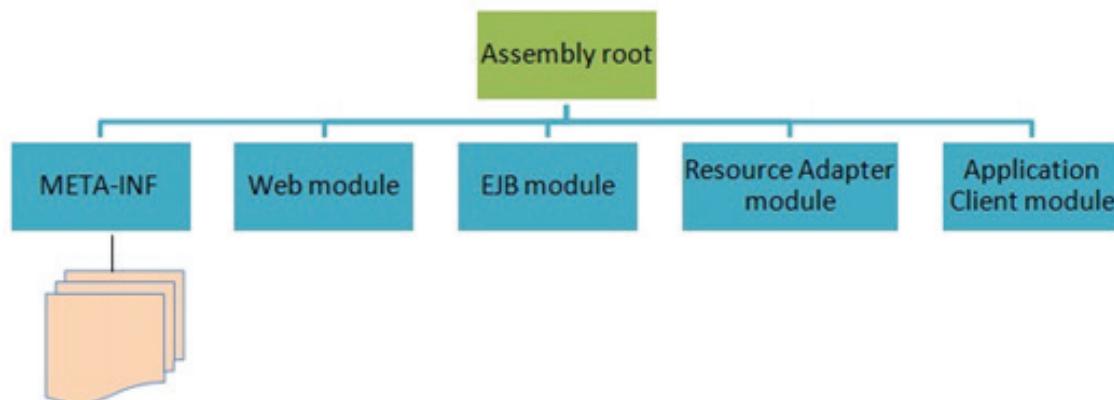


Figure 4.1: Structure of a Generic EAR File

The META-INF files comprise metadata pertaining to the deployment of the application. It can have deployment descriptors, library files, and so on.

Java applications need deployment descriptors of two types:

1. Java EE deployment descriptor
2. Runtime deployment descriptor

The Java EE deployment descriptor defines the configuration settings for deployment on any Java compliant environment.

The runtime deployment descriptor is used to define runtime specific configuration settings. For example, different servers accommodate the assembly root directory at different locations of the file structure, such specific details are configured using runtime deployment descriptor.

Java EE modules are deployed in containers. Each container has several components of the application and each component in turn has its own deployment descriptor. The Java EE module can be deployed as a standalone unit and the deployment descriptor of the Java EE module describes the security mechanisms, transaction handling, and so on of the application. There are four types of modules deployed in the Java EE application:

1. EJB module
2. Web module
3. Resource adapter module
4. Application client module

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 HTML files, JSPs, JSFs, 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 deployment descriptor.

Resource adapter modules comprise components which implement resource adaptation. In scenarios where the application is communicating with the 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 such as user forms, standalone classes, and so on for the application. These modules are packaged as .jar files.

4.4.1 Packaging Enterprise Beans

Enterprise beans are Java classes which implement the business logic of the enterprise application. They can be invoked by Web components or directly by application clients to perform certain tasks in the context of the application domain. They can access the database and perform operations on it based on the user input.

The EJBs are either packaged in JAR or WAR files (when they are used by Web components of a Web application).

→ Packaging EJBs as a JAR File

An enterprise application may have more than one bean component. Each bean module is packaged as a .jar file. All the .jar files together are packaged as an .ear (Enterprise Archive) file. Figure 4.2 shows the organization of different modules in an EAR file.

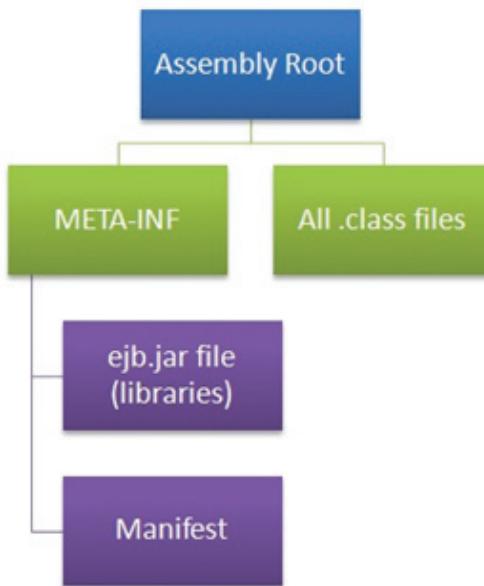


Figure 4.2: Structure of an EAR File

→ Packaging Enterprise Beans as WAR files

Web applications use beans to implement the business logic of the application. Since they operate on internal enterprise networks, therefore they require certain modules to make them compatible with the Internet. These modules are called Web components of the application.

The implementation of business logic of an application can be distributed between the Web components and Enterprise beans. The Web components access the bean components. The Web components can directly access the database or can access it through the bean components. The beans of the applications are packaged as .jar file, which together with other components of the Web application are packaged as .war (Web Archive) files.

The deployment of an application requires that all the components of the application follow certain standard location policy. This standard makes the deployment process simpler for any type of application.

The enterprise bean class files stored in a WAR module are placed in the WEB-INF/classes directory. If the bean components are provided as .jar files, then these .jar files are placed in the WEB-INF/lib directory of the WAR module.

The ejb-jar.xml file is a deployment descriptor which is not mandatory to be created for application deployment. However, if it exists, then it has to be placed in the WEB-INF directory.

4.4.2 Packaging Web Archives

Web archives are archive files for packaging Web applications. The smallest deployment unit in a Web application is a Web module. Web module comprises Web components such as static Web pages, images, and other resources required for the Web application. According to Java servlet specification, a Web module corresponds to a Web application.

Apart from Web pages and other resources, the Web application uses a set of utility classes both on the client-side and server-side. Similar to other archive files, Web modules also have a directory structure to comply with. This standard structure enables easy deployment of the application through any deployment tool.

Figure 4.3 shows the structure for arranging various components in a Web module.

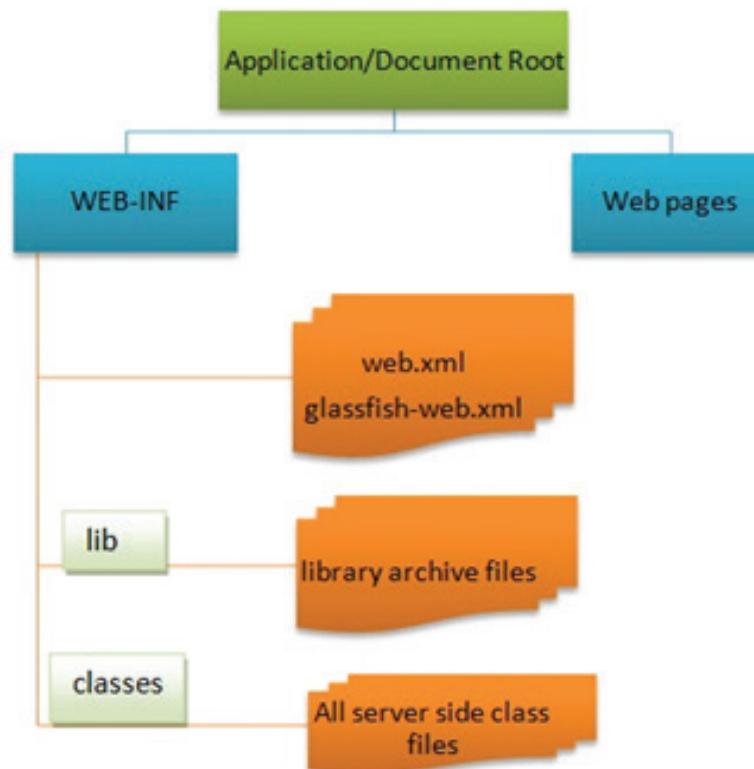


Figure 4.3: Structure of a Web Module

The left branch in figure 4.3 is also known as the document root which has the directories WEB-INF, lib, and classes.

Here, lib and classes are subdirectories of the directory WEB-INF. They comprise the following components:

- The classes directory includes all the bean classes, server side classes, servlets, and other utility classes.
- The lib directory contains all the beans and library files used in the application.

- The WEB-INF directory also has all the deployment descriptors: web.xml and ejb-web.xml.

The right branch of figure 4.3 contains all the Web pages that are presentable to the end user. The deployment descriptor web.xml is required if the application is using JavaServer Faces technology for creating the user interface. The deployment descriptor in that case has all the security specifications.

Application specific subdirectories can be created in the classes directory as per requirement.

It is not essential to package a Web application as a .war file for deployment. A runtime deployment descriptor can be created to deploy a Web application onto a Glassfish server. This runtime deployment descriptor maps the application specific resource names onto the Glassfish specific resource names. This deployment descriptor is saved as glassfish-web.xml and stored in the WEB-INF directory.

4.4.3 Packaging Resource Adapter Archives

Resource adapters are required in an application to make accessible the different types of resources used in an application. For instance, database driver is a resource adapter which enables the format change of data between the databases and applications. It is important to package these adapters to enable application deployment.

It can be packaged as a part of an enterprise archive file or as a separate .rar file.

A .rar file comprises a .jar file with the implementation classes of the resource adapter and an optional META-INF directory with the deployment descriptor of the resource adapter archive.

4.5 Check Your Progress

1. Which of the following is a naming service used by Java applications?

(A)	JNDI	(C)	XSL
(B)	CDI	(D)	Resource injection

2. Which of the following methods is used to obtain the object reference given the name of the object?

(A)	bind()	(C)	reference()
(B)	lookup()	(D)	None of these

3. Which of the following about PooledConnection object are true?

a.	Reusable connection object to datasource
b.	Connection objects are returned to a pool after completing the usage
c.	Used by application tier

(A)	a, b	(C)	b, c
(B)	a, c	(D)	None

4. Which of the given file is not a deployment descriptor?

(A)	web.xml	(C)	ejb.xml
(B)	ejb-jar.xml	(D)	None of these

5. Which of the following statements about JNDI are true?

a.	JNDI specifies an initial context for the location of application objects.
b.	bind() and lookup() are methods of the Context class used by JNDI.
c.	JNDI does not organize the objects in any form of hierarchy.
d.	All of these.

(A)	b, c	(C)	a, b
(B)	a, c	(D)	None of these

4.5.1 Answers

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

Summary

- An application has various components such as database, mail service providers, messaging services and so on. All these are collectively termed as resources in the application.
- Java does not instantiate the resources during compilation. In order to keep the application loosely coupled, the references to the resource instances are instantiated at runtime.
- Instead of injecting the resources into the application, annotations are added at places where the resource reference is required.
- Java implements dependency and resource injection through Context Dependency Injection (CDI) service.
- Enterprise applications are packaged as EAR files and Web applications are packaged as WAR files.
- The archive files may or may not have deployment descriptors. Each archive file has a specific structure. Different files are arranged in various directories of the archive files according to a standard structure.



To enhance your knowledge,
visit the **REFERENCES** page



Session - 5

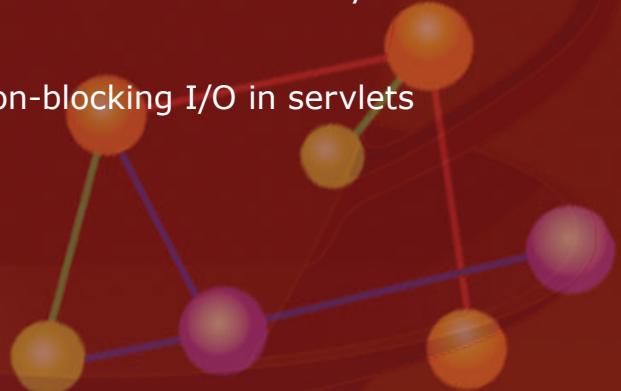
Java Servlets

Welcome to the Session, **Java Servlets**.

This session describes servlets and its lifecycle. It explains how various events in the lifecycle of a servlet are handled, the configuration of a servlet, writing service methods for servlets, how Web resources can be accessed, and finalizing of the servlet. This session also describes asynchronous processing in a servlet and implementation of non-blocking Input-Output (I/O) in servlets.

In this Session, you will learn to:

- Describe servlet and its lifecycle
- Describe how various events in the lifecycle of the servlet are handled
- Explain configuration of the Servlet
- Explain creating and handling HTTP requests and responses
- Describe writing service methods for Servlets
- Describe how Web resources can be accessed by the Servlet
- Explain finalization of a servlet and asynchronous processing in a servlet
- Describe implementing non-blocking I/O in servlets



5.1 What is a Servlet?

Servlet is a server-side component of a Java Web application, which receives request from the client and processes it. The client can be a JSP, HTML, or any other Web client. It receives the request over Internet through HTTP protocol. A servlet can access database, Java beans, and other resources of the application to process the client request. After processing the request, the servlet constructs the response for the request and returns it to the client through HTTP protocol on the Internet.

The implementation of servlets in Java is through Java servlet API which is contained in the Java package hierarchy, javax.servlet. The javax.servlet.http class is responsible for handling the HTTP related tasks of the Web application.

5.2 Lifecycle of a Servlet

The servlet of an application is invoked by the Web container in which it is placed. The Web container loads the servlet when it receives a request for the servlet. There are three phases in the lifecycle of the servlet.

1. Initialization
2. Service to the Web application
3. Destruction

→ Initialization

A servlet is initialized when a request for the servlet is received from the client. The javax.servlet interface consists of an init() method which initializes the servlet. The servlet is initialized after it is loaded by the container. The initialization can be done either after loading the servlet into the container or when the servlet receives a client request. The initialization of the servlet is done only once in the entire lifecycle of the servlet. After initializing, the servlet can serve any number of client requests before it is destroyed by the container.

→ Service

Once the servlet is initialized, the service provided by the servlet is defined in the service() method. The service method has two parameters of type ServletRequest and ServletResponse. After initialization, the servlet remains in the memory of the server. Whenever there is a client request for the servlet, the service() method of the servlet is invoked for processing and responding to the request.

→ Destruction

The servlet is unloaded from the memory by invoking the destroy() method. On invoking the destroy() method, the servlet is unloaded from the container and the memory allocated for it is deallocated. Both init() and destroy() methods are invoked only once throughout the lifecycle of the memory.

Certain classes in the application may require information regarding when the servlet is initialized or other events pertaining to the lifecycle of the servlet. These classes can create listener objects to keep track of the lifecycle events of a servlet. A developer can define these listener objects. In order to create listener objects, listener classes have to be defined by implementing the appropriate listener interface.

Following are the various listener objects provided by the Servlet API which can be invoked in response to events:

- **javax.servlet.AsyncListener** – This listener object is notified if there is a change of state of an asynchronous operation which is initiated by a ServletRequest object. The asynchronous event which is triggered due to a ServletRequest is of type javax.servlet.AsyncEvent.
- **javax.servlet.ServletContextListener** – This listener object is notified when a lifecycle event such as initialization of a servlet occurs. The corresponding event object is javax.servlet.ServletContextEvent.
- **javax.servlet.ContextAttributeListener** - This listener object is notified when there is a change in the attributes of the lifecycle events of the servlet. The corresponding event is javax.servlet.ContextAttributeEvent.
- **javax.servlet.ServletRequestListener** – This listener object is notified if the servlet receives a request or sends out a request from the Web application. The corresponding event is javax.servlet.ServletRequestEvent.
- **javax.servlet.ServletRequestAttributeListener** – This listener object is notified if there is a change in the attributes of the servlet requests. The corresponding event is javax.servlet.RequestAttributeEvent.
- **javax.servlet.http.HttpServletRequestListener** – This listener object is notified when there are HTTP requests being received from other components or when HTTP requests are sent out to other components of the application.
- **javax.servlet.http.HttpSessionBindingListener** – This listener object is notified when a certain object is bound or unbound to an HTTP session. The corresponding event is javax.servlet.http.HttpSessionBindingEvent.
- **javax.servlet.http.HttpSessionAttributeListener** – This listener object is notified when there is a change in the attributes of an HTTP session.
- **javax.servlet.http.HttpSessionActivationListener** – HTTP sessions can be passivated instead of removing them from the memory and again activated where there is a request for the servlet. When HTTP session activation occurs, this listener object is notified. The javax.servlet.http.HttpSessionEvent is an event object which is created whenever an event with respect to an HTTP session occurs.

5.2.1 Creating a Servlet in NetBeans IDE

To create a new Servlet, create a new package in the Web application by right-clicking the Source Packages folder and selecting New → Java Package. Next, right-click the package and select New → Servlet as shown in figure 5.1.

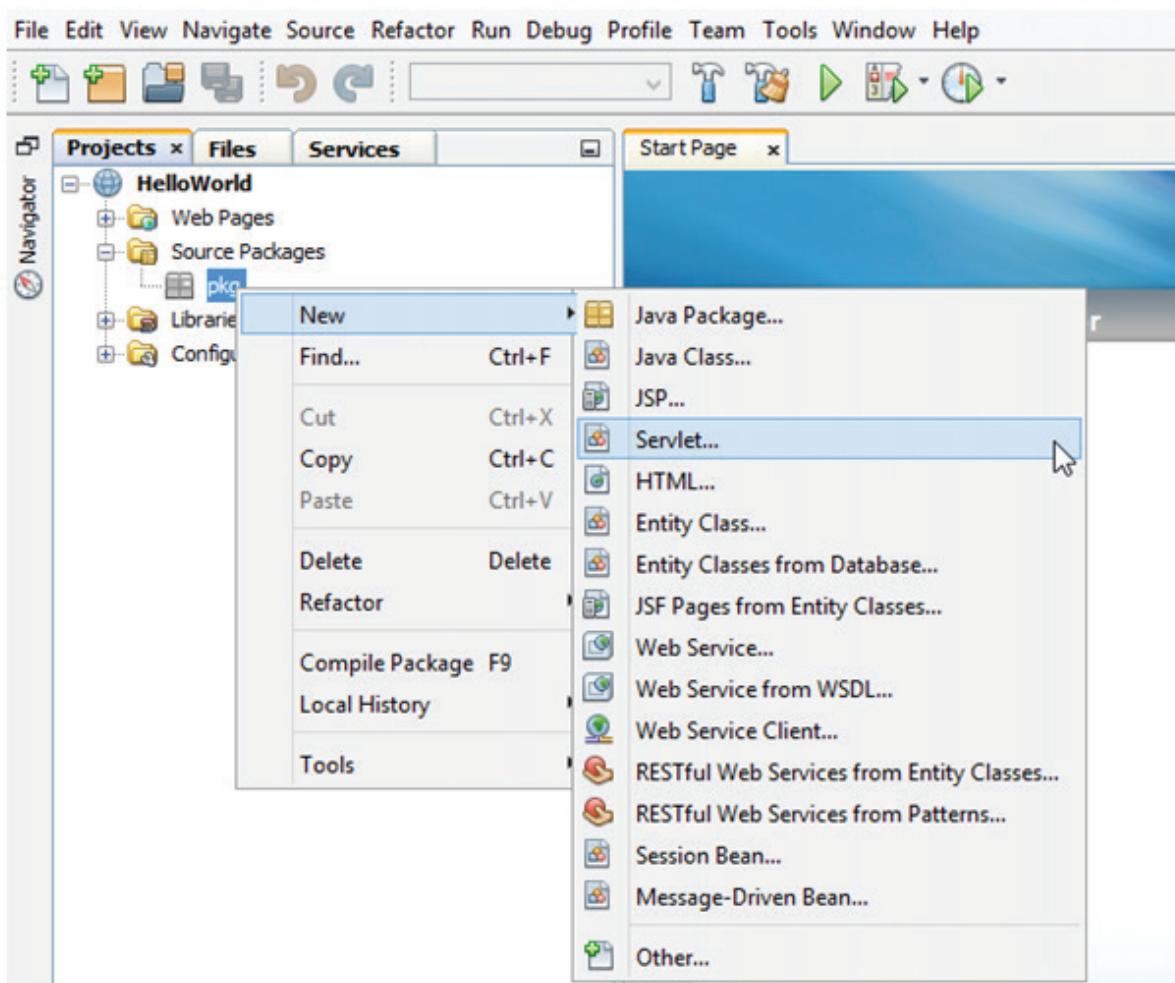


Figure 5.1: Creating a New Servlet

The web.xml file will be created automatically if the Add information to deployment descriptor (web.xml) checkbox is selected while creating a servlet as shown in figure 5.2.

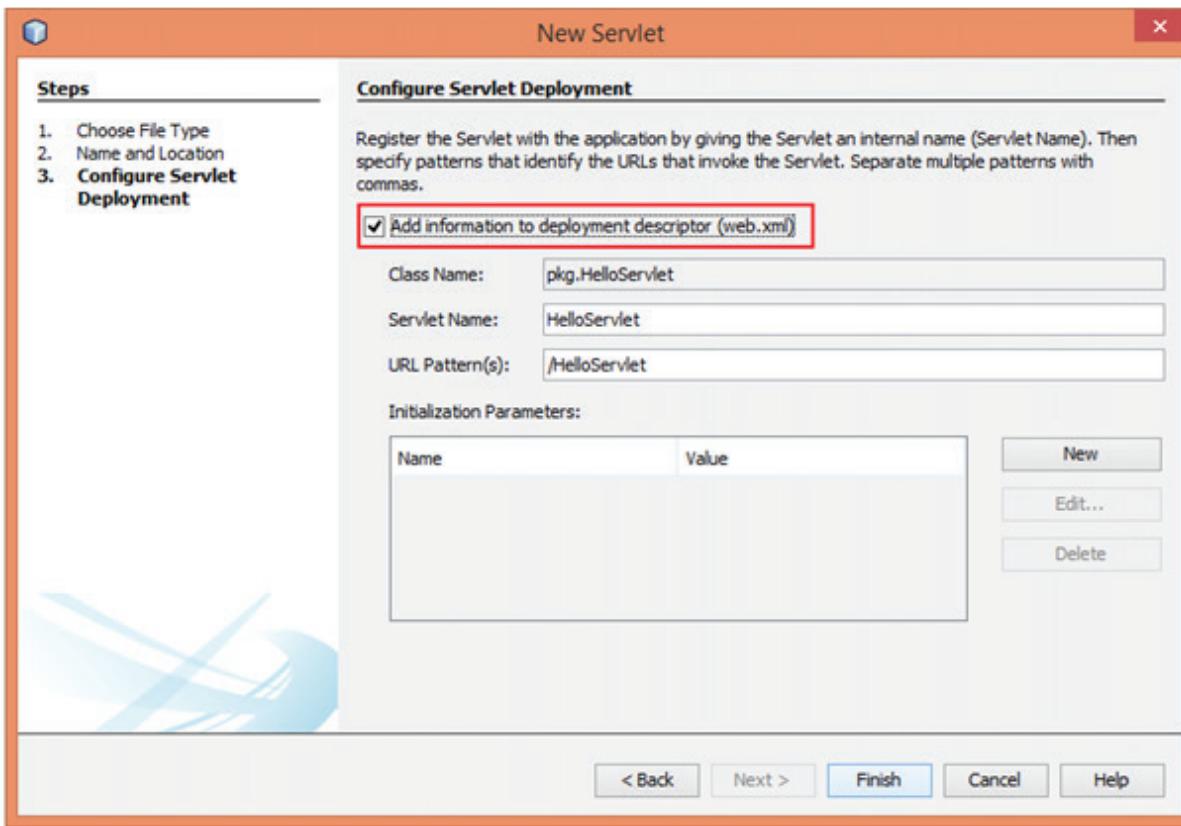


Figure 5.2: Selecting the Deployment Descriptor

When a Servlet is created, the container automatically generates some code as shown in Code Snippet 1.

Code Snippet 1:

```

import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(urlPatterns = {" /Servlet_Example"})

public class HelloServlet extends HttpServlet {
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html; charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            /* 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 First_Servlet</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Servlet First_Servlet at " + request.getContextPath() + "</h1>");
        out.println("</body>");
        out.println("</html>");
    }
}

@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    processRequest(request, response);
}
@Override
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    processRequest(request, response);
}

@Override
public String getServletInfo() {
    return "Short description";
} // </editor-fold>

}

```

Code Snippet 1 has methods such as `processRequest()`, `doGet()`, `doPost()`, and `getServletInfo()`.

The method `processRequest()` is used to write the processing code to process the incoming request. If the incoming request has to perform an HTTP operation, the `doGet()` and `doPost()` methods are invoked based on the type of request, that is GET or POST. These methods, in turn invoke the `processRequest()` method.

5.3 Servlet Context and Configuration

A servlet context defines various attributes of the servlet and the various resources that can be accessed by the servlet. The context of a servlet is defined by the class `javax.servlet.ServletContext` in Java servlet API. The `ServletContext` is initialized when the servlet is initialized and it sets the environment of the servlet.

When a servlet is created through NetBeans IDE, the context of the servlet is set during the deployment of the application. The deployment descriptor, web.xml is updated in order to set the context of the file.

The web.xml file can be accessed in the WEB-INF directory as shown in figure 5.3.

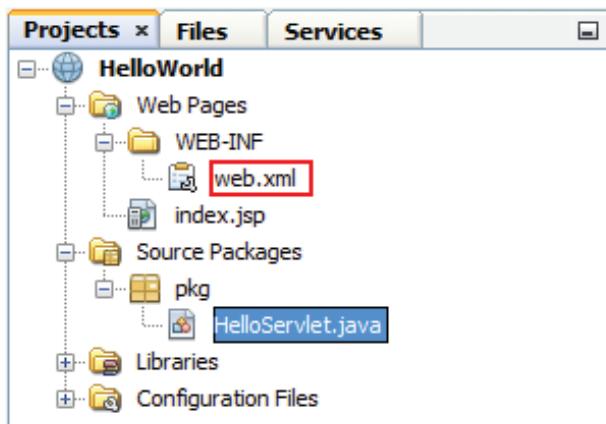


Figure 5.3: web.xml

The web.xml file contains the deployment configuration information of the Web application. Code Snippet 2 shows the default code in the deployment descriptor.

Code Snippet 2:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">

    <servlet>
        <servlet-name>HelloServlet</servlet-name>
        <servlet-class>HelloServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>HelloServlet</servlet-name>
        <url-pattern>/HelloServlet</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
</web-app>
```

Code Snippet 2 shows the default code in web.xml which is used for the deployment of the Web application, except for the highlighted code.

To add context parameters to the web.xml file, add the following tags to the deployment descriptor:

```
<context-param>
    <param-name>n1</param-name>
    <param-value>100</param-value>
</context-param>
<context-param>
    <param-name>n2</param-name>
    <param-value>200</param-value>
</context-param>
```

The code adds two parameters namely, n1 and n2 with their respective values. Code Snippet 3 shows how to access the context parameters in the servlet within the processRequest() method.

Code Snippet 3:

```
.....
ServletContext sc = getServletContext();
String para1 = sc.getInitParameter("n1");
String para2 = sc.getInitParameter("n2");
out.println("First parameter: "+para1);
out.println("Second parameter: "+para2);
.....
```

In Code Snippet 3, the ServletContext interface is used to fetch the context parameters using the getInitParameter() method. The import statement for javax.servlet.ServletContext must be added to the servlet. It may be required to add the @WebServlet annotation above the servlet class declaration, so that the servlet can be identified as follows:

```
@WebServlet (name="HelloServlet", urlPatterns={"/HelloServlet"})
public class HelloServlet extends HttpServlet {
.....
}
```

To execute the servlet code, right-click anywhere on the servlet and select Run File. Figure 5.4 shows the output of the application.



Figure 5.4: Output

The output of the servlet displays the values of the init parameters specified in the web.xml file and the output of the method `getContextPath()` where `HelloWorld` is the context path of the application. When the `ServletContext` is set for the Web application, it remains same for all the servlets of the application.

Every `ServletContext` object has an associated `ServletConfig` object. It has all the configuration details of the servlet generated by the container that are used while loading the servlet. The `ServletConfig` object provides the initialization parameters defined in `web.xml` to the `ServletContext` object. Following are the methods defined in the `ServletConfig` interface:

- `getInitParameter()` – This method accepts the initialization parameter name and returns the initialization parameter value.
- `getInitParameterNames()` – This method returns the names of all the initialization parameters.
- `getServletContext()` – This method returns the context in which the servlet is executing.
- `getServletName()` – This method returns the name of the servlet.

5.4 Request and Response Methods

When a servlet is loaded and initialized by the Web container, it is ready to accept and service requests from clients. Following are the steps in the request-response cycle:

1. Servlet receives a request
2. Accesses any external resources to service the request
3. Generates the response to the request

5.4.1 Request

The request can be a `ServletRequest` or `HttpServletRequest` object and as a response, the servlet may generate `ServletResponse` or `HttpServletResponse` object. All the non-HTTP requests implement the `ServletRequest` interface. The methods implemented by the interface access the following information:

- Parameters of the `ServletRequest` which are the medium of communication between the client and the server
- Object valued attributes are used for communication between container and servlet or between different components of the application
- Information regarding the protocol used to send the request to the servlet

The incoming request can be parsed manually using classes such as `BufferedInputStream`. All the HTTP servlet requests are passed to the servlet as `HttpServletRequest` object. Following is the format of an HTTP request:

`http:// [host] : [port] [request-path] ? [query-string]`

The host can be a local host or a remote host, port number for HTTP local host is 8080. The request path defines the context path, servlet path within the context, and other information pertaining to the path. The query string comprises the values which are passed as parameters to the servlet.

Following methods are used along with servlet requests to retrieve data pertaining to the servlet requests:

- **getContextPath()** – Returns the context path from the request
- **getServletPath()** – Returns the location of the servlet with respect to the context of the application
- **pathinfo()** – This method gives additional information about the servlet which is neither part of the context path nor the servlet path
- **getParameter Method**

The ServletRequest interface provides the getParameter method to fetch the request parameters on the servlet. Request parameters are extra information sent with the request. The syntax of the method is as follows:

```
String getParameter(String name)
```

where,

name – is a String specifying the name of the parameter

The method returns the value of a request parameter as a String or null if the parameter does not exist.

In case of HTTP servlets, the parameters are specified in the query string or posted form data.

This method should be used when the parameter has only one value. If the parameter is liable to have more than one value, use getParameterValues(java.lang.String).

→ RequestDispatcher

RequestDispatcher receives request from the client and forwards it to any resource (such as a servlet, HTML file, or JSP file) on the server. The RequestDispatcher object is created by the servlet container. It acts as a wrapper around a server resource located at a particular path or given by a particular name.

A RequestDispatcher object is used to forward a request to the resource or to include a resource in a response. The resource can be dynamic or static.

A relative pathname can be specified, however, it cannot extend outside the current servlet context.

If the path begins with a “/” it is interpreted as relative to the current context root.

It consists of two methods as follows:

- **forward**

```
void forward(ServletRequest request, ServletResponse response)
```

This method forwards a request from a servlet to another resource such as a servlet, JSP file, or HTML file on the server.

- **include**

```
void include(ServletRequest request, ServletResponse response)
```

This method is used to include the content of a resource such as a servlet, JSP page, or HTML file in the response.

The ServletRequest interface provides the getRequestDispatcher method to retrieve the RequestDispatcher object.

The syntax of the method is as follows:

```
RequestDispatcher getRequestDispatcher(String path)
```

where,

path – is a String specifying the pathname to the resource. If it is relative, it must be relative against the current servlet.

The method returns a RequestDispatcher object that acts as a wrapper for the resource located at the given path.

The difference between this method and ServletContext.getRequestDispatcher(java.lang.String) is that this method can take a relative path.

5.4.2 Response

A response is a communication from the servlet to the client. This response is built based on the input parameters received from the client which are later processed by the servlet. All the responses from the servlet implement the ServletResponse interface. In order to send data to the client, the interface defines appropriate methods.

Following are the steps involved in generating the response to the client requests:

1. The data to the client can be sent through an output stream. The output stream can be a PrintWriter stream for character data and for multimedia response the output is sent in Multimedia Internet Mail Extensions (MIME) format on a ServletOutputStream object.
2. In multimedia content, there can be different types of content such as text, html, pdf, and so on which can be included in the response. Methods in the ServletResponse interface are used to indicate the content type of the response.
3. The response of the request from the server may also require some buffer to be allocated at the receiving end, that is, the client. There are methods which can be used to communicate the requirement of the buffer at the client end.

4. In case of HTTP responses, status codes are also included in the response. In case of failure of generating response, a response status code is sent to the client. The important status codes are listed in table 5.1.

HTTP Status Code	Message
400 Bad Request	Syntax error
404 Not Found	The requested page could not be found on the server
408 Request Timeout	The client has waited for more than the time out period and the server did not respond during this time
412 Precondition Failed	The required pre-condition for the request is not fulfilled

Table 5.1: HTTP Status Codes

5. HTTP sessions may also require cookies to be set on the client-side. These cookies have the session information to be sent to the client from the server.

5.5 Filtering Requests and Responses

Filters are objects which can access, process, and transform the header information of servlet requests and responses. Filters encapsulate certain recurring tasks of the application such as formatting the data and apply them on the servlet requests and responses.

Following are some of the tasks performed by filters:

- Performing authentication on the headers and blocking the requests, if required
- Auditing the users of the applications and also, profiling the users who are accessing the application
- Compressing data
- Applying localization methods such as transforming the character set
- Interacting with external resources

Filters are defined on the requests and responses with the help of API filter. In order to define a filter, following steps are implemented using filter API:

1. Querying the request and response headers
2. Blocking the requests from propagating further based on the data in the header
3. Modifying the request and response headers and their data

5.5.1 Programming Filters

The filter API is implemented as a part of javax.servlet package and consists of the Filter, FilterChain, and FilterConfig interfaces. In order to define a filter, the Filter interface is implemented.

Classes implementing the Filter interface are annotated through @WebFilter annotation. The doFilter() method performs the actual function of the filter. Following are the actions performed by the doFilter() method:

- Examines the request headers
- Customizes or blocks the request
- Customizes or blocks the response of the servlet
- There might be multiple filters applied on a servlet request. In such a case, the next filter in the sequence is invoked after the current filter completes its processing
- The filter objects also have an init() and destroy() methods which are invoked when the service of the filter is required and when the service is terminated.

While using NetBeans, IDE Filter object can be created by adding a new file to the Web application context as shown in figure 5.5.

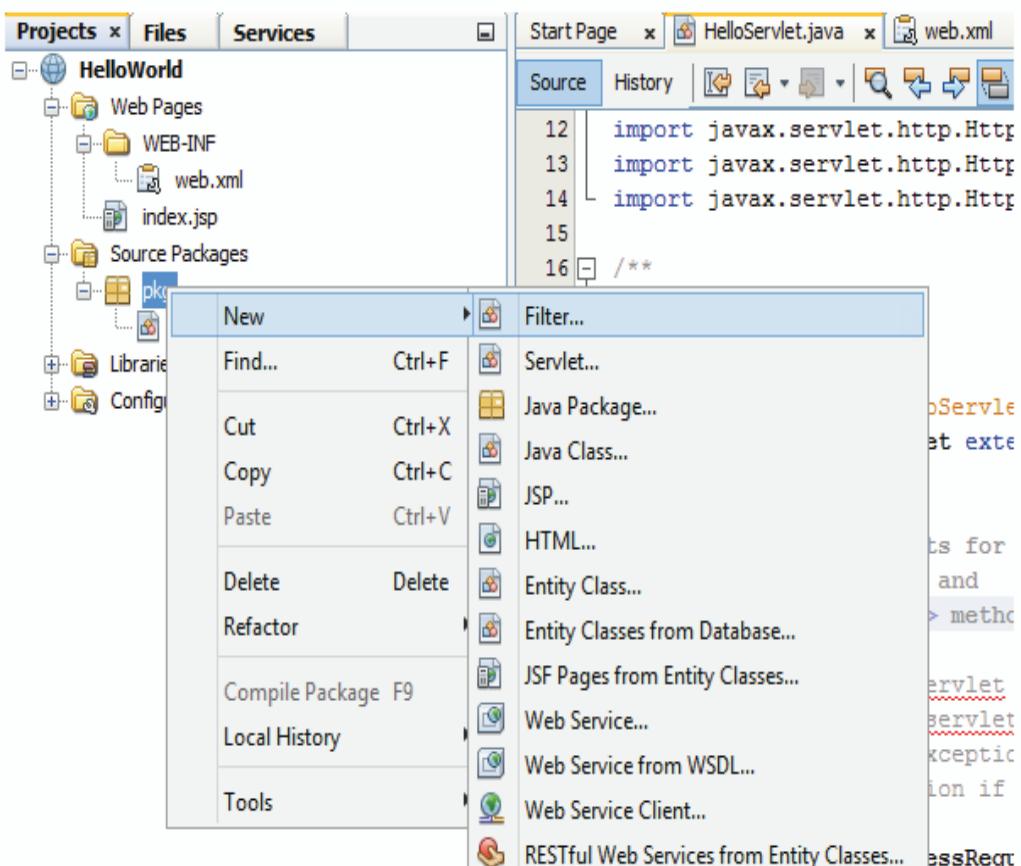


Figure 5.5: Creating a Filter in NetBeans

In case the Filter option is not visible in the menu, click Other, and then select Filter from the Web category.

The IDE generates all the default code required for the definition of the filter, so that it is compatible with the existing Web application context. The filter implementation code must be written in the doFilter() method.

5.5.2 Programming Customized Requests and Responses

While programming a filter, the filter may have to capture the request and modify the header. Also, the filter should not allow the request to propagate further. To achieve this, the filter passes an additional stream to the servlet and does not allow the servlet to close the original response stream. As the servlet generates the response, the filter modifies the servlet generated response as per the filter definition.

This additional stream is passed to the servlet by defining a wrapper to the server generated stream (which can be objects returned by getOutputStream or getWriter methods).

Other wrapper classes such as ServletRequestWrapper and HttpServletRequestWrapper are used to implement the filter actions on the request. Similarly, wrapper classes such as ServletResponseWrapper and HttpServletResponseWrapper are used for wrapping the responses.

5.5.3 Specifying Filter Mappings

All the filters defined in the Web application are not applicable to all the servlets or Web resources in the application. Therefore, certain mapping information specifying which filter to apply to which resource is essential. The filter mapping matches a filter to a Web component by name or by the URL pattern of the resource.

The filter mappings are done in the deployment descriptor of the application. In case of NetBeans, the mappings are done in web.xml. Otherwise, an XML file can be created to define the filter mappings, if they are to be done manually.

Code Snippet 4 shows an example of filter mapping done by name added to the web.xml.

Code Snippet 4:

```
.....
<filter>
    <filter-name>HelloServletFilter</filter-name>
    <filter-class>pkg.HelloServletFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>HelloServletFilter</filter-name>
    <servlet-name>HelloServlet</servlet-name>
</filter-mapping>
.....
```

The filter HelloServletFilter is applied to the HelloServlet servlet.

5.6 Maintaining a Client State

When an HTTP session is used in the Web application, a series of HTTP requests may be initiated from a single client and these requests are interrelated. The HTTP requests, in such a case are expected to save the state of the HTTP session. In order to support such applications where the HTTP servlets have to maintain the state, the servlet technology API provides various mechanisms for implementing and managing the sessions.

Every HTTP session with the servlet is represented by an HttpSession object. The getSession() method is used to return the session associated with the current servlet request. This method is also used to create an HTTP session if the current servlet request does not have an HTTP request associated with it.

5.6.1 Associating Objects with a Session

Every session is associated with different objects of the application. A session tracking mechanism is required which will map the objects of the session to their respective values. This kind of object mapping can be done with the help of object names. As there is no naming conflict within a Web context, no additional identifier is required to uniquely identify the objects.

Listener objects can be defined to keep track of object mappings when they change. Following are the listener objects which can be used for this purpose:

- javax.servlet.http.HttpSessionBindingListener interface which keeps track of the events of an object being bound to an HTTP session
- javax.servlet.http.HttpSessionActivationListener interface which keeps track of the event when an object associated with a session is activated or passivated

5.6.2 Session Management and Tracking

Every active HTTP session has certain server resources allocated to it. A session cannot block the resources indefinitely, if it is not accessing any service from the servlet. Therefore, every HTTP session is associated with a time out. The resources allocated to the session can be reclaimed after the time out period expires. While creating a new servlet, the default timeout is set to 30 in the <session-timeout> tag of the web.xml file.

To avoid session expiry due to time out, the session should be periodically accessed. This regular access of session resets the time out period, thus avoiding the session expiry. The time out interval can be accessed and set through getMaxInactiveInterval() and setMaxInactiveInterval() methods.

The session has to be invalidated on the server side after the purpose of the session is served. This can be done through invalidate() method.

In NetBeans IDE, session time out can be set in the web.xml file by modifying the <Session-timeout> value.

In order to track a session, every session is associated with a session identifier which is shared between both the client and the server of the session. The client can maintain the identifier in a session cookie or by sending the session id in the URL during every HTTP interaction. Session id in the URL is used only when cookies are disabled.

5.7 Finalizing a Servlet

The Web container which is responsible for loading and initializing the servlet is also responsible for removing the servlet from the Web container. The Web container removes the servlet from the memory by invoking the `destroy()` method. Before removing the servlet from memory, the Web container has to check on the following parameters of the servlet:

- All the service methods of the servlet should have returned calls, their execution should be complete
- Any pending service methods should be given certain grace period to complete the service requests before invoking the `destroy()` method
- Any service request that is pending must be identified even after the grace period
- The Web container must notify the long running service requests and wait until these threads complete their operation

In order to track the currently running service requests, the servlet class must maintain a counter. The counter is incremented when a service request is invoked and decremented, when a service request is complete. The counter reflects the number of requests the servlet is currently serving.

The servlet cannot make a clean shutdown until all the method requests are serviced. To ensure this, the Web container must notify all the active methods to shutdown. To notify the methods, the `setShuttingDown()` method is used.

The long running methods should either shutdown or gracefully exit to enable the shutdown of the servlet.

5.8 Developing and Deploying a Servlet Application

Creation of a new project in NetBeans creates various sub-directories which are used for purposes such as testing the application, holding the class files pertaining to the application, deployment descriptors to deploy the application, and so on.

The application can be developed by creating class files in the Source Packages folder. JavaScript pages, HTML pages, and other Internet related files are created in the Web Pages directory. Once all the directories and sub-directories of the class are created, the deployment descriptor, `web.xml` is defined in the application. The `web.xml` file is stored in the WEB-INF directory of the application.

By default, Java applications are deployed on the Glassfish server in the NetBeans IDE. There are other servers also, which can be used for the deployment of the application. The deployment configuration is retrieved from the web.xml file. To build the application, right-click the application and select Clean and Build. This will generate a .war file in the dist folder of the application. This .war file can then be deployed on the Web server for global access.

5.9 Asynchronous Processing

The servlet requests are serviced by generating one thread per request to create the response. In order to improve the efficiency of serving the applications, none of the threads of the application should be idle. Any processing thread can become idle in the following situations:

- The servlet thread is waiting for the allocation of some resource or for the process of generating the response to complete
- The thread is waiting for an external event to occur

The communication between the client and server can be asynchronous or synchronous. In case of synchronous communication, the client sends a request and waits for the server to respond. However, in case of asynchronous communication, the client sends a request and does not wait for the response.

Asynchronous Processing in Servlets

While executing servlets and filters in an application, if the execution reaches a blocking state, then the execution thread is handed over to an asynchronous context. After this, the thread returns without generating the response to the thread. The asynchronous context completes the blocking operation and generates the response to the context. If the asynchronous context cannot complete the request, then the thread is delegated to another servlet.

Asynchronous processing within the service methods is done through the functionality provided by the javax.servlet.AsyncContext interface. The startAsyncContext() method is invoked to generate an asynchronous context.

Following are some of the methods provided by the AsyncContext class:

- **void start(Runnable r)** – Through this method, the container provides a new thread instance to perform the operation which is blocking the application servlet from being removed from the container.
- **getRequest()** – It returns a ServletRequest object to return the request which has initiated the process of setting up an asynchronous context.
- **getResponse()** – This method returns a ServletResponse object. It can be used to generate the response to the servlet request from the asynchronous context.

- **complete()** – It completes the asynchronous operation and returns the response to the client which has requested the operation.
- **dispatch()** – It accepts the dispatch path as an argument and dispatches the requests and responses of asynchronous context.

To summarize the asynchronous processing mechanism, the blocking operations which are not allowing the servlet to be removed from the container are switched to an asynchronous context. A new thread object is created in the asynchronous context to handle the blocking operation. The run() method executes the new thread and the blocking operation. The complete() method is invoked to commit the process.

5.10 Non-blocking I/O

In order to improve the performance of Java EE applications, non-blocking I/O is used to process request and responses. Following steps summarize the implementation of non-blocking I/O for processing requests and writing responses:

4. Initiate an asynchronous mode for the request/response
5. For the request or response objects in the service methods, obtain corresponding input/output streams
6. Assign listeners to the input and output streams, a read listener to the input stream and a write listener to the output stream
7. Process the requests and responses in the listener callback methods

Following are the methods provided by the javax.servlet.ServletInputStream for non-blocking I/O:

- **void setReadListener(ReadListener r1)** – assigns a listener object to the input stream, which has callback methods that asynchronously read data from the input stream
- **isReady()** – returns a boolean value indicating that data can be read without blocking
- **isComplete()** – returns a boolean value to indicate completion in reading all the data in the stream

Following are the methods provided by javax.servlet.ServerOutputStream for non-blocking I/O:

- **void setWriteListener(WriteListener w)** – assigns a listener object to the output stream which has callback methods that asynchronously write data to the output stream
- **isReady()** – returns whether the write operation can be completed without blocking

5.11 Check Your Progress

1. Servlet filter mappings and initialization parameters are defined in the _____.

(A)	servlet class	(C)	deployment descriptor
(B)	servlet constructor	(D)	None of these

2. Which of the following event object notifies when certain object is bound with an HTTP session?

(A)	HttpSessionEvent	(C)	ServletContextEvent
(B)	HttpSessionBindingEvent	(D)	None of these

3. Which of the following functions is not performed by a filter?

(A)	Logging	(C)	Authentication
(B)	Auditing	(D)	All of these

4. Which of the following statements about session management are false?

(A)	A session id is associated with some sessions and cookies are associated with others.
(B)	If cookies are disabled, the session id is transmitted along with the URL in requests and responses.
(C)	Every valid session has a session id.
(D)	None of these.

5. Which of the following statements about finalizing a servlet are true?

a.	All the ongoing servlet request services should complete before destroying the servlet.
b.	The pending requests are given a time out period and then terminated.
c.	The pending requests are transferred to asynchronous context and executed within the same thread.
d.	None of these

(A)	a, c	(C)	a, b
(B)	b, d	(D)	c, d

5.11.1 Answers

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

Summary

- Servlets can process both HTTP and non-HTTP requests.
- A servlet is loaded by the container and initialized with init parameters. The servlet services the client requests and is removed by the container when the servlet is no longer requested by clients.
- Filters can be defined on the servlet requests and responses to perform tasks such as authentication, logging, and so on.
- Servlet API supports sessions and allows tracking of the sessions through different listener objects.
- A servlet, when removed from the container, must complete all the requests it is currently processing before releasing the memory space.
- Asynchronous processing of requests is implemented to avoid blocking of resources by the servlet.



Login to www.onlinevarsity.com

Session - 6

JavaServer Pages

Welcome to the Session, **JavaServer Pages**.

This session describes JavaServer Pages that are used to create dynamic Web pages. It also discusses various aspects of JavaServer Pages and explains how JavaServer Pages are written for applications.

In this Session, you will learn to:

- Describe JavaServer Pages (JSP)
- Compare it with other technologies
- Create JavaServer Pages
- Deploy JavaServer Pages in an application
- Understand how the servlets and JSPs interact
- Describe expression language used by JavaServer Pages



6.1 Introduction

JavaServer Pages are used to develop dynamic Web pages. It enables adding dynamic Web content to existing static Web content written in HTML. Dynamic Web pages are interactive with the ability to modify the content based on different parameters set for the Web page. For instance, identity of the user who has logged in is one parameter based on which the content to be displayed may change.

JSP comprises a tag library which contains various mark up elements similar to that of HTML tags. The difference between HTML and JSP tags is that JSP tags allow the server to modify the Web page through dynamic content. JSP also allows creation of custom tags. The final content seen by the user is a mixture of the static HTML content and the dynamic content generated by the JSP elements. JSP also allows accessing databases, Enterprise Java Beans, and generate HTML tags to present application specific data to the clients.

JSPs and Servlets both can perform similar functions. The JSP tags invoke the servlets which perform the required processing and return the response to the client Web page. The servlet may return the response as HTML tags or delegate the task to the JSP tags to generate the Web page. Generally, servlets do not focus on the presentation of the data whereas JSPs are mostly used for creating the presentation logic. Servlets are primarily used for request processing, which may in turn require access to database and bean components.

6.1.1 Need for JSP

JavaServer Pages are lightweight as compared to its counterpart Common Gateway Interface (CGI). For every client request, CGI would initiate a process on the server. This was not efficient as there was a lot of overhead on the server. Other programming technologies like servlets also generate dynamic Web content but the HTML content is delivered as part of the program. This merges the business logic and the presentation of the content, thus, increasing the program complexity. Also, the developer has to concentrate on both the presentation and business logic of the application and it makes code maintenance and debugging tasks very difficult. JSP helps in separating the presentation logic from business logic. This separation modularizes the application development process.

Apart from JSP, there are other technologies such as PHP, ASP, Coldfusion, and so on, which function in a similar manner as JSP does.

- Active Server Pages (ASPs) are developed to be compatible with Microsoft's .NET framework. It is built to be compatible with Internet Information Services (IIS). JSPs in contrast are platform independent and are compatible with any Web server.
- Hypertext Preprocessor (PHP) is also a similar technology as JSP and is open source. JSP being written in Java has an advantage of better tools and inherent mechanisms for database access, and so on.
- JavaScript is also used to generate dynamic Web page content but JavaScript runs only on the client-side without communicating with the servlet on the server. JavaScript code provides interactivity within the Web browser, it communicates with the server asynchronously. JavaServer Pages works on server-side and interacts with the servlets. JavaScript does not compete with JavaServer pages but complements them.

6.1.2 Uses of JSP

- While execution, JSP pages are translated into servlets. The advantage in terms of utility of JSPs over servlets is better maintainability and separation of presentation logic from the actual processing.
- JSP also enables embedding Java code into the Web page through JSP scriptlets.
- The dynamic elements of the Web page are embedded as part of the HTML tags in JSP. The JavaServer Pages Standard Tag Library (JSTL) has number of tags defined for this purpose.

6.1.3 Benefits of JSP

The benefits of JSP are as follows:

- Allows delegating of tasks or development of various components of the application to independent teams as the components are well separated.
- Supports the loosely coupled component architecture of Java. The components can be developed independently and later integrated.
- Introduces well-defined modules in the application as they separate presentation from processing of data.
- The presentation of the application can be improved by using sophisticated Web development tools for presentation. This enhancement to the user interface can be achieved independent of the underlying servlets.

6.1.4 JSPs in Web Applications

A Web application may contain various components such as Java classes, servlets, Web pages, and so on. Each of them are organized in different directories. When the application is deployed, the deployment descriptor is used to integrate all the components. The mapping of all the components is based on the mappings provided in the deployment descriptor.

The JSPs are stored along with the HTML pages in the Web Pages directory. Whenever there is a request for a JSP, it is converted into a servlet and then, executed in the container. The static content is converted into output stream and remaining elements in the JSP are converted into servlet code. The execution parameters of JSP can be manipulated through JSP page directives.

JSP configuration information is specified in the deployment descriptor web.xml through a <jsp-config> tag.

6.2 Authoring JSP Pages

A JavaServer Page has three types of constructs:

- ➔ Scripting elements
- ➔ Directives
- ➔ Actions

Scripting elements allow embedding Java code in the JSP page. This code later becomes part of the resultant servlet.

Directives are those constructs which control overall structure of the servlet.

Actions are those constructs which are used to control the behavior of the JSP engine or allow the developer to make use of already existing components in the application.

6.2.1 JSP Scripting Elements

Scripting elements are enclosed between <%...%> symbols. They are used to write Java code in JSP.

Following are the three types of scripting elements used to embed Java code in JSP:

1. Expressions
2. Declarations
3. Scriptlets

➔ JSP Expressions

Expressions are used to insert values directly to the output. The values of certain attributes can also be provided as expressions. JSP uses the following syntax to use expressions:

Syntax:

```
<%= Expression %>
```

Code Snippet 1 shows the usage of expressions in JSPs.

Code Snippet 1:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
    <title>Expressions Demo</title>
    </head>
```

```

<body>
    <h4>Date: " <%= new java.util.Date()%>"</h4>
    <p> Your IP address : "<%= request.getRemoteAddr()%>"</p>
</body>
</html>

```

In Code Snippet 1, values of two expressions are being retrieved.

The expression `<%= new java.util.Date()%>`, retrieves the value of date.

The expression `<%= request.getRemoteAddr()%>`, retrieves the value of IP address of the host. In this statement `request` is a predefined object.

JSP consists of many predefined objects. Following are some of the frequently used predefined objects. These predefined objects are also known as implicit objects.

- **request** – Whenever JSP is accessed, the JSP engine instantiates an `HttpServletRequest` object. The implicit object `request` refers to the request received. Data corresponding to the request can be retrieved with the help of `request` object.
- **response** – The `response` object refers to the response generated by the JSP for a particular request. The `response` is an instance of `HttpServletResponse`.
- **session** – The implicit object `session` is an instance of the `HttpSession` which is currently active.
- **out** – This object refers to an object used to write the output of JSP. This is usually a `PrintWriter` object.
- **application** – The `application` object refers to the context of the servlet. It is an object of type `ServletContext`.
- **config** – The `config` object refers to the configuration of the application to which the current JSP belongs. It is an object of type `ServletConfig`.
- **pageContext** – Refers to an object of type `PageContext` and server features pertaining to the environment of the Web page.
- **page** - Refers to the current JSP.

Using Expressions as Attribute Value

The attributes in a JSP refer to the properties with respect to the tags of JSP. Usually, these attributes are expected to be constant values. However, some of the attributes allow expressions as values to the attributes. Following are the elements which allow expressions to be assigned as values to the attributes:

- **<jsp:setProperty ...>**: This element has name and value attributes. It allows expressions to be assigned to its attributes as follows:

```
<jsp:setProperty name="entry" property="itemID" value='<%= request.getParameter("itemID") %>' />
```

- **<jsp:include >**: This element allows you to import files or pages into the current JSP. The page to be imported can be defined using an expression.
- **<jsp:forward >**: This element links the servlets and the JSPs. When the JSP is executed, the request is forwarded to the appropriate servlet.
- **<jsp:param>**: This element is included in the JSP or forwarded. If additional parameters need to be passed, then this element is useful.

→ JSP Declarations

Declaration statements in JSP are required to declare one or more variables which may be used in scriptlets. The syntax for declaring the variables is as follows:

Syntax:

```
<%! declaration %>
```

The XML equivalent for declaration is:

```
<jsp:declaration>
code fragment
</jsp:declaration>
```

Code Snippet 2 demonstrates the usage of JSP declarations.

Code Snippet 2:

```
<HTML>
<HEAD>
<TITLE>JSP Declarations</TITLE>
</HEAD>
<BODY>
<H1>JSP Declarations</H1>
<%! private int accessCount = 0; %>
<H2>Accesses to page since server reboot:
<%= ++accessCount %></H2>
</BODY>
</HTML>
```

In Code Snippet 2, a variable `accessCount` has been declared which counts the number of times the page is accessed after the corresponding servlet is loaded into the container. The variable is incremented every time the page loads into the Web browser.

→ JSP Scriptlets

Scriptlets are used to embed Java code into JSPs. Scriptlets are declared using the following syntax:

Syntax:

```
<%.....code fragment.....%>
```

Scriptlets can be used to perform tasks such as setting the headers of an HTTP response, setting the HTTP status codes, executing code that contains loops and other programming constructs, and accessing databases.

Code Snippet 3 shows the usage of scriptlets in JSPs.

Code Snippet 3:

```
<html>
<head>
    <title>Order form</title>
</head>
<body>
    <h3>Choose your products:</h3>
    <form method="get">
        <input type="checkbox" name="product" value="iPod">iPod
        <p></p>
        <input type="checkbox" name="product" value="Mobile Phone">Mobile
        Phone
        <p></p>
        <input type="checkbox" name="product" value="Toaster">Toaster
        <p></p>
        <input type="submit" value="Order">
    </form>

    <%
    String[] products = request.getParameterValues("product");
    if (products != null) {
    %>
        <h3>You have selected <%= products.length%> product(s):</h3>
        <ul>
    <%
        for (int i = 0; i < products.length; ++i) {
    %>
        <li><%= products[i] %></li>
    <%
        }
    %>
        </ul>
        <a href="<%=" request.getRequestURI() %>">>BACK</a>
    <%
    }
    %>

</body>
</html>
```

In Code Snippet 3, the user interface presents choice of products to the user. Based on the user's choice, it returns the list of the products ordered by the customer. The output of the program is as shown in figure 6.1.

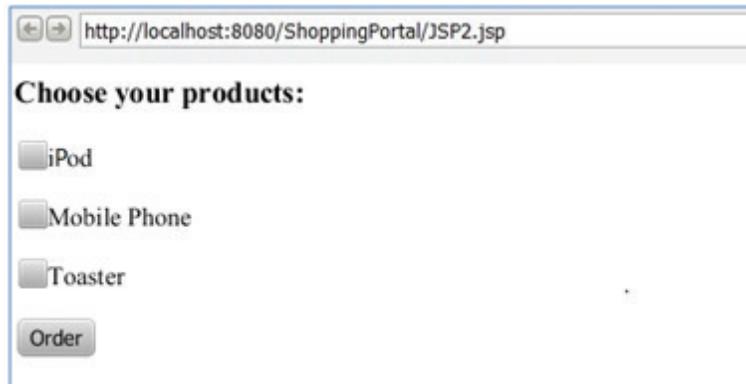


Figure 6.1: Output

After selecting the products iPod and Mobile Phone, the user clicks the Order button and the output will be as shown in figure 6.2.

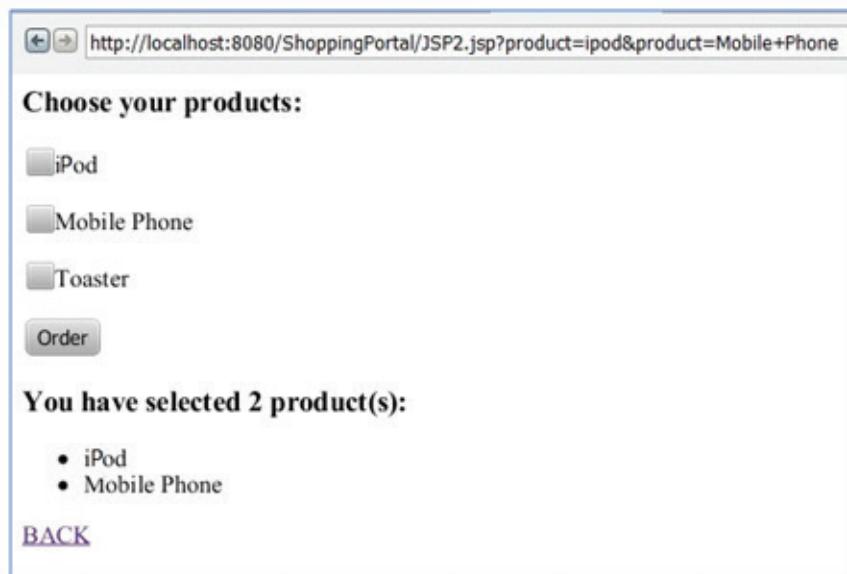


Figure 6.2: Output after Product Choice

6.3 JSP Directives

A JSP directive defines the structure of the servlet which is generated as a result of the JSP page. Following is the syntax for providing a directive for a JSP:

Syntax:

```
<%@directive attribute = "value" %>
```

A directive defining multiple attributes can be defined as follows:

```
<%@directive attribute1 = "value" attribute2 = "value" ...%>
```

JSP supports three types of directives:

- @page
- @include
- @taglib

6.3.1 @page directive

The @page directive is used to define page level attributes in the JSP. It can be placed anywhere in the JSP. It defines attributes such as content type of the page, importing classes for the page, and so on. A page directive can appear anywhere within the document.

Following are the attributes which can be defined through the page directive:

- **import** – The import attribute specifies the packages that require to be imported by the servlet class to which the current JSP will be translated during execution. The usage of the @page directive to import a package is as follows:

```
<%@page import = "java.util.*" %>
```

Apart from predefined Java classes, the import attribute can be used to import custom classes defined by the developer for the servlet.

- **contentType** – The contentType attribute is used to set the HTTP response header content type. HTTP response is generated by the servlet while processing an HTTP request. The content type header informs the receiver about the type of data present in the response based on which the client can invoke appropriate plug-ins or applications to receive the HTTP response. The content type should be any one of the Multipurpose Internet Mail Extensions (MIME) types. Following is the usage for contentType attribute:

```
<%@page contentType = "text/plain" %>
```

- **isThreadSafe** – The isThreadSafe attribute defines whether the JSP implements a SingleThreadModel or not. This implies whether the servlet shares resources with other servlets. In a SingleThreadModel, there is no sharing of resources hence, consistency is ensured. If it is not using SingleThreadModel, then multiple threads might access the resource which may lead to inconsistencies. The attribute can take either true or false as values. Following is the usage:

```
<%@page isThreadSafe = "true" %>
```

- **session** – The session attribute specifies whether the current JSP page participates in HTTP sessions or not. The session attribute can assume true or false values. Following is the usage:

```
<%@page session = "true" %>
```

- **buffer** – The buffer attribute specifies if the current response requires some buffer to be allocated on the client side. This attribute can be used to specify the buffer size required for the page. Following is the usage:

```
<%@page buffer = "none" %>
```

- **autoflush** – The autoflush attribute specifies whether the output buffer should be automatically cleared after the response is received. Following is the usage:

```
<%@page autoflush = "true" %>
```

- **extends** – The extends attribute specifies the classes from which the current page can be inherited. The values of the extends attribute are the super classes from which the page has been inherited. Following is the usage:

```
<% @page extends = "package.class" %>
```

- **info** – The info attribute takes a string value which can be returned when the getServletInfo() method is called. Following is the usage:

```
<% @page info = "Information about the page" %>
```

- **errorPage** – This attribute specifies the URL of the page which will be generated when an exception occurs in the Web page. Following is the usage:

```
<%@page errorPage = "/404.html" %>
```

- **isErrorPage** – This attribute informs whether the current page is an error page or not. If it is an error page, then it is invoked only when an exceptional condition occurs. This attribute will assume a true or false value. Following is the usage:

```
<%@page isErrorPage = "true" %>
```

- **language** - The language attribute defines the underlying scripting language in JSP. Following is the usage of the attribute:

```
<%@page language = "java" %>
```

6.3.2 @include directive

The @include directive is used to include files to the current JSP. These files can be HTML files, text files, applets, or any other files which may be required by the JSP. This directive enables the developer to use elements such as tables, images, and so on from the files.

Following is the usage of the @include directive:

```
<%@include = "/404.html" %>
```

6.3.3 @taglib directive

JSP allows users to create their own set of tags with attributes. The interpretation of the tags can be defined by the developer according to the requirement of the application. This set of user defined tags is termed as tag library. The tag library comprises a tag library descriptor and tag handlers. The tag handlers are responsible for the implementation of the tags definitions and the tag library descriptor defines the mapping of the tags to appropriate handlers. The tag handlers are generally Java classes.

These user-defined tag libraries can be included in the JSP files through the @taglib directive. Following is the usage of the @taglib directive:

```
<%@taglib uri = "location of the library" prefix = "prefixOfTag" %>
```

On using this directive, the JSPs can use the tags defined in the tag library.

6.3.4 Thread Safe JSPs

Multithreading is an execution mechanism in Java, where each execution flow is divided into independent threads and executed simultaneously. However, in certain scenarios the multithreaded execution can lead to inconsistent results. Using a single thread of execution in such a case is preferred. Java defines synchronized blocks to avoid inconsistencies due to multiple threads executing simultaneously.

In case of JSPs, single threaded programs can be executed by implementing the interface SingleThreadModel. The servlet container specification states that when SingleThreadModel interface is implemented, no two threads will simultaneously access the service () method of the servlet.

Apart from implementing the SingleThreadModel, thread safety can also be implemented through the @page directive where the entire page can be declared thread safe through the attribute isThreadSafe.

6.4 Processing Data Received from Servlets in JSPs

The user interface elements in JSPs are displayed using form tag. The forms have various input elements such as radio buttons, textboxes, comboboxes, and so on. The standard way of handling forms in JSP is through Java Beans. Bean class has variables equal to the number of input elements in the form and the corresponding getter and setter methods, to set and get the values of these variables.

The values of the input elements in JSP are set through get property and set property tags in JSP. Once the interface between the servlet and JSP is set, the servlet can perform the required operations on the input values read through the form.

Code Snippet 4 shows a JSP code which invokes a bean.

Code Snippet 4:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>JSP Page</title>
  </head>
  <body>
    <jsp:useBean id="probean" scope="session" class="Product.Products-
Bean" />
    <jsp:setProperty name="probean" property="proName" value="Galaxy"/>
    <jsp:setProperty name="probean" property="description" val-
ue="Touchphone"/>
    <p>Product Name:<br/>
      <jsp:getProperty name="probean" property="proName"/>
    </p>
    <p>Product description:<br/>
      <jsp:getProperty name="probean" property="description"/>
    </p>

  </body>
</html>
```

This code invokes a bean through useBean tag of JSP. The statement

```
<jsp:useBean id="probean" scope="session" class="Student.ProductsBean" />
```

indicates that the current JSP makes use of the Java Bean class Product.ProductsBean. The scope attribute of the use bean determines the extent of usage of the Java bean. The scope of a useBean can be as follows:

- **request** – which implies that the bean will be used only for the current request
- **session** - which implies that the bean can be used only for the current session
- **page** - which implies that the bean can be used throughout the current page
- **application** – which implies that the bean can be used through the entire application of which the current JSP is an integral part

The setProperty and the getProperty tags of the JSP are used to set the variables of the bean class and retrieve the variables from the bean class respectively. The bean class has getter and setter methods to use the values from the JSP. Code Snippet 5 shows a bean class.

Code Snippet 5:

```

package Student;

public class ProductsBean
{
    private String proName;
    private String description;
    //private int age = 0;

    public ProductsBean() {
    }
    public String getProName() {
        return proName;
    }
    public String getDescription(){
        return description;
    }
    //public int getAge(){
    //    return (int)age;
    //}
    public void setProName(String firstName) {
        this.proName = firstName;
    }
    public void setDescription(String lastName) {
        this.description = lastName;
    }
    //public void setAge(Integer age) {
    //    this.age = age;
    //}
}

```

Code Snippet 5 shows the Java Bean ProductsBean. The JSP code shown in Code Snippet 4 sets the values of the variables in the bean class. The bean class can define methods to manipulate the data retrieved from the JSP.

Figure 6.3 shows the output of the code.

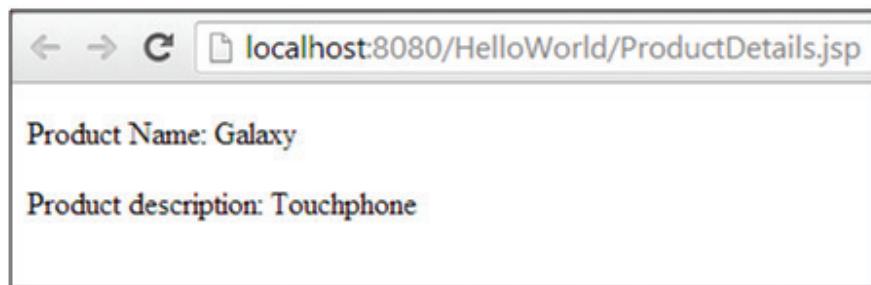


Figure 6.3: Data Retrieved from the Bean Class

6.5 JSP Standard Tag Library (JSTL)

JSTL is a repository of tags which are used to create JSPs. JSTL has tags for implementing various tasks which can be classified are as follows:

- Core tags
- XML tags
- Formatting tags
- SQL tags
- JSTL functions

These tags support functions such as iterations, implement conditions, access databases, and so on. Apart from the standard tag library, JSP allows for custom defined tags. A user can define a set of tags. In order to use these user defined tags, the tag library has to be included in the JSP through the @taglib directive as discussed earlier. Following is the format to include the standard tag library in the JSP:

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

6.5.1 Core Tags

All the core tags are prefixed with the letter ‘c’. These tags support features such as variable support, flow control, URL management, and so on. The URL for the core tag library is as follows:

<http://java.sun.com/jsp/jstl/core>

Core tags perform the following categories of functions:

- **Variable support** – Supports operations such as setting the values of the variables. The set and remove tags belong to this category.
- **Flow control** – These are similar to conditional and loop statements in core Java and are used for iterations in scriptlets. The tags in this category are choose.....when.....otherwise, forEach, forTokens, and if.
- **URL management** – Supports importing resources, redirecting requests, and other URL related operations. The tags in this category are import.....param, redirect....param, and url....param.
- **Miscellaneous** - These tags provide functions for error handling. The tags in this category are catch and out.

6.5.2 XML Tags

All the XML tags are prefixed with 'x'. There are three categories of XML tags - core, flow control, and transformation. The URL for XML tags is:

<http://java.sun.com/jsp/jstl/xml>

The functions performed by each category are as follows:

- **Core tags for XML** – The core tags provide the basic function of parsing and interpreting XML tags. The core XML tags are out, parse, and set.
- **Flow control tags for XML** – The flow control tags are used to control the interpretation order of XML tags. The tags used for this are choose...when...otherwise, forEach, and if.
- **Transformation tags** – The transformation tags perform functions such as styling of the document. The tags used for transformation are transform.....param.

6.5.3 Formatting Tags

Formatting tags are used for formatting the XML documents based on local parameters such as local time and so on. The prefix for formatting tags is 'fmt' and the formatting tags library is located at:

<http://java.sun.com/jsp/jstl/fmt>

The tags in this category are formatNumber, formatDate, parseDate, parseNumber, setTimeZone, and timeZone.

6.5.4 SQL Tags

SQL tags are used for database operations. All the SQL tags are prefixed with 'sql'. The URL for SQL tags is as follows:

<http://java.sun.com/jsp/jstl/sql>

The functions of the SQL tag library are setting up the data source and performing the SQL operations.

The tag used for setting up the data source is setDataSource. The tag for SQL operations are query...dataParam...param, update...dataParam...param, and transaction.

6.5.5 JSTL Functions

There are a set of predefined functions in JSP which are used over a group of objects. They are prefixed with 'fn'. The URL for the location of the JSTL functions is:

<http://java.sun.com/jsp/jstl/functions>

The functions are mainly operations on Collections and Strings for string manipulation.

Code Snippet 6 shows the use of some commonly used JSTL tags.

Code Snippet 6:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>

        <p><b>Using forEach</b></p>
        <c:forEach var="i" begin="101" end="105">
            Item <c:out value="${i}" /><p>
        </c:forEach>

        <p><b>Using formatDate </b></p>
        <c:set var="now" value="<%=new java.util.Date()%>" />
        <p>Formatted Date: <fmt:formatDate type="date" value="${now}" /></p>

        <p><b>Using forTokens</b></p>
        <c:forTokens items="John,Mary,Rosy" delims="," var="name">
            <c:out value="${name}" /><p>
        </c:forTokens>

        <p><b>Using choose</b></p>
        <c:set var="price" scope="session" value="${200*2}" />
        <p>Product price is : <c:out value="${price}" /></p>
        <c:choose>
            <c:when test="${price <= 0}">
                The product is for free.
            </c:when>
            <c:when test="${price < 1000}">
                Price is nominal.
            </c:when>
            <c:otherwise>
                Price is very high.
            </c:otherwise>
        </c:choose>

        <p><b>Using import</b></p>
        <c:import var="data" url="http://www.tutorialspoint.com"/>
        <c:out value="${data}" />
    </body></html>
```

The code shows the use of `forEach`, `formatDate`, `forTokens`, `choose`, and `import` tags of JSTL.

Figure 6.4 shows the output of the code.

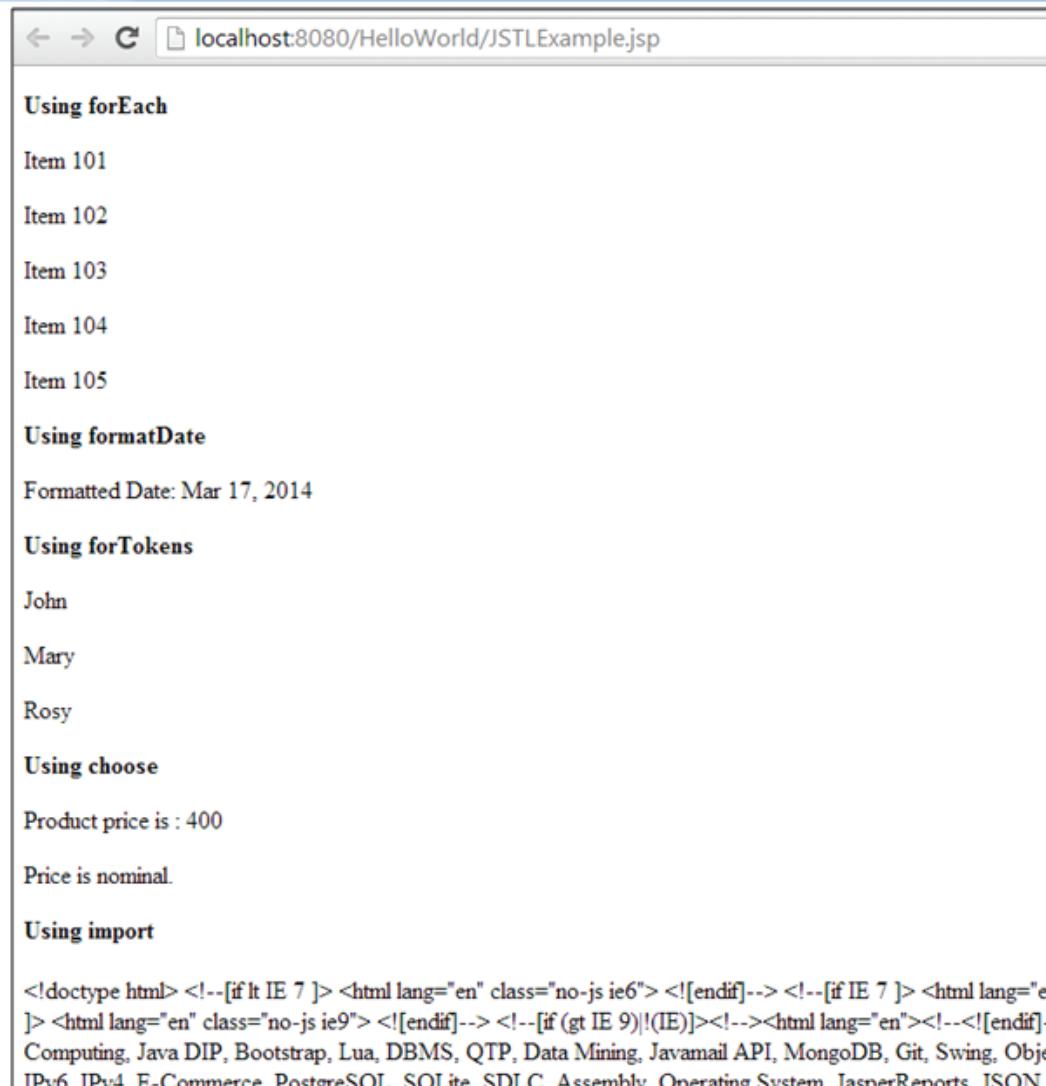


Figure 6.4: Using JSTL Tag Library

6.5.6 Custom Tags

Using JSP, one can create a custom tag or user-defined tag. When JSP is translated into a servlet, the tag is converted to operations on an object called a tag handler. Next, the Web container invokes these operations when the JSP page's servlet is executed.

The custom tags can be inserted directly into a JSP in the same manner as built-in tags. A custom tag can be empty or may have a body. The custom tag may also contain attributes. The class that handles the custom tag is called the tag handler. The class must extend SimpleTagSupport or BodyTagSupport class and override the doTag() method.

The code to be generated on execution is placed within the doTag() method. A Tag Library Descriptor (TLD) file which is an XML document containing information about a library as a whole and about each tag contained in the library also must be created. TLDs are used by a Web container to validate the tags. The TLD file is saved with the .tld extension. It is saved in the tlds folder during file creation.

To create a <ex>Hello /> tag, you need to first create a Tag Library Descriptor (TLD). To create the TLD file, right-click the project and select New → Tag Library Descriptor. Code Snippet 7 shows the TLD file HelloTLD.tld.

Code Snippet 7:

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.1" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:x-
si="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-jsptaglib-
rary_2_1.xsd">
    <tlib-version>1.0</tlib-version>
    <short-name>hellotld</short-name>
    <uri>/WEB-INF/tlds/HelloTLD</uri>
    <tag>
        <name>HelloTag</name>
        <tag-class>pkg.HelloTag</tag-class>
        <body-content>empty</body-content>
    </tag>
</taglib>
```

The file consists of the description for the tag, HelloTag. Please note that the tag <tag> is not added by default. It will be added during creation of the corresponding tag handler class. To create the tag handler class, right-click the project and select New → Tag Handler. Remember to check the TLD file created earlier as the TLD for the current tag handler. This will add the description of the tag and its handler in the TLD file. Code Snippet 8 shows the HelloTag.java file which is the tag handler for the HelloTag.

Code Snippet 8:

```
import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.io.*;

public class HelloTag extends SimpleTagSupport {
    public void doTag() throws JspException, IOException {
        JspWriter out = getJspContext().getOut();
        out.println("Hello! This is my first Custom Tag.");
    }
}
```

The code prints ‘Hello! This is my first custom tag.’ when the JSP is executed.

Code Snippet 9 shows the use of custom tag Hello:

Code Snippet 9:

```
<%@ taglib prefix="ex" uri="WEB-INF/tlds/HelloTLD.tld"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>

        <ex:HelloTag />

    </body>
</html>
```

The output of the code is shown in figure 6.5.

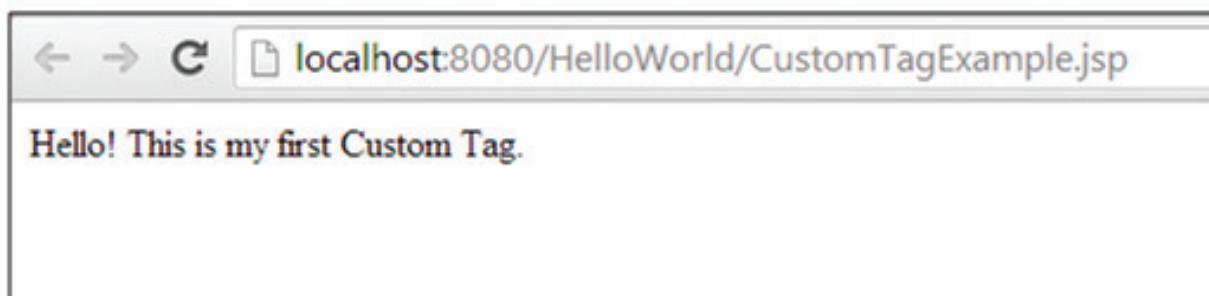


Figure 6.5: Custom Tag

Similarly, one can create custom tags with body and attributes.

6.6 Unified Expression Language

Expressions are used in JSPs to retrieve values in the scriptlets, setting values to the attributes, parameters, and so on. Apart from JSPs, expressions are also used in JSFs (JavaServer Faces). JavaServer Faces is a Java technology for developing user interfaces. Unified Expression language is used to make the expressions of both JSF and JSP compatible with each other.

Following are the advantages of having a unified expression language which is compatible with both JSP and JSF:

- It allows deferred evaluation of expressions.
- It supports expressions which can set values for the variables and also invoke methods.
- JSTL iteration tags can be used along with deferred evaluation of expressions.

Expressions in Unified Expression Language can be evaluated immediately or deferred for evaluation later.

Immediate evaluation of expressions implies that the expressions are evaluated by the JSP engine and read only values that are available for the user interface.

In case of deferred evaluation, the expressions are not evaluated by JSP but passed onto JSF or any other component which uses unified EL technology to evaluate the expression.

There are two types of expressions supported by EL:

- Method expressions
- Value expressions

Method expressions enable invoking of methods to retrieve the values of the expressions for the component which the current tag is representing.

Value expressions are those where a value is explicitly provided for the expression. There are two types of values – Rvalue and Lvalue. Rvalues are those values which cannot be modified and Lvalues can be modified.

All the expressions are resolved through a generic `ELResolver` class. The `ELResolver` identifies the expressions and evaluates the Java bean components. For instance, the following expression returns the context path of the page:

```
 ${pageContext.request.contextPath}
```

6.7 Check Your Progress

1. Which of the following statements about a JSP are true?

a.	JSP is converted into a servlet for execution
b.	JSTL is the standard tag library for JSPs
c.	Scriptlets are tags which allow Java code
d.	All of these

(A)	c	(C)	d
(B)	b	(D)	a

2. _____ is an implicit object in JSP.

(A)	request	(C)	pageContext
(B)	response	(D)	All of these

3. Which of the following attributes of @page directive ensures that the page is synchronized?

(A)	Extends	(C)	isThreadSafe
(B)	Import	(D)	SingleThreadModel

4. If 'c' implies core JSP tag then _____ implies an XML tag.

(A)	'c'	(C)	'fn'
(B)	'x'	(D)	'sql'

5. Which of the following is not a JSP directive?

(A)	page	(C)	include
(B)	request	(D)	taglib

6.7.1 Answers

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

Summary

- JavaServer Pages is a server side technology used to develop dynamic Web pages.
- JSPs work along with servlets to handle client request and responses. The client requests and responses are HTTP requests and responses.
- Using JSPs is advantageous over other similar technologies used on Internet due to platform independence and portability of Java.
- At runtime, the JSPs are converted into servlets and executed.
- JSPs have three types of elements – scriptlets, expressions, and directives.
- JSP has a repository of tags to perform various functions known as JSP Standard Tag Library (JSTL).
- Unified Expression Language (UEL) is used to evaluate JSP expressions such that they are compatible with JSFs.

Technowise



Are you a
TECHNO GEEK
looking for updates?

Login to

www.onlinevarsity.com

Session - 7

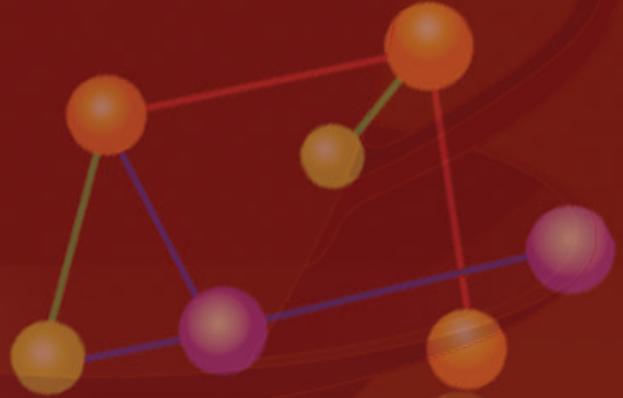
Introduction to JavaServer Faces

Welcome to the Session, **Introduction to JavaServer Faces**.

This session provides an introduction to JavaServer Faces (JSF) technology. It describes how to use the technology for developing user interfaces along with validators and listeners. It also describes how to develop a sample JSF application.

In this Session, you will learn to:

- Describe the basic structure and usage of JSF
- Compare it with other technologies used to develop user interfaces
- Explain the tag library of JSF
- Develop a sample application using JSF
- Explain the component model of UI
- Explain the navigation model
- Describe the lifecycle of JSF applications



7.1 Introduction to JSF Technology

JSF is a Java based framework used to build and maintain server-side user interface components of the Web applications. The server-side user interface component implies the response generated by the servlet residing on the server. The JSF technology supports the user interface development through an API and tag library.

- The API has implementations for page navigation, event handling, server-side validation, and so on.
- The tag library defines various tags used to connect the user interface components to Web pages and server-side objects. It provides a framework for the development of custom components and custom tag libraries.
- JSF allows defining user interface by dropping UI components on to the form. It connects the UI components with the application code on the server. It constructs the UI from reusable components and is capable of saving and restoring UI state beyond server request.

JSF uses XML files known as view templates or facelet views to generate the UI component on the client side.

Figure 7.1 demonstrates the functioning of an application using JSF technology.

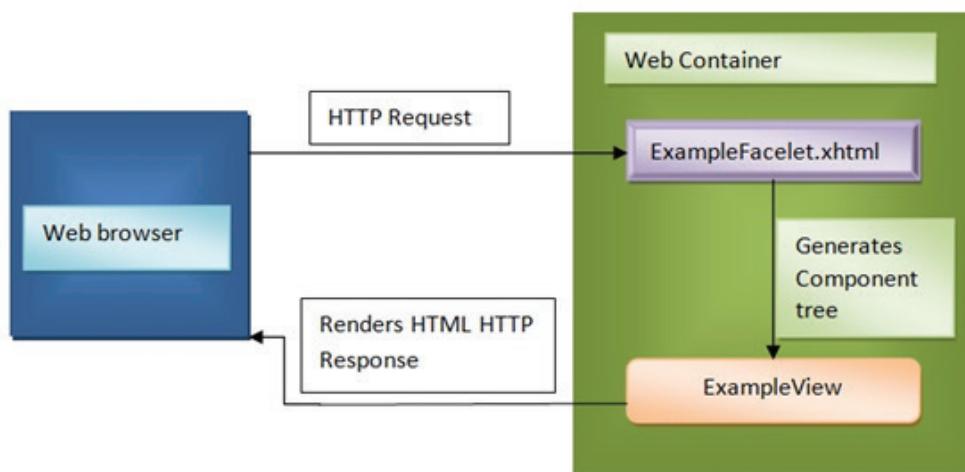


Figure 7.1: JSF Application

Figure 7.1 graphically represents how various components of the JSF application function together. When an HTTP request is received from the client, a JSF Web page is invoked by the Web container on the server. When a JSF page is developed, the receiving element is built using component tags. The JSF page may also be composed of objects such as eventlisteners, validators, and converters.

The JSF page processes the request and generates a response through a component tree and a view template.

7.2 Benefits of JSF Technology

Following are the advantages of JSF technology:

- JSF separates the presentation layer and application logic of the application. This clear separation of presentation layer and application logic is well defined on client-side through HTML and other technologies. JSF extends it to the server-side.
- This separation of presentation layer and application logic enables a simpler development process, where separate teams can develop the UI components and application logic of the application respectively. The architecture enables simple interconnection of these components after development.
- JSF does not limit the developer in terms of using a specific scripting language or markup language.
- JSFs can be used as an additional layer over JSPs of the application or as a layer over the servlets. The Unified Expression language implemented for JSF enables developers to collaborate with both JSPs and servlets. This layering of JSFs extends the user interface of the application onto a wide range of devices.
- Facelets used by JSF technology enable reuse of code and extending functionality over several components through templates and composite component features.
- Implicit navigation rules allow the developers to quickly configure page navigation.

7.3 JSF Application Demonstration

A JSF application comprises a set of Web pages written in XHTML. These Web pages in turn have JSF tags and managed beans. The configuration file, faces-config.xml was essential in case of JSF 1.0 but in JSF 2.0 this configuration file is optional.

7.3.1 Creating the User Interface

A JSF application is created in the same manner as other Web applications, by choosing a Web application while creating new projects in Netbeans. Figure 7.2 demonstrates this.

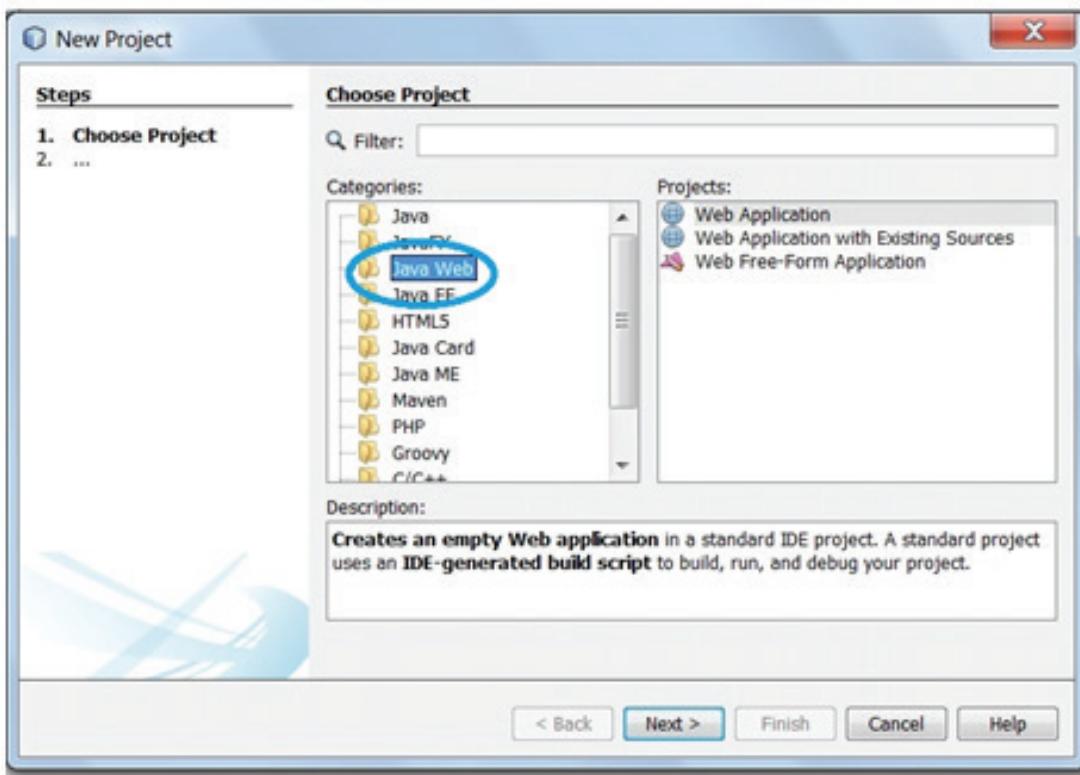


Figure 7.2: Creating a Web Application

To create a JSF application an additional step is added at the end of the project creation process which is the choice of the framework. To create a JSF project, select the JavaServer Faces checkbox as shown in Figure 7.3.

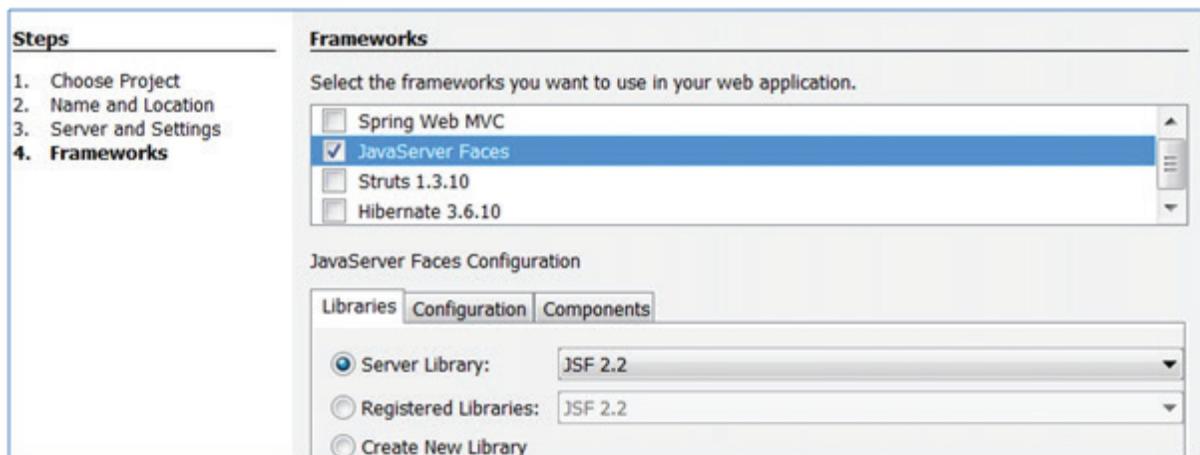


Figure 7.3: Selecting the Framework

After creating the JSF project, Netbeans creates a project with the directory structure as shown in figure 7.4. A JSF project named ‘UserRegistration’ has been created here. The files web.xml and index.xhtml are created by default.

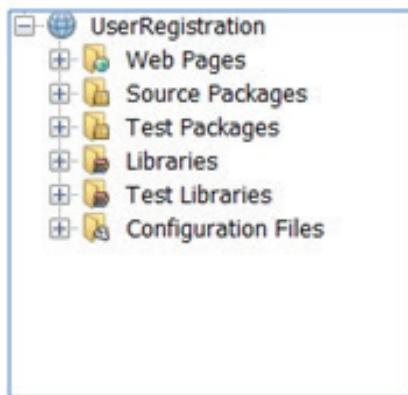


Figure 7.4: Directory Structure

The project stage is set to ‘Development’ by default. The developer can check the stage of the project in web.xml file. When a project is set to development stage, certain page navigation messages are defined by default. These messages enable the developer to understand when there are errors in the code and the page is unable to navigate to the next Web page in the flow. Apart from ‘Development’ the project can be in - ‘Production’ stage, ‘System test’ stage, and ‘Unit test’ stage.

index.xhtml is the Web page created by default in the JSF application. Web application execution can begin from an .html, .xhtml, or .jsp page. Code Snippet 1 shows the code generated for index.xhtml file created for the project ‘UserRegistration’.

Code Snippet 1:

```

!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"
< html xmlns="http://www.w3.org/1999/xhtml" xmlns:h=http://xmlns.jcp.org/jsf/html >
<h:head>
    <title> Facelet title</title>
</h:head>
<h:body>
    Hello from Facelets
</h:body>
</html>
    
```

Code Snippet 1 shows the default .xhtml document created when a facelets project is created in NetBeans IDE. The code shown is an XHTML file with facelet specific namespaces. In the code, the JSF specific namespace being used is declared through the following statement:

`xmlns:h=http://xmlns.jcp.org/jsf/html`

This allows usage of the JSF component library. The prefix ‘h’ represents HTML. For the default index file, UI components can be added through drag and drop.

Ctrl+Shift+8 displays the palette for the UI components. These components can be dragged and dropped in the body section of the index.xhtml.

Code Snippet 2 shows the modified index.xhtml.

Code Snippet 2:

```

!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"
< html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://xmlns.jcp.org/jsf/html" >
<h:head>
    <title> Facelet title</title>
</h:head>
<h:body>
<f:view>
    <h:form>
        <h:outputLabel value="Name:" for="name"/>
        <h:selectOneMenu id="prefix" value="#{Members.reg.RegistrationManagedBean.prefix}" >
            <f:selectItem itemLabel="Mr." itemValue="Mr"/>
            <f:selectItem itemLabel="Mrs." itemValue="Mrs"/>
            <f:selectItem itemLabel="Ms" itemValue="Ms"/>
        </h:selectOneMenu>

        <h:inputText id="Name" label="Name" required="true" value="#{Members.reg.RegistrationManagedBean.Name}"/>
        <h:message for="firstName" />
        <p></p>
        <h:outputLabel value="Phone:" for="phone"/>
        <h:inputText id="contactno" label="Phone" required="true" value="#{Members.reg.RegistrationManagedBean.contactno}">
        </h:inputText>
        <p></p>

        <h:outputLabel for="age" value="Age:"/>
        <h:inputText id="age" label="Age" size="2" value="#{Members.reg.RegistrationManagedBean.age}"/>

        <p></p>

        <h:commandButton id="register" value="Register" action="confirm" />
    </h:form>
</f:view>
</h:body>
</html>
```

Here, the bean is RegistrationManagedBean. All the input controls in the Web page are mapped to the attributes in the bean. For instance, the variable value="#{Members.reg.RegistrationManagedBean.Name}" accesses the Name variable of the managed bean.

The given code generates the output as shown in figure 7.5.

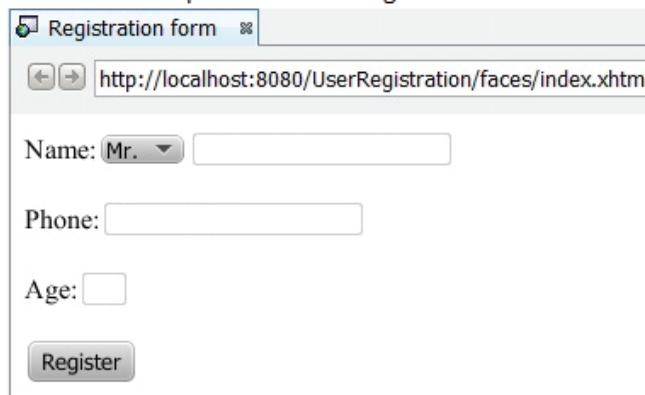


Figure 7.5: Output

The value attribute for each input type has a reference to the managed bean. The value in each input field is mapped to a variable created in the managed bean. The variable is identified along with the package.

7.3.2 Creating a Managed Bean

The managed bean implements the logic for the application. A JSF Managed Bean can be created by selecting the option from the New submenu provided while creating a new file as shown in figure 7.6.

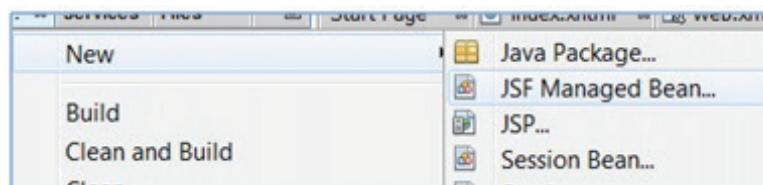


Figure 7.6: Creating JSF Managed Bean

The New JSF Managed Bean dialog box is displayed as shown in figure 7.7. Here the developer needs to select various options pertaining to the managed bean.

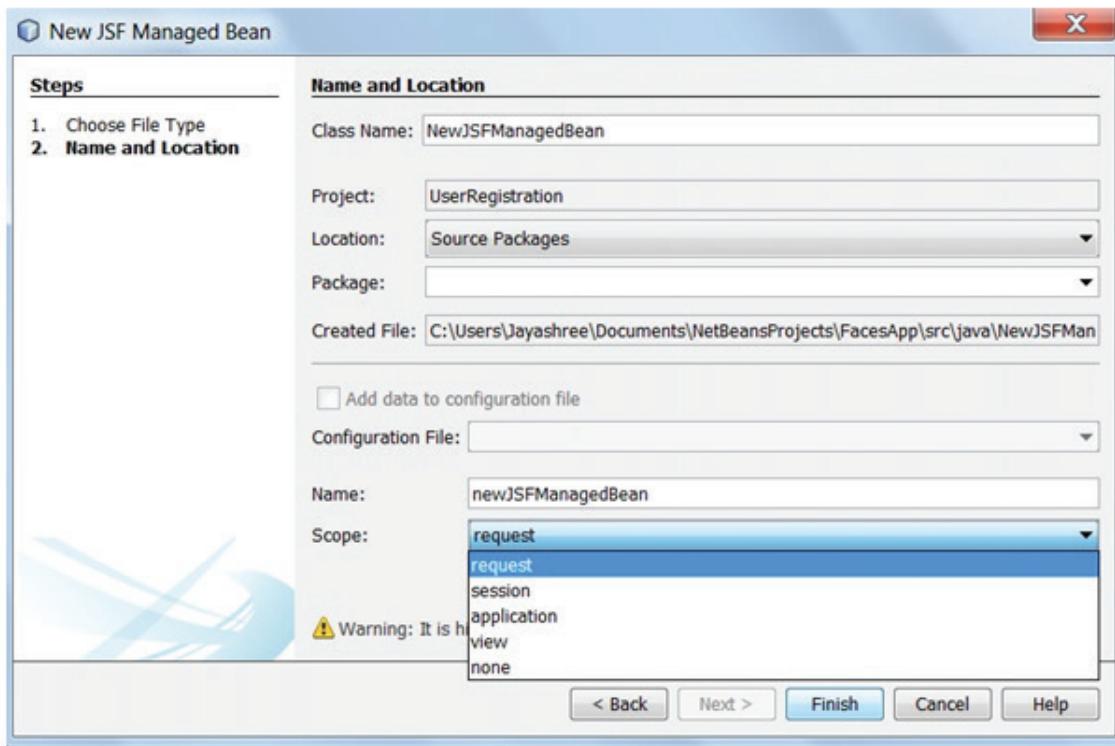


Figure 7.7: Selecting Options for the Managed Bean

The developer must select among the five options for the scope of the managed bean. Each of these options has the following interpretation:

- **request** - implies that the bean is available only for a single HTTP request.
- **session** – implies that the bean is available for the entire HTTP session.
- **application** – implies that the bean is available as long as the application is active on the server and to all the users using the application.
- **view** – implies that the bean is available for Ajax requests.
- **none** – implies that the bean is not created for any particular scope but is invoked on demand.

As per the requirement of the application, the appropriate scope for the bean is selected while creating the bean. In the given example, select request as the scope.

The managed bean is created with the class name provided in the dialog box. Once the managed bean is created, the variables are added to the managed bean class. There should be one variable for every data field read from the form. The getter and setter methods are invoked for each of the variables. The getter and setter methods of the bean shall be discussed in detail in later sections.

7.3.3 Creating the Response Page

A response xhtml page is created for the index page and named as specified in the action field of the index.xhtml file as shown in Code Snippet 2 where, action = confirm. The response page 'confirm' is defined as shown in Code Snippet 3.

Code Snippet 3:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <h:head>
        <title> Confirmation</title>
    </h:head>
    <h:body>
        <h:outputText value="Registration confirmed"/>
        <p></p>

        <h:outputText value="Name:"/>
<h:outputText
    value="#{Members.reg.RegistrationManagedBean.prefix}" />

        <h:outputText value="#{Members.reg.RegistrationManagedBean.name}" />
        <p></p>
        <h:outputText value="Phone:"/>
        <h:outputText value="#{Members.reg.RegistrationManagedBean.contactno}" />
        <p></p>
        <h:outputText value="Age:"/>
        <h:outputText value="#{Members.reg.RegistrationManagedBean.age}" />
        <p></p>
    </h:body>
</html>
```

Code Snippet 3 shows the code for the response page. For the values to be displayed as response it accesses the values from the managed bean. The h:outputText tag is an HTML tag which is used to display text in the Web page.

Code Snippet 4 shows the RegistrationManagedBean used in this application.

```
package Members.reg;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;
import javax.faces.bean.SessionScoped;
public class RegistrationManagedBean {
    public RegistrationManagedBean() {
    }
```

Code Snippet 4:

```

private String prefix;
    private String name;
    private String contactno;
    private String age;
public String getPrefix() {
    return prefix;
}
public void setPrefix(String prefix) {
    this.prefix = prefix;
}
public String getName() {
    return Name;
}
public void setName(String Name) {
    this.Name = Name;
}
public String getContactno() {
    return contactno;
}
public void setContactno(String contactno) {
    this.contactno = contactno;
}
public String getAge() {
    return age;
}
public void setAge(String age) {
    this.age = age;
}
}

```

On running the project, the output of the execution is as shown in figure 7.8.

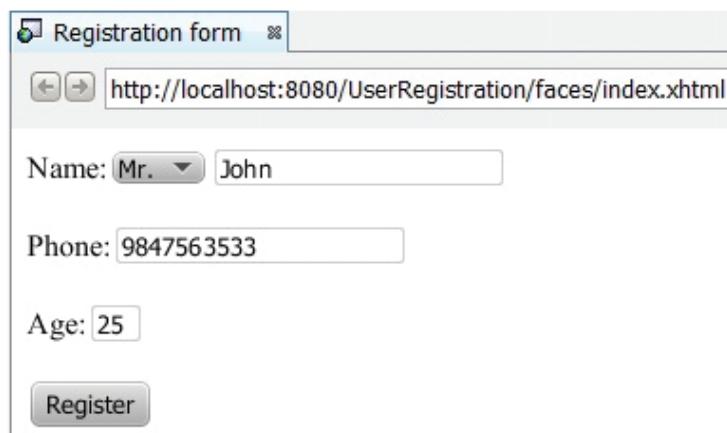


Figure 7.8: JSF Application Execution

Click Register. The response is generated as shown in figure 7.9.

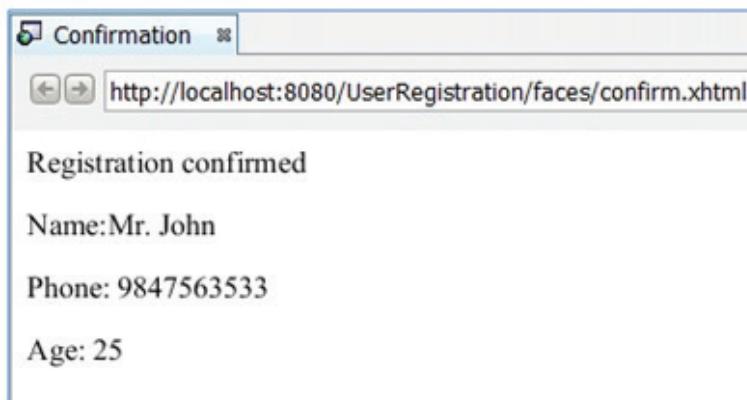


Figure 7.9: JSF Response Page

The data provided by the user is stored in the bean and retrieved on the confirm.xhtml page using the JSF `outputText` tag.

7.4 UI Component Model

JavaServer Faces provides a repository of user interface components which can be used to create JavaServer Faces view. This view can be defined as a UI or a non-UI component. These components are simple, reusable elements which can be used to create JSF applications. The components can be as simple as a button and as complex as a table which in turn is made of multiple components.

The JSF component architecture has the following elements:

- `javax.faces.component.UIComponent` classes are used to define the components, their state, and behavior.
- A rendering model defines the usage of the components.
- A conversion model defines the conversion of data format between components and registers data converters on the component.
- An event and listener model handles the events occurring with respect to the components.
- A validation model enables validators to be registered on a component to validate the data handled by the component.

7.4.1 Component Classes of the User Interface

A predefined set of UI component classes and associated behavioral interfaces are provided by JSF technology which can be used to create user interface components. This basic set of components can be extended which enables the developers to create customized components. The base class for all the classes is `javax.faces.component.UIComponent`. This is an abstract class. The base class of all the components is defined in the class `UIComponentBase` that defines the basic functionality of a class.

Table 7.1 shows some component classes that are available as a part of JavaServer Faces technology.

Component	Description
UIData	An instance of this class is bound to the collection of data which is an instance of javax.faces.model.DataModel. The collection can be a database.
UIColumn	An object which represents a column of data from the UIData object.
UICommand	An instance of this class fires an action in the user interface.
UIForm	An instance of UI form has several child UI components which can accept user input.
UIGraphic	An instance of UIGraphic is used to display an image.
UIInput	UIInput instance can be used to accept input from the user.
UIMessage	An instance of UIMessage is used to display an error message.
UIMessages	Used to hold set of error messages.
UIOutcomeTarget	Used to display a hyperlink.
UIOutput	Used to display output.

Table 7.1: Component Classes of UI

The component classes have certain behavior which is implemented through behavioral interfaces. Table 7.2 shows various behavioral interfaces provided by JSF.

Interface	Description
ActionSource	This interface implies that the current component can fire an action.
ActionSource2	Extends ActionSource functionality and allows the components to use expression language while referencing to the event handlers.
EditableValueHolder	Allows providing additional features to input data holders.
NamingContainer	This container holds all the objects which require an unique identifier.
StateHolder	Defines that the component has a state that need to be saved between requests.
ValueHolder	It implies that there is a local value associated with the current component.
javax.faces.event.SystemEventListenerHolder	Maintains a set of event listener instances for various events.
javax.faces.component.behavior.ClientBehaviorHolder	Instances of client behavior can be attached through this interface.

Table 7.2: Behavior Interfaces

7.4.2 Rendering Components on Web Pages

JavaServer Faces architecture separates the definition of components and rendering the components. This separation enables reuse of code, where a component class can be implemented once and rendered in different ways based on facts such as hardware of end user, that is, whether the Web page is rendered on a mobile device or a machine.

The rendering of a component is implemented as a renderer class. A render kit is used to determine how component classes are mapped onto the component tags based on the client. A render kit defines a set of javax.faces.render.Renderer class for all the supported components.

For instance, a UIselectOne component allows the user to select one option out of several options, which can be rendered on the client side as a combobox, listbox, or as a group of options.

When a custom tag is defined in the render kit, it requires definition of component functionality and rendering attributes. JSF technology provides a custom tag library for rendering components of HTML.

7.4.3 Conversion Model

Data is associated with various instances in the application. The managed bean interprets the data as variables and invokes methods on this data, whereas data in the user interface is closer to the user perspective of the application. There is a difference in the data interpretation format in such scenarios. The application has two views of data; one is the model view which interprets the data as objects and variables and the other one is the presentation view which interprets the data in a format understandable to the end user.

JSF technology enables automatic conversion among these two formats of data. When there is a requirement of converting the format of the data, JSF allows registering a javax.faces.convert.Converter implementation on UIOutputComponent. When a converter implementation is registered on a component, the converter is responsible for conversion of data.

7.4.4 Event and Listener Model

The event and listener model of JSF is similar to that of Java Beans event listener model. There are three types of events according to JSF specification; application events, system events, and data model events.

Events of an application are generated by objects of type UIComponent. An event object identifies the component which has generated the event. The application has to be notified about the event which is done by the listener objects. The component whose events are supposed to be captured by the listener has to be registered with the listener object.

UI components may generate two types of events –action events and value change events.

An action event generates an object of class javax.faces.event.ActionEvent, when the user activates a component that implements the behavioral interface ActionSource.

Similarly, a value change event generates an object of class javax.faces.event.ValueChangeEvent. This event occurs when certain value of a component which accepts user input, get change. These are instances of UIInput.

A value of the attribute `immediate` defines the time of handling of the event. Based on this value, the event can be handled during the invoke application phase or apply request values phase of the application execution. Value change events can be handled during process validation phase.

When an action event or a value change event occurs in the application, the system can react through an event listener or by invoking a managed bean to handle the event.

System events occur due to objects in the application but not due to UIComponents. These events have to be handled in the application definition.

A data-model event occurs when there is a change to the underlying database of the application.

7.4.5 Validation Model

JSF provides validation of data in the user input components. The validation checks are performed through a set of standard classes. There are a set of standard tags which are used for validation in JSF. These tags are implemented through `javax.faces.validator.Validator` implementations.

Most of the JSF tags have attributes defined to perform the validation function. In addition to the validators on individual components, the developer can also register default validators on the application. In addition to the predefined attributes for validation, custom validators can also be created on the application. There are two ways to implement custom validation:

1. By implementing a Validator interface

While implementing the validator interface, the validator must be registered with the application and corresponding custom tag must be created.

2. By implementing a managed bean

7.5 Navigational Model

Navigational model defines the order in which the pages in the application should be traversed according to the control flow of the application. In JSF, navigation is the set of rules for selecting the subsequent page to be loaded by the application.

Navigation can be implicit or user defined. Implicit navigation is defined by the events on the Web page. The `action` attribute of the event defines the Web page to be loaded. This is defined by the developer while implementing the logic of the application.

Consider the following statement:

```
<h:commandButton value="submit" action="response">
```

According to the statement, when the command button is clicked, it implies an event of submit. The corresponding action that must be taken involves loading the response Web page. The navigation handler searches for the page in the context of the application and loads the appropriate page.

User defined navigation implies navigating through the Web pages according to the explicit configuration in the config files of the application.

Navigation rules are defined in the configuration files such as faces-config.xhtml.

The default structure of a navigation rule is as follows:

```
<navigation-rule>
    <description></description>
    <from-view-id></from-view-id>
    <navigation-case>
        <from-action></from-action>
        <from-outcome></from-outcome>
        <if></if>
        <to-view-id></to-view-id>
    </navigation-case>
</navigation-rule>
```

A navigational rule comes into action when there is some action in the current Web page or some event or value change in the current Web page. The characteristic of the event which triggers the page navigation is defined in the navigation rule.

Consider the following example of a navigation rule:

```
<navigation-rule>
    <from-view-id>/Registration.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>success</from-outcome>
        <to-view-id>/Confirmation.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

The given navigation rule defines navigation from page Registration.xhtml to Confirmation.xhtml page. The current page displaying is Registration.xhtml. On successfully navigating this page, the next page to be loaded is Confirmation.xhtml. Similarly, the navigation can also be triggered on actions such as button click in the initiating Web page. The navigation rule can be made more complex by using multiple if conditions leading to different navigation cases.

Implicit navigation of pages is carried out by the NavigationHandler. This NavigationHandler can be invoked by EventListener objects on the Web pages. The NavigationHandler invokes the appropriate page implicitly otherwise the navigation rule in the configuration file is checked to locate the appropriate page.

7.6 Lifecycle of JavaServer Faces Application

The lifecycle of a JSF application begins when a client makes an HTTP request and is completed when the response is rendered to the client. There are mainly two phases in the lifecycle; execute and render.

The execute phase is organized according to a component tree, including various tasks such as validation of the input data, handling of component events, and so on. This component tree is also known as view. JSF stores the state of the Web page. Whenever the Web page is loaded the view is restored.

Following is the order of events in JSF application lifecycle:

1. Restore view of the Web page.
2. Apply requests to the Web page.
3. Process the events that trigger in the Web page.
4. Process the validations of the data in the Web page.
5. Connect to the managed bean to update the model values of the application.
6. If updating the model values triggers any events, handle them.
7. Render the response.

The typical lifecycle may comprise all these steps or a response may be generated at an appropriate intermediate step which may complete the lifecycle.

There are two types of requests handled during the lifecycle of the application; initial requests and postback requests. Initial request implies a new request for the Web page, whereas postback request implies submitting the data to an already loaded Web page of the application.

The events that occur during the lifecycle of an application can be categorized into the following phases:

- Restore view phase
- Apply request values phase
- Process validations phase
- Update Model values phase
- Invoke application phase
- Render response phase

7.6.1 Restore View Phase

When a JSF application is invoked, it is done through either an initial request or a postback request. When an initial request is made, all the components, their event handlers, listeners, validators, and managed beans are loaded. All these components can handle FacesContext instance. When it is an initial request, the next phase would be a Render response phase, where all the components are loaded on the client side.

If it is a postback request such as submission of data through a form, then a view corresponding to this page present in the FacesContext of the application is restored. During restore view phase, all the values or request parameters are retrieved and stored locally on each component.

7.6.2 Apply Request Values Phase

The request parameters retrieved in the restore view phase are restored. If this triggers any events in the application, the event handlers are invoked based on the value of the immediate attribute. If the immediate attribute is set to true, then the event is handled immediately, otherwise it is deferred. At the end of this phase, the components are set to their new values and corresponding messages and events of the application are queued.

7.6.3 Process Validations Phase

Once all the values are set, then the validation of the values is done in this phase. FacesContext uses the validate method for this purpose. If any conversions of data from the components are required, it is done in this phase. If any error occurs during validation, then an error message is generated and the execution advances to the render response phase. If any event handlers have invoked the render response phase, then the response is rendered and the application execution is completed. Similarly, the execution completes if there is a redirection to another application.

7.6.4 Update Model Values Phase

After completion of data validation, the values of the components are set to the local values. The conversion of the component data to the corresponding session bean data is done in this phase. The application enters the render response phase, if any error occurs during the conversion or if the application is redirected, or if render response phase is invoked by any of the event handlers.

7.6.5 Application Invoke Phase

In this phase, all the application level tasks are carried out such as submitting the form. This may, in turn, lead to navigation to another page of the application, further leading to the render response phase of the application.

7.6.6 Render Response Phase

The components of the Web page are rendered on the client-side according to the JSF specification. If it is an initial request, then the page is added to the component tree.

7.7 Check Your Progress

1. Which of the following statements about JSF are true?

a.	JSF is a server-side technology.
b.	JSF can only develop UI components for the application.
c.	The navigation rules for JSF should always be explicitly defined.
d.	None of these.

(A)	a, c	(C)	a, c
(B)	a, b	(D)	a

2. Which of the following is not a JSF component?

(A)	faces-config.xml	(C)	Table
(B)	Button	(D)	Image

3. Which of the following phases in JSF lifecycle validates the values of the UI components?

(A)	Invoke Application phase	(C)	Update model values
(B)	Apply request values phase	(D)	Process validations phase

4. Which of the following UI components access databases?

(A)	UIData	(C)	Both a and b
(B)	UIColumn	(D)	Only a

5. Which of the following values can be used for the scope of the managed bean so that the bean sustains a lifecycle of the JSF?

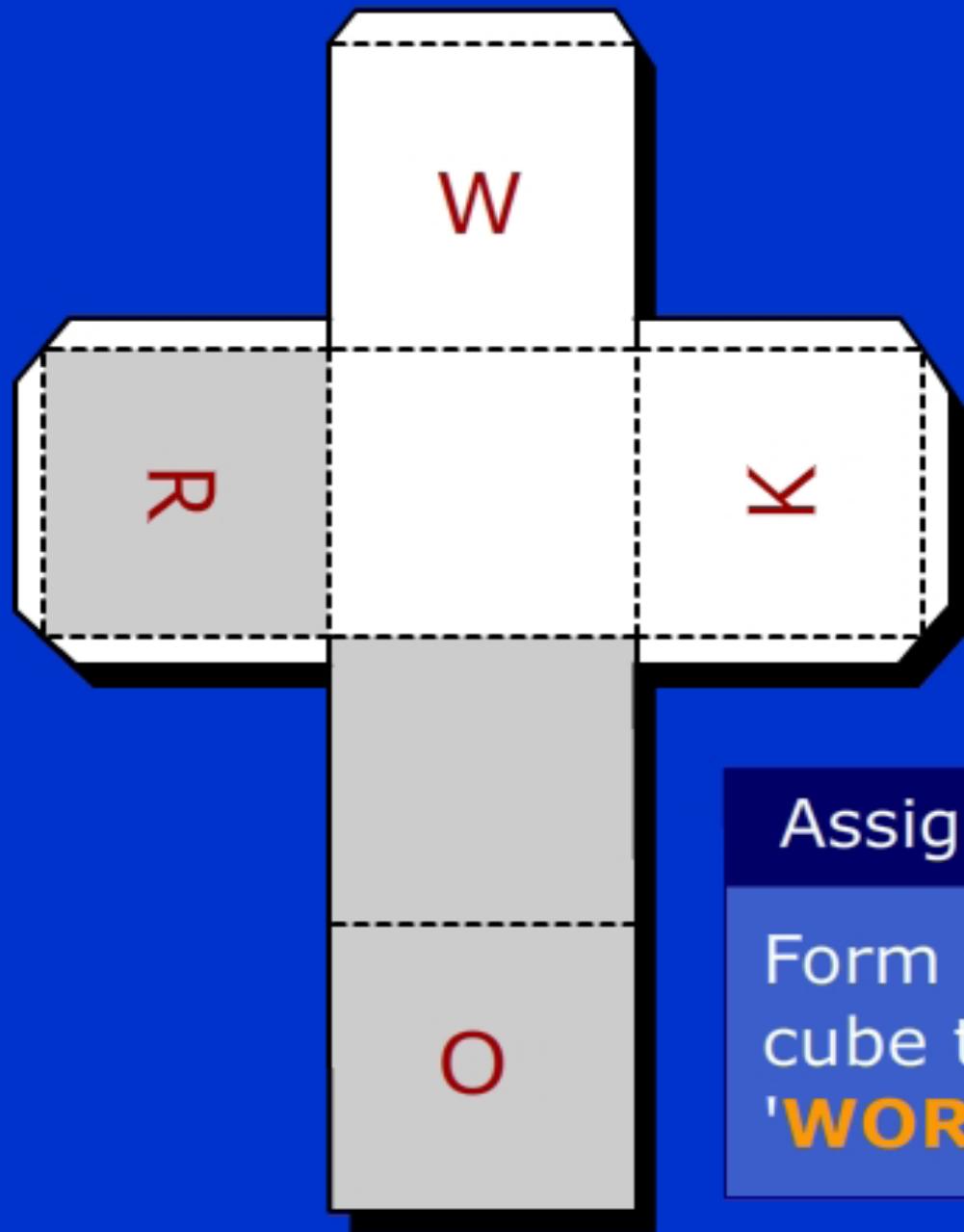
(A)	Request	(C)	Page
(B)	Session	(D)	All of these

7.7.1 Answers

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

Summary

- JavaServer Faces is an application framework used to develop user interfaces based on templates.
- JSF follows component-based development of the UI and uses a standard tag library. It also allows developers to define custom tags for application development.
- The response of the request can be generated by communicating with a managed bean or without using the bean.
- The page navigation in JSF can be implicit or user-defined. If it is user-defined navigation, the navigation rules are defined in the configuration files.
- There are two important phases in the lifecycle of the JSF application, execute phase and render phase. The execute phase has other phases based on the inputs received and events triggered during the application execution.
- The lifecycle of JSF application can start with restore view phase if the request is a postback request. The lifecycle ends with a Render response phase.



Assignment

Form the
cube to read
'WORK'.

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

For perfection, solve the assignments @

www.onlinevarsity.com

Session - 8

Java Server Faces as Web Pages

Welcome to the Session, **Java Server Faces as Web Pages**.

This session introduces the process of creating Web pages through JavaServer Faces (JSF). It describes how to use various component tags of JSF and connect them to the validators, listeners, and so on. It also explains how to develop Java beans and connect the bean components with JSF pages.

In this Session, you will learn to:

- Develop Web pages using JSF
- Add various components to the Web pages through JSF tags
- Describe the use of converters, validators, and listeners to add additional functionality to the Web pages
- Develop server-side components such as managed beans and converters for JSF
- Explain how to connect all the components together to develop a comprehensive application



8.1 Using JavaServer Faces Technology in Web Pages

JSF technology provides component tags which can be used for creating the Web pages of an application. Functionality to these component tags is added by registering converters, validators, and listeners with them. Managed beans are also associated with the components in the Web page. The managed beans provide business logic to Web pages.

8.1.1 Web Page Setup Using JSF

A typical JSF Web page comprises the following components:

- Namespace declarations which are used to declare JSF tag libraries.
- HTML head and body tags. However, these are optional.
- Form tag which represents the user input components.

There are various tag libraries defined to create JSF pages. The developer has to declare the tag library which will be used in the definition of the current Web page. Every JSF page requires access to two tag libraries – JavaServer Faces HTML render kit and JavaServer Faces Core tag library. JSF also has a library which defines the tags for HTML user interface; this tag library is known as HTML standard tag library. The HTML standard tag library is associated with an HTML render kit. In order to use certain tag library in the JSF Web page, it has to be declared in the namespace section.

When multiple tag libraries are included for certain Web page, then a tag library specific prefix has to be added to the tag. For instance, for all the HTML tags a letter ‘h’ is prefixed to the tag.

8.1.2 HTML Tag Libraries in JSF Pages

The HTML tag library in JSF is used to add HTML components to the Web page. Similar to an HTML page these components are used to display data on the Web page, accept user data as input, and so on. Based on the client who is invoking the Web page, each component of the JSF can be rendered in a different way.

Table 8.1 shows some of the important HTML tags used in JSF pages.

Tag	Description
h:form	Represents an input form to which other components can be added
h:datatable	Creates an HTML table which can be modified dynamically
h:graphicImage	Displays an image
h:column	Represents a column of data in a data table of the database
h:commandLink	Used to create a hyperlink
h:inputText	Used as text area which can accept user input
h:message	Used to display a localized message

Tag	Description
h.outputText	Allows display of plain text
h:selectOneListBox	Allows selecting one element from a list of options
h:selectOneMenu	Allows selecting one item from a menu
h.selectRadioOne	Allows choosing one option from a list of options

To add an HTML page structure to the JSF Web pages, <h:head> and <h:body> tags need to be added to the JSF page.

8.1.3 Component Tag Attributes

Each component tag has certain attributes associated with it. Following are the common attributes associated with each tag:

- **binding** – This property binds the component with a bean, which will be invoked for any action with respect to the component.
- **id** – Uniquely identifies the component. This attribute is not usually required for the component but is used when the component has to be referred by another component. The id need not be explicitly defined by the developer as the JSF implementation automatically generates an id for the components.
- **immediate** – This attribute can only assume a value of true, when the value is set to true. It indicates that any validations and conversion with respect to the component should happen immediately.

Consider an instance where the Web page has two components – a text area and a button.

If the immediate attribute for both the components is set to true, then the value entered in the text area is used immediately when the button is clicked.

If the immediate attribute of the text area is not marked true and the immediate attribute of button is marked true, then the value is not considered as a parameter when the button is clicked. The value in the text area is considered for interpretation later.

- **rendered** - This attribute specifies the condition when the component should be rendered. If this condition occurs or holds true, then the component is rendered otherwise the component is not rendered.

A boolean EL expression is used along with the rendered attribute to determine whether the component should be rendered or not. For instance, in a bank portal every account holder has a login. When the user logs in, the details of the accounts held by the user are displayed. The Web page displaying the account details also has an area where the fixed deposits held by a user need to be displayed. The elements displaying the fixed deposits are rendered only if the account holder actually holds a fixed deposit otherwise, these elements are not rendered.

- **style** – specifies a CSS style for the component.
- **styleClass** – this attribute specifies a CSS class according to which the component has to be rendered.
- **value** – This specifies the value component in the form of a value expression.

Consider the following example:

```
<h:commandLink id="check"
    ...
    rendered="#{cart.numberOfItems > 0}">
    <h:outputText
        value="#{cart.numberOfItems}" />
</h:commandLink>
```

The value field in the code is the value to be displayed with the outputText tag which implies the number of items placed in the shopping cart in an online shopping portal. The rendered attribute ensures that the component is rendered only if the value of numberOfItems is greater than zero.

8.1.4 Using Text Component Tags

Text components of a JSF page are essential to view and display text on the Web page. There are three types of text components – Label, Text area, and Password.

- Label is used to display read only text on the Web page.
- Field allows the user to enter text data, which can be submitted along with the user form.
- Password field also allows the user to enter data, but the data entered in the field is concealed through characters such as asterisks.

There are four types of input tags – h:inputHidden, h:inputSecret, h:inputText, and h:inputTextArea.

- h:inputHidden tag allows accepting a hidden variable in the Web page
- h:inputSecret tag is used for text areas which accept password or any confidential values
- h:inputText tag is used to accept text input in a single line
- h:inputTextArea is used to accept input of multiple lines

Output tags are used to include the output components in the JSF Web page. There are four types of output components – h:outputFormat, h:outputLabel, h:outputLink, and h:outputText.

- h:outputFormat is used to display formatted output
- h:outputLabel is used to display a read-only label
- h:outputLink is used to display a hyperlink
- h:outputText is used to display a one-line text string

8.1.5 Using Command Component Tags

The command component tags are used to perform actions and navigation. h:commandButton tag is used to add a button component to the JSF page and h:commandLink tag adds a hyperlink to the JSF page.

Apart from the attributes such as binding, id, value, and so on which are added as attributes to the command component tags, an action and actionListener can also be added to the command component tags.

The action attribute can present a logical outcome or a reference to a bean which will result in a logical outcome. This outcome will determine the page to be accessed when the command component is activated.

The actionListener attribute refers to the bean method or listener which will process the component action.

8.1.6 Using Data Bound Table Components

The h:datatable component is used to add relational data to the Web page. This component enables binding to a collection of data objects.

The value attribute of the h:datatable tag is used to bind relational data to the component. The data object which can be added to the data table component can be a list of beans, array of beans, a single bean, javax.faces.model.DataModel object, java.sql.ResultSet object, javax.servlet.jsp.jstl.sql.Result object, or javax.sql.Rowset object.

8.1.7 Using Core Tags

Apart from the tags implementing the HTML functionality, JSF has a set of core tags used to perform core actions. Following are the categories of core tags supported by JSF:

- **Event handling core tags** – actionListener, phaseListener, setPropertyActionListener, valueChangeListener are the event listening core tags
- **Data Conversion core tags** – converter, converterDateTime, convertNumber are data conversion core tags

- **Facet Core tags** - Facet represents a named section within a container, facet is created through facet tag and the data pertaining to the facet is represented through the metadata tag
 - **Core tags used to represent items in a List** – f:selectItem and f:selectItems tags are used to represent items in a list
 - **Validator core tags** – These tags are used to validate input values or other values in JSF pages
 - Miscellaneous core tags
- All these core tags are inserted in the page definition with a prefix 'f'.

8.2 Using Converters, Listeners, and Validators

Converters, Listeners, and Validators are also components of JSF pages which process the input data of the form. Converters are used to convert the input data of the components. Listeners capture the events occurring in the page and perform actions according to the events. Validators are used to validate the data received from the input components.

8.2.1 Using Standard Converters

The JSF standard implementation provides a set of Converter implementations that can be used to convert component data. As discussed in earlier sections, the application data is represented in two views – a presentation view and a model view.

The presentation view uses the data format which can be used in the Web pages and the model view uses data format which can be operated by the application beans. The data has to be converted between these two formats while executing the application. The standard converter implementations present in javax.faces.Convert can be used to perform required conversions in the application. Each converter is associated with an error message. If a converter is unable to convert the input value of the component, then it returns the error message. Following are the converter implementations present in javax.faces.Convert package.

- BigDecimalConverter
- BigIntegerConverter
- BooleanConverter
- ByteConverter
- CharacterConverter
- DateTimeConverter
- DoubleConverter
- EnumConverter

- FloatConverter
- IntegerConverter
- LongConverter
- NumberConverter
- ShortConverter

In order to use a converter with a component, the converter has to be registered with the component first. Following are the ways in which the converter can be registered with a component:

- The converter tag is nested within the component tag. This is generally used for date and time converters.
- Bind the value attribute of the component with the managed bean property which in turn has the converter.
- Add a converter attribute to the component tag and refer to the converter from the converter attribute.
- A converter tag can be nested in a component tag. Then, one can use either a converter tag's converterId attribute or its binding attribute to reference the converter.

→ Using DateTimeConverter

The DateTimeConverter is used to convert the input text of a component into java.util.Date type. This conversion is possible by including a convertDateTime tag in the component tag as shown in the following code:

```
<h:outputText value="#{ShoppingBean.shipDate}">
<f:convertDateTime type="date" dateStyle="full" />
</h:outputText>
```

The value attribute is set to java.util.Date type and the date format is converted according to the tag definition.

The tag will display the date in the following format:

Saturday, September 21, 2013

The DateTimeConverter can have the following attributes:

- **binding** – It is used to bind the converter to a bean.
- **dateStyle** – This attribute defines the date format according to java.text.DateFormat.
- **for** – This attribute is used in case of composite components, in this case to bind the dateTimeConverter to certain component in the group.

- **locale** – This attribute is used to represent a date format which is part of predefined set of date styles.
- **pattern** – This attribute determines the formatting pattern of the date and time.
- **timeStyle** – This attribute is also used to define the format of the date and time.
- **timeZone** – This attribute of the DateTimeConverter defines the time zone in which the application is running.
- **type** – This attribute defines whether a string value has date, time, or both date and time.

→ Using NumberConverter

Numberconverter tag is used to transform the data from the input component to java.lang.Number type object. Similar to DateTimeConverter, NumberConverter also has several attributes which define the parameters of the conversion. Few of these attributes are binding, currencyCode, currencySymbol, for, groupingUsed, and so on.

8.2.2 Registering Listeners on Components

Listener objects are used to capture events in the Web page. These listeners can be implemented as classes or as managed bean methods. The listener method is referenced through actionListener or valueChangeListener if it is a managed bean method. If it is implemented as a class, then the listener is referenced through the tag actionListener or through tag valueChangeListener.

→ Registering a ValueChangeListener on a Component

The ValueChangeListener can be registered on a component which implements EditableValueHolder, by nesting a f:valueChangeListener tag within the component's tag on the JSF page.

The ValueChangeListener has two attributes namely, type and binding. The type attribute specifies the implementation of ValueChangeListener along with the tag, whereas the binding attribute binds the component to a managed bean.

In Code Snippet 1, an example is shown on the usage of the type attribute

Code Snippet 1:

```
<h:inputText id="name" size="30" value="#{ShoppingBean.name}"
required="true">
<f:valueChangeListener
type="Dealsstore.listeners.UserNameChanged" />
</h:inputText>
```

In the given code, the listener on the inputText component is a valueChangeListener. The Listener object is a user defined listener which keeps track of the changed name value in the inputText field and UserNameChanged event in the given hierarchy.

→ **Registering an ActionListener on a Component**

ActionListeners can be registered on command components such as buttons. The ActionListener can be registered by including a tag in the command component tag as in case of ValueChangeListener. The actionListener also has type and binding attributes which are used to reference the user defined listener implementations.

The core tag library also has an setActionListener tag which can be used to register an actionListener with an ActionSource instance.

8.2.3 Using Standard Validators

JSF provides a standard set of validators that can be used by the developers to validate component's data.

Following is the list of the standard validators present in JSF:

- **BeanValidator class** – It is a class used to register a bean validator for the component; the corresponding tag is validateBean.
- **DoubleRangeValidator class** – It is a class used to validate whether a certain float value of a component is within the specified range or not. The tag used for this validator is validateDoubleRange.
- **LengthValidator class** – This class is used to validate string inputs from the components. The validation is based on whether the string length is within the expected limit or not. The tag for the validator is LengthValidator.
- **LongRangeValidator** – This class is used to define validators which will check the range of floating type value. The tag used for this validator is validateLongRange.
- **RegexValidator** – This class is used to define a validator of an input component against a regular expression from the java.util.regex package. The tag used for this validator is validateRegEx.
- **RequiredValidator** – This class is used to define a validator which checks that the current value is not empty. This is applied on a component which accepts input. The tag for this validator is validateRequired.

All the standard validator classes implement the Validator interface. User-defined validators can also be written in an application implementing the Validator interface. These validators can be associated with standard error messages which can be displayed if the validation fails on the component data.

The validations of the input data can also happen through bean validation, where the developer can specify bean validation constraints on the managed bean properties. Upon specifying the constraints on the managed bean properties, the constraints are automatically specified on the corresponding user interface fields.

→ Validating a Component's Value

Similar to converters, the validators must be registered on the components where validation is required. One can register the validators with the components in one of the following ways:

- Nest the validator tag within the corresponding component tag.
- Refer to a method which performs the validation as an attribute to the component tag.
- Nest the validator tag within the component tag and use the validatorId or binding attribute to refer to the validator.

Validation can be performed only on those components which implement EditableValueHolder.

Following is an example of using a validator tag:

```
<h:inputText id="check" size="4" value="#{cartItems.quantity}">
<f:validateLongRange minimum="1"/>
</h:inputText>
```

In the example, the validator tag is nested within the inputText tag on which the validation is to be performed. This tag is responsible for validating the value entered in the textbox and imposes a condition that the minimum value should be 1. The attributes of all the validator tags accept Expression Language (EL) expressions.

8.2.4 Referencing a Managed Bean Method

Components can be associated with managed beans through attributes. Following are attributes which enable referencing managed bean methods that can perform certain functions on the component:

- **action** – The action attribute refers to a managed bean. This attribute is used on command components or on components which have certain event listeners registered.
- **actionListener** – This attribute refers to a managed bean which can respond to event actions.
- **validator** – This attribute refers to a managed bean method which can validate the component value.
- **valueChangeListener** – This attribute refers to a managed bean method, which can handle the change of values in various components.

Components implementing ActionSource interface can use the action and actionListener attributes and components implementing EditableValueHolder interface can use the validator and valueChangeListener attributes.

The managed bean has to be defined to handle all the required validations, which can later be used in attributes of the components.

8.3 Developing Applications with JSF Technology

Managed beans are bean classes which add functionality to the JSF page. They comprise getter and setter methods for the properties defined in the bean and also business logic relevant to the application.

8.3.1 Creating a Managed Bean

Managed bean in Netbeans can be created by right-clicking the JSF project and adding the managed bean element to the project. If the developer is defining the class manually, then it should be a class defined with default constructor, a set of properties associated with the editable text area, and a set of accessor and mutator methods for each component. The managed bean properties can be bound to any one of the following:

- Component value
- Component instance
- Converted value
- Listener instance
- Validator instance

A managed bean component can validate component's data, handle an event fired by a component, and define page navigation.

→ Using Expression Language to Reference Managed Beans

Expression language can be used with the component tags to reference managed beans. Expression language offers features such as deferred evaluation of expressions, use value expression to both read and write data, and method expressions.

There are various phases in the application lifecycle of a JSF application. It will be optimal to defer the evaluation of an expression until the validation and conversion phases are complete. Code Snippet 2 demonstrates how the values entered in the Web page are associated with the beans.

Code Snippet 2:

```
<h:inputText id="payamount" size="30" value="#{ShoppingBean.price}"
required="true">
converter = "#shoppingBean.IntegerConverter"
validator="#{shoppingBean.validateNumberRange}"
</h:inputText>
```

In Code Snippet 2, the input text area is an editable place holder which expects numerical data. First, it converts the input text to integer and then, checks the value of the integer. According to the application domain, since the entered value is a currency, it should always be positive. This can be ensured by defining the minimum attribute of the validator.

All JavaServer Pages attributes accept EL expressions. Apart from referencing bean properties, these attribute expressions can also refer to implicit objects, lists, arrays, and maps.

8.3.2 Writing Bean Properties

A component tag value can be bound with the managed bean property through the value attribute and the component instance can be bound to the managed bean through binding attribute.

The ‘binding’ attribute of all the converters, listeners, and validators can be used to bind them to the managed bean. To successfully bind the converters, listeners, and validators to a component, the managed bean property must accept and return same type of converter, listener, and validator object. Similarly, to bind the component values with the managed bean properties the corresponding types of the data must match.

→ Writing Properties Bound to Component Values

When the developer intends to write the component value as a bean property, then the data type of the bean property and the component value must be same.

Following are the component classes supported by javax.faces.components and acceptable types of their values.

- The components of class UIInput, UIOutput, UISelectItem, and UISelectOne can have data of any primitive Java data type for which a converter is defined in the javax.faces.convert.Converter class.

In a JSF page they are represented through the tags <h:input> and <h:output>, and the values are bound through the value attribute.

- The component class UIData reads a group of data items from the component, it can have data of types, array of beans, List of beans, single bean, java.sql.ResultSet, javax.servlet.jsp.jstl.sql.Result, and javax.sql.RowSet.

In JSF the UIData component is represented through the tag <h:dataTable>.

- The components of class UISelectBoolean can have variables of type class Boolean or the primitive type boolean.

In a JSF page, the UISelectBoolean component is represented through the tag <h:selectBooleanCheckbox>.

- The components of class UISelectItem can be bound to properties of type java.lang.String, Collection, Array, and Map.
- The components of class UISelectMany can be bound to properties of type array or List.

The corresponding JSF tag for the UISelectMany component is <h:selectMany>.

→ **Writing Properties Bound to Component Instances**

When a property is bound to component instance, it will return a component instance instead of values. For instance, in a travel portal, the payment options on the Web page may not be rendered initially when the page is loaded. When a button is clicked, a component event listener is associated with the managed bean. The event listener component invokes the managed bean, which may in turn render the payment options component, which is an aggregation of other components. Code Snippet 3 shows the example of binding with a component instance.

Code Snippet 3:

```
<h:selectBooleanCheckbox id="payment"
binding="#{payBean.paymentGateway}" />
<h:outputLabel for="payment" rendered="false"
binding="#{payBean.cardDetails}" />
</h:outputLabel>
```

In Code Snippet 3, a checkbox is rendered which is connected to the paymentGateway object in the managed bean. The card details are initially not rendered but they are rendered after the checkbox instance representing the paymentGateway object is selected. Once the component is rendered after checking the checkbox, the card details label is rendered. The for attribute of the outputLabel tag is bound with the checkbox control using the id attribute of the checkbox.

→ **Binding Properties to Converters, Listeners, and Validators**

The converter, listener, and validator implementations are bound to managed bean attributes. This binding enables the managed bean to manipulate their implementations and add additional functionality to them.

8.3.3 Writing Managed Bean Methods

Similar to any other class, methods can be defined in managed beans also. These methods perform several application specific functions such as processes associated with the page navigation, handling action events, performing validation on the component values, and handling value change events.

It is a good programming practice to implement all the validator, listener, and converter methods in the same managed bean to which the attribute values of the JSF page are bound. This enables safe access to the attributes of the managed bean by the methods of the bean.

→ **Writing a Method to Handle Navigation**

An action method is bound to the action attribute of the component tag. This method should be a public method which accepts no parameters and returns an object which is a logical outcome of the action attribute determining the page that is to be displayed by the navigation system.

The action method typically returns a string outcome. An Enum class can be defined to encapsulate all the possible outcome strings.

→ **Writing a Method to Handle an Action Event**

A method that handles an action event should be public in the managed bean. This method will accept the action event as a parameter and return void. The action method handling an action event is referenced by the component `actionListener` attribute. The action method handling an action event can be implemented only by components that implement `javax.faces.component.ActionSource`.

→ **Writing a Method to Perform Validation**

A method can be included in the managed bean to validate input instead of implementing the `javax.faces.validator.Validator`. A managed bean method which performs the validation must accept the `javax.faces.context.FacesContext`, this object represents the component whose data must be validated.

The validator method of the managed bean is referred through the `validate` attribute of the component. Only the components of the class `UIInput` can be validated.

→ **Writing a Method to Handle a Value Change Event**

Managed beans can handle the value change event in the components through a method which accepts a value change event and returns void. The method handling a value change should be public. The method is referenced from a `valueChangeListener` attribute of the component.

8.4 Configuring JSF Applications

For large applications, configuration of the application is essential to ensure appropriate functioning of the application. Following are the tasks required to configure the applications:

- Registering the managed beans with the application. This ensures that all the classes and components of the application can access the managed beans.
- Configuration of the managed beans so that the beans are instantiated with appropriate values whenever referenced.
- Defining the navigation rules among the pages of the application.
- Packaging the application to include all the resource files in the application.

JSF provides a portable configuration document which is an XML file to configure the resources of the application. Each application may have more than one application resource configuration files. The `faces-config.xml` file is one such configuration file used in JSF applications.

Each configuration file must include the following information in the given order:

- XML version number with an encoding attribute.
- A `faces-config` tag with all the declarations of the tag libraries and name spaces used in the application.

- The application can have more than one configuration file which can be located in any one of the following ways:

- Locating a resource /META-INF/faces-config.xml in the application's /WEB-INF/lib/ directory.
- A context initialization parameter javax.faces.application.CONFIG_FILES in the Web deployment descriptor specifies the path to the multiple configuration files of the Web application.
- A resource named faces-config.xml is the configuration file for simple Web applications.

An instance of javax.faces.application.Application class is created for all the Web applications in JSF. This object helps in accessing all the resources pertaining to the application.

As the configuration information of the application is stored in multiple configuration files, it is important to define an order in which these files are accessed. This order of access is defined by an ordering XML element. The ordering of the configuration files can be absolute ordering or relative ordering.

If the ordering is an absolute ordering then the files listed are processed in the order specified. A relative ordering element has sub elements before and after. The configuration files are specified with these sub elements to determine the order accessing the configuration files.

8.5 Using Validators and Converters in a JSF Application

To use validators and converters in a JSF page, you can apply the code given in Code Snippet 4. In Code Snippet 4, two validators have been defined each on the Product textbox and Cost textbox. The product name cannot be less than 4 characters and the cost of the product cannot be less than 200.

A standard converter is also used to convert the cost of the product into a number with precision of two digits after the decimal point.

Code Snippet 4:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
    <h:head>
        <title>Purchase form</title>
    </h:head>
    <h:body>
        <h:form>
            <f:view>
                <h:outputLabel for="Name" value="ProductName:"/>
                <h:inputText id="Name" label="ProductName" required="true" >
                    <f:validateLength minimum="4"></f:validateLength>
                </h:inputText>
            </f:view>
        </h:form>
    </h:body>
</html>
```

```

<h:message for="ProductName" />
<p></p>
<h:outputLabel for="cost" value="Cost:"/>
<h:inputText id="cost" label="Cost" size="2" >
    <f:validateLongRange minimum="200" />
    <f:convertNumber maxFractionDigits ="2" />
</h:inputText>
<p></p>
<h:commandButton id="register" value="Register"
    action="confirm" />
</f:view>
</h:form>
</h:body>
</html>

```

Figure 8.1 shows the output page when the validators on the page come into action and find the data in the page elements is not according to what is prescribed by the validator.

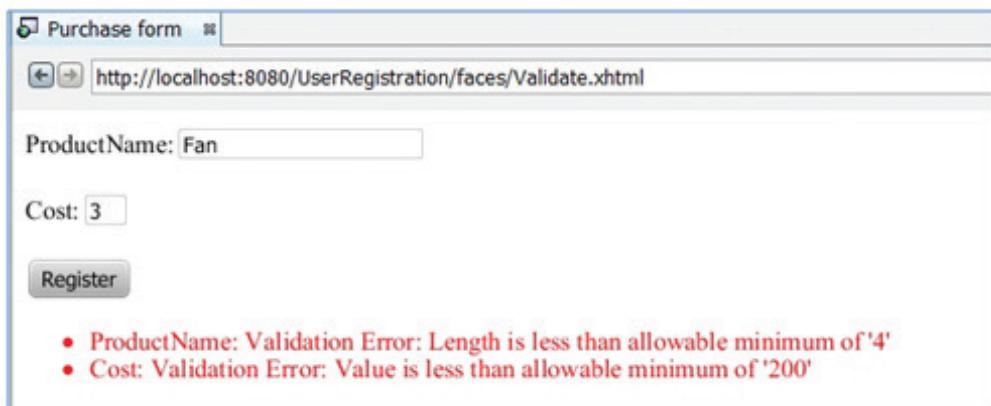


Figure 8.1: Validators in JSF Page

→ Converter Implementation through Managed Beans

Here the application accepts the value in rupees and converts the currency to dollars.

Code Snippet 5 shows the code for the index page where the application execution begins.

Code Snippet 5:

```

<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"

      <h:head>
      </h:head>

```

```

<h:body>
    <h:form>
        <h3> Currency Converter </h3>
        <p></p>
        <h:outputLabel for="rupees" value="Amount in Rupees:"/>
        <h:inputText id="rupees" value="#{currencyBean.rupees}" />
        <h:commandButton id="submit" value="Submit" action =
    "response"/>

    </h:form>
</h:body>
</html>

```

Code Snippet 6 shows the managed bean associated with the JSF page.

Code Snippet 6:

```

package converter;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
@ManagedBean (name = "currencyBean")
@SessionScoped
public class CurrencyBean {

    int rupees;
    int dollars;
    public CurrencyBean() {
    }

    public int getRupees() {
        return rupees;
    }
    public void setRupees(int rupees) {
        this.rupees = rupees;
    }
    public int getDollars() {
        this.dollars = ((this.rupees)*60);
        return dollars;
    }
    public void setDollars(int dollars) {
        this.dollars = dollars;
    }
}

```

In Code Snippet 6, two instance variables rupees and dollars are given whose values are encapsulated through getter and setter methods. These values are used in the Web pages.

Code Snippet 7 shows the code for the response page.

Code Snippet 7:

```

<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <h:head>
    </h:head>
    <h:body>
        <h3> Currency Converter </h3>
        <h:outputText id="output" value="The converted value is:
$"/>
        <h:outputText id="out" value="#{currencyBean.dollars}" />
    </h:body>
</html>

```

In Code Snippet 7, the response page displays the output of the conversion by accessing the dollars attribute of the currencyBean, which is a managed bean. Figure 8.2 shows the Web page when the code given in Code Snippet 5 runs successfully.

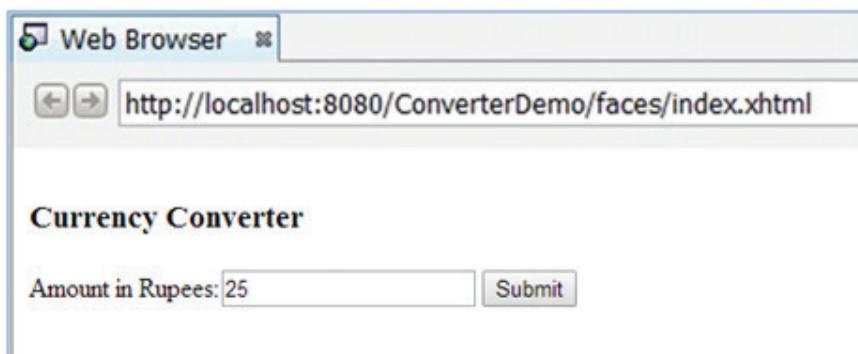


Figure 8.2: Index Page for the Converter

Figure 8.3 shows the converted value on the response.xhtml page after clicking Submit.

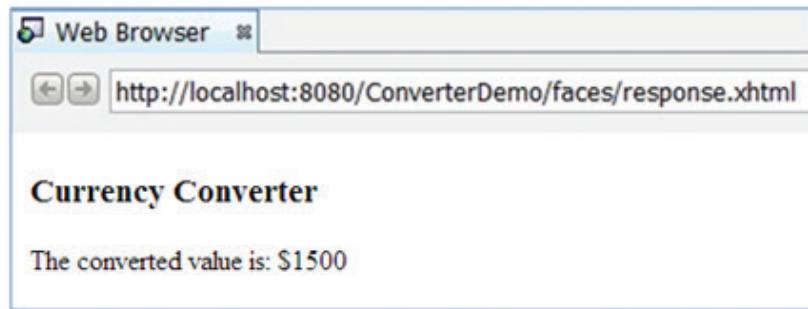


Figure 8.3: Output of Currency Converter

Custom Converters for complicated conversions can be created by implementing the Converter interface.

8.6 Check Your Progress

1. Which of the following components should be present in JSF page declarations?

(A)	Namespace declaration	(C)	URLs of tag libraries
(B)	Tag library declaration	(D)	All of these

2. Which of the following statements about the attributes of the component tags are true?

a.	id attribute is assigned to identify the component		
b.	binding attribute freezes the placement of the component on the form		
c.	immediate attribute can take non-boolean values		
d.	rendered attribute can assume only boolean values		

(A)	a, d	(C)	b, c
(B)	b, c	(D)	b, d

3. Which of the following components are used for password fields?

(A)	h:inputHidden	(C)	H:inputTextArea
(B)	h:inputSecret	(D)	All of these

4. Which of the following statements about listeners are true?

a.	Components implementing ActionSource interface can use action and actionListener attributes.
b.	Components implementing EditableValueHolder interface can use validator and valueChangeListener attributes.
c.	Components implementing EditableValueHolder interface can use action and actionListener attributes.
d.	Components implementing ActionSource interface use validator and valueChangeListener attributes.

(A)	a, b	(C)	c, d
(B)	b, c	(D)	a, d

5. A component tag value can be bound with the managed bean property through the _____ attribute?

(A)	rendered	(C)	value
(B)	binding	(D)	action

8.6.1 Answers

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

Summary

- JSF tags can implement what HTML tags can along with additional functionality through managed beans.
- Each tag has various attributes to define the behavior of the components associated with the tag.
- Apart from the HTML tags JSF also has a set of core tags, which are used for event handling, data conversion, and so on.
- JSF defines a standard set of validators, listeners, and converters which can be used to act along with the components.
- Validators, listeners, and converters can be implemented through managed bean methods.
- Methods and properties of the components can be implemented through managed beans.
- The configuration information of JSF applications is stored in XML files. Each application can have multiple configuration files.



Visit the
Frequently Asked Questions
section @

Session - 9

Facelets

Welcome to the Session, **Facelets**.

This session describes Facelets. It explains the lifecycle of Facelets and the various components and resources used in Facelets. It also describes how to use expression language with Facelets.

In this Session, you will learn to:

- Define Facelets
- Describe the lifecycle of Facelets
- Describe various components and resources used in Facelets
- Describe various resource library components of Facelets
- Describe how Facelets is compatible with HTML markup
- Explain how to use expression language with Facelets



9.1 Facelets

Facelets is a Web template system which is used to create Web pages with the help of a view declaration language. View declaration language provides a default view handling mechanism for the JSFs. It is an open source technology and is compatible with HTML markup and builds component trees. The components in a JSF are arranged according to a tree structure called component tree. The navigation through the components of JSF is determined by this tree structure. Facelets are independent of containers.

Following are some of the important features offered by JSF:

- Allows specifying the UIComponent trees in separate files
- Provides templating and decorators for the pages
- Supports expression language in the tags and allows build time validation of EL expressions
- Has a Facelet tag library which supports view definition along with the JSF tags
- Is compatible with XHTML and HTML
- Facelet APIs are not dependent on the container
- XML configuration files are not essential

Advantages of Facelets

- Facelets allow the developer to create templates and composite components. This in turn, allows code reuse. Templating is the process where the developer defines the look and feel of the Web application. This skeletal definition of the Web page is known as a template. Developers can reuse these templates and composite components for developing other Web pages. A composite component also comprises other components and tags which can later be reused in the Web pages.
- JSF allows for content reuse through referencing a file or through defining custom tags.
- The compilation time for Facelets is less and it allows compile time validation of EL expressions.
- Facelets are independently rendered through render kit and are compatible with any render kit.

Facelet views are created as XHTML pages. Facelets use XML namespace declarations to support JavaServer Faces tag library mechanism. Table 9.1 depicts the tag libraries which can be supported by the JSF technology and can be used to add components to a Web page.

Tag Library	Prefix	Utility of the Tags
JavaServer Faces Facelet tag library	ui:	Tags are used for templating the Web pages
JavaServer Faces HTML tag library	h:	JavaServerFaces component tags for all UIComponent objects
JavaServer Faces Core tag library	f:	This library comprises tags for JSF custom actions. These tags are independent of any render kit
PassThrough Elements tag library	p:	Tags to support markup which is HTML friendly
JSTL Core tag library	c:	These are JSTL 1.2 core tags
JSTL Functions tag library	fn:	These are tags used by JSTL functions for version 1.2

Table 9.1: Tag Libraries Supported by JSF Technology

9.2 Lifecycle of a Facelets Application

A Facelets application executes as a part of JavaServer Faces application. The JSF application executes in two phases – execute and render. Following are the steps in the lifecycle of a Facelets application:

1. Whenever a page created through Facelets template is requested, a new component tree is created whose root is javax.faces.component.UIViewRoot object.
2. Based on the component tree defined by the UIViewRoot, the view is populated with the components for rendering. Figure 9.1 shows a sample component tree.

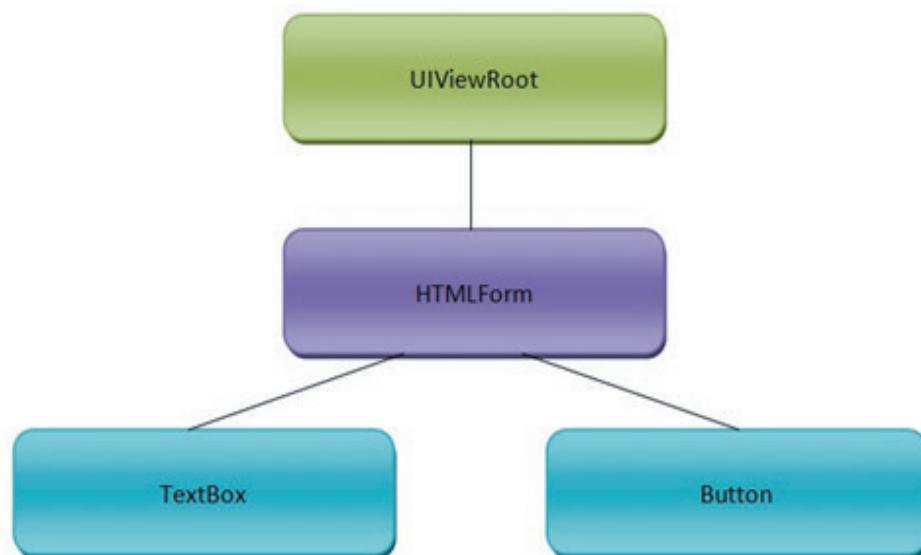


Figure 9.1: Sample Component Tree

3. Based on the client request, certain response is generated by the JSF application. This response is rendered back to the client.
4. The state of the response view is saved for the next request. This state includes the state of input components and the form data.
5. Based on the response received by the client if the client makes another request, then the saved state of the Web page is restored.
6. The process is repeated again for a new request, with intermediate validations as applicable to the request.

9.3 Creating a Simple Facelets Application

A user registration module is used by most of the Web applications which intend to keep track of the information of users who access the portal. The typical flow of data in this scenario is as follows:

1. The user sends details required for the registration as a request to the server.
2. Server invokes servlets or managed beans to handle the request.
3. The managed bean checks with the database whether the requested user name is available for allocation to the current user or not.
4. If the user name can be allotted to the current user, then the registration is confirmed otherwise the application prompts for a new user name and password. This process continues until a valid user name is allotted to the user.
5. The final page is a confirmation page with the details entered in the login page.

When such an application is implemented as a JSF application, then following tasks in the application are handled by Facelets:

- ➔ Generating the views of the application through a component tree
- ➔ Page navigation of the application

9.4 Using Facelets Templates

Template defines a reusable component or format in a generic context. In the context of Web applications, template defines the format of the Web pages. Facelets also support composition of components which is a single logical page view.

Code Snippet 1 depicts the template for header content of a Website for an 'XYZ' bank created in a file named headertemplate.xhtml.

Code Snippet 1:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets">
  <head>
    <title>TODO supply a title</title>
    <meta name="viewport" content="width=device-width"/>
  </head>
  <body>
    <div>
      <ui:composition>
        <header> XYZ Bank</header>
      </ui:composition>
    </div>
  </body>
</html>
```

Code Snippet 1 depicts a template for header defined through Facelets. This template can be used in a Web page.

In Code Snippet 1, the default code is generated when a Web page is created through Facelets. Additionally a ‘ui’ component is added to it. The ‘ui:’ component is added by including the corresponding namespace <http://java.sun.com/jsf/facelets>. Every tag is defined in a namespace. The tag ‘ui’ has component definitions such as ‘composition’. The ‘composition’ tag is an aggregation of different sub-components.

Code Snippet 2 depicts the template holding default footer of the Web page which is stored in a .xhtml file named footertemplate.xhtml.

Code Snippet 2:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets">
  <head>
    <title>TODO supply a title</title>
    <meta name="viewport" content="width=device-width"/>
  </head>
  <body>
    <div><ui:composition>
      <footer>XYZ Bank...Best in class</footer>
    </ui:composition></div>
  </body>
</html>
```

Code Snippet 3 depicts the template created for representing the default content of the Web page, which is stored in a file named contenttemplate.xhtml.

Code Snippet 3:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets">
    <head>
        <title>TODO supply a title</title>
        <meta name="viewport" content="width=device-width"/>
    </head>
    <body>
        <div>
            <ui:composition>
                <p>Welcome to XYZ Bank</p>
                <p> . </p>
                <p> . </p>
            </ui:composition>
        </div>
    </body>
</html>
```

Code Snippet 4 depicts the usage of templates created in Code Snippets 1, 2, and 3.

Code Snippet 4:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.
w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:ui="http://java.sun.com/jsf/facelets">
    <h:head>
        <title>XYZ Bank</title>
    </h:head>
    <h:body>
        <ui:insert name="header" >
            <ui:include src="header.xhtml" />
        </ui:insert>
        <ui:insert name="content" >
            <ui:include src="contenttemplate.xhtml" />
        </ui:insert>
        <ui:insert name="footer" >
            <ui:include src="footertemplate.xhtml" />
        </ui:insert>
    </h:body>
</html>
```

Earlier in Code Snippets 1, 2, and 3 three different templates have been defined. In the Web page created through Code Snippet 4, the predefined templates are being used to create a new Web page.

The code uses the ‘ui:’ tag with include attribute. The include attribute allows the developer to include the templates for the ui component. The templates for header, content, and footer are included in the current Web page through the <ui: include src....> tag. Figure 9.2 shows the Web page generated by executing the code in Code Snippet 4.

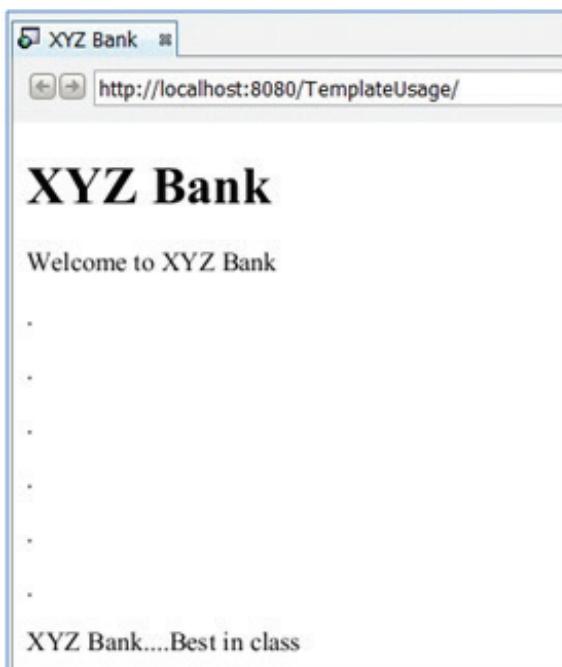


Figure 9.2: Web Page Generated from Templates

Templating helps in maintaining a standard look across multiple Web pages in an application. The Web pages which are created from the template page are known as client pages. Facelets define a set of templating tags which are used to define templates. Table 9.2 depicts the tags used for creating templates.

Tag	Description
ui:component	An user interface component is created and added to the component tree. A table in a Web page can be defined through the component tag.
ui:composition	This tag uses a template optionally to define the set of components present in a Web page. All the content outside the composition tag is trimmed while rendering.
ui:debug	This tag is used to define and add a debug component to the Web page.
ui:decorate	The utility of this tag is similar to that of ui:composition tag. However, the content outside this tag is not trimmed.
ui:define	This tag defines the content which is expected to be inserted into the page through a template.
ui:include	This tag is used to reuse code in a Web page from other existing Web pages.
ui:insert	This tag is used to insert content into the Web page.

Tag	Description
ui:param	This tag is used to add parameters to an included file.
ui:repeat	This tag is equivalent to loop constructs in Java. As in loop constructs the repeat tag also repeats a block of statements.
ui:remove	This tag is used to remove content from the Web page.

Table 9.2: Templating Tags

9.5 Composite Components

Composite component is defined like a template with entities such as validators, converters, and tags put together. All these can further be used as a component. It can be generalized as an aggregation of various Facelet components which are made to function as a single component. This composite component can be reused, with the listeners, validators, and their responses defined.

Similar to a template, any XHTML page with its components and markup tags can be treated as a composite component. In order to define a composite component, a composite namespace has to be declared in the XHTML file. All the composite components are defined in this namespace.

Following are the steps for creating a composite component:

1. Composite components are defined through the components in the namespace as follows:

```
xmlns:composite="http://java.sun.com/jsf/composite"
```

Include the namespace declaration in the xhtml file in which the composite components are being defined for the application.

2. Every composite component has an interface and implementation associated with it. The definition of the interface and implementation is done through the tags composite:interface, composite:attribute, and composite:implementation.

The interface defines the configurable values which can be used by the developer to use the component being defined. The implementation declares all the XHTML mark-ups which are part of the composite component.

3. The .xhtml file in which the composite components are defined is to be placed in the resources folder of the project.

Table 9.3 depicts the tags that can be used to define a composite component.

Tag	Description
composite:interface	This tag is used to determine the variables which are exposed to the user to enable usage of the composite component.
composite:implementation	If there is an interface element defined in the composite:interface, then there should be a corresponding implementation element to be defined.

Tag	Description
composite:attribute	This tag defines the attributes of the composite components. These attributes may assume a value when the composite component is created.
composite:insertChildren	Composite attributes allow for the definition of component trees within the composite component. This tag indicates the location where the component tree has to be added.
composite:ValueHolder	This tag allows accommodating a ValueHolder object within the component.
composite:EditableValueHolder	This tag allows for holding an EditableValueHolder object.
composite:ActionSource	This tag accommodates an ActionSource object in the component.

Table 9.3: Tags to Define a Composite Component

Code Snippet 5 depicts the code for creating a composite component.

Code Snippet 5:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:composite="http://java.sun.com/jsf/composite"
      >
<composite:interface>
    <composite:attribute name="usernameLabel" />
    <composite:attribute name="usernameValue" />
</composite:interface>
<composite:implementation>
<h:form>
    #{cc.attrs.usernameLabel} : <h:inputText id="username" value="#{cc.
    attrs.usernameValue}" />
    <h:commandButton value="Submit" action="response"></h:commandButton>
</h:form>
</composite:implementation>
</html>
```

In Code Snippet 5, a composite component is created which has 'username' label and a textarea which accepts text input. There are two components to define a composite component – composite interface and composite implementation. Composite interface comprises the user interface components and the composite implementation defines how various user interface components are linked together.

The #{cc.attrs....} is a variable component which can be given a user defined value. In Code Snippet 6, values of these variables are substituted according to the usage of the component.

This code is created in a file named loginComp.xhtml and stored in resources directory in the Web Pages directory of the project. The hierarchy of directories is as shown in figure 9.3.

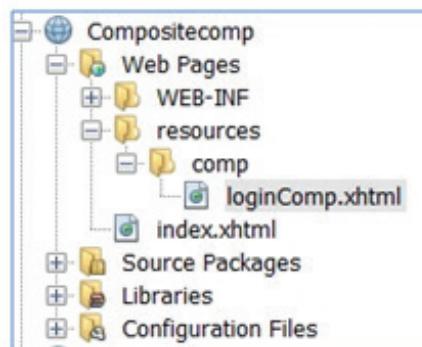


Figure 9.3: Directory Location of Composite Components

The composite component can later be used in another Web page. Here, this component is used in the index.html of the project. Code Snippet 6 depicts the code for the same.

Code Snippet 6:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:j="http://java.sun.com/jsf/composite/comp">
<h:form>
    <j:loginComp usernameLabel="Enter User Name: "
                 usernameValue="#{userData.name}" />
</h:form>
</html>
```

In Code Snippet 6, a namespace and a custom component tag are defined. The prefix for the namespace is 'j:' and the namespace path has the directory name in which the definition of the custom tag is present. The composite component is defined in the comp directory therefore, the namespace for the 'j:' tag has been defined accordingly. The value accepted in the text area is bound to the bean property userData.name. Code snippet 7 depicts the code for the bean class UserData.

Code Snippet 7:

```
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean
@SessionScoped
public class UserData {

    private String name;
    public UserData() {
    }
```

```

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

}

```

Code Snippet 7 depicts the bean class which has the attribute name to which the variables in the Web pages are bound.

Code Snippet 8 shows the code for the response page.

Code Snippet 8:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <head>
        <title>Response</title>
        <meta name="viewport" content="width=device-width"/>
    </head>
    <body>
        <h:outputText id ="output" value="The name entered is "/>
        <h:outputText id ="output" value="#{userData.name}" />
    </body>
</html>

```

In Code Snippet 8, the page displays the name entered in the index page. The name is displayed as part of outputText tag.

On deploying the application the Web browser displays the page as shown in figure 9.4.

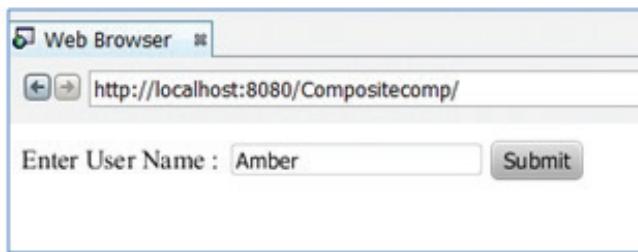


Figure 9.4: Composite Component Usage in a Web Page

All the three components in the Web page are included through a single component loginComp.

On submitting the values, the response page is as shown in figure 9.5.

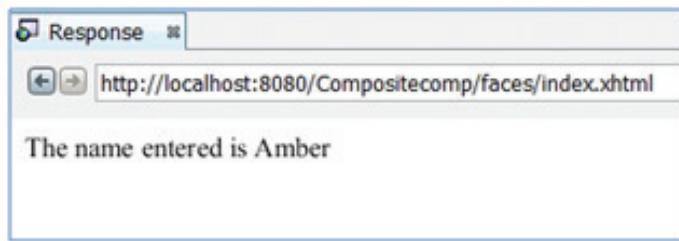


Figure 9.5: Response Page

9.6 Web Resources

Web resources refer to those software elements of the application which are required for proper rendering of the Web pages. These elements include images, scripts, stylesheets, and so on. There are definite locations where these resources are placed in the application so that they can be accessed while execution of the application.

- Resources of an application can be packaged in a directory. They must be located as a subdirectory of the /resources directory of the application.
- If a resource is packaged in the class path of the Web application, then it must be located as a subdirectory of META-INF/resources of the application.

Every resource of the application is allotted a resource identifier, which is a unique string and identifies the resource uniquely. Following is the format of the resource identifier:

```
[locale-prefix/] [library-name/] [library-version/] resource-name [/resource-version]
```

All the elements in the resource identifier except the resource name are optional. All the resources can be considered as a library location. This library may comprise a template, image, and/or a composite component used in the application.

9.7 Relocatable Resources

Relocatable resources in a Web page are those which are referenced in one section of the Web page but are rendered in another part of the Web page. In order to achieve this, the target attribute of the tag is used. The target attribute can assume any one of the three values – head, form, or body.

Relocatable resources can be used for composite components that use stylesheets or the composite components that use JavaScript.

9.8 Resource Library Contracts

Resource library contracts are used to define the look of the Web pages. It is possible in JSF that a subset of pages in the application has a certain look and feel and another set of Web pages have a different appearance. This is achieved by defining the contracts in the application.

Consider a Web application packaged as a WAR file. All the resource library contracts are defined in the contracts folder of the application. Each contract folder has a CSS file and a template file which together define the appearance of the Web pages. Each template has insertion points defined for the resources. The template, the insertion points, and CSS files together form the resource library contract.

Resource library contracts can be defined according to the pattern as shown in figure 9.6.



Figure 9.6: Format of Resource Library Contracts

The resources are mapped onto the Web pages in the faces-config.xml. Code Snippet 9 depicts the format of mapping the contracts to the Web pages.

Code Snippet 9:

```

<application>
  <resource-library-contracts>
    <contract-mapping>
      <url-pattern> /RegisteredUsers/* </url-pattern>
      <contracts>colored</contracts>
    </contract-mapping>
    <contract-mapping>
      <url-pattern> * </url-pattern>
      <contracts>colorless</contracts>
    </contract-mapping>
  </resource-library-contracts>
</application>
  
```

In Code Snippet 9, a contract is defined for certain set of users designated as RegisteredUsers specified in the tag URL pattern. The display theme is colored. For all other users specified through the regular expression ‘*’, the display theme is colorless. The display theme in both the cases is specified through contracts tag. The URL for this application is given with the url-pattern tag.

9.9 HTML5 Friendly Markup

JSF allows the user to use HTML components in the Web pages or create custom components that suit the application. HTML5 offers a wide variety of elements and attributes which can be used in the Web pages. JSF allows using the pre-existing HTML5 components rather than defining new components. JavaServer Faces attributes can in turn be used in HTML5 elements.

The HTML5 support to JSF can be categorized into the following:

- ➔ Pass-through elements
- ➔ Pass-through attributes

The behavior of the HTML5 elements used in Facelets is determined by the Facelets page author but not by the HTML components. Pass-through elements are rendered as Facelet tags and pass-through attributes are rendered through the browser directly without any intermediate interpretation. All the pass-through elements and attributes are supported through the namespace, xmlns:p="http://xmlns.jcp.org/jsf/passthrough"

Pass-through elements enable the usage of HTML5 tags but they are treated as JSF components. These components are associated with a server side UIComponent instance. Pass-through elements imply that the HTML5 elements are rendered as JSF components. An HTML 5 element can be rendered as a JSF component by including at least one attribute of jsf: namespace.

In general, a non-JSF element can be made a pass-through element by using at least one of the attributes from the namespace http://xmlns.jcp.org/jsf.

For a non JSF element to be made as pass-through element, the namespace has to be declared in the beginning of the .xhtml page along with other namespaces. The pass-through HTML elements are rendered as Facelets tags through a combination of HTML tags and an identifying jsf: attribute. Table 9.4 depicts some of the pass-through elements of HTML5 which are represented as a combination of element and jsf attribute.

HTML Element	jsf Attribute	Facelet Tag
A	jsf:action or jsf:actionListener	h:commandLink
A	jsf:value	h:outputLink
A	jsf:outcome	h:link
Body		h:body
Button	jsf:outcome	h:button
button		h:commandButton
Input	type = "button"	h:commandButton
Input	type = "checkbox"	h:selectBooleanCheckBox
Input	type = "*"	h:inputText

Table 9.4: Pass-through Elements for HTML 5 Elements

Pass-through Attributes

Pass-through attributes allow passing JavaServer Faces attributes to the browser without any intermediate interpretation. A pass-through attribute can be specified as a JavaServer Faces UIComponent.

Pass-through attributes can be specified in any of the following ways:

- By using the JavaServer Faces namespace for the pass-through attributes. Following is the namespace used which has to be specified in the JSF page:

```
xmlns:p=http://xmlns.jcp.org/jsf/passthrough
```

After defining the namespace, the pass-through attributes are added to the component with the prefix 'p:'. Following is an example of the pass-through attributes:

```
<h:inputText p:type = "number" p:min = "1" p:max ="100">
```

In the given code, the attributes Type, min, and max are specified by prefixing p: to the inputText element.

- Pass-through attributes can also be added through f:passThroughAttribute tag. This tag is used if there is only a single attribute to be passed as pass-through attribute to the element.
- To pass a group of non-JavaServer Faces attributes, f:passThroughAttributes tag can be used within the component tag.
- A set of values can be passed as map declarations where each entry of the Map object is name-value pair.

9.10 Check Your Progress

1. Which of the following statements about Facelets are true?

a.	Facelets are dependent on the Web containers		
b.	Facelets declare views		
c.	XML configuration files are essential for rendering Facelet views		
d.	It is compatible with XHTML but not HTML		

(A)	a, b	(C)	c, d
(B)	b, c	(D)	a, d

2. Match the following JSF tag libraries with the corresponding utility.

	Tag Library		Utility
a.	JavaServer Faces core tag library	1.	Provides templating
b.	JSF Facelets tag library	2.	Provides tags for all UIComponent objects
c.	Passthrough attributes tag library	3.	Defines custom actions independent of render kit
d.	JavaServer Faces HTML tag library	4.	Supports HTML5 attributes

(A)	a-4, b-1, c-3, d-2	(C)	a-3, b-1, c-4, d-2
(B)	a-3, b-4, c-2, d-1	(D)	a-2, b-3, c-4, d-1

3. All the JSF views are rendered through _____.

(A)	Template	(C)	Component tree
(B)	Facelet	(D)	XML page

4. Page navigation rules for Facelets are defined in which of the following:

(A)	faces-config.xml	(C)	Managed bean
(B)	index.xhtml	(D)	None of these

5. Which of the following statements about composite components are true?

(A)	Every composite component must have either an interface or implementation	(C)	Composite components are not reusable components
(B)	They are built by including the namespace, xmlns:composite=http://xmlns.jcp.org/jsf/composite	(D)	None of these

9.10.1 Answers

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

Summary

- Facelets is a view declaration language used to define views for a Web application.
- It can also define reusable templates which can be used among multiple Web pages.
- Facelets have a set of core tags and can use other tag libraries to define the view.
- Facelets provide for definition of user defined tags, composite components, and templates.
- Web resources are included in the resources directory and the resource library contracts in the contracts directory of the application.
- Through resource library contracts facelets provides a uniform look and feel among the Web pages of the application. It also allows defining the appearance of a subset of pages.
- HTML 5 tags and non JavaServer Faces attributes can be implemented through pass-through elements and pass-through attributes.

GROWTH
RESEARCH
OBSERVATION
UPDATES
PARTICIPATION



Session - 10

Enterprise JavaBeans

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

This session describes Enterprise JavaBeans (EJB) and its usage in enterprise applications. It also describes its variants and how different types of beans can be used for developing Web and enterprise applications. The naming conventions used in Enterprise JavaBeans and the lifecycle of enterprise beans are also explained.

In this Session, you will learn to:

- Explain enterprise beans
- Use variants of enterprise beans
- Describe different ways through which the enterprise beans can be accessed
- Describe the lifecycle of the beans
- Create and deploy a sample enterprise bean using Netbeans IDE



10.1 Enterprise Beans

Enterprise beans are server-side components used in enterprise and Web applications which enable modularized development. The bean components, by default, have the transaction management, scalability, and security authorization implemented and the developer must only develop the business logic of the application.

Different enterprise beans of an application are put together in a container and the container in turn has interfaces with the application server on which they are located. These entity beans also interact with the persistent storage which can be database, a file, or any other application. For instance, consider a case where the business logic based on certain conditions wishes to abort an executing transaction. In this situation, the bean communicates with the container to abort the transaction. The actual process of aborting the transaction is taken care by the EJB container.

EJB has evolved from version 1.0 to the current version 3.2, where the prime focus has been to enable distributed application development. The evolution has also focussed towards making the EJB architecture lightweight and enabling easy deployment of Web services.

Enterprise bean architecture is used for application development when the following features are needed in the application:

- Scalability
- Data and transaction integrity
- When the application will have clients with varying requirements

10.1.1 Types of Beans

There are three types of beans as follows:

- Entity beans (deprecated)
- Session beans
- Message driven beans

In the earlier versions of EJB there were three types of beans – entity beans, session beans and message driven beans. However, in recent versions, the entity beans have been replaced by the usage of Java Persistence API.

Note - Enterprise JavaBeans refer to bean components which can run on multiple address spaces and are implemented by importing the package javax.ejb. However, Java beans refers to beans which run in a single address space and are implemented by importing the package java.beans.

10.2 Session Beans

Session beans are components which are defined to handle a specific client of the application. It is invoked by the client to access the application server and perform various tasks on behalf of the client. The session bean accesses the server and performs required operations by invoking session bean methods. These bean methods invoked by the client communicate with the server and invoke appropriate business components on the application server to service the client request. Session beans are not persistent; they do not survive server crashes.

Consider an example of a bank transaction where the customer logs into his/her account and accesses specific details. The interface invoked by the client to access the bank application server is a session bean. The session bean is active for the duration the client accesses the application. Code Snippet 1 demonstrates the usage of session beans.

Code Snippet 1:

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
public interface TravelAgent extends EJBObject {
    public void reserveTicket(Customer cust, Train no ri, Date journey_date, ResClass c) throws RemoteException;
    public TrainInfo availability(ResClass loc, Date journey_date) throws
        RemoteException;
}
```

Code Snippet 1 shows the interface `TravelAgent` that can make ticket bookings on a train. It has two methods namely, `reserveTicket()` and `availability()`. The method `reserveTicket()` is used to reserve a ticket based on the parameters passed to the method. These parameters might be filled in by the user in a Web form. The `availability()` method returns the information of various trains based on user input for reservation class and journey date which are taken as parameters.

10.2.1 Types of Session Beans

There are three types of session beans as follows:

1. Stateful
2. Stateless
3. Singleton

State of an object is represented by a set of variables. A stateful session bean is an extension to the server. It is created by the client and maintains the state of the current session between the client and the server. The state of these beans is manipulated by the client and they are removed once the interaction between the client and server ends.

Shopping cart of an e-commerce portal is an example of stateful session bean. This bean is created when a user logs into the e-commerce portal for shopping. It keeps track of the items selected by the user as the user browses through the portal. This session bean ends either when the user completes the purchase or logs out of the session without making a purchase or the user accidentally terminates the session with the server.

A developer uses stateful beans in an application when:

- Methods are invoked based on the state of the bean
- Certain information has to remain constant between method invocations
- If the bean has to interact with various components of the application

Stateless bean does not maintain any instance variables of the session, that is, the conversational state of the session is not maintained. The state of the instance variables used during a session is not retained across multiple methods of class. Consider the earlier example of shopping cart.

An e-commerce client may have two methods such as `browse_through_products()` and `purchase()`. A stateful bean is used for a shopping cart which will be updated while executing the method `browse_through_products()`. After the user completes browsing of products the user may or may not purchase the products. If the user calls the method `purchase()` then the state of the cart is passed onto the `purchase()` method to generate the cost of all the products the user intends to buy. This is possible only if a stateful bean is used. In case of stateless beans, the state of the instance variables is not maintained across the method calls.

The bean can be identified as stateful or stateless with the `@Stateful` and `@Stateless` annotations respectively. Code Snippet 2 shows the code of a stateless bean on the server-side.

Code Snippet 2:

```
package Converter;  
  
import java.lang.Float;  
  
import javax.ejb.Stateless;  
  
import javax.ejb.LocalBean;  
  
@Stateless  
  
@LocalBean  
  
public class ConverterBean {  
  
    private Float euroRate = new Float("83.4093016");  
  
    public Float dollarToEuro(Float dollars) {  
  
        Float result = (dollars * euroRate);  
  
        return result;  
    }  
}
```

Code Snippet 2 has a method which generates the euro equivalent of dollars, given number of dollars as input.

Code Snippet 3 provides the code of a bean on the client-side.

Code Snippet 3:

```
package Converter;

import javax.ejb.Stateless;
import javax.ejb.LocalBean;
import java.lang.Float;
@Stateless
@LocalBean
public class Client1 {
    public static void main(String args[]) {
        ConverterBean conObj = new ConverterBean();
        Float res;
        Float input;
        System.out.println("Enter the value");
        input = Float.parseFloat("20.00");
        input = Float.parseFloat("20.00");
        res = conObj.dollarToEuro(input);
        System.out.println(res);
    }
}
```

Code Snippet 3 is the client code which can reside in a servlet or any other client-side script. The client-side script sends the dollar value as parameter for the dollar to euro conversion and makes a method call. The server-side code executes the method and returns the result.

A developer uses stateless beans in an application when:

- ➔ The state of the bean does not have client specific data
- ➔ The bean in question is performing a generic task, like the one described in Code Snippet 2

Web services when implemented with stateless beans are more efficient.

A singleton session bean is created only once throughout the application and exists throughout the lifecycle of the application. The state of a singleton session bean is expected to be retained across method invocations but not across server crashes or shut down.

A singleton session bean can be used to implement Web services end points. Implementation of an application hit counter can be achieved through a singleton session bean.

A developer can use a singleton session bean when:

- Certain tasks need to be executed when the application starts up and when the application shuts down
- Certain information needs to be shared across methods of the application
- An enterprise bean needs to be accessed by multiple threads concurrently

10.3 Message Driven Beans

Message driven beans process asynchronous messages in an application. These messages are usually Java Message Service (JMS) messages. In an application, the message driven beans are placed in a container which in turn provides the transaction management, security, and concurrency services to the bean. Hence, a message driven bean is a stateless, transaction aware component which exists on the server-side. In order to write message driven beans, the package javax.jms.message must be imported.

Though message driven beans are also stateless like stateless session beans, the clients cannot access message driven beans through interfaces. They can be invoked only through messages. A single message driven bean is capable of processing messages from multiple clients. The container holding the message driven beans can pool messages and process them concurrently.

When a message is received by a message driven bean it invokes the onMessage() method. The received message can be any one of the five types – Stream, Map, Text, Object, or Bytes. The bean maps the message to any one of these types and processes it according to the business logic mentioned in the onMessage() method. The onMessage() method can call other methods or session beans in the course of handling the message.

Session beans can also send and receive messages using JMS but only synchronously. Message driven beans are always used for asynchronous communication and hence, they do not block server resources.

10.4 Accessing Enterprise Beans

Among all the variants of enterprise beans, only session beans can be accessed explicitly as they provide an interface for access. Message driven beans do not have an interface and cannot be invoked explicitly. They only respond to asynchronous JMS messages. The bean access techniques discussed in the following section apply to session beans only.

A client can access session beans in the following two ways:

1. No-interface view
2. Business interface

In no-interface view, the public methods of enterprise bean implementation class are provided to the client. The client can use any of these public methods or any super class methods which are accessible through these public methods. For using no-interface view, the client and target bean should be packaged in the same application.

Business interface comprises the business methods of the enterprise bean. These are context specific methods defined by the developer to implement the business logic of the application.

An enterprise bean can be accessed only through no-interface view or business interface. This isolates the client from the complexity of bean implementation as the client is only aware of the method signature or prototype to be accessed but not with the implementation of these methods. The process of modifying the bean implementation is also simplified due to these interfaces.

10.4.1 Using Session Beans on the Client Side

As Java supports distributed location of various components of application, a mechanism is needed for accessing the components of the application. Various mechanisms used for this purpose are Java Naming and Directory Interface (JNDI), Java annotations, dependency injection, and so on. All these mechanisms provide a reference of the required component to the requesting component.

Dependency injection is the most commonly used technique for accessing the enterprise beans in Web applications, Web services and so on.

JNDI provides explicit look up through a global syntax for Java EE and Java SE components.

Portable JNDI Syntax

JNDI is used for identifying different components of the applications through JNDI names. JNDI defines namespaces and naming conventions so that different components of the application can be identified. It provides a mechanism of binding an object to a name, provides an interface to lookup the directory and locate the required component, and also keeps track of any changes made to the names of the components.

JNDI defines three namespaces as follows:

- ➔ java:global
- ➔ java:module
- ➔ java:app

java:global namespace is used to locate remote enterprise beans through JNDI lookups. Following is the format for accessing an enterprise bean in global namespace:

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

JNDI hierarchically organizes the objects and binds the objects with their respective references in the directory. The application name refers to the name of the application whose enterprise bean is being accessed. This is required if the application is packaged in an Enterprise Archive (EAR) file. The module name is the module within the application to which the bean belongs.

The enterprise bean is accessed through its interface. Access through the global namespace is required only when the beans and clients are remotely located.

If the client and the enterprise bean are collocated, java:module namespace is used. Following is the format of the java:module namespace:

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

In this case the enterprise beans are located in the same module.

An enterprise bean can provide more than one interfaces for access. If the enterprise bean has only one interface, the interface name need not be mentioned explicitly.

In order to access Java enterprise beans located within the same application java:app namespace is used. Following is the format of a java:app address

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

For example, if an enterprise bean, CartBean, is part of Web application ShoppingCart.

- The JNDI name according to the java:global namespace is

```
java:global/ShoppingCart/CartBean.
```

- The module name is ShoppingCart. Within the module namespace, the bean is referred through the portable JNDI name, that is, java:module/CartBean.

- java:app/CartBean will be the name according to java:app namespace.

When an enterprise application is designed and developed, the clients can access the enterprise beans in three modes: remote, local, and Web service. It is important to decide first as to how the various components will be placed on the application infrastructure to get good application performance. This decision of placing the application components is based on the following factors:

- **The nature of design in linking the components** – If the components of the application are tightly coupled, that is, there are strong dependencies among the components then it will be appropriate to provide local access of beans to the clients.

For tightly coupled applications, collocating the bean and the client gives better performance. For instance, consider there is a bean which will process the client request, which in turn calls another bean which is supposed to update the database. In this case, locating these two components on the same machine will prove positive on the performance of the application.

- **The type of clients** - There are different types of clients for an enterprise bean. In certain cases, the enterprise bean may interact with an external client and in other cases an enterprise bean might be invoked by another enterprise bean or a Web component.

It is considered good design if an enterprise bean is accessed by a client remotely. However, in the second case it is a design choice to be made keeping in mind the application performance.

- When the components of the application are distributed over the application infrastructure it becomes mandatory that the enterprise bean provides remote access to the clients.

The design choice of locating the components remotely or locally is to be made considering factors such as network latency and other hardware related issues. The purpose of distributing the components of the application is to improve the overall performance of the application despite of the network delays.

It is possible to give both remote and local access to the enterprise beans by providing appropriate annotations `@Remote` and `@Local` and by defining different interfaces for remote and local access respectively.

10.4.2 Different Types of Clients Accessing Enterprise Beans

There are primarily three types of clients as follows:

1. Local clients
2. Remote clients
3. Web service clients

→ **Local Clients**

A client to an enterprise bean is termed as a local client if it runs in the same application as that of the bean and the absolute location of the enterprise bean is known to the client. The client can be a Web component or another enterprise bean but it must belong to the application whose enterprise bean is being accessed.

The enterprise bean can be accessed through either no-interface view or business interface. The no-interface view provides all the public methods of the bean through which the bean can be accessed.

The business interface defines all the business logic methods and bean lifecycle methods for the enterprise bean. If an annotation specifies remote or local access (`@Remote` and `@Local`) in both the bean class and the bean interface, then the default business interface is local interface.

A local business interface can be defined in one of the following ways:

1. A bean can be accessed in no-interface view by providing annotations as shown in the following example in the enterprise bean implementation class.

```
@Session
```

```
public class Bean_Impl{....}
```

2. A no interface view can also be defined as follows, where the business interface is annotated as local.

```
@LocalBean
```

```
public class Bean_Impl{....}
```

With this declaration all the public methods are accessible by the local clients.

3. By specifying the interface class using the @Local annotation and then implementing the interface through the bean class as follows:

```
@LocalBean(ExampleBean.class)  
Public class Bean_Impl implements ExampleBean{.....}
```

Dependency injection and JNDI look up can also be used by the local clients to get access to the enterprise beans. In order to obtain a reference to the local enterprise bean through dependency injection, the annotation @EJB is used and the local business interface name of the enterprise bean is specified as follows:

```
@EJB  
Business_interface_name name;
```

In order to obtain a reference to the local business interface through JNDI lookup, the lookup() method of javax.naming.InitialContext can be used as follows:

```
Local_Business_interface example = (Local_Business_interface)  
InitialContext.lookup("java:module/Local_Business_interface");
```

Here, the reference of local business interface is being obtained through the lookup() method.

→ Remote Clients

A client of an application is termed to be a remote client if it runs on a different machine and different JVM than that of the enterprise bean. A remote client can be a Web component, an application client, or another bean. However, the location of the enterprise bean it is trying to access is transparent to the client. Remote clients cannot access the enterprise bean through no-interface view therefore, the enterprise bean must have a business interface so that remote clients can access it.

There are two ways of defining the business interface through @Remote annotation:

1. @Remote

```
public interface Interface_Name { ... }
```

The annotation here defines that the interface can be used to invoke remotely located enterprise bean.

2. @Remote(Interface _ Name.class)

```
public class Bean1 implements Int_Name { ... }
```

The annotation @Remote here identifies an interface class name which can be used to access the enterprise bean Bean1.

The business interface accessing an enterprise bean located remotely performs the operations pertaining to the business logic of the application.

Consider an e-commerce portal. The client (user logged into the portal) may access the business interface. The business interface in turn may implement methods such as `confirm_order()`, `process_payment()`, and so on which are executed by the enterprise bean say, `Sell_products`, located remotely.

Client access to the business interface located remotely can also be processed using dependency injection and JNDI lookup as in the case of local clients.

→ Web Services

The platform independence of Java makes it very useful for implementing Web services. Stateless session beans are used for providing Web services. A Web service is implemented through stateless session beans which can be invoked by any client on the Internet which uses standard protocols such as SOAP, HTTP, and WSDL. The client accessing a Web service need not be a Java client. It should run on any hardware.

Message-driven beans cannot be invoked through Web service clients. All the public methods defined in the session bean class can be used by the Web service clients. `@WebMethod` annotation is used to expose the methods of the bean class to the Web service clients and also to customize the methods used by the Web services.

When bean methods are called by remote clients, the remote clients act on a copy of the parameter but not on the actual parameter. However, when the bean methods are invoked by local clients, then the method would act on the actual parameter instead of a copy of the parameter. There are chances that the remote client may change data after sending the parameter to the bean. In order to avoid the effect of change of parameters which are already submitted, the bean methods act on the copy of the parameter.

The remote clients are slower than the local clients due to the network latency in getting response from the bean. Therefore, due care must be taken to pass the parameters to the bean method in a manner that the effect of the network latency is not significant.

10.5 Naming Conventions of Enterprise Beans

An enterprise bean comprises different components that work together to accomplish a task. Following are the different components an Enterprise bean application would have:

- Bean class implements all the business methods expected to be provided by the bean application.

Consider an example of a railway reservation system. The bean class for this will have all the methods pertaining to ticket reservation, such as `reserve_ticket()`, `cancel_ticket()`, `get_status()`, and so on. Code Snippet 4 shows a hypothetical definition of the bean class that implements the `Train_Res` interface.

Code Snippet 4:

```
class Train_ResBean implements Train_Res{
    String reserve_ticket(){
        // implementation of the method
    }
    void cancel_ticket(){
        //implementation of the method
    }
    void get_status(){
        //implementation
    }
}
```

- Business interface declares the business methods of the bean class that it wishes to expose. Code Snippet 5 shows the business interface for the Train_ResBean class.

Code Snippet 5:

```
public interface Train_Res{
    String reserve_ticket();
    void cancel_ticket();
    void get_status();
}
```

Apart from the Bean class and Business interface, there are other classes which are required for implementing the Bean class. These classes are termed as Helper classes.

With different components of the application, it is important to have proper naming convention to access these components efficiently. Table 10.1 shows the convention to be used. (Deployment Descriptor has been referred as DD in the table)

Component	Convention	Example
Enterprise bean name (DD)	<name>EJB	Train_ResEJB
EJB JAR display name (DD)	<name>JAR	Train_ResJAR
Enterprise bean class	<name>Bean	Train_ResBean
Business interface	<name>	Train_Res

Table 10.1: Naming Conventions for Bean Components

10.6 Lifecycle of Enterprise Beans

Enterprise beans go through different states during their lifecycle. These states are different for stateless, stateful, and singleton beans.

→ Stateful Session Beans

The various stages in the lifecycle of a stateful session bean can be generalized as follows:

- **Instantiation** - Initially the bean does not exist. The bean is brought into existence by instantiating the bean class.
- **Dependencies injected** - Once the bean is instantiated, dependency injection is performed on the bean to relate the bean with other components of the application. After this process, the bean transits to the pooled state.
- **Activated** - The bean is activated when it is invoked through business interface or through no-interface view.
- **Passivated** - After the purpose of activating the bean is served, the bean becomes passive and is returned to the pool of beans. This is however applicable only to the stateful session beans. The stateless session beans do not achieve the passive state.
- **Removed** - The bean class is finalized and removed from the pool if it is not invoked by any client for certain timeout period. This is done as part of the memory management process.

The transition among different stages of the life cycle of the bean is done through various methods such as:

`setSessionContext()` and `newInstance()` – methods are used to instantiate the bean.

- **`ejbActivate()`** – method is used to switch the bean from passive state to active state.
- **`ejbPassivate()`** – method is used to switch the bean from active state to passive state.
- **`ejbRemove()`** – method is used to deallocate the memory of the bean instance.

→ Stateless Session Beans

The initial steps of the life cycle of the stateless session beans are same as that of the stateful session beans. The stateless session beans do not keep track of the properties pertaining to the session. When stateful session beans become passive, the state of the bean is maintained for a certain time period and discarded after the timeout period expires. Since stateless session beans do not maintain any client state, they never achieve the passive state. Following are the states in the life cycle of the stateless session bean:

- **Instantiated** - The bean is brought into existence by instantiating the bean class.
- **Dependencies injected** - Once the bean is instantiated dependency injection is done on the bean to relate the bean with other components of the application. After this process the bean is in a pooled state.

- **Activated** - The bean is activated when it is invoked through business interface or through no-interface view.
- **Removed** - After serving the purpose the stateless bean reverts to the ready state and when the application shuts down it is removed from the ready state through ejbRemove() method with @Remove annotation.

→ Singleton Session Beans

The singleton session bean is created only once in the application. It is created with @Startup annotation. The singleton session bean is never passive. Following are the stages in the life cycle of a singleton session bean.

- **Instantiated** - It is brought into existence by creating an instance of it.
- **Dependencies injected** - Dependency injection is performed on the bean to connect the bean with other existing components of the application.
- **Removed** - It is removed from the memory only when the application shuts down.

→ Lifecycle of Message Driven Beans

Message driven beans are similar to session beans. Following are the stages in the lifecycle of message driven beans:

- **Instantiated** - They come into existence when instantiated.
- **Dependencies injected** - Dependency injection is performed on it to connect with other components of the application.
- **Activated** - They are invoked by the client using the onMessage() method.
- **Removed** - They are removed from active state with an @Remove annotated method.

Like stateless session beans, the message driven beans are never passive.

10.7 Applying Enterprise Beans

Applying enterprise beans involves creation of a simple Enterprise JavaBean and observing the basic functioning of the bean.

The basic requirements include NetBeans IDE, and Java EE 7 installed on your system. The IDE should be configured with Glassfish server.

As discussed earlier, an enterprise bean will have a bean class and a corresponding interface. The bean class has the definition of business methods. The interface consists of declarations of the exposed methods that can be used to access the bean. The bean class implements the interface and overrides the methods in the bean class to provide the method definition.

The NetBeans IDE is shown in figure 10.1.

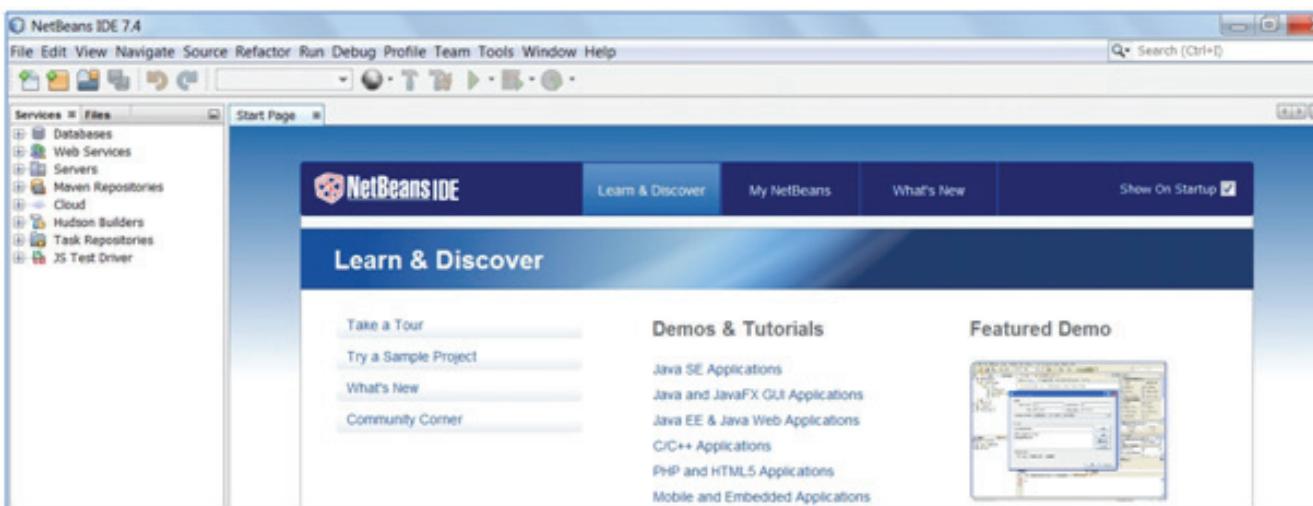


Figure 10.1: NetBeans IDE

Create a new project by selecting File → New → Java EE → EJB module in the New Project dialog box as shown in figure 10.2.

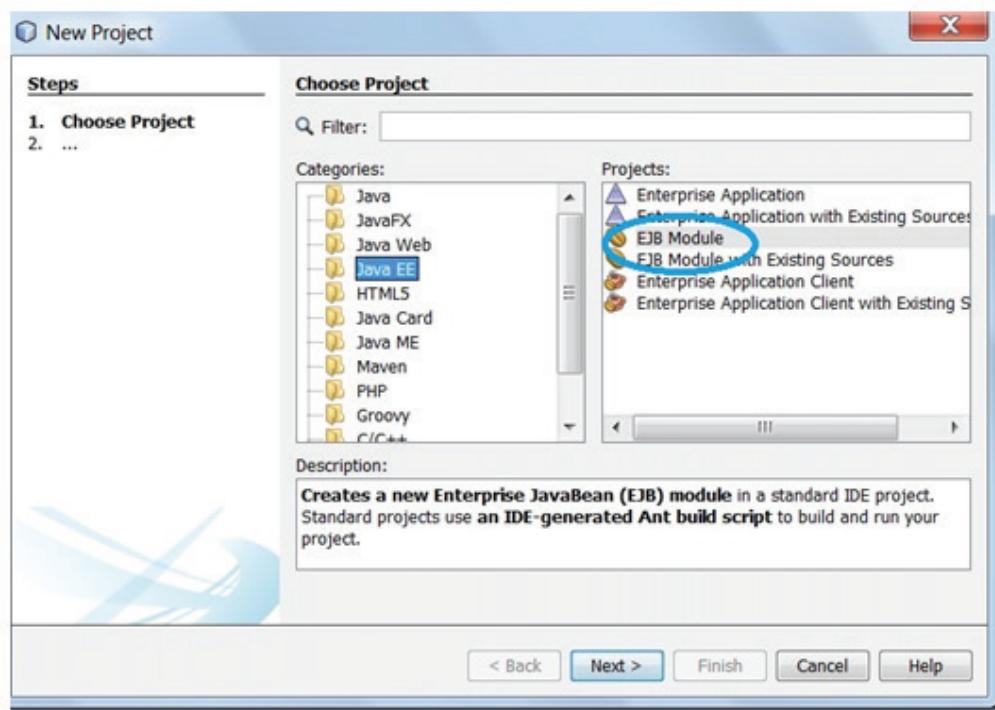


Figure 10.2: Project Creation

Click Next. It leads you to another window where you can choose the project name. Choose an appropriate project name as shown in figure 10.3.

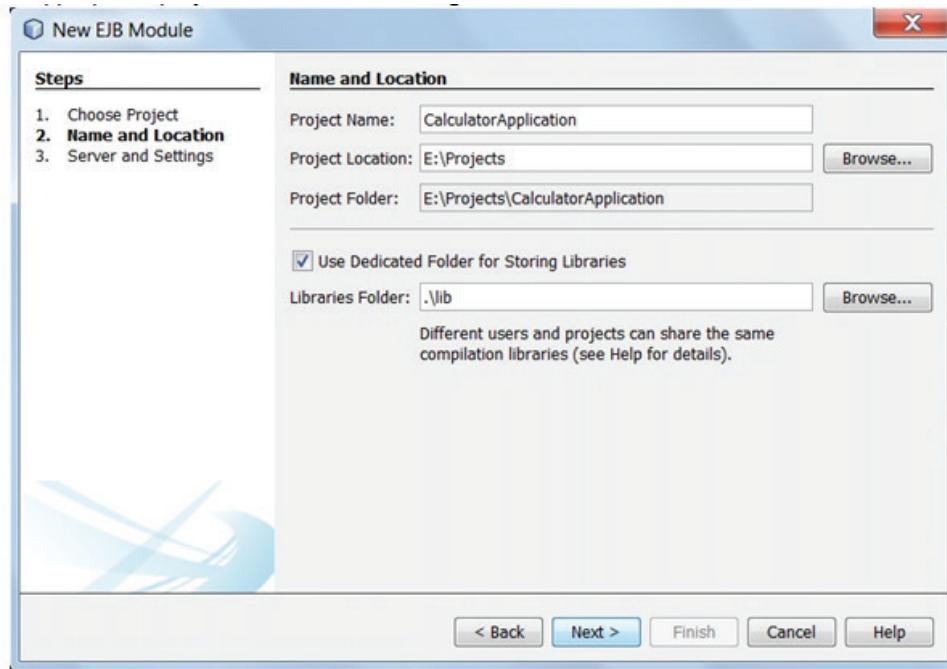


Figure 10.3: Choosing the Project Name

After specifying the project name, select the server you wish to use for the application and click Finish. This will create the project with various sub-folders as shown in figure 10.4.

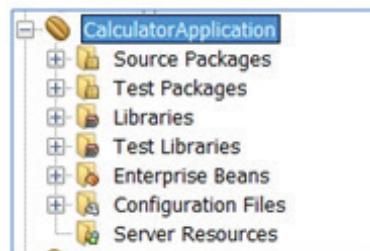


Figure 10.4: Project Window

The project is now created with various sub-folders. Source Packages is the folder in which the beans are stored. Right-click the Source Packages folder and create a new package within the Source Packages folder as shown in figure 10.5.

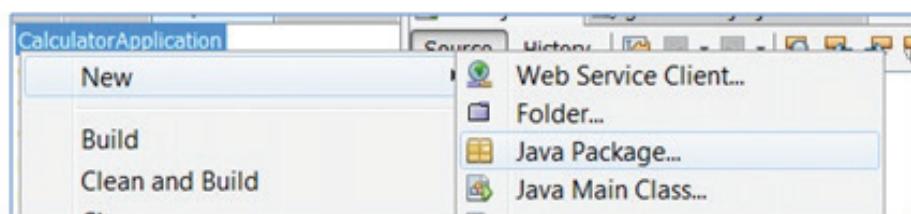


Figure 10.5: Creating a New Test Package for the Bean

Select New → Java Package. Create a package and give it an appropriate name. Here, the package is named Test as shown in figure 10.6.

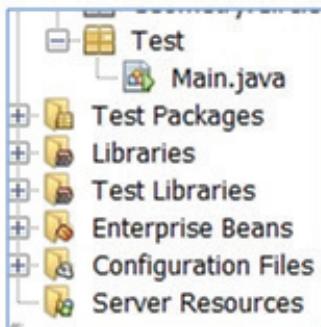


Figure 10.6: Bean Package

The Main.java class has been added to the Test package. This class will be the client that will invoke the bean methods. Now, you can add Calculator interface and CalculatorImplBean to the application.

The Calculator interface has the code as shown in Code Snippet 6.

Code Snippet 6:

```
package Test;
import javax.ejb.Remote;

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

The interface consists of two public methods that can be accessed by clients. The Calculator interface has been created as a remote interface using the @Remote annotation so that it can be accessed by remote clients. The Calculator interface is implemented by a stateless session bean CalculatorImplBean. The bean will implement the methods of the Calculator interface as shown in Code Snippet 7.

Code Snippet 7:

```
package Test;

import javax.ejb.Stateless;
import javax.ejb.LocalBean;
@Stateless
@LocalBean

public class CalculatorImplBean implements Calculator{
    @Override
    public String sayHello(String name) {
        return "Welcome to EJB"+name;
    }
}
```

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

Code Snippet 7 shows the stateless session bean CalculatorImplBean implementing the Calculator interface. It overrides the methods that are declared in the Calculator interface and provides the method definition. The session bean is invoked by the client, that is, Main.java as shown in Code Snippet 8.

Code Snippet 8:

```
package Test;

import java.util.Properties;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Main {

    public static void main(String[] args) throws NamingException {
        // TODO code application logic here

        InitialContext c = new InitialContext();
        Object l;
        l = c.lookup("caljndi");
        Calculator r=(Calculator)l;
        System.out.println(r.sayHello(" Bruce "));
        System.out.println( r.addition(5,10));

    }
}
```

Code Snippet 8, which is the client class, invokes the session bean. The session bean can also be invoked through Web clients such as a servlet.

In Code Snippet 8, the initial context of the bean is setup and also a reference to a JNDI name is created through the lookup() method. The 'caljndi' parameter of the lookup method is the JNDI name of the bean. This mapping has to be mapped in the configuration files.

In order to define the configuration, create a new file with a GlassFish Descriptor as shown in figure 10.7.

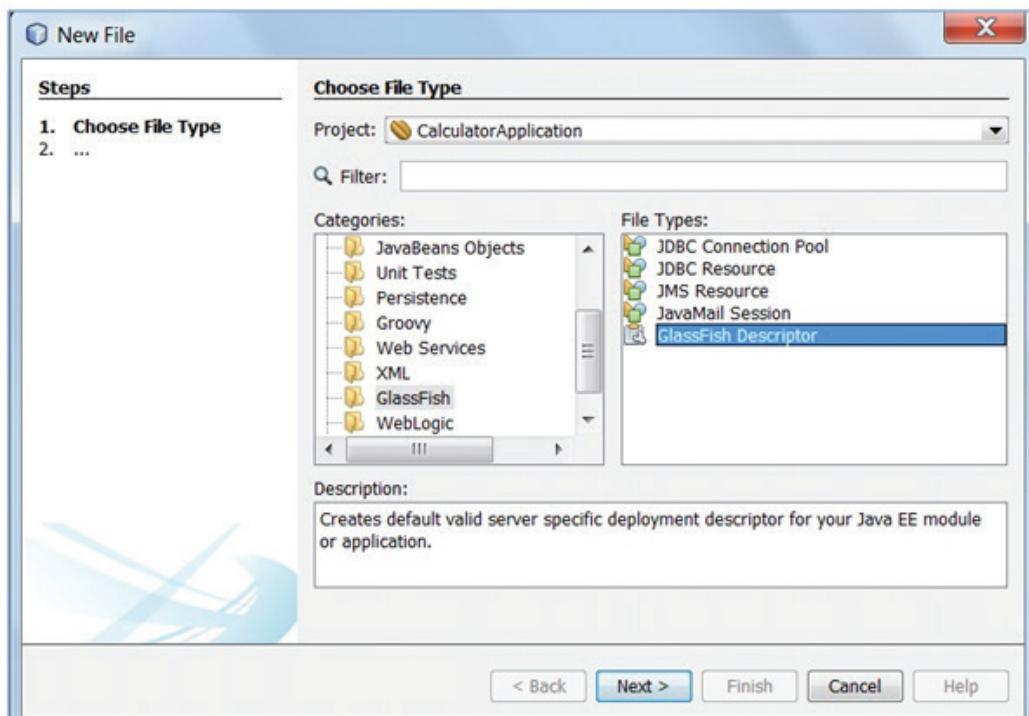


Figure 10.7: Defining GlassFish Descriptor

There will always be only one glassfish descriptor in the package. The file is created with the name glassfish-ejb-jar.xml. Edit the XML file to specify the JNDI name. The XML file will have the code as shown in Code Snippet 9.

Code Snippet 9:

```
<?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>CalculatorImplBean</ejb-name>
      <jndi-name>caljndi</jndi-name>
    </ejb>
  </enterprise-beans>
</glassfish-ejb-jar>
```

Set the JNDI name in the XML file to caljndi which is given as parameter to the lookup() method in the client file (Main.java).

Clean and build the application once the GlassFish server configuration is done.

10.8 Deploying the Code

Once the build is successful, deploy the code by right-clicking the package. Run the application after deploying the code. Successful deployment of the code is possible only if the server configuration is properly done. Figure 10.8 shows the output of the application.



The screenshot shows the NetBeans IDE's Output window. It has three tabs at the top: Java DB Database Process, GlassFish Server, and CalculatorApplication (run). The GlassFish Server tab is selected and displays the following log output:

```
RUN:
Welcome to EJB Bruce
15
BUILD SUCCESSFUL (total time: 11 seconds)
```

Figure 10.8: Output

10.9 Creating a Web Client and an External Application Client for an Enterprise Bean

In this example, an enterprise bean is accessed by an external application client. A remote interface and session need to be defined for the bean application. An application client will access the bean methods through remote interface. The following components are created to create the communication bean and an application client:

1. The remote interface
2. An enterprise application with a bean method. This bean method is created in a session bean so that it can be accessed by request
3. Application client
4. Web client

→ Creating the Remote Interface

In order 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 10.9.

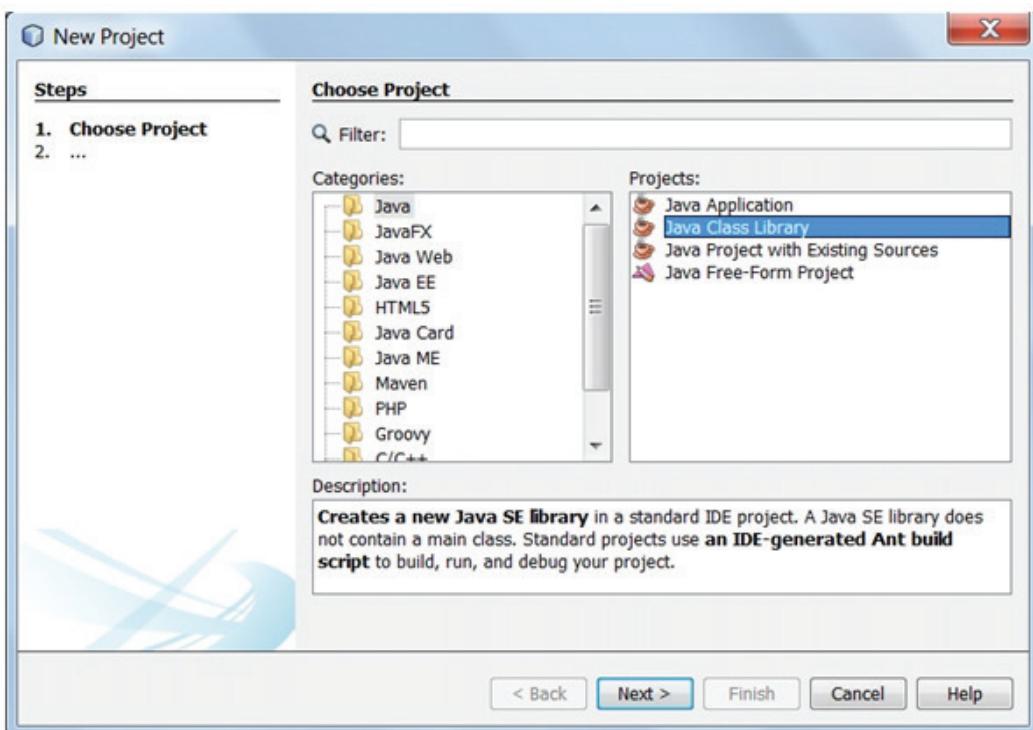


Figure 10.9: Creating a Class Library for Remote Interface

Name the session interface as ‘FirstRemoteInterface’ as shown in figure 10.10.

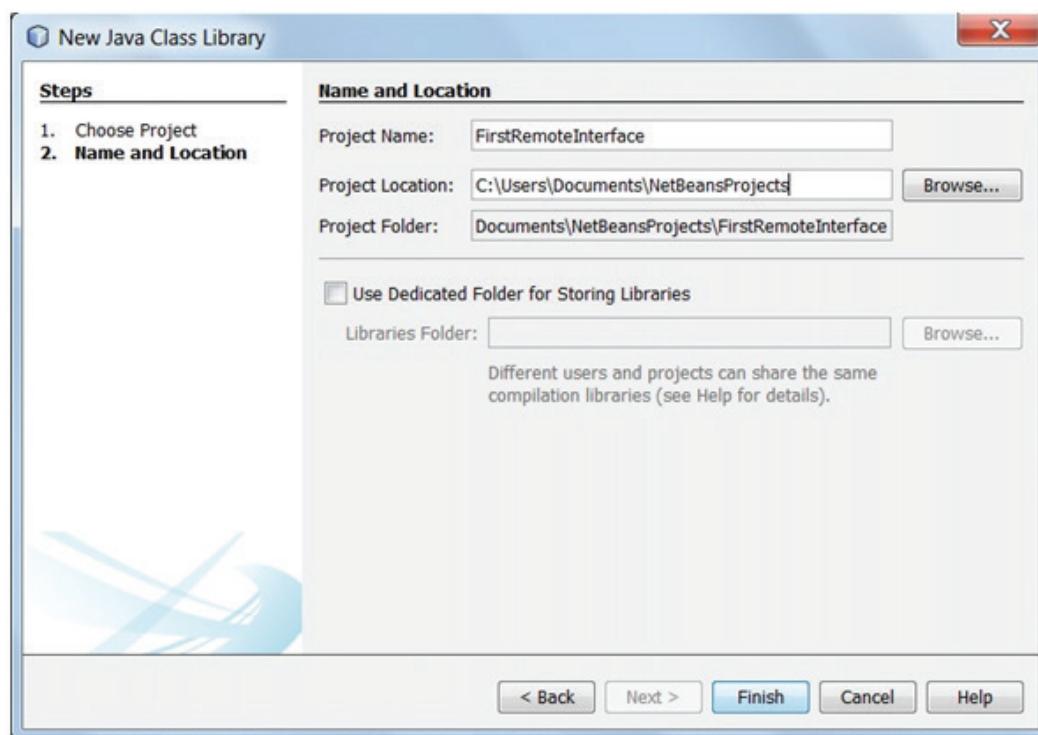


Figure 10.10: Creating Remote Interface

Once the remote interface is created, create an Enterprise Application which consists of the enterprise bean. Configure this enterprise bean to be accessed by the application client through the remote interface.

→ Creating the Enterprise Application

To create an enterprise application named ‘BeanApplication’ with EJB module, click File → New Project → Java EE → Enterprise Application as shown in figure 10.11.

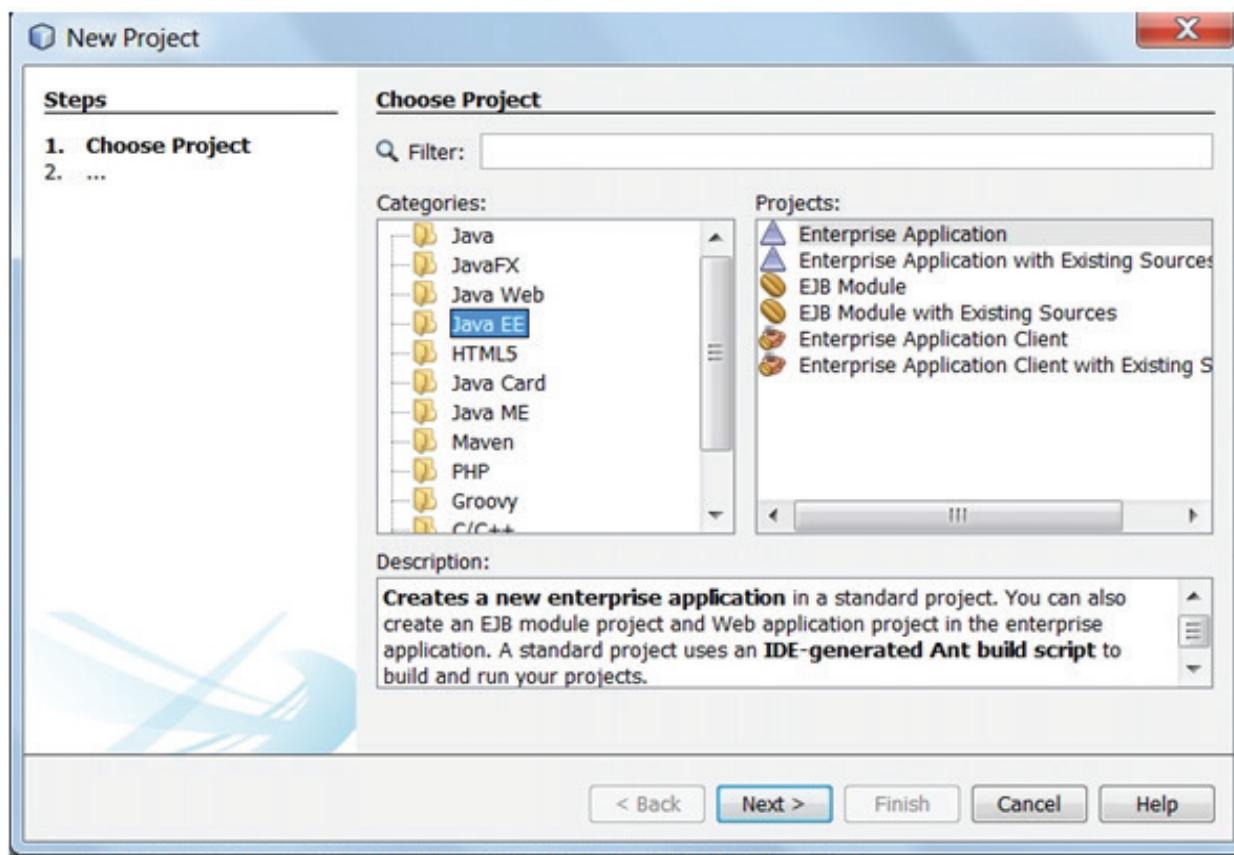


Figure 10.11: Creating an Enterprise Application

Click Next.

Specify the name of the application as BeanApplication and click Next.

Select the options from the Server and Settings screen as shown in figure 10.12.

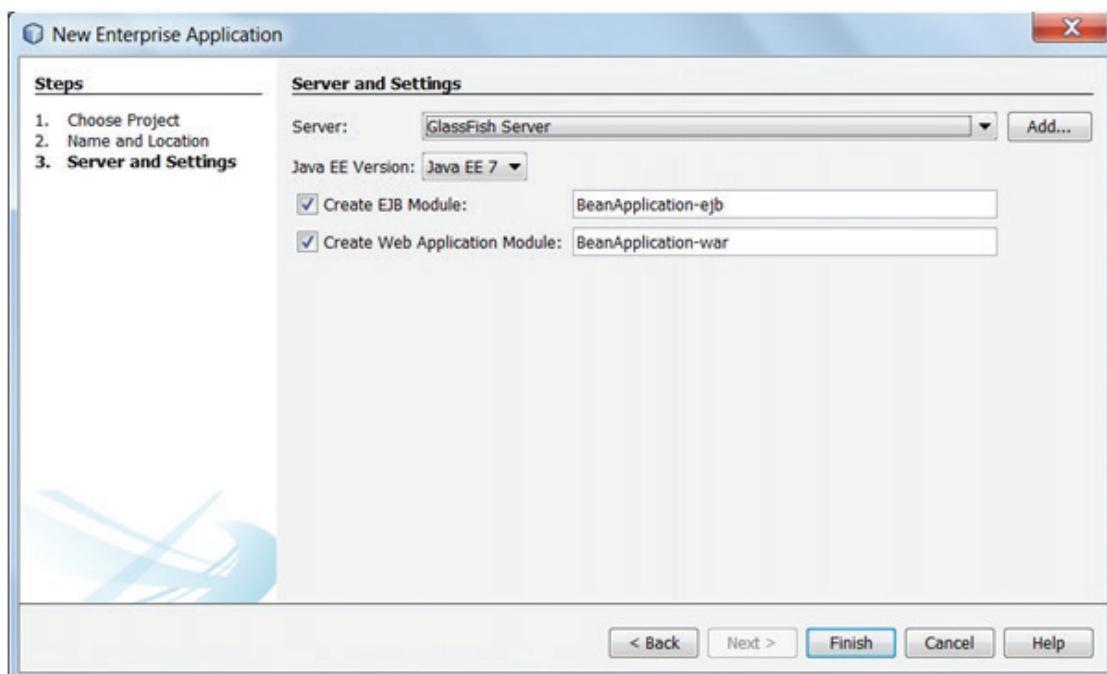


Figure 10.12: Creating an EJB Module in the Enterprise Application

Ensure that both the Create EJB Module and Create Web Application Module checkboxes are selected.

Click Finish.

An Enterprise Application with the name 'BeanApplication' is created with the BeanApplication-ejb and BeanApplication-war modules.

→ Creating the Session Bean

To create a session bean in the EJB module, right-click the BeanApplication-ejb project and select New → Other → Enterprise JavaBeans → Session Bean as shown in figure 10.13.

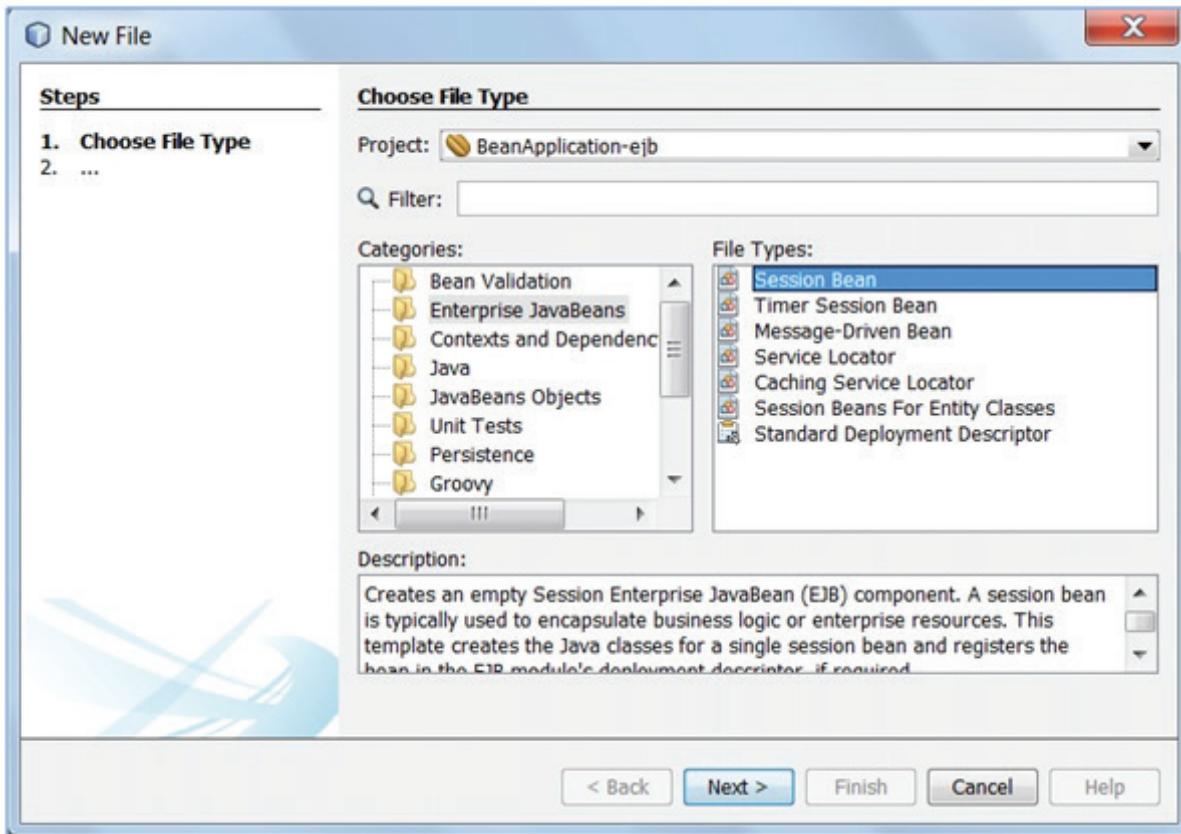


Figure 10.13: Creating a Session Bean in the EJB Module

Click Next.

Specify the name of the bean as ApplicationSession and package name as ejb.

Ensure that Stateless option is selected.

Select the Remote in project checkbox. The FirstRemoteInterface created earlier will automatically get added to the drop-down.

This is shown in figure 10.14.

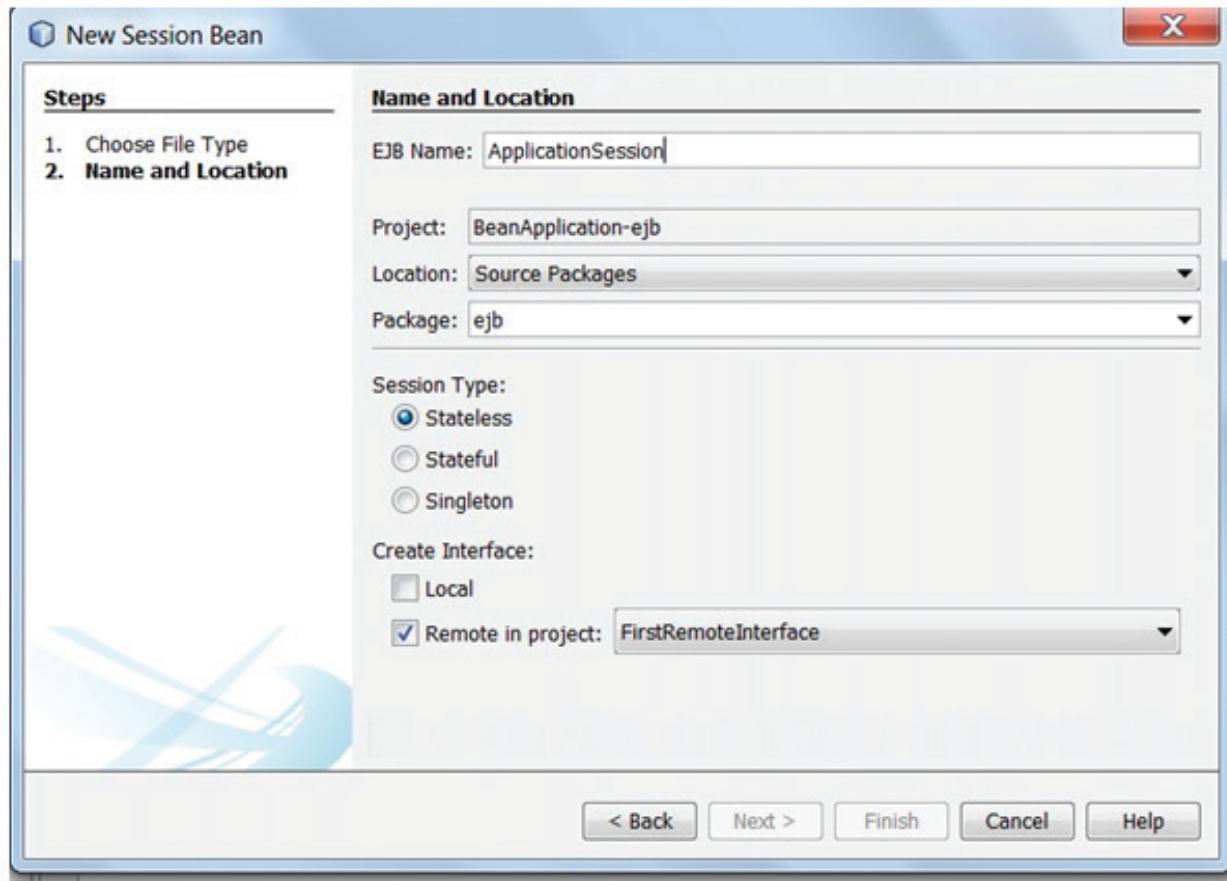


Figure 10.14: Session Bean Settings

Click Finish. The Session Bean is created within the ejb folder as shown in figure 10.15

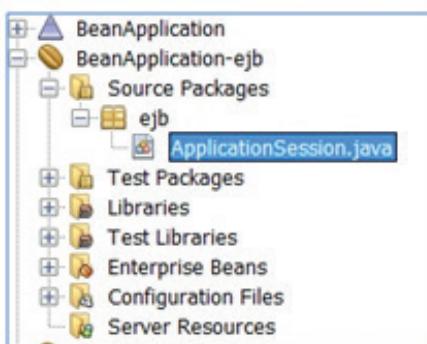


Figure 10.15: Session Bean Created in the EJB Module

A corresponding remote interface named ApplicationSessionRemote is automatically created in the FirstRemoteInterface project as shown in figure 10.16.

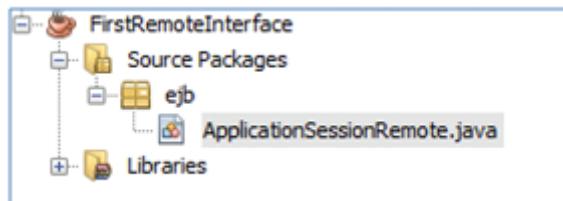


Figure 10.16: Remote Interface Added to the FirstRemoteInterface Project

The ApplicationSession implements the remote interface defined earlier.

→ Creating Business Methods

In order to define business methods in the bean that can be accessed by the client, the IDE provides options for the code to be inserted directly by clicking Alt+Insert. It gives the options as shown in figure 10.17.

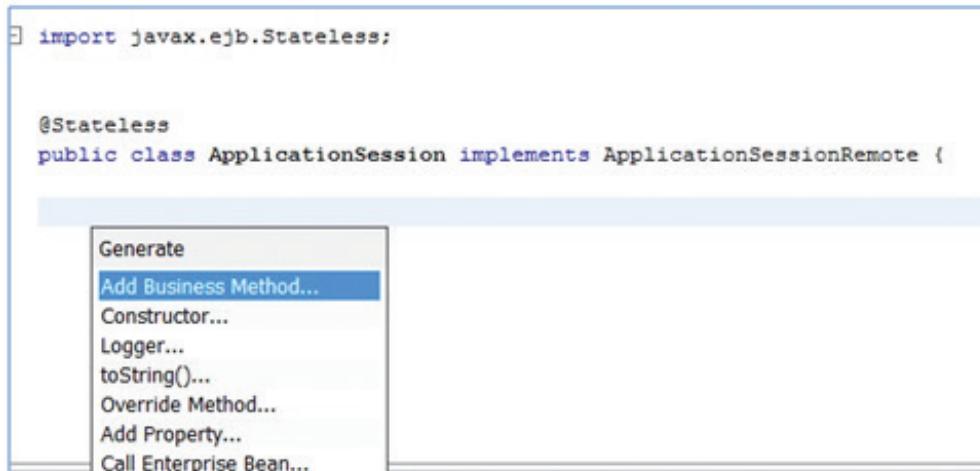


Figure 10.17: Adding Business Methods to the Session Bean

Note - The menu can also be opened by right-clicking the editor and selecting Insert Code.

Click Add Business Method. The Add Business Method dialog box is displayed.

Specify the method details in the dialog box as shown in figure 10.18.

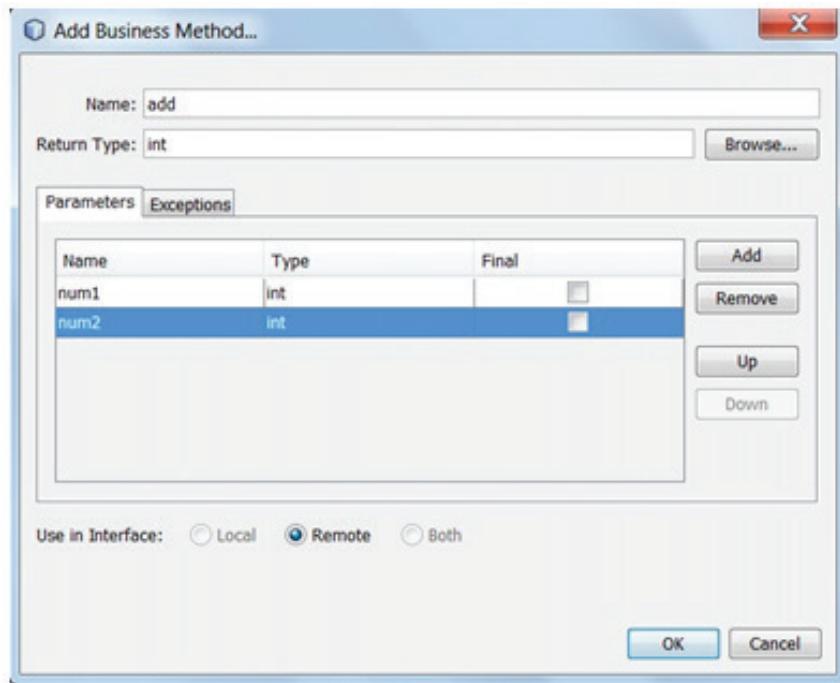


Figure 10.18: Defining the Business Method Prototype

Click OK. The method will be added to the ApplicationSession bean as shown in figure 10.19.

```
@Stateless
public class ApplicationSession implements ApplicationSessionRemote {

    // Add business logic below. (Right-click in editor and choose
    // "Insert Code > Add Business Method")
    @Override
    public int add(int num1, int num2) {
        return 0;
    }
}
```

Figure 10.19: Business Method Added to the Session Bean

Simultaneously, the method signature is also added to the ApplicationSessionRemote file as shown in figure 10.20.

```
package ejb;

import javax.ejb.Remote;

@Remote
public interface ApplicationSessionRemote {

    int add(int num1, int num2);

}
```

Figure 10.20: Method Signature Added to the Remote Interface

Modify the code in the ApplicationSession bean's add method as shown in Code Snippet 10.

Code Snippet 10:

```
public int add(int num1, int num2) {
    return num1+num2;
}
```

Build and deploy the enterprise application after defining the method in the session bean.

Right-click the BeanApplication and select Deploy. This will deploy the application on the server. Its deployed application can be viewed in the Services tab under the GlassFish server as shown in figure 10.21.

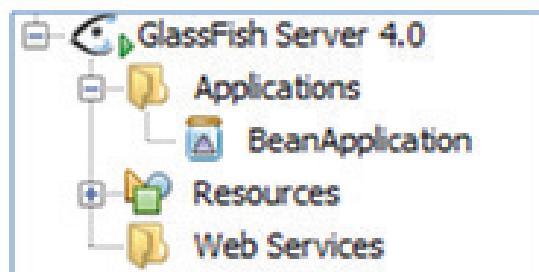


Figure 10.21: Enterprise Application Deployed on the Server

The deployed jar and war files of EJB and Web modules respectively can be viewed in the Files tab.

→ Creating the Application Client

To create an external enterprise application client, click File → New Project → JavaEE → Enterprise Application Client as shown in figure 10.22.

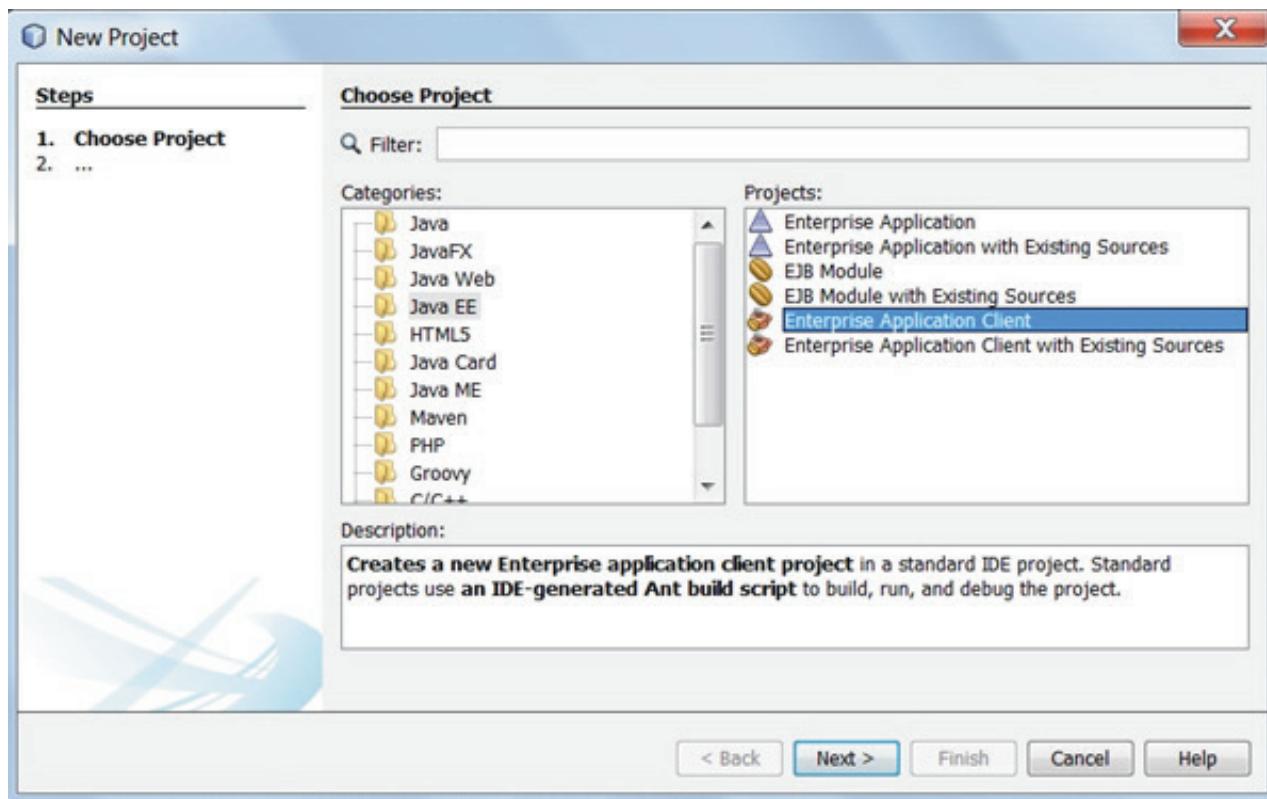


Figure 10.22: Creating Enterprise Application Client

Click Next. Name the project as Client and click Next.

The Server and Settings screen is displayed. Since the project is going to be external, we need not add it to any Enterprise Application. Therefore, leave the Add to Enterprise Application drop-down blank.

Select the server as Glassfish and JavaEE7 as the Java EE Version and click Finish. The hierarchy of the Client application is shown in figure 10.23.

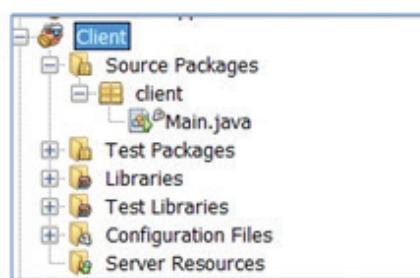


Figure 10.23: Directory Hierarchy in the Client Application

To invoke the ApplicationSession bean method from the client, you need to add a reference to the remote interface.

To do this, press Alt+Insert and select ‘Call Enterprise Bean’ as shown in figure 10.24.

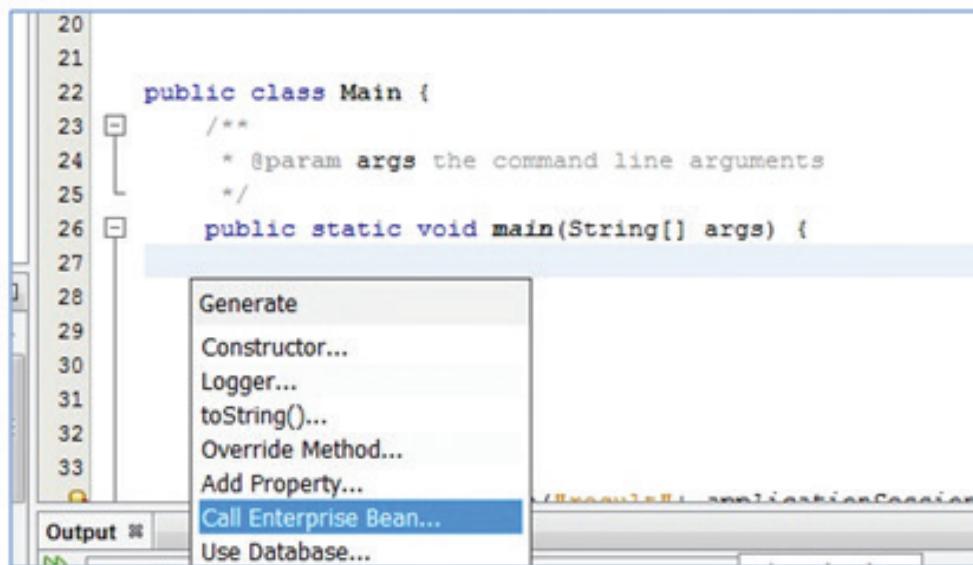


Figure 10.24: Calling Enterprise Bean from the Client

On selecting the option ‘Call Enterprise Bean’, the IDE will pop a wizard in which the developer can choose the bean to be invoked.

Select the ApplicationSession bean as shown in figure 10.25.

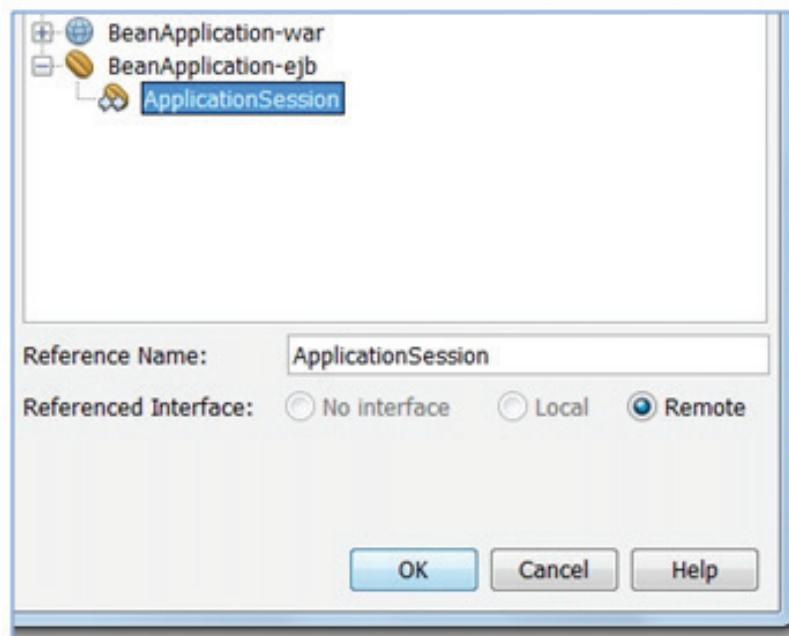


Figure 10.25: Selecting the Session

This will add the reference of the remote interface to the main class of the client application as shown in figure 10.26.

```
public class Main {
    @EJB
    private static ApplicationSessionRemote applicationSession;

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
    }
}
```

Figure 10.26: Remote Interface Reference Added to the Client

After adding the reference, the developer can invoke the methods of the bean exposed by the remote interface. Code Snippet 11 shows the code used by the client application to invoke the add method of the ApplicationSession bean.

Code Snippet 11:

```
package client;

import ejb.ApplicationSessionRemote;
import javax.ejb.EJB;

public class Main {
    @EJB
    private static ApplicationSessionRemote applicationSession;

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

In Code Snippet 11, the add method of the bean is accessed using the reference of the remote interface.

On deploying and running the application client, the output is shown in figure 10.27.

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

Figure 10.27: Output on Running the Application Client

→ **Creating the Web Client**

To create the Web Client, in the BeanApplication-war module, right-click the Web Pages folder and select New → JSP. The New JSP dialog box is displayed. Specify the name as index as shown in figure 10.28.

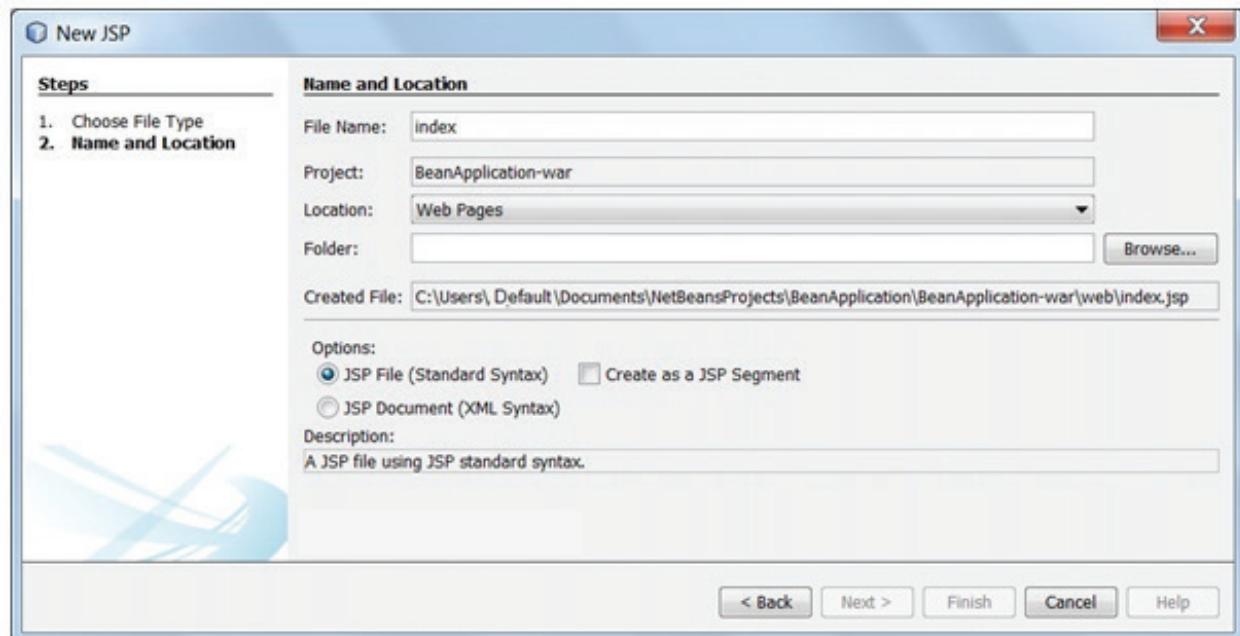


Figure 10.28: Creating a JSP in the Bean Application

Add input components to the JSP page. In order to add input components activate the ‘Palette’ with a combination of Ctrl+Shift+8. This will invoke the palette as shown in figure 10.29.

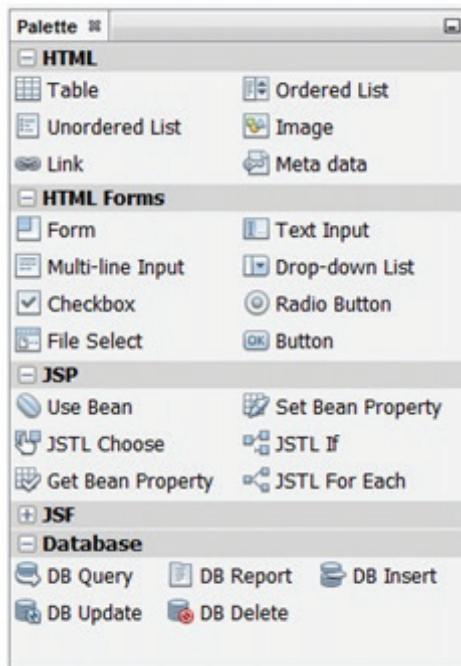


Figure 10.29: Palette for JSP Components

Drag and drop the components from the palette into the body section of the JSP. All the UI components should be placed in a form component, therefore, first drop the form component in the JSP page.

Code Snippet 12 shows the code of the index.jsp file.

Code Snippet 12:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Web Client</title>
    </head>
    <body>
        <form name="Addition" action="Addition" method ="GET">
            <input type="text" name="num1" value="" size="5" />
            <input type="text" name="num2" value="" size="5" />
            <input type="submit" value="Add" />
        </form>
    </body>
</html>
```

In Code Snippet 12, a form component is added to the JSP first. The form component has attribute ‘action’ which specifies the servlet to be invoked for the current JSP and also the HTTP method to be used. The GET method is invoked in this case. Two text boxes and a button have been added to the JSP.

Note - Since the GET method has been used, the parameters passed in the textboxes will be visible in the URL string in the Address Bar of the browser.

→ Creating the servlet for the JSP

The servlet will be used to write the business logic to invoke the ApplicationSession bean. In order to create the servlet, right-click the BeanApplication-war project and select New → Other → Web → Servlet as shown in figure 10.30.

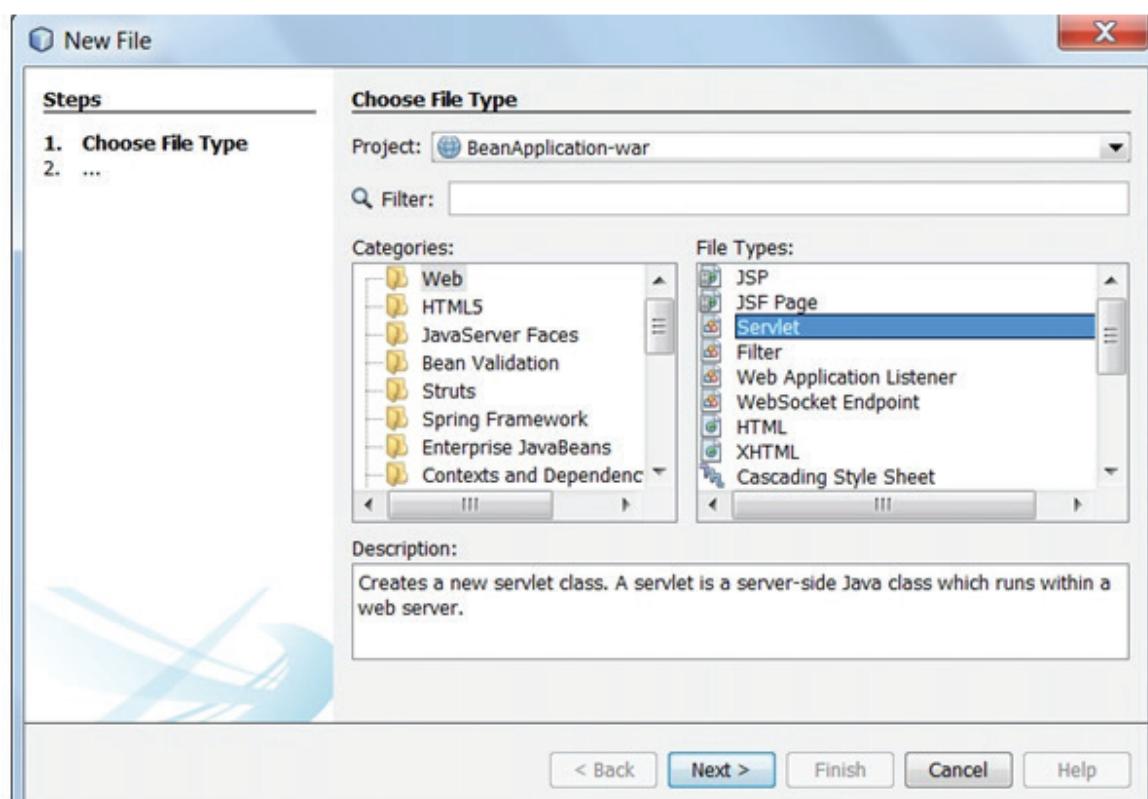


Figure 10.30: Creating a Servlet

Specify the package and name of the servlet as shown in figure 10.31.

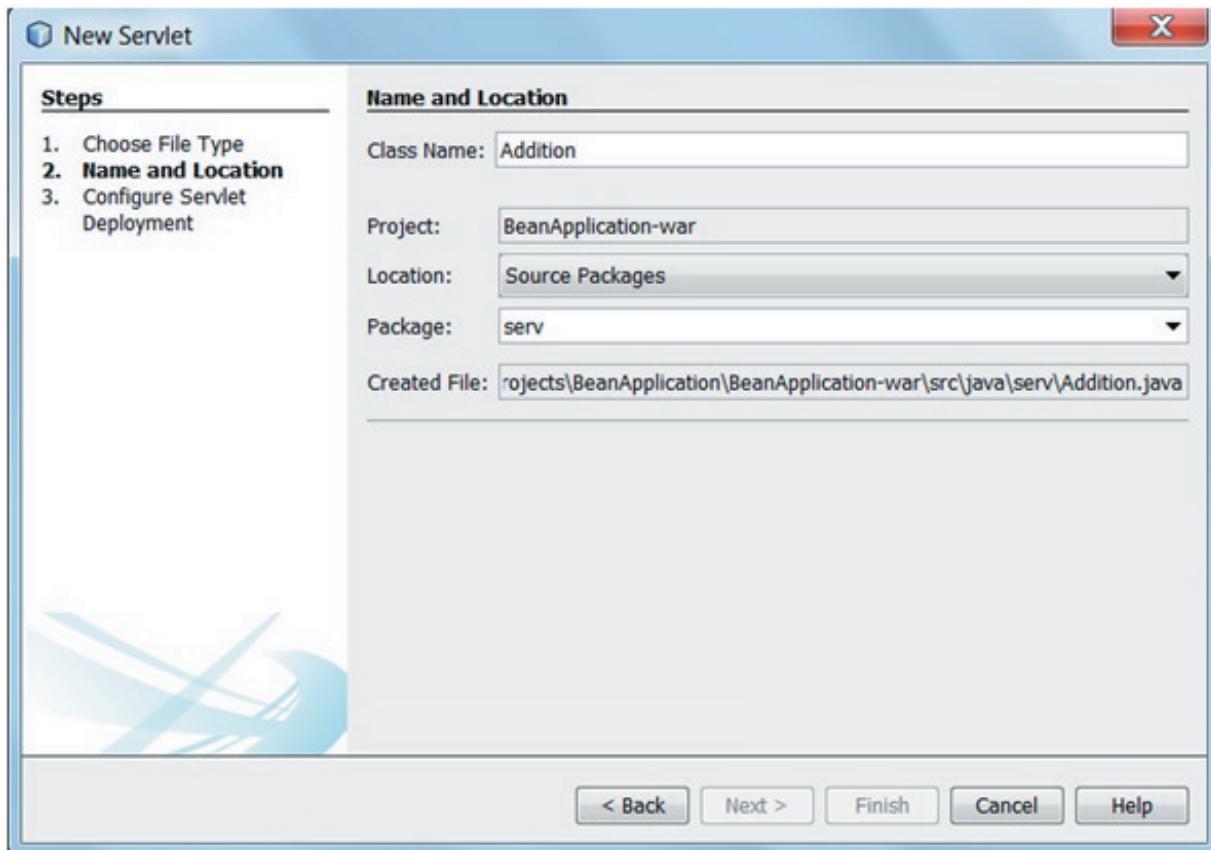


Figure 10.31: Specifying the Package and Name of the Servlet

Click Next. On the next screen, select the 'Add information to deployment descriptor (web.xml)' checkbox.

Click Finish. The servlet named Addition will be created with the processRequest, doGet, and doPost methods.

Now, add reference of the remote interface of the ApplicationSession bean to be invoked.

Right-click in the editor and select Insert Code → Call Enterprise Bean.

Select the ApplicationSession bean as shown in figure 10.32.

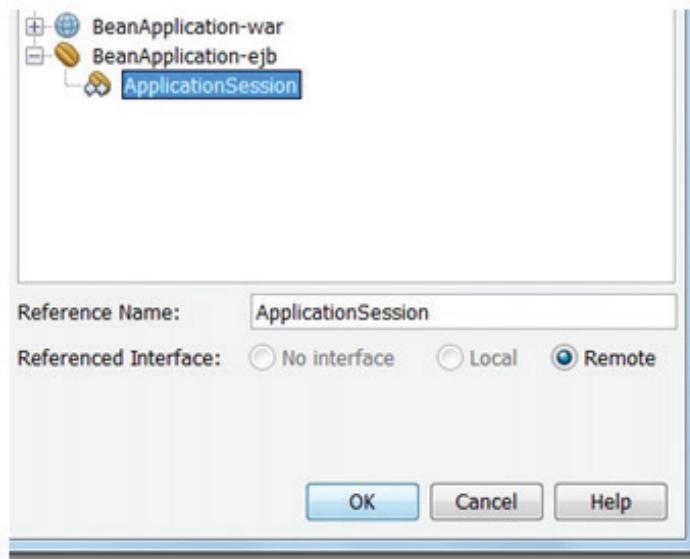


Figure 10.32: Selecting the Enterprise Bean

Click OK. The remote interface reference will be added to the servlet as follows:

```
@EJB
```

```
private ApplicationSessionRemote applicationSession;
```

Add the following code in the try block of the processRequest() method.

```
.....
```

```
int n1,n2;
```

```
n1 = Integer.parseInt(request.getParameter("num1"));
```

```
n2 = Integer.parseInt(request.getParameter("num2"));
```

```
.....
```

This piece of code converts the String parameters read from the JSP page to 'int' type.

Add the following code in the HTML block of the processRequest() method to invoke the addition operation.

```
.....
```

```
out.println("<h2> Result: " + applicationSession.add(n1, n2)+ "</h2>" );
```

```
.....
```

These statements should be added in the body section of the HTML code generated from the servlet. Code Snippet 13 shows the final code of the servlet.

Code Snippet 13:

```

package serv;

import ejb.ApplicationSessionRemote;
import java.io.IOException;
import java.io.PrintWriter;
import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class Addition extends HttpServlet {
    @EJB
    private ApplicationSessionRemote applicationSession;

    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            int n1,n2;
            n1 = Integer.parseInt(request.getParameter("num1"));
            n2 = Integer.parseInt(request.getParameter("num2"));
            /* 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 Addition</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h2> Result: " +
applicationSession.add(n1, n2)+ "</h2>" );
            out.println("</body>");
            out.println("</html>");
        }
    }

    @Override
    protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
        processRequest(request, response);
    }
}

```

```

@Override
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    processRequest(request, response);
}
@Override
public String getServletInfo() {
    return "Short description";
}// </editor-fold>

}

```

Code Snippet 13 is the updated code which accesses the ApplicationSession method add().

Expand the Configuration Files folder and double-click web.xml file to open it.

Click Pages tab and specify index.jsp in the Welcome Files box.

Note - You can also use the Browse button to select the index.jsp file.

Deploy the BeanApplication enterprise application.

Note - Deploying the Web module alone will generate an error. Hence, you need to deploy the entire enterprise application.

Right-click the BeanApplication enterprise application and select Run. The index.jsp page is displayed. Specify 2 and 3 in the textboxes as shown in figure 10.33.

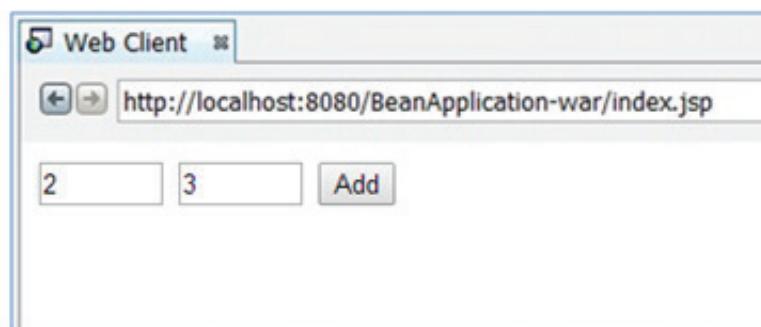


Figure 10.33: Execution of Servlet in the Web Module

Click Add.

The resultant page generated by the servlet which displays the result is shown in figure 10.34.

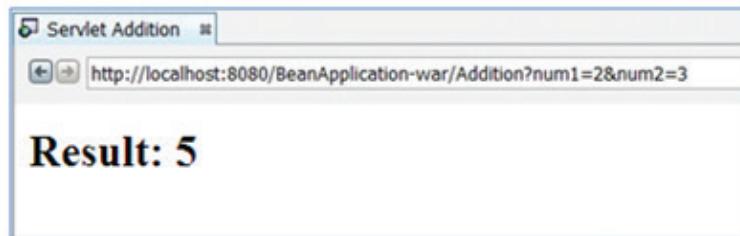


Figure 10.34: Output Page Generated by the Servlet

10.10 Check Your Progress

1. _____ has to be provided for the session beans to enable clear mapping by other components to the application.

(A)	Business interface	(C)	JNDI name
(B)	No-view interface	(D)	None of these

2. Which of the following types of beans can be passive?

(A)	Stateless beans	(C)	Message driven beans
(B)	Stateful beans	(D)	Singleton session beans

3. According to the naming conventions of session beans, which of the following is the name of a bean class corresponding to the Business interface XBean?

(A)	XBeanjar	(C)	XBean ejb
(B)	XBean bean	(D)	XBean

4. _____ method is used for invoking message driven beans.

(A)	onMessage()	(C)	setSessionContext()
(B)	lookup()	(D)	ejbActivate()

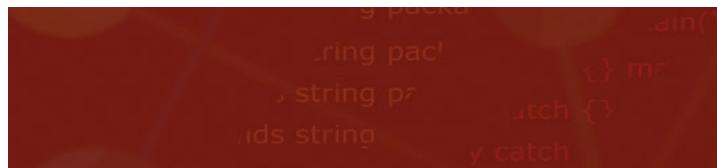
5. Which of the following is a GlassFish Descriptor?

(A)	HomeImpl.java	(C)	glassfish-ejb.xml
(B)	glassfish-jar-ejb.xml	(D)	glassfish-jar.xml

10.10.1 Answers

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

Summary



- Enterprise beans can be session beans and message driven beans.
- There are three variants of session beans – stateful, stateless, and singleton session beans.
- There are three types of clients which access beans – local clients, remote clients, and Web services. These clients can access the beans through business interface and no-interface views.
- Each session bean goes through various states during its life cycle. These states are different for stateless, stateful, and singleton beans.
- Stateful sessions can be passive but stateless, singleton, and message driven beans cannot be passive.

**Get
WORD WISE**



Visit
the **Glossary** section
@

www.onlinevarsity.com

Session - 11

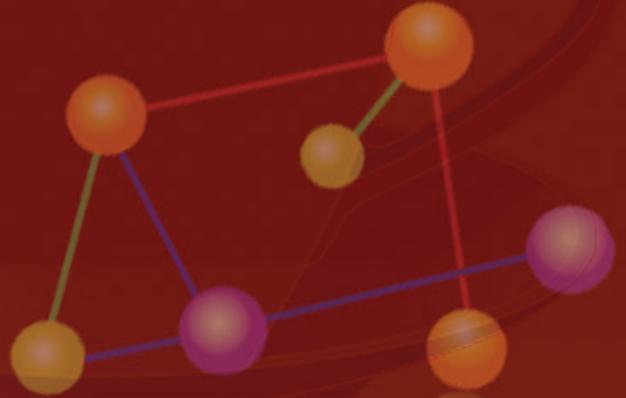
Java Persistence API

Welcome to the Session, **Java Persistence API**.

This session describes Java Persistence API and how it works. It also describes entities and how to manage and query entities. It provides detailed explanation on how to create database schema and basics of Java Persistence Query Language (JPQL).

In this Session, you will learn to:

- Define Java Persistence API
- Describe how Java Persistence API works
- Describe entities and their mutual interactions
- Describe managing and querying the entities
- Explain creation of database schema
- Explain the basics of Java Persistence Query Language



11.1 Introduction

Java Persistence API (JPA) was introduced in Java EE 5 as a standard persistence API. A Java application is modelled as an object-oriented application and databases are stored as relational databases. There is an alternate technique of modelling the databases known as Entity-Relational ER model. An ER model can be translated into an object-oriented application where entities can be mapped to an object and relationships among the objects can be converted to methods in the class.

Similarly, ER model of data can be systematically converted into relational database. The ER model is the basis of Java Persistence API, javax.persistence.Entity, which manages the data as entities.

11.2 Entities

An entity is an object in the Java application whose state has to be saved to the database. In terms of a relational database, a class representing a set of objects can be mapped to the table definition where both tables and classes are the blue prints of data storage in the application. Each object of the class can be mapped to each row of the corresponding table. Whenever the state of an object is accessed, the row corresponding to the entity is retrieved from the table. The state of the persistent data object is represented through the fields pertaining to the entity in the database.

Consider an example where a class Student is defined in the application with attributes roll no, name, and admission date. In the relational database, this data is represented as a table with fields roll no, name, and admission date.

If an object of type Student has values 10001, Andy, and 01-10-10, then these values depict instance variables in the class which are represented as a row (record) in the table.

According to the Persistence API, following are the requirements of the entity classes:

- Every entity class must have a default constructor with public or protected access. Apart from the default constructor, there can be other overloaded constructors defined in the class.
- The entity classes cannot be defined as final. No methods and variables within the entity class should be declared final.
- If the persistent object is passed through a session bean's remote interface, then the entity should be synchronized. To achieve this, the entity class should extend the Serializable interface.
- The variables of the persistent objects must be declared protected or private. The instance variables should be accessible only by methods of the class.
- An entity class may extend other entity or non-entity classes.

The entity class variables can assume any one of the primitive data types or wrappers of primitive data types. It can also be an object of any one of the following types:

- java.math.BigInteger
- java.math.BigDecimal
- java.util.Date
- java.util.Calendar
- java.sql.Date
- java.sql.Time
- java.sql.Timestamp
- User-defined serializable types
- byte[]
- Byte[]
- char[]
- Character[]
- Enumerated types
- Other entities and/or collections of entities

A persistent field is one whose values correspond to instance variables of an object and are obtained at runtime. These variables do not have the getter and setter methods explicitly defined in the entity class.

A persistent property gets the values through the getter and setter methods defined in the entity class. The Netbeans IDE generates these methods for the attributes as shown in figure 11.1.

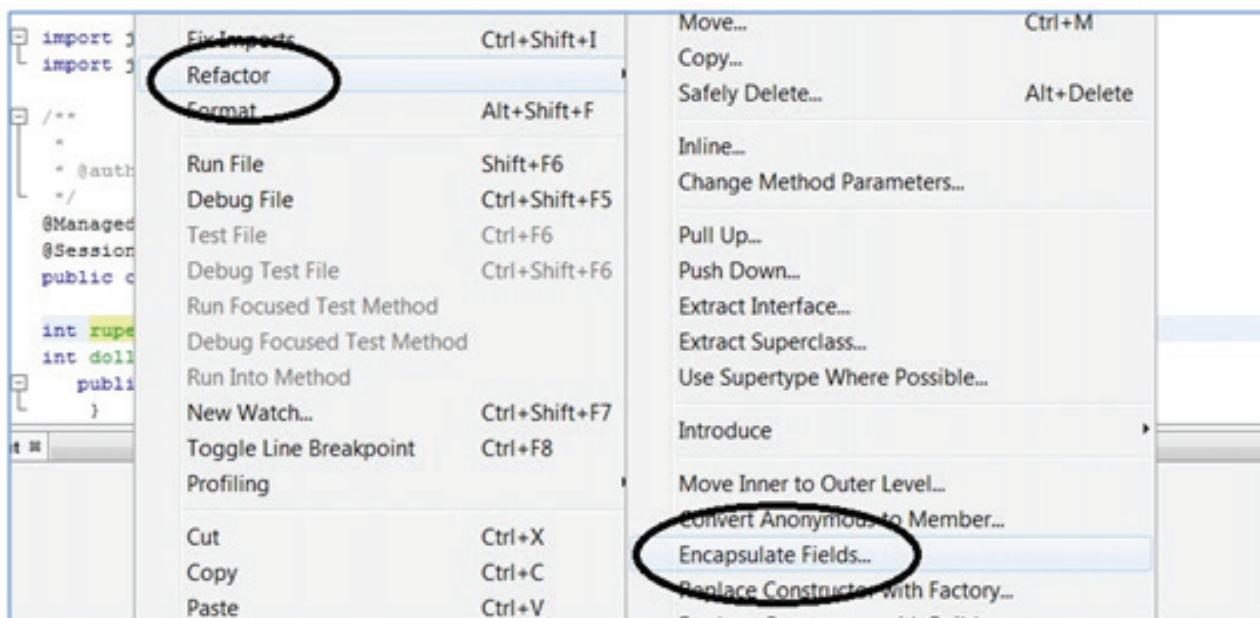


Figure 11.1: Encapsulating the Attributes

The menu in figure 11.1 appears on right-clicking the variable that is to be encapsulated. The screen in figure 11.2 appears through which the getter and setter methods can be created.

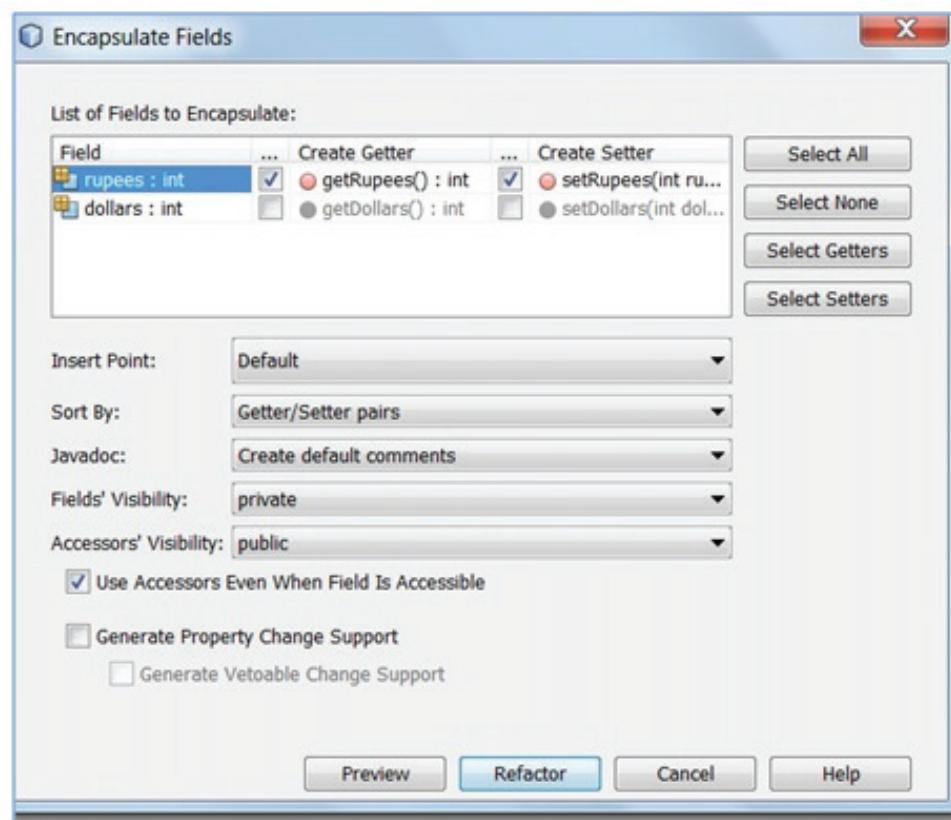


Figure 11.2: Getter and Setter Methods for Attributes

These getter and setter methods are used to fetch and return the values for the attributes.

When the entity class is using persistent fields, all the non-transient fields (fields which are supposed to be written to storage) are mapped and persisted to the database. In contrast, transient fields are those whose values are not stored onto the database. They are created using the `@Transient` annotation. It is used to annotate a property or field of an entity class, mapped superclass, or embeddable class.

Forexample:

```
@Entity
public class Student {
    @Id int id;
    @Transient User currentUser;
    ...
}
```

When the entity class has persistent properties, it must also define the getter and setter methods according to the method conventions of Java Bean properties. The getter and setter methods must have the following signature as per the method convention:

```
Type getProperty()
setProperty (Type t)
```

where property refers to the identifier representing the property of the entity class.

→ Collections as Entity Fields and Properties

An entity may also have a collection as a property, for example a Student object can opt for a set of courses. This can be represented in the class through a Collection or Set of course objects. The entity field may use any one of the following interfaces to represent a collection of objects:

- `java.util.Collection`
- `java.util.Set`
- `java.util.List`
- `java.util.Map`

The getter and setter methods must be appropriately defined for these collections in the entity class.

Unlike Collection, Set, and List objects, Map objects contain a key-value pair. These objects are generally used to speed up the data access from the database by building indexes, and so on. Code Snippet 1 shows the getter and setter methods for a Collection object coll

Code Snippet 1:

```

package Product;

import java.util.Collection;
import java.util.Iterator;
import javax.ejb.Stateless;

public class ProductsBean implements java.io.Serializable{
    private String proName;
    private String description;
    private Collection coll = new Collection() {
        /*implement all the abstract methods of the Collection interface*/
    }
    public Collection getColl() {
        return coll;
    }
    /**
     * @param coll the coll to set
     */
    public void setColl(Collection coll) {
        this.coll=coll;
    }
}

```

→ Validating Persistent Fields and Properties

The Java Beans Validation API provides mechanisms for validating application data. The validation mechanism is embedded into the Java EE containers and it can be applied to the persistent entity classes. The Persistence provider will automatically invoke the Java Bean validation methods on persistent properties and fields if they are marked with an appropriate annotation with bean validation constraints.

Note: Annotation is a form of metadata which is not a part of the program.

Bean validation provides for defining a set of constraints, both default and custom constraints on the persistence fields and properties. Each constraint is associated with a validator class which is responsible for validating the persistence field or property.

Table 11.1 lists Bean validation's built-in constraints, defined in the javax.validation.constraints package that can be applied to the value of a field or property.

Constraint	Description	Example
@AssertFalse	The value must be false.	@AssertFalse boolean isAbsent;
@AssertTrue	The value must be true.	@AssertTrue boolean isActive;
@Digits	The value must be a number within a specified range.	@Digits(integer=5, fraction=2) BigDecimal amount;
@Max	The value must be an integer value lower than or equal to the number in the value element.	@Max(10) int total;
@Min	The value must be an integer value greater than or equal to the number in the value element.	@Min(5) int quantity;
@NotNull	The value must not be null.	@NotNull String firstname;
@Null	The value must be null.	@Null String tempData;
@Pattern	The value must match the regular expression defined in the regexp element.	@Pattern(regexp="\\"(\d{3}\")\\"d{3}-\d{4}") String phone;
@Size	The size must match the specified boundaries.	@Size(min=3, max=150) String message;

Table 11.1: Bean Validation Built-in Constraints

Code Snippet 2 shows the usage of bean validation constraint to specify non-null persistent fields.

Code Snippet 2:

```
public class Login {
    @NotNull
    private String username;
    @NotNull
    private String password;
}
```

Here, the `@NotNull` annotation is used to specify that the fields `username` and `password` should not be null when they are persisted.

More than one constraint can also be applied on a single JavaBeans component object. For example, you can place an additional constraint for size of field on the `username` and the `password` fields as follows:

```
public class Login {
    @NotNull
    @Size(min=1, max=20)
    private String username;

    @NotNull
    @Size(min=1, max=20)
    private String password;
}
```

→ Primary Keys

Every row in a database has to be uniquely identified through primary key. This concept of primary key has to be implemented in case of entity classes also. There are two variants of a primary key – a simple primary key and a composite primary key. In case of simple primary key, there is only one attribute for unique identification whereas in case of composite primary key, a combination of attributes can be used to uniquely identify an entity.

Primary keys are implemented through attributes of an entity. It is a good design practice to avoid floating point data type as the data type of the key attributes since integral type of data is more portable.

If the primary key of an entity is implemented as a class, then following are the requirements of the class:

- The primary key class must be declared public or protected.
- The class must be serializable.
- The primary key class must have a default constructor defined.
- It must implement the `hashCode()` and `equals()` methods.
- If the primary key is a composite key, then it must be mapped to multiple fields or properties of the entity class. It should be represented as an embeddable class.
- The names and data types of the attributes of the entity class and names of the attributes of the composite primary key should match.

→ Implementing Entity Relationships

The entities in an object-oriented application are related with each other. There are variations in the association of different entities. The association can be one-to-one association, one-to-many association, many-to-one association, and many-to-many association. The EntityManager class manages the entities and their association in the domain.

- An association between two types of entities is said to be one-to-one when one entity of a type is associated with exactly one entity of another entity type. For example, a customer entity can be associated with a bank account entity as a one-to-one association, if all the accounts in the domain are single account holder accounts. This association is represented in the Java program through @OneToOne annotation.
- One-to-many association is where an entity of one type can be associated with more than one entities of another type. Consider an example of Student entities travelling in a school bus, many Student entities are associated with a single Bus entity. This association is represented through @OneToMany annotation.
- Many-to-one association implies that many entities of certain type are associated with one entity of another type. This association in the code is represented through @ManyToOne annotation.
- Many-to-many association implies that every entity of certain type can be associated with many entities of the other type and vice-versa. For instance, when a university is offering multiple courses, each student can enrol into multiple courses offered by the university and several student entities can enrol into a single course. This is a many-to-many association. In Java programs this association is represented through @ManyToMany annotation.

→ Directionality of Relationships

The direction of relationship among entities can be unidirectional or bidirectional.

Unidirectional Relationships

In case of a unidirectional relationship, there is an owner of the relationship. In a unidirectional relationship, only one entity has a relationship field or property that refers to the other. For example, an Order would have a relationship field that identifies Product, but Product would not have a relationship field or property for an Order. In other words, Order knows about Product, but Product does not know which Order instances refer to it.

Bidirectional Relationships

In case of bidirectional relationship, among the two participating entities, one of the entity is the owning side and the other entity is termed as inverse side. For instance, primary-foreign key relationship is a bidirectional relationship.

Consider an example of Employee entity and Department entity. The Department number acts as the foreign key for the Employee which defines the relationship between the Employee and Department entities. In the direction Employee → Department, it is a one to one relationship where an employee entity can be associated with only one Department entity.

In the direction Department → Employee, the relationship is a one-to-many relationship where many employees may be working in a single department.

Following are the annotations associated with the bidirectional relationships:

- In a bidirectional relationship, the inverse side of the relationship uses the mappedby attribute of the annotations @OnetoOne, @OnetoMany, and @ManyToOne to identify the owner of the entity.
- In a many-to-one relationship, represented using the mappedby attribute, the many side of the relationship should be the owner side of the relationship.
- In case of many-to-many relationships, any of the sides can be designated as the owner side.
- In case of one-to-one relationships, which are usually primary key-foreign key relationships, the owning side is the side containing the foreign key.

Cascade Operations and Relationships

Relationships among entities in a domain introduce dependencies in the application. For instance, if a Department is deleted from the database then the association of the Employees in that Department has to be changed and associate the entities to another Department entity or delete the employees associated with the deleted Department entity. This is a cascade operation. Table 11.2 shows the cascade operations supported by the persistence API for the entities.

Cascade Operation	Description
ALL	The operation is cascaded to all the related entities.
DETACH	If the owner entity is removed from the persistence context, then the related entity is also removed.
MERGE	When the owner entity is merged with the persistence context, then the related entity is also merged.
PERSIST	If the owner entity is persisted to the storage, then the related entities are also persisted.
REFRESH	When the owner entity is refreshed in the persistence context, then the related entities are also refreshed.
REMOVE	When the parent entity is deleted from the persistence context then all the related entities are also removed from the persistent context.

Table 11.2: Cascade Operations

→ Embeddable Classes in Entities

Embeddable classes of entities are classes which represent a set of entities whose identity is dependent on the containing entity class. Each embeddable entity class shares the identity of the containing entity class. For instance, in a company scenario, when the company provides insurance to all the dependents of the employee, then the identity of the dependents of the employee is based on the identity of the employee entity. In this case the dependent entity is an embeddable class.

Given here is an embeddable class ZipCode.

```
@Embeddable  
  
public class ZipCode {  
  
    String zip;  
  
    String plusFour;  
  
    ...  
  
}
```

While creating the embeddable class, the @Embeddable annotation is applied and where it is used in the code, the @Embedded annotation is applied.

Code Snippet 3 gives an example of usage of an embeddable class.

Code Snippet 3:

```
@Entity  
  
public class Profile {  
  
    @Id  
  
    protected long SSN;  
  
    String first_name;  
  
    String last_name;  
  
    @Embedded  
  
    ZipCode zipCode;  
  
    String nationality;  
  
    ...  
}
```

11.3 Entity Inheritance

Inheritance is used in Java to ensure code reuse and to distinguish various groups of entities in the domain. A hierarchy of entities can be created in the application domain through inheritance, where certain entity sub-classes another entity. When a database query is submitted for the base class of the entity hierarchy then the query implementation is treated against the entire hierarchy.

→ Abstract Entities

According to basic object-oriented programming, abstract classes cannot be instantiated. Similarly, in case of persistence API, the abstract entities and concrete entities are treated in a similar manner, but the abstract entities cannot be instantiated.

However, both concrete entities and abstract entities can be queried. If the target of a query is an abstract entity, the query will also operate on all the concrete subclasses of the abstract entity.

To declare an abstract class, @Entity annotation is specified on the class.

Code Snippet 4 shows an example of abstract entity.

Code Snippet 4:

```
@Entity
public abstract class ConceptCar{
    @Id
    protected String carId;
    ...
}

@Entity
public class SupersonicConceptCar extends ConceptCar{.....}

@Entity
public class MinimumfuelConceptCar extends ConceptCar{.....}
```

Any query on the abstract entity class ConceptCar is further redirected to the concrete entity subclasses SuperSonicConceptCar and MinimumFuelConceptCar.

→ Mapped Superclasses

Unlike abstract classes, mapped superclasses are not entities. Abstract superclasses cannot be instantiated but mapped superclasses can be instantiated. The @Entity annotation is not used with this type of superclass. Also, it is not mapped as an entity by the Java Persistence provider. These superclasses are used when there is certain common state and mapping information pertaining to all the sub-classes of the super class.

You cannot query Mapped superclasses nor use it in EntityManager or Query operations. For this purpose, you have to use the entity subclasses of the mapped superclass. Mapped superclasses can be abstract or concrete but cannot be targets of entity relationships.

There cannot be any tables corresponding to the Mapped superclasses in the datastore. The table mappings are defined by the entity subclasses that inherit from the mapped superclass. For example, in Code Snippet 5, the mapped tables would be CS_STUDENT and EC_STUDENT, but there will not be any STUDENT table.

To create a Mapped superclass, specify the @MappedSuperClass annotation on top of the class.

Code Snippet 5 shows an example of Mapped superclass.

Code Snippet 5:

```
@MappedSuperClass
public class Student{
    @Id
    protected String studId;
    .....
}
public class CS_Student extends Student{
    .....
}
public class EC_Student extends Student{
    .....
}
```

In this scenario, there is certain state associated with every student of the university which is represented in the class Student. The scenario also suggests that if there is a student existing in the university, then the Student entity has to be mapped to some department such as CS or EC. Here, the Student class is the mapped superclass, whose entities are not created.

→ **Non-Entity Superclasses**

Other than Abstract and Mapped superclasses, entities may also have non-entity superclasses. Again, these superclasses may be abstract or concrete. However, the non-entity superclasses do not have persistent state. Also, any state inherited from the non-entity superclass by an entity subclass is nonpersistent. Similar to Mapped superclasses, non-entity superclasses also may not be used in EntityManager or Query operations. Any mapping or relationship annotations in non-entity superclasses are ignored.

→ **Entity Inheritance Mapping Strategies**

The entities of the application domain have to be systematically mapped to the tables of the relational database to ensure efficient retrieval of the data from the database. Following are the strategies of mapping the entities onto the database:

- A single table per class hierarchy
- A table per concrete entity class
- A join strategy

The default mapping strategy is single table per class hierarchy.

The strategy element of @Inheritance annotation is used to set the strategy. The possible values of strategy are defined in the javax.persistence.InheritanceType enumerated type as follows:

```
public enum InheritanceType {
    SINGLE_TABLE,
    JOINED,
    TABLE_PER_CLASS
};
```

→ A Single Table Per Class Hierarchy

This strategy corresponds to the default InheritanceType.SINGLE_TABLE. According to a single table per class hierarchy, all the classes in a hierarchy are mapped to a single table. There is a discriminator column in the table which identifies which class the entity belongs to.

The discriminator column has a combination of four properties to identify the class of the hierarchy to which it belongs. Following are the properties of a discriminator column:

- **String name** – name of the discriminator column
- **Length** – The length of the string discriminator, it is ignored if it is a non-string discriminator
- **String columnDefinition** – database specific column type which is used by the SQL queries in the database
- **DiscriminatorType** – This is an enumerated type describing the strategy of mapping the database onto the entities

→ Table Per Concrete Entity Class

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.

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.

→ The Joined Subclass Strategy

This strategy corresponds to InheritanceType.JOINED. This strategy uses SQL join operation to merge the data from different classes in the hierarchy. The root of the class hierarchy is represented by a single table and every sub-class of the hierarchy is represented through a different table. Every table has a primary key. The superclasses and sub-classes are linked through the primary key and foreign key relationships (foreign key is an attribute which refers to the attribute of the entity super class associated with the entity subclass).

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.

11.4 Managing Entities

Java Persistence API uses entity managers to manage the entities. These entity managers are instances of javax.persistence.EntityManager interface. An instance of entity manager is associated with a persistence context. A persistence context implies a set of entities in the data store of the application. The entity manager is responsible for managing the persistence context which includes creating persistent instances of entities, persisting this data on to the database, and removing these entities from the data store.

→ EntityManager Interface

JPA has an EntityManager interface which is responsible for creating persistent entities and removing them from the data store, retrieving the data from the database based on the primary key values, and executing SQL queries on these entities.

→ Container Managed Entity Managers

All the components of an application are present within a container and the lifecycle of the entity manager is managed by the container. A container managed entity manager is responsible to maintain a consistent persistence context across all the components of the application which use EntityManager instance.

A transaction refers to a set of operations carried out in an application which store or retrieve persistent entities from the data store. A transaction would require making calls to multiple components of the application and all the components of the application should access a single persistence context. The EntityManager automatically propagates the persistence context among the components of the application. This removes the need of passing explicit references to the EntityManager instances which can be further used by other components in a single transaction.

An instance of EntityManager can be created as follows:

```
@PersistenceContext  
EntityManager e;
```

→ Application Managed Entity Managers

In case of application managed entity managers, the lifecycle of the entity manager is managed by the application. An application managed entity manager is used when an isolated persistence context is required.

Applications create entity manager instances through the method createEntityManager() using an instance of EntityManagerFactory.

An instance of EntityManager can be created by creating the instance of EntityManagerFactory by using the createEntityManager() method as follows:

```
@PersistenceUnit  
EntityManagerFactory ef;  
EntityManager em = ef.createEntityManager();
```

The transaction manager has to explicitly start the transaction and define the scope of the entities while performing the entity operations.

11.4.1 Entity Lifecycle

The entity manager also manages the lifecycle of the entity instances. There are four stages in the lifecycle of an entity instance – new, managed, detached, and removed.

- New entity instances have no persistent state or identity. These instances are not associated with any persistence context.
- Managed entity instance has a persistent state and identity. They are associated with a persistence context.
- Detached entity instances are not associated with any persistence context but they have an identity.
- Removed entity instances have a persistent identity, are associated with a persistent context, and are scheduled for removal from the data store.

The EntityManager interface has various methods which can be used to perform operations and manipulate the state of the entities in the persistence context. These methods are as follows:

- find() method is used to find an entity instance in the data store with the help of primary key of the entity instance.
- persist() method is invoked by EntityManager to make the new entity instances persistent and managed. That is, to associate the entity with an identity and a persistence context.
- remove() method is invoked by the EntityManager to remove managed entity instances from the data store.
- flush() method is used to synchronize the managed entity with the data store.

Persistence Unit

Persistence unit refers to all the entity classes in the application, which are stored in a single data store and are managed by a single EntityManager. The persistence unit of an application is defined by the persistence.xml configuration file.

In this configuration file, the name of the database, the database driver required to access the database, and the entity classes of the database are defined.

Code Snippet 6 illustrates how to define a persistence unit in a persistence.xml file.

Code Snippet 6:

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

<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">

  <persistence-unit name="SamplePersistentUnit">
    <description>For illustration</description>
    <provider>com.objectdb.jpa.Provider</provider>
    <mapping-file>META-INF/mappingFile.xml</mapping-file>
    <jar-file>Company.jar</jar-file>
    <class>Company.Employee</class>
    <class>Company.Department</class>
    <properties>
      <property name="javax.persistence.jdbc.url"
        value="objectdb://localhost/my.odb"/>
      <property name="javax.persistence.jdbc.user" value="admin"/>
      <property name="javax.persistence.jdbc.password" value="admin"/>
    </properties>
  </persistence-unit>
</persistence>
```

11.5 Querying Entities

In order to query entities, Java Persistence API has various methods as follows:

- Java Persistence Query Language (JPQL)
- Criteria API

Java Persistence Query Language is a string based language used to query databases like SQL from the object-oriented context. JPQL queries are very similar to SQL queries and can be defined in the entity class or deployment descriptor. Type casting of the results of JPQL queries is required in the entity class.

Criteria API are also used to create type safe queries from a Java program. Criteria queries are written in the business logic of the application. Unlike JPQL they do not require type casting of the query result.

11.6 Database Schema Creation

The deployment descriptor persistence.xml can be used to configure the application to automatically create and drop database schemas and also load data into the tables. Scripts for performing these tasks can be explicitly defined and used in the persistence.xml file to perform the required operations.

Java Persistence API provides various properties which can be used in the process of creating the database schema. Following are the properties provided by the Java Persistence API which can be used along with the XML tags to define the properties of the persistence unit in the application.

- javax.persistence.schema-generation.database.action

This property is used to specify the action to be taken when the application is deployed. This property enables creating and dropping the data sources in the current persistence unit.

This property can assume any one of the following four values:

- **none** – The none value implies that no database is to be created.
- **create** – The create value of the property implies that a database schema has to be created.
- **drop-and-create** – The drop-and-create value of the property implies that any existing database has to be dropped and a new schema overwriting the old schema has to be created.
- **drop** – The value drop implies that the database schema has to be dropped.

- The properties javax.persistence.schema-generation.create-source and javax.persistence.schema-generation.drop-source are used to define how the database source is created. There are four options to choose from while creating and dropping the source.

- **metadata** – This value of the property implies that the object/relational metadata can be used to create or delete the data sources.
 - **script** – The value script refers to a SQL script that can be used to create or delete the database. When the value is provided to the script then the path to locate the script has to be provided which is relative to the root of the persistence unit.
 - **metadata-then-script** – This value of the property indicates the usage of metadata first and then the SQL script to create or delete the database sources.
 - **script-then-metadata** – This value of the property indicates the usage of script first and then metadata to create or delete the data sources.
- javax.persistence.sql-load-script-source property can be used to define the SQL script to load the data from the data source. This script location has to be specified relative to the persistence unit.

11.7 Java Persistence Query Language

Java Persistence Query Language enables creation of SQL like queries which can be used by the application to access and manipulate a database. The entity manager creates required queries for the application using the methods `createQuery` and `createNamedQuery`.

The `createQuery` method is used to create dynamic queries in the application's business logic, where the parameters of the query are accepted at runtime and `createNamedQuery` is used to create static queries, where the parameters of the query are passed as part of the code.

Code Snippet 7 shows the method of creating a dynamic query using `createQuery` method.

Code Snippet 7:

```
.....
public List findStudent(String name) {
    return em.createQuery(
        "SELECT s FROM Student c WHERE c.name LIKE :Name")
        .setParameter("Name", name)
        .setMaxResults(10)
        .getResultList();
}
.....
```

In the given query, all the entity instances of entity class `Student` are retrieved through the query. The required parameter for the query is set through the `setParameter` method. The method `setMaxResults()` limits the number of instances that can be returned from the execution of the query and the `getResultList()` method reads the result list and provides required memory for saving the list.

The `createNamedQuery` method is used to create static queries, which implies that the parameters for the query are not accepted dynamically at runtime.

Code Snippet 8 shows an example for `createNamedQuery`.

Code Snippet 8:

```
.....
students = em.createNamedQuery("SELECT s FROM Student c WHERE c.name LIKE :Name")
    .setParameter("StudentName", "Alice")
    .getResultList();
.......
```

Code Snippet 8 uses `createNamedQuery` where the parameter value is set to 'Alice'. This parameter value cannot change during execution.

There are two variants of parameters of the queries:

1. Named parameters
2. Positional parameters

The named parameters in a query are prefixed with a colon ':'. In Code Snippet 7 '`:Name`' is an example of the named parameter. The values to the named parameters are set through the method `setParameter`.

Positional parameters are prefixed with a '?' symbol followed by the numeric position of the parameter in the query. In order to set this parameter, an overloaded form of the `setParameter` method is invoked which accepts the numerical position and the corresponding parameter as argument.

Code Snippet 9 shows the usage of positional parameters.

Code Snippet 9:

```
Query query = em.createQuery(
    "SELECT * FROM Student s WHERE s.roll_no= ?1 AND s.course= ?2");
query.setParameter(1, 1001);
query.setParameter(2, "CS");
```

11.7.1 Writing SQL Queries in JPQL

The data in the database is accessed through SQL queries. Java Persistence API provides Java Persistence Query Language (JPQL) to write queries on the database.

→ **Select Query**

Select query is the most commonly used SQL query in database transactions. Following is the format for writing a JPQL statement to accommodate a select query.

```
QL_statement ::= select_clause from  
[where] [group by] [having] [orderby]
```

In the given syntax, the select clause and from keyword are essential. Remaining clauses in the square brackets are optional in a SELECT statement. The select_clause has all the properties or fields of the entity class which are expected to be returned by the query.

The from clause, holds the reference to the data source objects from which the data has to be retrieved. This can be a schema name, a collection, or an element of a single valued relationship.

The where clause, represents the condition based on which objects are expected to be retrieved from the entity classes.

The groupby clause is used to create logical groups among the data objects. This clause can also be used along with having clause when data is to be retrieved based on some condition (filter criteria) on the group.

The orderby clause is used to specify some order on all the entities of the entity class that are being retrieved by the query. Update and Delete statements can also be written on entities as follows:

```
upd_stmt ::= update_clause [where]  
dlt_stmt ::= delete_clause [where]
```

The where statement is optional in the Update and Delete statements. It is used only when there is a condition to be applied for the operation.

11. 8 Application Using Persistence API

A Web application using Persistence has several components and which need to be interconnected to demonstrate the usage of persistence in an application.

→ Creating a Web Application Using JSF

To understand the use of the Persistence API, create a Web application using JSF in the IDE by selecting File → New Project → Java Web → Web application. The New Web Application dialog box is displayed as shown in figure 11.3.

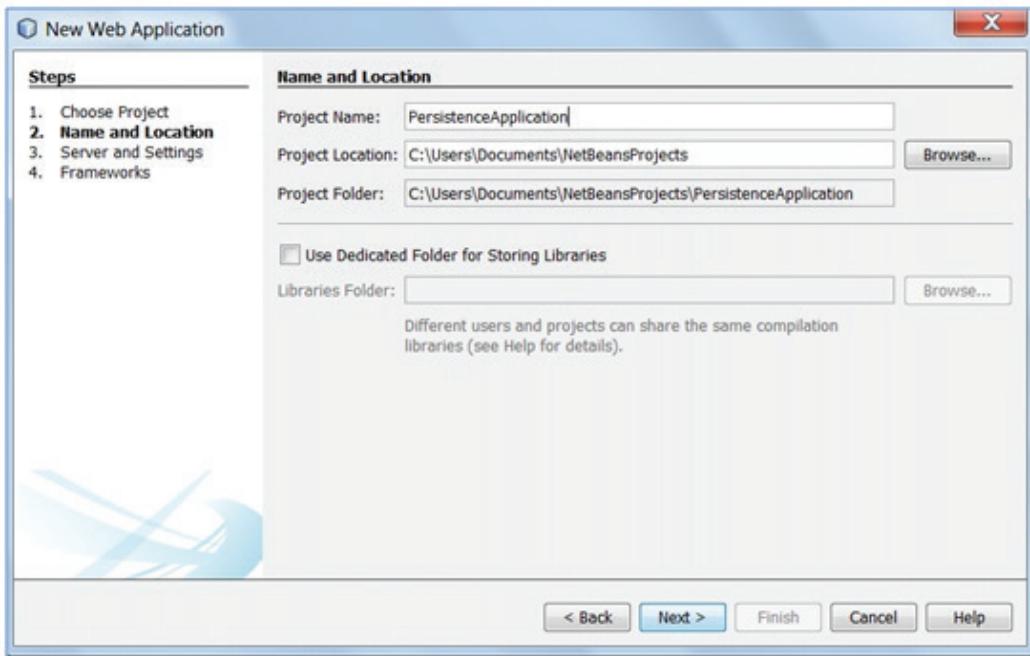


Figure 11.3: Creating a Web Application

Specify the name of the application as PersistenceApplication and click Next.

Select the GlassFish server and Java EE 7 version and click Next.

Select the JavaServer Faces checkbox and click Finish.

→ Creating the Entity Class and Persistence Unit

Once the Web application is created, add an entity class named Message to the application to store the messages. .

To create the entity class, right-click the project and select New → Other → Persistence → Entity Class from the New File dialog box as shown in figure 11.4

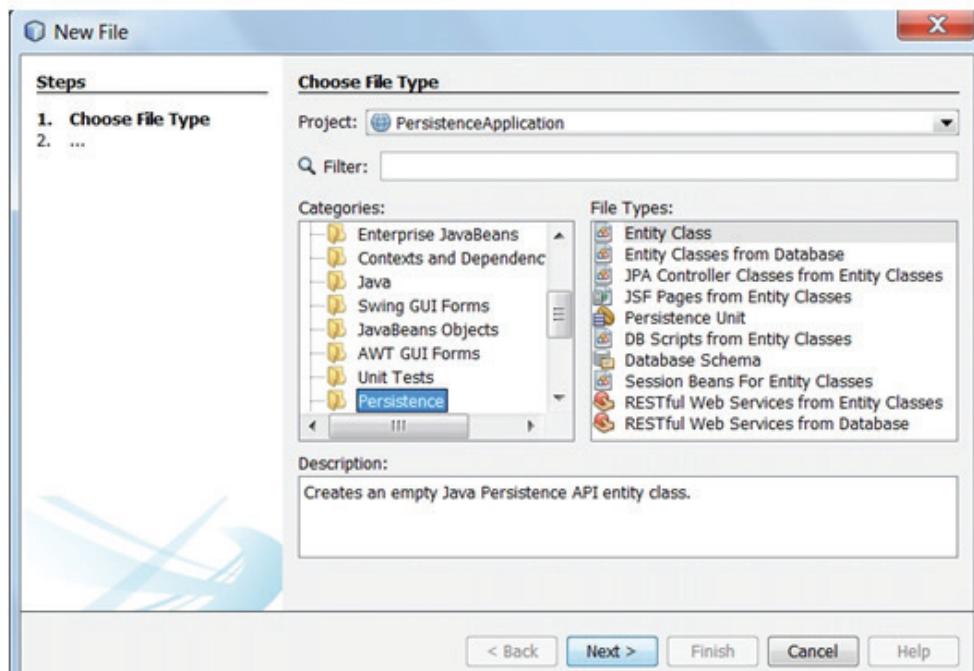


Figure 11.4: Creating an Entity Class

Specify the Name and Location details as shown in figure 11.5.

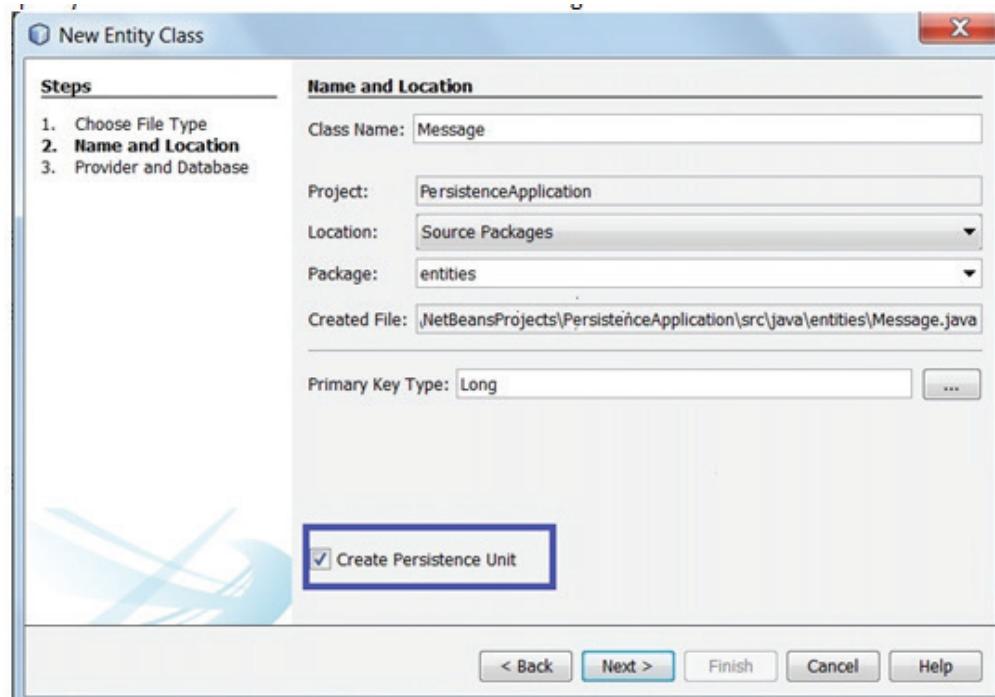


Figure 11.5: Choosing the Entity Class Name and Package

Note that every entity class must have a primary key. By default the primary key type is selected as Long.

Select the Create Persistence Unit checkbox. This will create a Persistence Unit automatically on completion of the Entity class creation.

Click Next. The Provider and Database screen is displayed. Specify the settings as shown in figure 11.6.

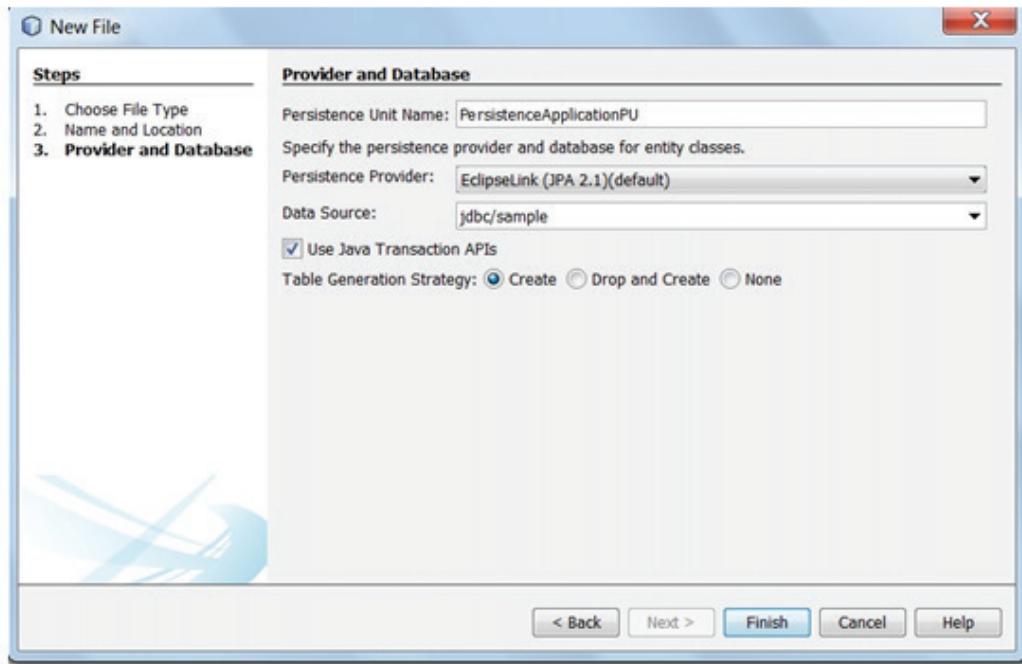


Figure 11.6: Provider and Database Settings

Here, the Persistence Unit Name is taken by default and the Persistence Provider is set to EclipseLink. You can modify these if needed.

Select the DataSource as jdbc/sample from the drop-down. The jdbc/sample database comes bundled with JavaDB in the server installation directory.

Ensure that the Use Java Transaction APIs checkbox is selected and the Table Generation Strategy is set to Create.

Click Finish.

The IDE creates the entity class and opens the class in the editor.

You can see that the IDE generated the id field private Long id; and annotated the field with @Id and @GeneratedValue(strategy = GenerationType.AUTO).

This means that Id is the primary key of the Entity class Message and it is an auto-generated value.

Add a new attribute named message in the Message class as shown in Code Snippet 10.

Code Snippet 10:

```
package entities;

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 id;
    private String message;

    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id=id;
    }
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message=message;
    }
}
```

```

@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 + " ]";
}
}

```

The code generated comprises the attribute ‘Id’ and getter and setter methods. 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.

The attribute message and its getter and setter methods have been added. Connect the Entity class to the JavaDB database which is an integral part of the NetBeans IDE.

Note: To generate getter-setter of the message attribute, right-click in the editor and choose Insert Code (Alt-Insert) and then, select Getter and Setter. In the Generate Getters and Setters dialog box, select the message field and click Generate. The IDE generates getter and setter methods for the field message.

The entity class created will represent a table in the database. When this application is executed, a table for the Message entity will be automatically created in the database. The table will contain the columns id and message.

In the persistence unit (created in the Configuration Files folder), you can see JTA mentioned in the transaction-type in the XML editor (transaction-type="JTA"). This indicates that the container is assigned the responsibility for managing the lifecycle of entities in the persistence context. This reduces the coding effort as the entity lifecycle will be managed by the container and not by the application.

→ Creating the Session Façade (Session Bean)

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.

To create a session bean, right-click the project and select New → Other → Enterprise JavaBeans → Session Beans For Entity Classes as shown in figure 11.7.

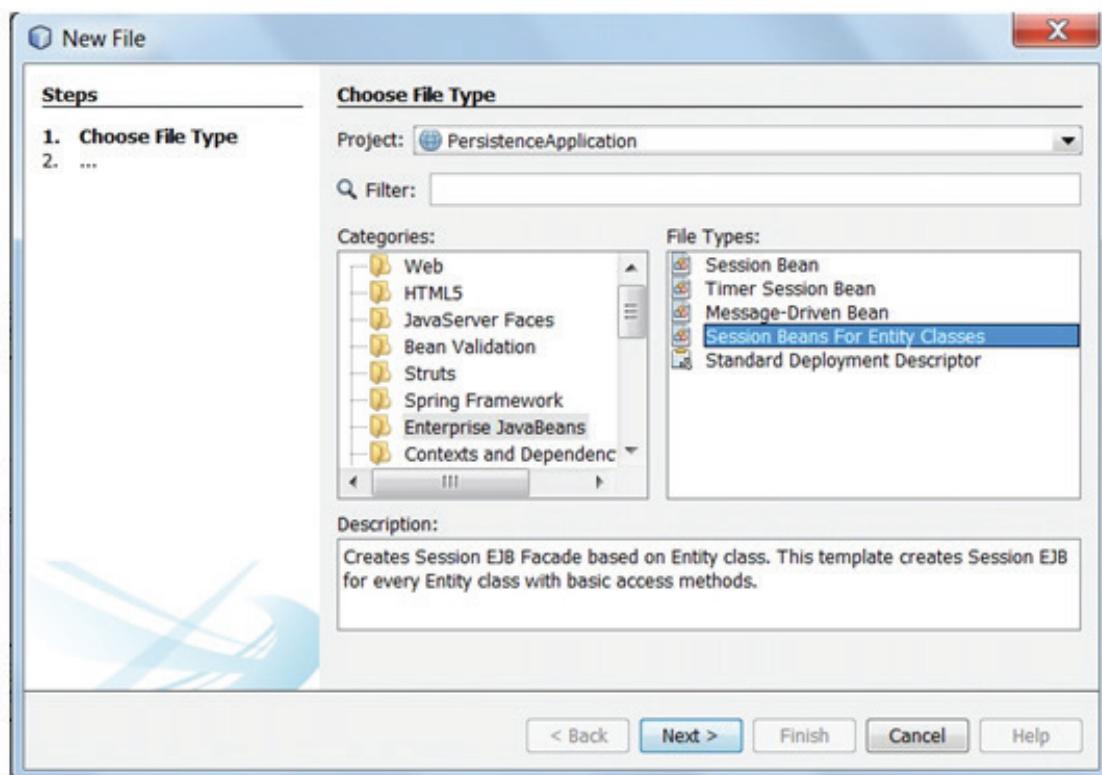


Figure 11.7: Creating a Session Bean for Message Entity Class

Click Next. Select entities.Message from the Available Entity Classes and click Add. The Message entity will be added to the Selected Entity Classes list as shown in figure 11.8.

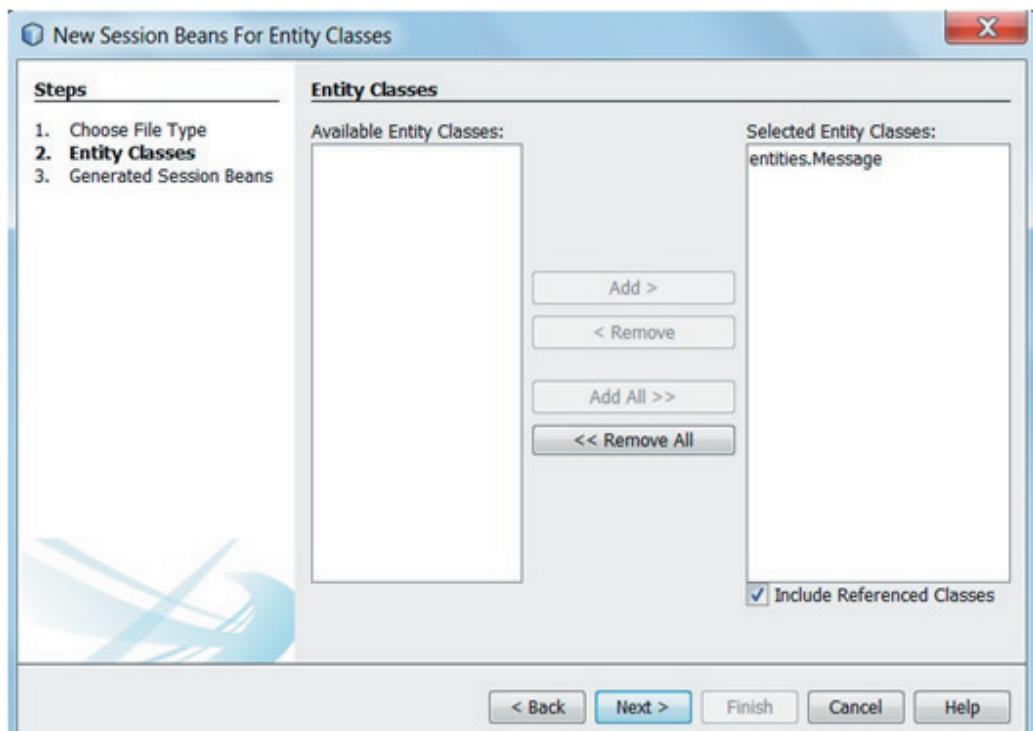


Figure 11.8: Adding Entities to the Session Bean

Click Next. Create the session bean in a package ‘façade’, this package name is based on the developer’s choice. Figure 11.9 shows how the session bean is created using the wizard.

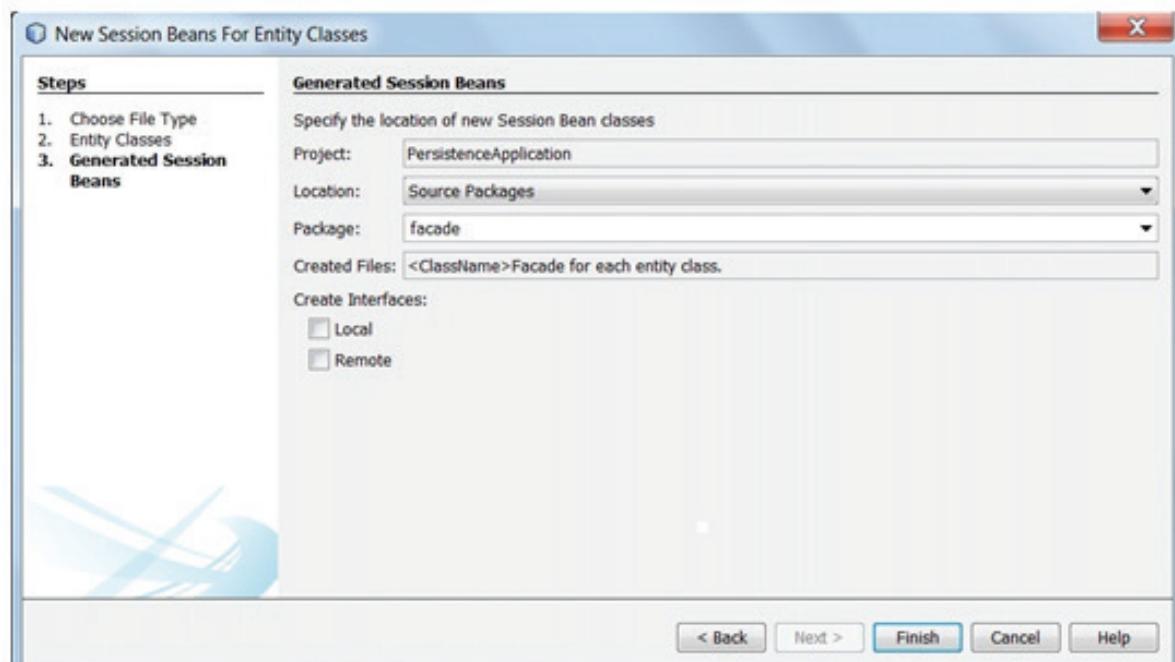


Figure 11.9 Creating the Session Bean for Entity Classes in Package

Note that it is not required to create a business interface for the session bean. Instead, in this example the bean will be exposed to a local managed bean using a no-interface view.

Click Finish.

The IDE generates the session facade class named MessageFacade.java and AbstractFacade.java.

The annotation @Stateless is used to declare MessageFacade.java as a stateless session bean component.

Also, MessageFacade.java extends AbstractFacade.java, which contains the business logic and manages the transaction.

All this is done to create a local managed bean.

Code Snippet 11 shows the code in AbstractFacade.java, this code is generated by the IDE.

Code Snippet 11:

```
package facade;  
  
import java.util.List;  
  
import javax.persistence.EntityManager;  
  
public abstract class AbstractFacade<T> {  
    private Class<T> entityClass;  
  
    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 11 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. `findAll()` method retrieves all the entities in the database and `count()` method is used to count the number of entities in the database. `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. The EntityManager is part of Persistence API, which is used to access the entities in the database.

When the session facade for the entity is created using the wizard, by default the IDE adds the PersistenceContext annotation (@PersistenceContext(unitName = "PersistenceApplicationPU")) to inject the entity manager resource into the session bean component and to specify the name of the persistence unit.

Code Snippet 12 shows the code of MessageFacade class.

Code Snippet 12:

```
package facade;

import entities.Message;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.PersistenceContext(unitName = "PersistenceApplicationPU")
private EntityManager em;

@Override
protected EntityManager getEntityManager() {
    return em;
}
public MessageFacade() {
    super(Message.class);
}
```

In Code Snippet 12, the code injects the resource of the Persistence unit through an annotation and then invokes an entity manager for this context.

The method getEntityManager returns an object of type EntityManager and the constructor invokes a super class constructor.

→ Defining the User Interface

In this example, the user interface will be created using JSF.

First create a JSF managed bean by right-clicking the project and selecting New → Other → JavaServer Faces → JSF Managed Bean as shown in figure 11.10.

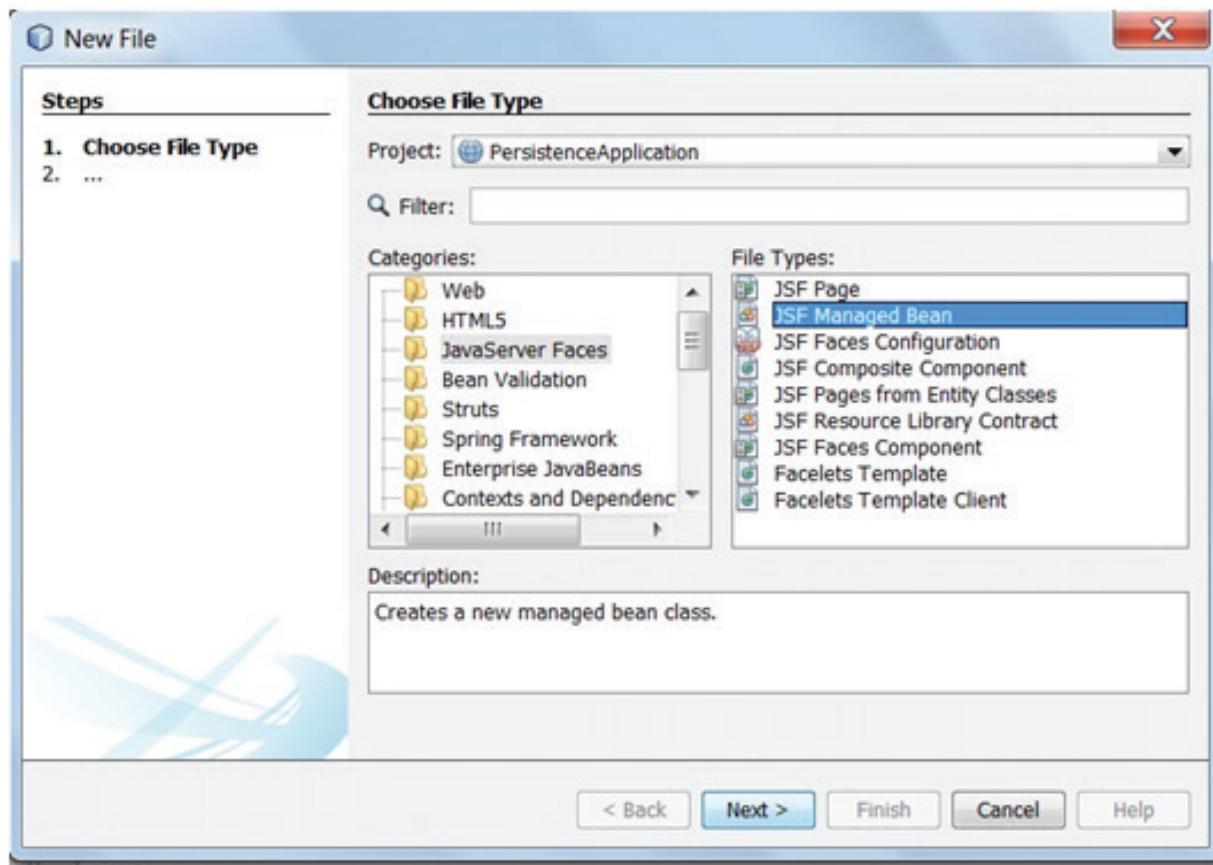


Figure 11.10: Creating a JSF Managed Bean

Click Next.

Specify appropriate values for the options in the wizard as shown in figure 11.11.

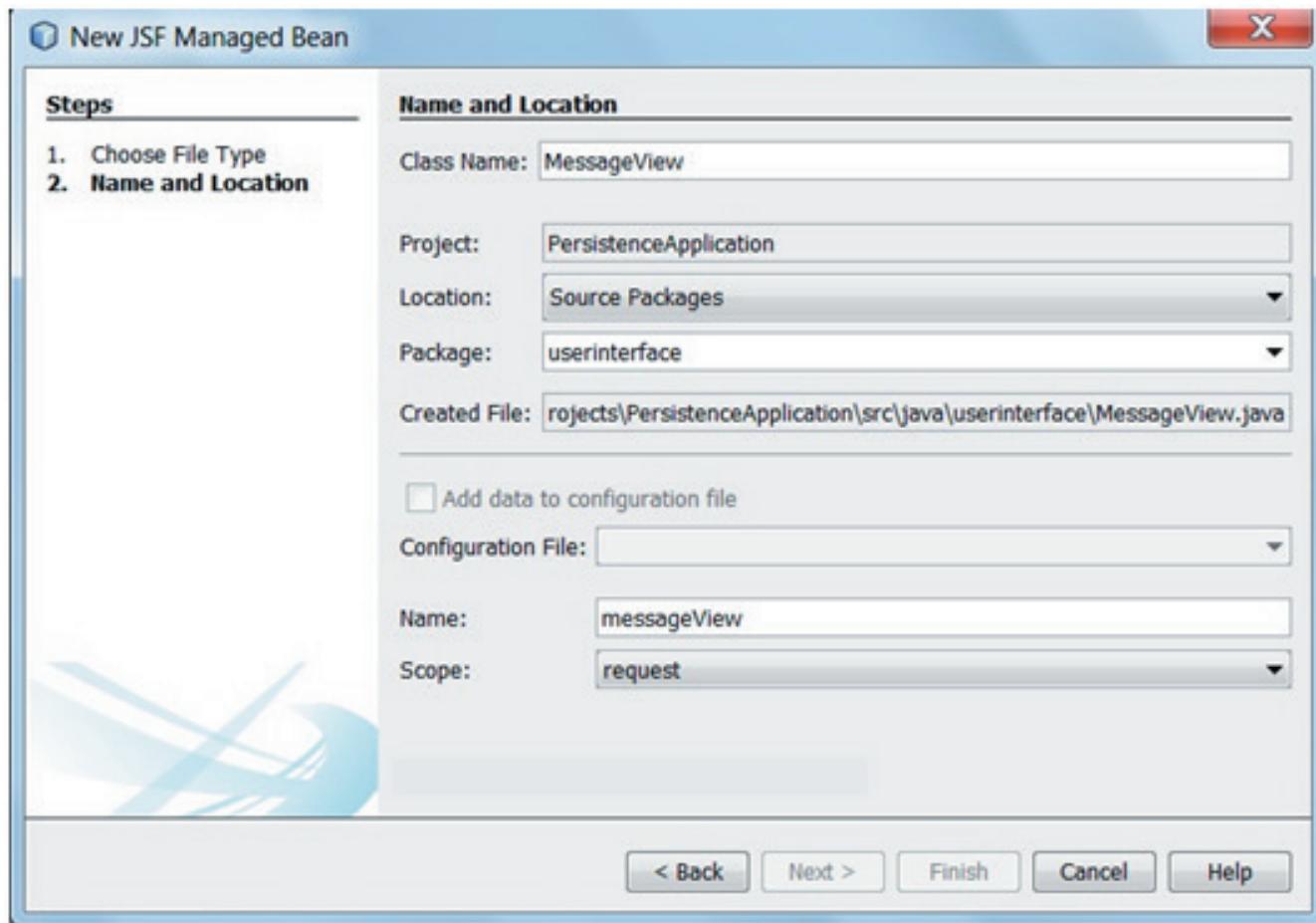


Figure 11.11: New JSF Managed Bean

Click Finish. The JSF ManagedBean is created and opened in the editor.

The IDE adds the @ManagedBean and @RequestScoped annotations and the name of the bean. Now, a reference of the Session bean created earlier must be added to the ManagedBean to invoke its methods.

To add the session bean reference, press Alt+Insert. This provides options of code that can be added in the MessageView class. Select 'Call Enterprise bean' option in the menu as shown in figure 11.12.

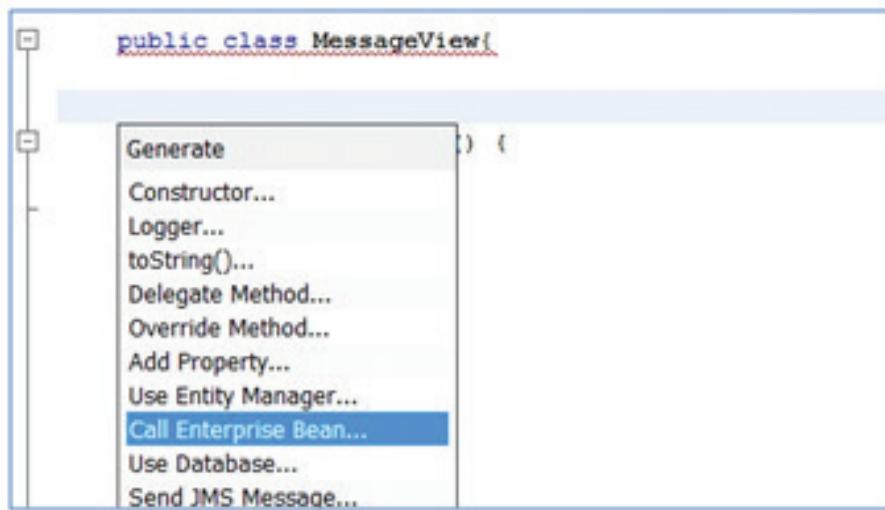


Figure 11.12: Adding an Enterprise Bean to the MessageView Class

Select the MessageFacade bean as shown in figure 11.13.

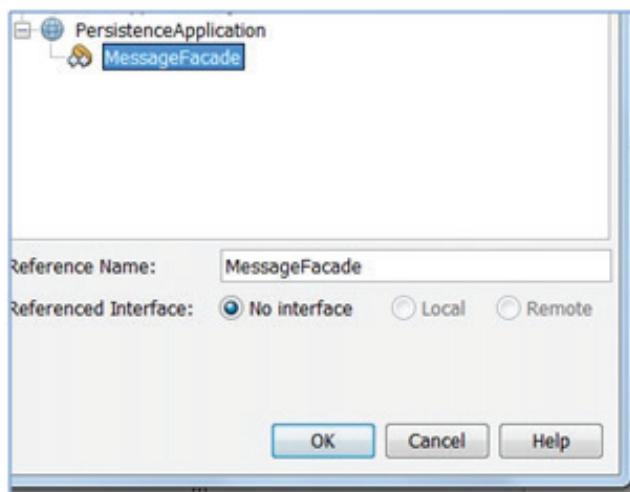


Figure 11.13: Selecting the Enterprise Bean

Notice that by default, the Referenced Interface is set to No interface. This is because the Session bean was created with a no-interface view.

Click OK.

A reference to the Session bean is added to the MessageView class using the @EJB annotation as depicted in Code Snippet 13.

Code Snippet 13:

```
public class MessageView {  
    @EJB  
    private MessageFacade messageFacade;  
  
    /**  
     * Creates a new instance of MessageView  
     */  
    public MessageView() {  
    }  
}
```

Modify the class by adding the message variable, initializing it in the constructor, getter method, and adding the getNumberOfMessages and postMessage methods as depicted in Code Snippet 14.

Code Snippet 14:

```
public class MessageView {  
    @EJB  
    private MessageFacade messageFacade;  
    // Creates a new field  
    private Message message;  
    /**  
     * Creates a new instance of MessageView  
     */  
    public MessageView() {  
        this.message = new Message();  
    }  
    // Calls getMessage to retrieve the message  
    public Message getMessage() {  
        return message;  
    }
```

```
// Returns the total number of messages
public int getNumberOfMessages() {
    return messageFacade.findAll().size();
}

// Saves the message and then returns the string "theend"
public String postMessage() {
    this.messageFacade.create(message);
    return "response";
}
```

Code Snippet 14 has three methods – getMessage(), getNumberofMessages() and postMessage().

The getMessage() retrieves the message from the user interface. The method getNumberofMessages() returns the number of messages in the database.

The method postMessage() creates a message entity in the database and the create() method also defined in the AbstractFacade class, is accessed through MessageFacade class. To remove the error symbols that appear in the code, right-click in the editor and choose Fix Imports.

Note that the postMessage method returns the string “response”.

The JSF 2.x specification allows using implicit navigation rules in applications that use Facelets technology.

In this example, no navigation rules have been configured in the faces-config.xml file. Instead, when the postMessage method is invoked, the navigation handler will try to locate a suitable page in the application, in this case, a page named response.xhtml.

→ Creating the index Page

Modify the index page of the application according to the code shown in Code Snippet 15 to add user interface components to the Web page.

Code Snippet 15:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <h:head>
        <title>Persistence Application</title>
    </h:head>
```

```

<h:body>
    <h:form>
        <h:outputLabel value="Message:"/><h:inputText value="#{messageView.message.message}" />
        <h:commandButton action="#{messageView.postMessage()}" value="Post Message" />
    </h:form>
</h:body>
</html>

```

In Code Snippet 15, user interface components such as text label, input text field and a command button are added. The command button action invokes the `postMessage()` method in the JSF Managed bean. The `postMessage()` method returns a String object which refers to the ‘response’ page mentioned in Code Snippet 14.

Note: In case you see an error/warning symbol in the left margin next to the line containing `<f:view>`, click the symbol and select Add `xmlns:f="http://xmlns.jcp.org/jsf/core"` library declaration.

→ Creating the response Page

Create a new JSF page with the name `response.xhtml`. Code Snippet 16 shows the response page `response.xhtml`.

Code Snippet 16:

```

<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <h:head>
        <title>Response</title>
    </h:head>
    <h:body>
        <h:outputLabel value="There are ">
        <h:outputText value="#{messageView.numberOfMessages}" />
        <h:outputLabel value="messages!" />
    </h:body>
</html>

```

In Code Snippet 16, the bean method `numberOfMessages` is invoked which returns the number of messages in the database.

On executing the application, the `index.xhtml` page is displayed. Type the message 'Good Morning' in the textbox as shown in figure 11.14.

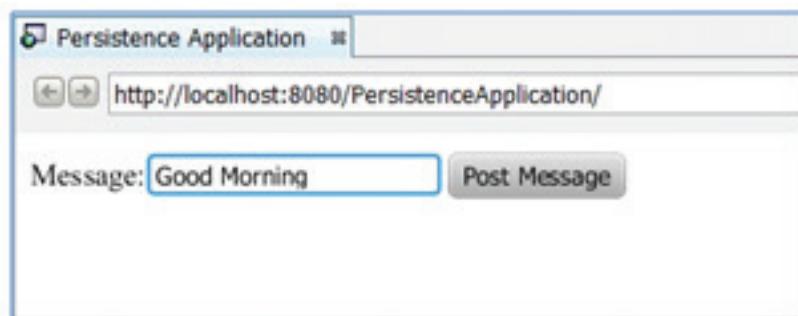


Figure 11.14: Running the Persistence Application

Click the Post Message button. The response page is displayed as shown in figure 11.15.

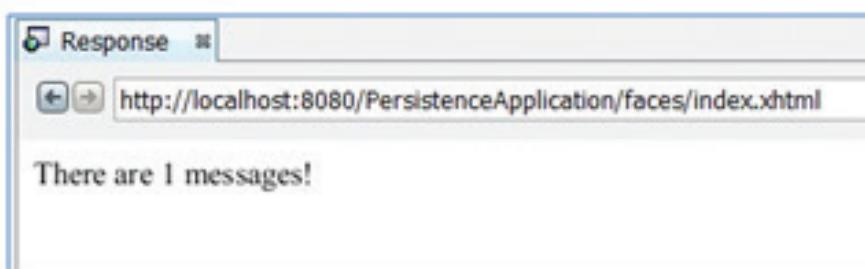


Figure 11.15: Response Page

The response page displays the number of messages in the database.

11.9 Check Your Progress

1. An entity class variable can assume which of the following data types?

(A)	Primitive data types	(C)	Collections
(B)	Wrapper classes for primitive data types	(D)	All of these

2. Which of the following statements about an entity class are false?

(A)	An entity class should be thread safe.	(C)	It should have a default constructor.
(B)	An entity class cannot be further extended.	(D)	It can have overloaded constructors.

3. Which of the following statements about primary key classes are false?

(A)	The primary key class should be an embedded class.	(C)	It should have a default constructor.
(B)	It should define the hashCode() and equals() methods.	(D)	None of these

4. Which of the following are the two variants of parameters in queries?

a.	Named parameters
b.	String parameters
c.	Positional parameters
d.	Hash values

(A)	a, c	(C)	a, b
(B)	b, d	(D)	c, d

5. Which of the following operations is not a cascading operation on the entities?

(A)	JOIN	(C)	DETACH
(B)	REFRESH	(D)	PERSIST

11.9.1 Answers

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

Summary

- Java Persistence API performs the object-relational mapping of data, where the database stores the data in the form of tables and the application handles the data in the form of objects.
- The entities are modelled as entity classes by the JPA.
- The entity classes can accept the data in both static and dynamic forms through entity properties and fields.
- The entities in the application can be associated with each other in one-to-one, one-to-many, many-to-one, and many-to-many relationships.
- The entity inheritance in the object-oriented mode can be mapped to relations through single table per class hierarchy, a table per concrete entity class, or through join strategy of the relations.
- JPQL is used to implement the SQL operations on the database.



To enhance your knowledge,
visit the **REFERENCES** page



Session - 12

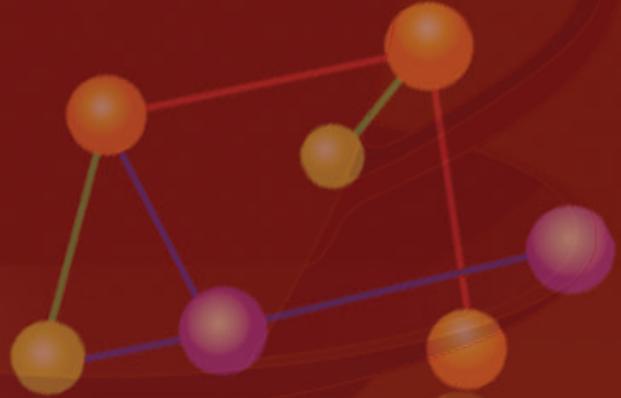
Transactions

Welcome to the Session, **Transactions**.

This session describes transactions and how to handle transactions in Java EE applications. It explains container managed, bean managed transactions, and transactions that update multiple databases. It also describes transaction management in Web applications and how to control concurrent data access using persistence API.

In this Session, you will learn to:

- ➔ Describe handling transactions in Java EE applications
- ➔ Describe container managed and bean managed transactions
- ➔ Describe managing transactions that update multiple databases
- ➔ Explain transaction management in Web applications and Web components
- ➔ Explain how to control concurrent data access through Persistence API



12.1 Transaction

Transaction refers to a series of actions performed on the database, with a condition that either the entire set of actions should execute and complete or none of the actions are executed. If a partial set of actions are executed then these actions should be rolled back. A transaction is supposed to have the following essential properties known as ACID properties:

- **Atomicity** – Atomicity of a transaction implies that either all the operations of the transaction execute/commit or none of the operations in the transaction execute.
- **Consistency** – The execution of the transaction on a set of data should give consistent results. This is relevant when multiple operations of different transactions are inter leaved to achieve better performance.
- **Integrity** – The transaction, when executed, should not alter the database state in a way that the domain constraints are violated.
- **Durability** – The changes made by the transactions, once committed, should be retained in the database irrespective of situations such as system reboot or recovery from a failure.

Java EE provides a Java transaction API which allows applications to access transactions in a secure manner ensuring the integrity of data.

Consider the scenario of operating a bank account. Here, the account holder deposits cash into the account and then, checks the balance. Following are the operations involved in accessing a bank account:

1. Logging into the bank application
2. Depositing money into the account
3. Checking the new balance in the account
4. Logging out of the account

Either all the operations in this transaction have to execute or none of them are executed.

A transaction may end in two ways - commit or rollback. When a transaction is committed, all the changes made by the transaction to the database are made permanent in the database. When a transaction is rolled back the effect of all the operations in the transaction are reversed.

Java supports programmatic transactions (Bean Managed Transactions or BMT) and Java Transaction API (JTA) transactions (Container Managed Transactions or CMT). In programmatic transactions, the developer writes code for creating and managing the transactions through JDBC. In JTA, transactions are created by using the interfaces provided by Java APIs.

Transactions can be Container Managed or Bean Managed transactions. In case of Container Managed Transactions, the lifecycle of the transaction is managed by the container. In case of Bean Managed Transactions the code for transaction management should be explicitly written by the developer.

The Java Transaction API specifies an interface between the transaction manager and other components participating in the transaction. Other entities in the transaction may be the client, database, bean components, and so on. The JTA defines a UserTransaction interface which can be used by applications to start, commit, or rollback transactions.

12. 2 Container Managed Transactions

In case of container managed transactions, 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. There is no definite demarcation of the transaction in this case, that is, 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 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 methods of `java.sql.Connection`
- `getUserTransaction` of `javax.ejb.EJBContext`
- all methods of `javax.transaction.UserTransaction`
- **Transaction Attributes**

A transaction attribute is required to define the scope of the enterprise bean methods and transactions associated with these methods in the deployment descriptor. There are six possible values of a transaction attribute as follows:

- Required
- RequiresNew
- Mandatory
- NotSupported
- Supports
- Never

Required attribute is associated with an enterprise bean method. It implies that the enterprise bean method must be executed in a transaction context. If a user transaction already exists, the container will execute the method in the current transaction context. If there is no pre-existing transaction context, the container will create a new transaction context and commit the transaction before the method. Required transaction attribute is an implicit transaction attribute to all the enterprise bean methods.

RequiresNew attribute implies that a method with this attribute requires a new transaction context. If the method is called by a component which is already in a transaction context, then the container suspends the current transaction context and creates a new transaction context. The method is executed in the new transaction context and the transaction is committed before exiting the method. The suspended transaction is resumed after the execution of the method is complete.

Mandatory attribute of a bean method implies that the method must be invoked from a transaction context only. If the component invoking the methods does not begin a transaction, the container will not create a transaction but it will throw a TransactionRequiredException and the method will not be invoked.

NotSupported attribute of an enterprise bean method implies that the method cannot execute in a transaction context. If the method is invoked from a transaction context then the transaction is suspended by the container and the method is invoked without being associated with a transaction. After the execution of the method is complete then the calling transaction is resumed.

If the method is not invoked from a transaction context then the container will not initiate a transaction while the method is executing.

Supports attribute of an enterprise bean method implies that if the method is invoked from a transactional context then the method will execute within that context. If the method is not invoked from a transactional context then the method will execute without invoking a transactional context. This attribute uses the transaction context if it exists otherwise it will execute without any exception.

Never attribute of a method bean implies that it should never be invoked from a transactional context. If the method is invoked from a transaction context, it will throw a RemoteException. The container will abort the method execution if it is invoked within a transaction context.

→ Rolling Back a Container Managed Transaction

The transaction is rolled back by the container whenever an exception is thrown. In case of user defined exceptions, the setRollBackOnly method of EJBContext has to be invoked.

Code Snippet 1 shows the structure of a Container Managed transaction.

Code Snippet 1:

```
@Stateless  
@TransactionManagement(TransactionManagementType.CONTAINER)  
public class StudentAdmissions {
```

```
@TransactionAttribute(TransactionAttributeType.REQUIRED) public void
addStudent(Connection c)

.... .

}
```

In Code Snippet 1, the transaction is being defined to be container managed transaction through the annotation. `addStudent()` method is declared with a transaction attribute type REQUIRED, which implies that a transaction context is required for the execution of the method. If there is an existing transaction then the method is executed in that context otherwise a new transaction context is created.

12.3 Bean Managed Transactions

Container managed transactions 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 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 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.

Code Snippet 2 shows the skeletal structure of a method that implements a bean managed transaction.

Code Snippet 2:

```
import javax.transaction.UserTransaction;

@TransactionManagement(TransactionManagementType.BEAN)

class UserTrans{

.... .
```

```
public void transactionMethod() {  
    try {  
        // obtain access to a UserTransaction  
        UserTransaction tx = myEJBContext.getUserTransaction();  
  
        // start a JTA transaction  
        tx.begin();  
  
        // call other objects and resources to be used in the transaction  
        ...  
  
        // complete the transaction  
        tx.commit();  
    }  
    catch (Exception e) {  
        // report any error as a system exception  
        throw new javax.ejb.EJBException(e);  
    }  
}  
...  
}
```

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.
- **Bean Method Returning without Committing**

In case of a stateless session bean, the transaction is committed before returning the method.

In a stateful session bean, using JTA transaction or JDBC the association between the transaction and the method is retained across multiple client calls.

12.4 Transaction Timeouts

Transactions cannot run for indefinite amount of time as they are resource intensive. Therefore, a certain timeout period is set for each transaction. In case of container managed transactions, the timeout period is set through administration console of the server.

Following are the steps to start the administration console in the NetBeans IDE.

1. Click Servers in the Services tab.

If the Services tab is not visible, then select it from the Windows menu as shown in figure 12.1

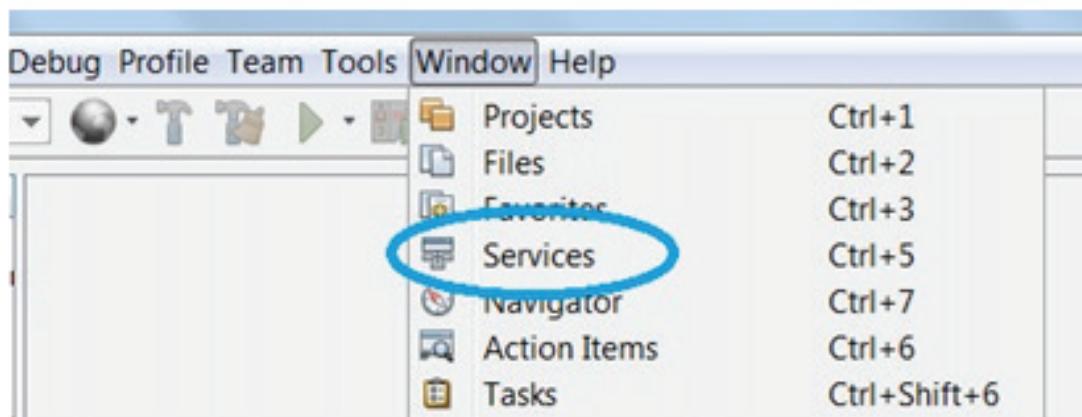


Figure 12.1: Selecting Services Window

2. Expand the Servers option to view and select the currently used server, for instance GlassFish Server.
3. Right-click GlassFish Server to open the context menu.
4. Select View Domain Admin Console for the server to set the transaction timeout as shown in figure 12.2.

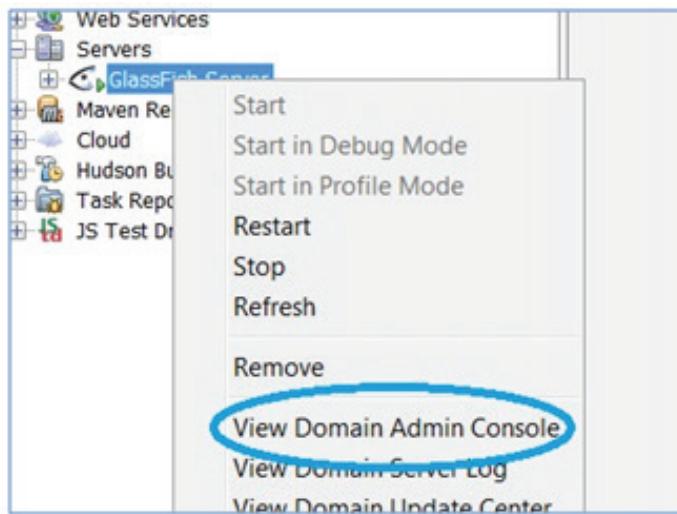


Figure 12.2: Accessing View Domain Admin Console in NetBeans

This opens a GlassFish server configuration page as shown in figure 12.3 through which a common timeout period for all the transactions can be set.



Figure 12.3: Setting Transaction Timeout Period through GlassFish Server Admin Console

Expand the Configurations in the left pane and click Transactions. The Transactions Service screen is displayed as shown in figure 12.4.

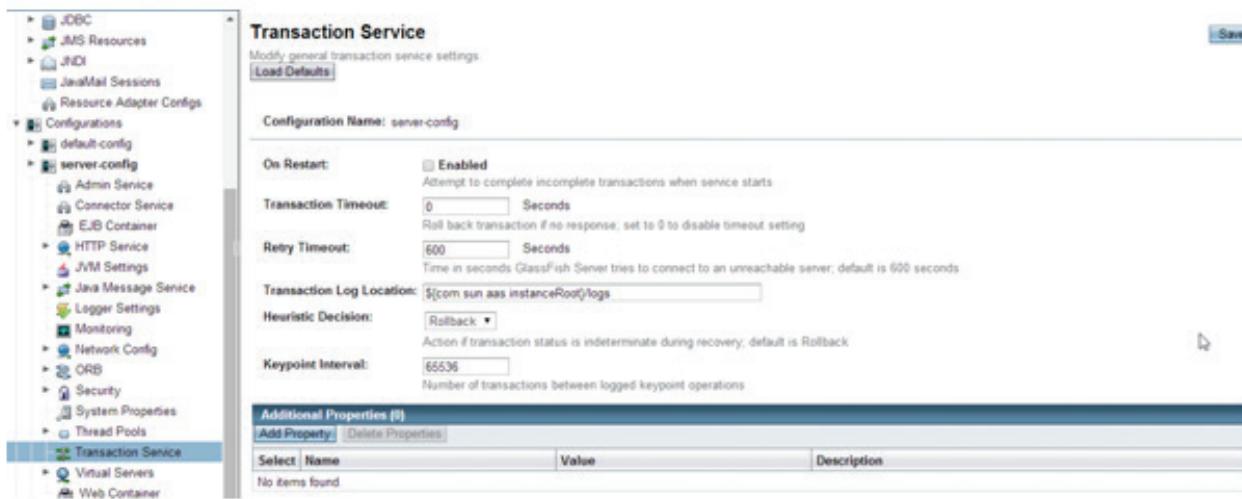


Figure 12.4: Transaction Service

Here, you can set the Transaction Timeout value and click Save. Note that, by default, the Transaction Timeout value is 0 second.

If the transaction timeout period has to be set through enterprise beans, the `setTransactionTimeout` method of the `UserTransaction` interface is used to set the timeout period.

12.5 Updating Multiple Databases

The transaction manager in an application allows for updating more than one database in a transaction through bean managed transactions. Following are two scenarios in which the bean methods will update a database:

1. A bean method makes a database connection with a database and executes transactions on the database. These business methods can in turn initiate another transaction on another database and execute methods.
2. A bean method may initiate multiple transactions on different database servers.

12.6 Transactions in Web Components

Web components such as Java Servlets, provide a handle to the current transaction by default through the `javax.transaction.UserTransaction` interface. These transactions do not span multiple client sessions.

The transactions are initiated in the `service()` method of the servlets and is committed before the method exits.

The Web components can execute in a multi-threaded environment, but the transaction resource objects are not shared by threads.

The Web component may access an EJB context, which in turn may connect to the database resource. The underlying platform is responsible for propagating the transaction from the Web component to EJB context.

12.7 Controlling Concurrent Access to Entity Data

Multiple transactions access data on the database through different applications. There are certain mechanisms provided by the platform to retain the integrity of the database. The transaction manager assumes that the DBMS provides locking mechanisms through read and write locks to ensure data integrity. On the application side, the transaction manager or the persistence provider delays the writes to the database until the transaction is complete.

By default, the persistence providers for the transactions use optimistic locking.

According to the optimistic locking technique, the transaction will not acquire any lock for reading the data and before writing the updated transaction data it will check whether the data has changed since it read it last. This is done through the version number column in the database. If the transaction finds that the data is modified since it read data for the transaction, then it will raise a javax.persistence.OptimisticLockException.

Pessimistic locking is another mechanism where the transaction obtains locks for all the data that might be used by the transaction and holds it until the transaction is complete. This will reduce the degree of concurrency as the data objects are locked for a longer duration of time. However, this technique is efficient if the underlying data is frequently accessed by different transactions.

The lock mode of an entity operation may be set to one of the lock mode types listed in table 12.1.

Lock Mode	Description
OPTIMISTIC	This lock mode obtains optimistic locks for all the data entries used in the transaction.
OPTIMISTIC_FORCE_INCREMENT	Apart from obtaining an optimistic read lock, this lock mode also increments the version attribute of the data entity.
PESSIMISTIC_READ	Pessimistic read lock mode does not allow any other transactions to modify or delete the data. This lock obtained is retained till the transaction which acquired it is complete.
PESSIMISTIC_WRITE	When a pessimistic write lock is obtained by a transaction the data is not read, written, or updated by any other transaction. This lock is released only when the transaction is complete.
PESSIMISTIC_FORCE_INCREMENT	This lock mode obtains a long term lock on the data object and increments the version attributes of the data.
READ	A read lock implies an optimistic read lock where other transactions can also read, write, and update the data.
WRITE	A write lock implies an optimistic write lock with an increment to the version attribute. The write lock is retained until the write operation is complete.
NONE	This lock mode implies that there is no lock required on the data object.

Table 12.1: Locking Modes

12.7.1 Setting a Lock Mode

Following are the methods which are used to set lock modes:

- lock() method of EntityManager interface accepts two parameters, the object on which the lock has to be set and the lock mode type along with the enumeration constant. Following is the usage of this method:

```
lock(Student s, LockModeType.READ)
```

- find() method finds an entity in the database and acquires a lock on this data type. It has three parameters – the entity type, the search attribute, and the lock type. Following is the usage of the find() method.

```
Student s = e.find(Student.class, Roll_no, LockModeType.WRITE);
```

where e, is an instance of EntityManager.

- refresh() method is used to add a lock or change the existing lock into a new type. Following is the usage of this method:

```
Student s = em.find(Student.class, Roll_no);
```

```
...
```

```
em.refresh(s, LockModeType.OPTIMISTIC_FORCE_INCREMENT);
```

- setLockMode method along with a query object. Following is usage of this method.

```
Query q = createQuery(...);
```

```
q.setLockMode(LockModeType.OPTIMISTIC_FORCE_INCREMENT);
```

→ Using Pessimistic Locking

When pessimistic locking technique is used, the locks are retained by the transaction for a longer duration. Pessimistic locking is used when data integrity is critical for the application and in situations where the application is resource intensive. When a transaction cannot obtain a pessimistic lock, then the transaction throws a PessimisticLockException. If a pessimistic lock cannot be obtained and this in turn does not trigger a transaction roll back, then it throws a LockTimeOutException.

The pessimistic locks cannot be retained by the transactions for indefinite amount of time as they block other applications. The timeout period can be defined through javax.persistence.lock.timeout property. If the lock is not released within this time duration then a LockTimeoutException is thrown.

12.8 Demonstrating Container Managed Transactions

Container manages transactions by defining the transaction attributes in the deployment descriptor.

Create a Web application named TransactionApplication.

Create a package named CMT and add a Stateless Session bean named StudentAdmissionsBean.

Add the code depicted in Code Snippet 3 that supports Container Managed Transaction.

Code Snippet 3:

```
package data;

import com.sun.rowset.CachedRowSetImpl;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import javax.sql.rowset.CachedRowSet;

@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)

public class StudentAdmissionsBean {
    @TransactionAttribute(TransactionAttributeType.MANDATORY)
    public String addStudent(Connection c) {
        String message;
        try{
            c.createStatement();
            CachedRowSet st = new CachedRowSetImpl();
            String query="insert into Students values('S007','Martha')";
            st.setCommand(query);
            st.execute(c);
            message="Row Inserted";
        }catch(SQLException e){
            message=e.toString();
        }
    }
}
```

```

        return message;
    }

@TransactionAttribute(TransactionAttributeType.NEVER)
public CachedRowSet display(Connection c) throws SQLException{
    Statement stmt = c.createStatement();
    CachedRowSet st = new CachedRowSetImpl();
    String query = "select * from Students";
    st.setCommand(query);
    st.execute(c);
    return st;
}
}

```

In the given code, the Session bean CMTBean has the TransactionManagementType set to Container. Two methods have been defined in the class StudentAdmissionsBean namely, addStudent() and display(). The addStudent method is used to add a new student to the Students table and the display method is used to display the student details of the students. It returns a CachedRowSet object.

Note - The Students table has been created in the sample database of JavaDB database server that comes bundled with the GlassFish installation. The Students table consists of the RollNo and Name columns with records of six students.

The TransactionAttributeType for addStudent method has been set to MANDATORY. This means that the method must be invoked within a transaction context by the caller. If it is executed without a transaction context, the container will throw the TransactionRequiredException.

Similarly, the display method has the TransactionAttributeType set to NEVER. This means that display method should not be called from a transaction context. If it is called from a transaction context, the container will throw a RemoteException and abort the method execution.

→ Creating the Bean Client

To create a client, create a new servlet named CMTServlet in the CMT package. Add reference to the StudentAdmissionsBean. Right-click in the editor and select Insert Code → Call Enterprise Bean → StudentAdmissionsBean from the dialog box that is displayed.

Add the code shown in Code Snippet 4 in the servlet.

Code Snippet 4:

```
public class CMTServlet extends HttpServlet {  
    @EJB  
    private StudentAdmissionsBean studentAdmissionsBean;  
  
    protected void processRequest (HttpServletRequest request,  
        HttpServletResponse response) throws ServletException, IOException {  
        response.setContentType ("text/html;charset=UTF-8");  
  
        PrintWriter out = response.getWriter();  
        try{  
            out.println ("<!DOCTYPE html>");  
            out.println ("<html>");  
            out.println ("<head>");  
            out.println ("<title>Servlet CMTServlet</title>");  
            out.println ("</head>");  
            out.println ("<body>");  
            out.println ("<h1>STUDENT 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);  
            out.println ("Connection Done");  
            out.println (studentAdmissionsBean.addStudent(conn));  
  
            out.println ("</body>");  
            out.println ("</html>");  
        }  
    }
```

```

        catch (EJBTransactionRequiredException ex) {
            out.println(ex + "- Error: Transaction Mandatory: Method must execute
within a transaction.");
        }

        catch (EJBException ex) {
            out.println(ex + "- Error: Method cannot execute in a transaction.");
        }

        catch (Exception ex) {
            out.println("Error: Other exception - " + ex);
        }
    }
...
}

```

In the given code, a database connection has been created with the sample database. Next, the addStudent method of the StudentAdmissionsBean has been invoked without using a transaction context. The required catch blocks have been added to catch different types of exceptions.

Note - You can also create a single multi-catch block instead of writing multiple catch blocks. Here, multiple catch blocks have been purposely used to display different statements for different types of exceptions.

Deploy the application and run the servlet. Right-click the CMTServlet and click Run File.

The output is shown in figure 12.5.



Figure 12.5: Transaction Attribute MANDATORY without Transaction Context

Note that the container issues the EJBTransactionRequiredException. This is because, the addStudent method had TransactionAttributeType set to MANDATORY. Since the method was invoked without a transaction context, the container raised this exception.

Now, add the following @Resource annotation in the servlet:

```
@Resource  
javax.transaction.UserTransaction utx;
```

This annotation is used to inject a resource into the file. Here, the UserTransaction reference has been added to create a transaction context.

Now, add the following statements before and after the print statement, out.println(studentAdmissionsBean.addStudent(conn)); as follows:

```
utx.begin();  
out.println(studentAdmissionsBean.addStudent(conn));  
utx.commit();
```

The begin and commit methods are used to define a transaction context within which the addStudent method of the StudentAdmissionsBean will be invoked.

Save the changes and run the servlet.

The output is shown in figure 12.6.

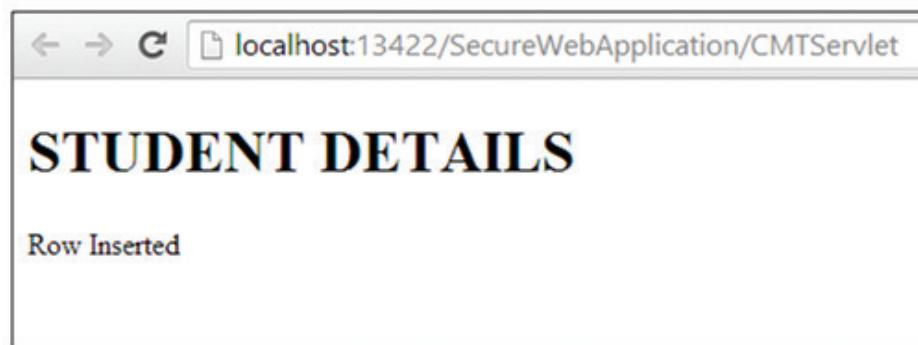


Figure 12.6: Transaction Attribute MANDATORY within Transaction Context

Note that the addStudent method executed successfully. This is because the container found a transaction context within which the method was invoked.

The record inserted in the Students table is shown in figure 12.7.

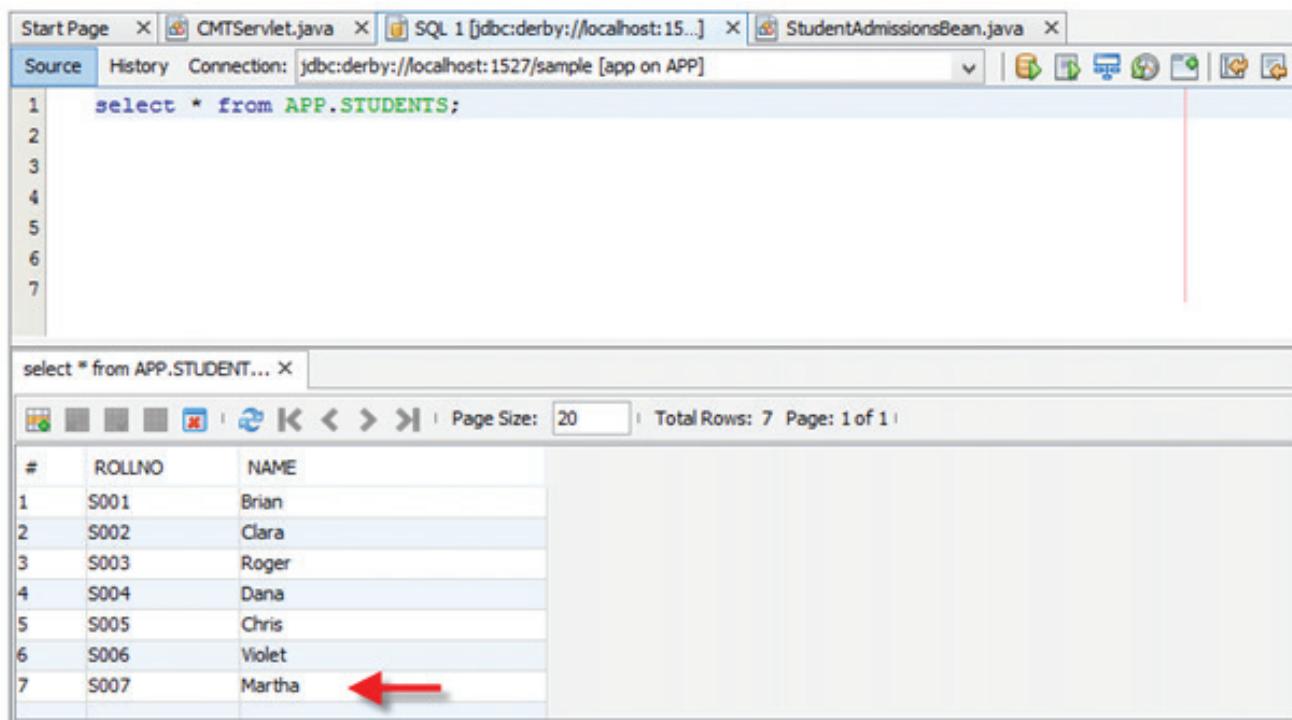


Figure 12.7: Record Inserted in the Students Table

Now, remove the addStudent method and invoke the display method within the transaction context as follows:

```

utx.begin();
CachedRowSet st =studentAdmissionsBean.display(conn);
while (st.next()) {
    out.println("Roll No: " + st.getString(1)+ ", Name: " + st.getString(2));
}
utx.commit();

```

Save the changes and run the servlet. The output is shown in figure 12.8.

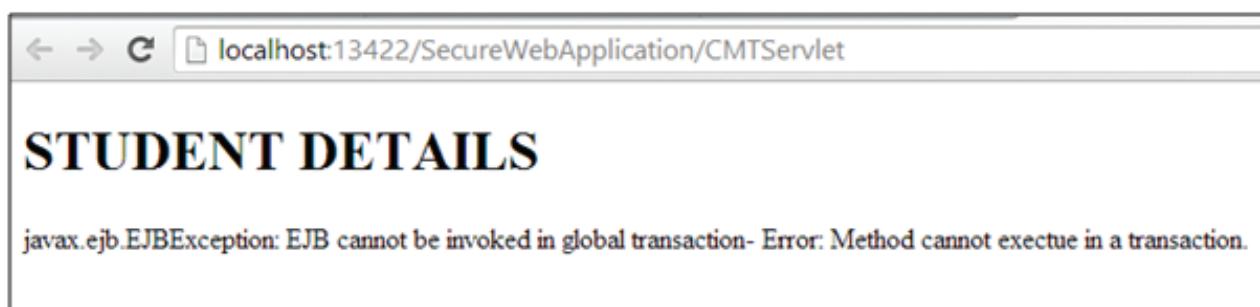


Figure 12.8: Transaction Attribute NEVER within Transaction Context

Note that the container throws the EJBException saying that the method cannot be invoked in a global transaction. This is because the display method has the TransactionAttributeType set to NEVER which does not allow a method to execute within a transaction context. Since, the display method was invoked within a transaction context, the container raised an exception and aborted the request.

Now, remove/comment the utx.begin() and utx.commit() statements from the code to execute the display method without a transaction context.

Save the changes and run the servlet.

The output is shown in figure 12.9.



STUDENT DETAILS

Roll No: S001, Name: Brian
 Roll No: S002, Name: Clara
 Roll No: S003, Name: Roger
 Roll No: S004, Name: Dana
 Roll No: S005, Name: Chris
 Roll No: S006, Name: Violet
 Roll No: S007, Name: Martha

Figure 12.9: Transaction Attribute NEVER without Transaction Context

Note that now the method display has executed successfully because the transaction context has been removed and so the container does not throw any exception. Also, the record inserted earlier of student S007 is visible in the list.

Similarly, other transaction attributes can be used with bean methods in a Container Managed Transaction scenario.

12.9 Demonstrating Bean Managed Transactions

To understand Bean Managed Transaction, create a new package BMT and add a Stateless Session bean named StudentBean in the package.

Code Snippet 5 shows the modified code of the StudentBean.

Code Snippet 5:

```

@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class StudentBean {

    // Add business logic below. (Right-click in editor and choose
    // "Insert Code > Add Business Method")
    public String addStudent(Connection c) {
        String message;
        try{
            c.createStatement();
            CachedRowSet st = new CachedRowSetImpl();
            String query="insert into Students values ('S008','Alice')";
            st.setCommand(query);
            st.execute(c);
            message="Row Inserted";
        } catch(SQLException e){
            message=e.toString();
        }
        return message;
    }
}

```

In Code Snippet 5, the same method, addStudent, has been created in the bean. However, the TransactionManagementType has been set to BEAN and therefore, the transaction attributes have also been removed.

Note - The transaction type can also be declared in the deployment descriptor. The declaration in the deployment descriptor overrides the annotations in the class file.

Create a new servlet in the BMT folder named BMTServlet and modify the code as depicted in Code Snippet 6.

Code Snippet 6:

```
public class BMTServlet extends HttpServlet {  
    @EJB  
    private StudentBean studentBean;  
    @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 CMTServlet</title>");  
            out.println("</head>");  
            out.println("<body>");  
            out.println("<h1>STUDENT 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();  
        } catch (Exception e) {  
            e.printStackTrace();  
        } finally {  
            if (out != null) {  
                out.close();  
            }  
        }  
    }  
}
```

```

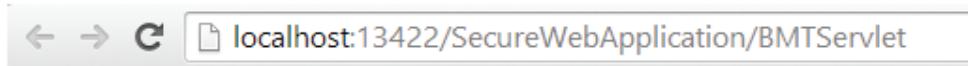
out.println(studentBean.addStudent(conn));
utx.commit();
out.println("</body>");
out.println("</html>");
}
catch(Exception ex) {
    out.println("Error: Other exception - " +ex);
}
...
}

```

The @Resource annotation has been used to inject a reference of UserTransaction.

The connection to the sample database has been obtained. The transaction has been initiated through begin() method. Apart from the begin method, the User Transaction interface provides other methods to commit and rollback the transaction. The addStudent method of the StudentBean has been invoked followed by the commit method.

Save the changes and run the BMTServlet file. The output is shown in figure 12.10.



STUDENT DETAILS

Row Inserted

Figure 12.10: Output of Bean Managed Transaction

The new row inserted in the Students table is shown in figure 12.11.

#	ROLLNO	NAME
1	S001	Brian
2	S002	Clara
3	S003	Roger
4	S004	Dana
5	S005	Chris
6	S006	Violet
7	S007	Martha
8	S008	Alice

Figure 12.11: Row Added to the Students Table

12.10 Check Your Progress

1. Which of the following statement about transactions is true?

(A)	Containers do not allow a bean method to initiate multiple transactions.	(C)	Container managed transactions are programmatic transactions
(B)	Bean managed transactions are declarative transactions.	(D)	None of these

2. Which of the following attributes should be declared in the deployment descriptor if a transaction context is essential for the bean method?

(A)	Required	(C)	RequiresNew
(B)	Mandatory	(D)	Never

3. Which of the following actions are taken by the transaction manager when a bean method fails after the transaction in the method commits?

(A)	The transaction remains committed	(C)	The transaction is rolled back
(B)	Exception handler has to be defined for this situation	(D)	None of these

4. Which of the following statements about optimistic locking mechanism are true?

(A)	Allows multiple transactions to read data objects simultaneously.	(C)	Allows multiple transactions to update data objects simultaneously.
(B)	Allows multiple transactions to write data objects simultaneously.	(D)	None of these

5. Which of the following operations is not a cascading operation on the entities?

	Method		Function
a.	setRollBackOnly	1.	Propagates the transaction state to the database
b.	rollback	2.	Returns the status of the transaction
c.	getStatus	3.	Reverses the transaction changes
d.	commit	4.	Ensures that the transaction is finally rolled back

(A)	a-3, b-4, c-2, d-1	(C)	a-2, b-4, c-1, d-3
(B)	a-4, b-3, c-2, d-1	(D)	a-4, b-3, c-1, d-2

12.10.1 Answers

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

Summary

- Transactions in Java EE applications are initiated by the bean methods. They can be container managed or bean managed.
- The context of container managed transactions is defined by the container. The developer cannot use transaction management methods in the bean.
- In bean managed transactions, the developer can explicitly invoke the transaction management methods.
- Transactions are associated with a timeout period. The transactions have to release the resources before the timeout period expires.
- Transactions can access multiple databases through bean methods.
- Optimistic and pessimistic locking techniques are used to ensure data integrity in the database.



Login to www.onlinevarsity.com

Session - 13

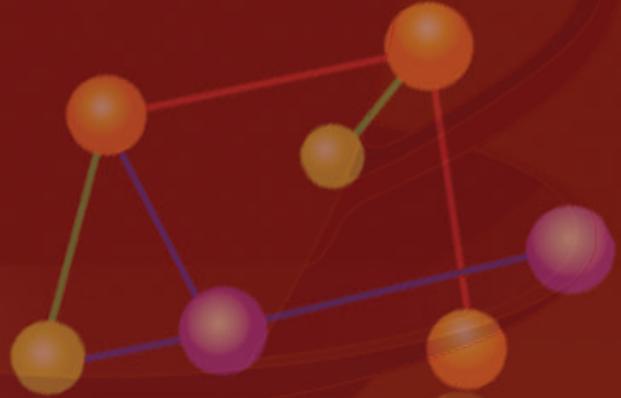
Java Message Service Components

Welcome to the Session, **Java Message Service Components**.

This session describes Java Messaging Service (JMS) API, its utility in an application, the architecture, and JMS programming model. It also describes models of message communication and the usage of JMS API in Java EE applications.

In this Session, you will learn to:

- Explain JMS API and its utility in an application
- Describe models of message communication
- Explain JMS API architecture
- Explain the JMS programming model
- Explain configuring JMS messages with options
- Use JMS API in Java EE applications



13.1 Overview of JMS

Java application architecture allows its components to be distributed geographically, where each component may be located on a different machine. Messaging enables integrating these components in a loosely coupled manner. It provides for communication between various components of the application. The communication is loosely coupled because the sender and receiver need not be available at the same time for communication.

The messaging infrastructure is also known as message oriented middleware, which isolates the developer from the implementation of the messaging system. The developer can use the predefined objects and methods provided by the messaging system to create, send, and receive messages for the component.

Java EE provides the Java Message Service (JMS) API to perform the messaging tasks for Java applications. The JMS API provides certain interfaces that enable programmers to develop the messaging system in applications. It provides portability and loosely coupled, asynchronous, and reliable communication mechanism.

In loosely coupled communication, there is an intermediary messaging agent. The sender sends the message to the agent and the receiver retrieves the message from this agent when it is available. A tightly coupled communication mechanism is one where an entity sends a message and waits for a response.

Asynchronous communication implies that the sender sends a message and does not wait for the response from the receiver. There is no simultaneous two way communication.

Reliability of the JMS API message implies that the message is received by a receiver only once ensuring that there are no duplicate messages.

→ JMS API and Java EE

JMS API can be used as an independent messaging system for an enterprise and can also be used as part of a Java EE application. JMS has the following features when used as part of a Java EE application:

- Application clients, Enterprise Java beans, and Web components of the application can send and receive asynchronous messages.
- JMS also allows the application clients to set a message listener which gives a notification to the client when a message is received from other components.
- The EJB container has message-driven beans which are responsible for receiving asynchronous messages. Multiple message-driven beans can be pooled together for concurrent processing of the messages received.
- JMS messages can be sent and received as part of Java transaction.
- A JMS provider can be integrated using Java EE connector architecture. A JMS provider is a messaging system which implements the interfaces provided by the JMS API to define a messaging system. The JMS provider can be accessed through a resource adapter on different platforms.

13.2 JMS API Architecture

A JMS API can be used to develop an independent application or a messaging module of an enterprise application. An independent JMS application would comprise the following components:

- ➔ JMS provider
- ➔ JMS clients
- ➔ Messages
- ➔ Administered objects

JMS provider implements the messaging functionality through the JMS API interface and also provides the administrative and control features.

JMS clients are the components which produce and consume messages.

Messages are the objects that are exchanged among JMS clients.

Administered objects are configuration objects for JMS clients. There are two types of configuration objects for JMS – destinations and connection factories. Destination object is used by a JMS client to specify the destination of the messages it generates and a connection factory object is used to create a connection with the JMS provider. The administered objects are created by the administrator and placed in the JNDI namespace. These objects are later used by the client through their JNDI names.

Figure 13.1 shows a diagrammatic representation of JMS Architecture.

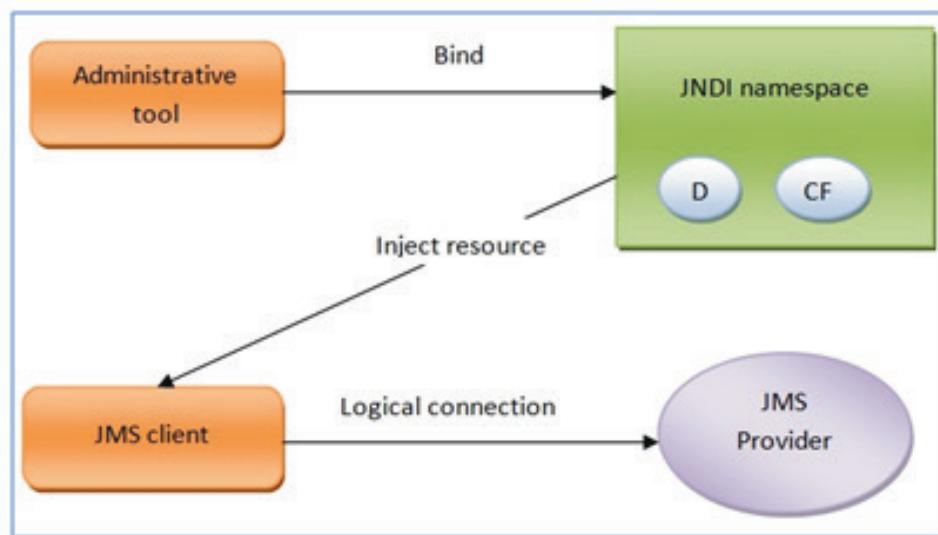


Figure 13.1 JMS Architecture

A JMS client injects resources into the application through administrative tools. The resources are bound with the help of JNDI namespace. The client then communicates with these resources by establishing a logical connection through the JMS provider.

13.3 Messaging Models in JMS

The messaging models of JMS define the mode of communication between two JMS entities. There are two models of messaging used by the JMS providers. Every JMS provider should implement either of the messaging models:

- ➔ Point-to-point messaging model
- ➔ Publish-subscribe messaging model
- ➔ **Point to point messaging model** – All JMS entities have message queues associated with them. Every message sent is destined to a message queue at the destination. The messages received by a JMS client are received from a message queue. Every message in this model has only one sender and one receiver. The receiver JMS client need not be available to receive the message as soon as it is sent. The message is placed in the receiving queue of the destination JMS client.

Figure 13.2 demonstrates the point-to-point messaging model.

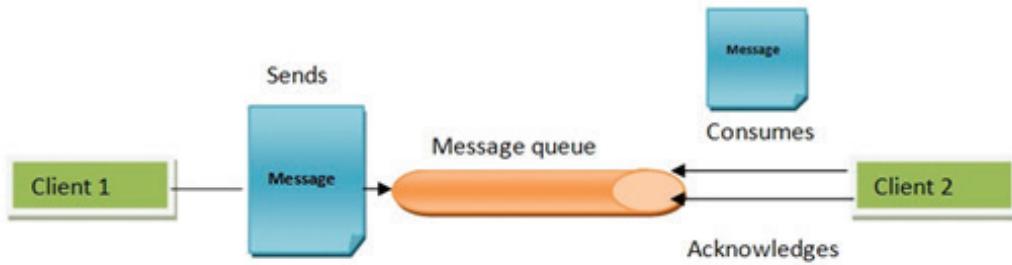


Figure 13.2 Point-to-Point Model

- ➔ **Publish-subscribe model** – This is a broad cast communication model. That is, a message in this model has one sender and one or more receivers. All JMS clients are associated with a topic. The topic object is similar to a bulletin board which can house messages from JMS clients. A JMS client which intends to send a message sends a message to the topic. The JMS client which intends to receive messages from a topic has to subscribe to the topic and then consume messages. When there are multiple subscribers to a JMS topic, then the JMS system is responsible for distributing the messages among the subscribers of the topic. A topic retains the message received until the messages are distributed to all the subscribers of the topic.

The consumer of the messages from the topic and subscription of the topic are two different JMS objects. The consumer object accepts the message and uses/processes the message received from the client. The subscription object is an entity of JMS provider. Once a subscription object is created for a topic and client, the client receives the messages from the topic every time a message is posted onto the topic.

Durable subscriptions of JMS provider enable the client to receive messages sent by the message producer while the receiver client is not available. Durable transactions provide the required flexibility and reliability.

Figure 13.3 demonstrates publish-subscribe messaging model.

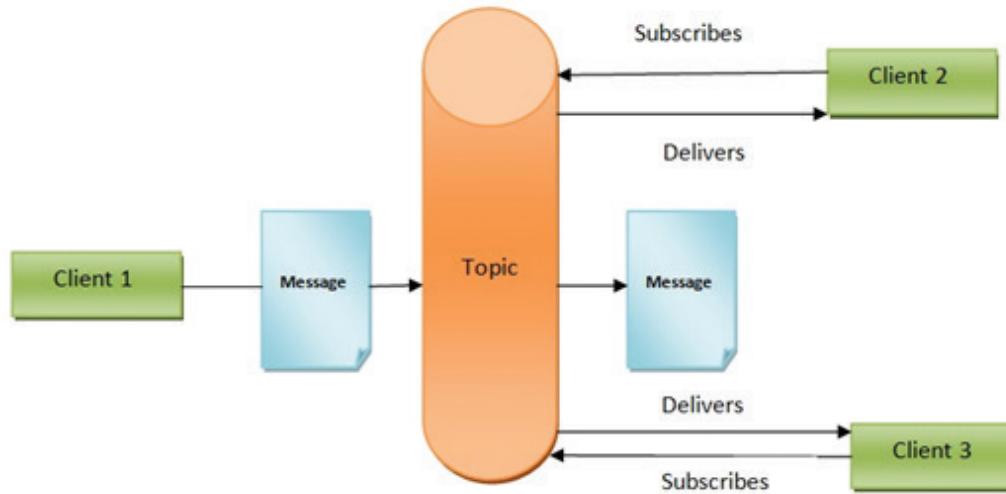


Figure 13.3 Publish-Subscribe Model

Message consumption in JMS application is asynchronous by default. However, a JMS client can establish a synchronous communication session by invoking the `receive()` method for another JMS client from which it intends to receive messages.

For asynchronous communication, JMS client can register a message listener with the consumer. When a message is received, the JMS provider delivers the message to the consumer by invoking the `onMessage()` method of the registered listener. In case of Java EE applications, the message-driven bean serves as message listener.

13.4 JMS API Programming Model

A JMS application comprises the following components:

- ➔ Administered objects
- ➔ Connections
- ➔ Sessions
- ➔ JMSContext objects
- ➔ Message producers
- ➔ Message consumers
- ➔ Messages

13.4.1 JMS Administered Objects

These are objects which are created by the administrator and are configured in the JNDI namespace of the application. The JMS clients can access these administered objects through resource injection. There are two types of administered objects in JMS:

1. Connection factories
2. Destinations

These objects are created through Administration Console or through asadmin create-jms-resource command in the form of connector resources. The resources can also be specified in the file named GlassFish-resources.xml. A wizard in NetBeans IDE can be used to create resources on GlassFish Server by clicking File → New File → GlassFish → JMS Resource. Figure 13.4 shows the wizard which creates a JMS resource for the Java EE application.

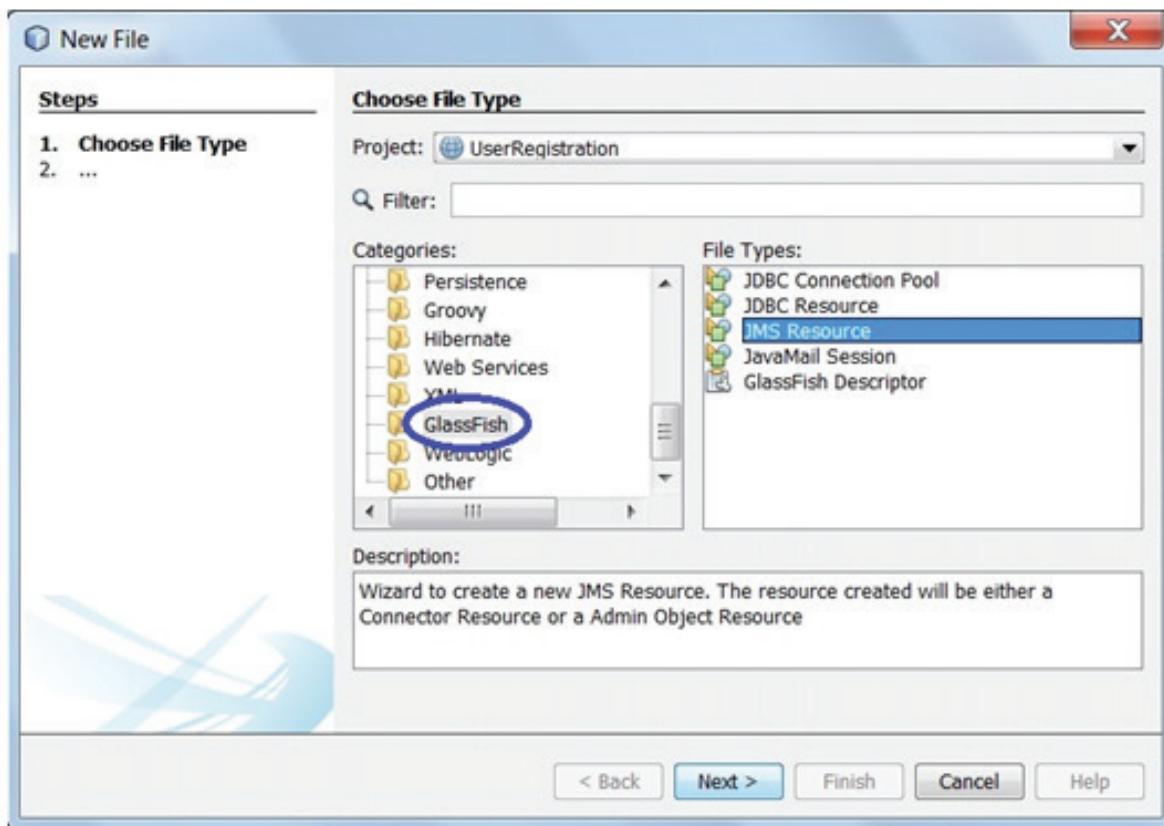


Figure 13.4: Creating a JMS Administered Object

An administered object can be created by adding a new JMS resource. This can be done by adding a new component to the Java EE project.

Click Next. The screen to specify JMS Resource attributes is displayed in figure 13.5.

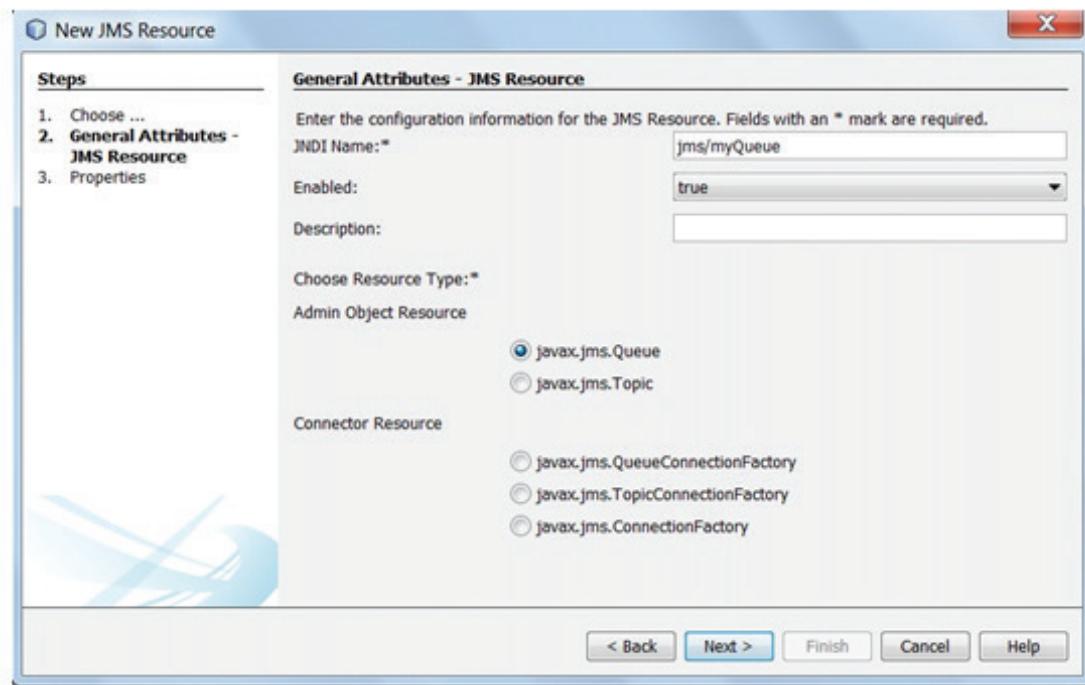


Figure 13.5: Selecting the JMS Resource Type

The resource type can be selected from the options given in the screen. For example, the JNDI Name provided here is jms/myQueue and the Admin Object Resource selected is javax.jms.Queue. Similarly, a Connector resource can be created. These will be used to create a Point-to-Point messaging model.

The connection factory objects and destination objects are the administered objects. A connection factory object is used to create a connection to a provider. The configuration parameters of this object are defined by the administrator.

A destination object is also created as an administered object. These objects are targets of the message objects created by a JMS client or the source of the JMS messages which the client will consume. They are also created through the wizard as shown in figure 13.5.

The Admin object resources in the figure imply the destination object which can be a queue or a topic.

13.4.2 Connections

A connection object in the context of JMS implies a virtual connection with the JMS provider. In case of JMS, a connection is created by creating an object of JMSContext.

13.4.3 Sessions

Sessions comprise message exchange among the message producers and consumers in certain application context. A session is created by creating a JMSContext object.

Sessions serialize the execution of message listeners, sequencing them according to the order in which messages are sent and received.

13.4.4 JMSContext Objects

JMSContext is a state of an application which associates a connection object with a session. A JMSContext can be used to create the following entities of a JMS system:

- ➔ Message producers
- ➔ Message consumers
- ➔ Messages
- ➔ Queue browsers
- ➔ Topics

A JMSContext object is created by calling the createContext() method on a ConnectionFactory object as follows:

```
JMSContext jms = ConnectionFactory.createContext();
```

A JMSContext has to be explicitly closed if it is not included in a try with resources code block.

- ➔ Message producers are objects that generate messages in a certain context. Every message producer is associated with a JMSContext object. Since the message producer objects are lightweight objects, an object is created each time the message producer has to generate a message to be sent. Once there is a JMSContext object, say jms, a message producer object can be created as follows:

```
JMSProducer jpro = jms.createProducer();
```

- ➔ Message consumers are objects that accept messages from JMS clients. A message consumer, like a message producer is also associated with a JMSContext object. Given a JMSContext object jms, a message consumer can be created through a createConsumer() method as follows:

```
JMSConsumer jcon = jms.createConsumer();
```

A message consumer can register with a JMS destination such as queue or topic. The JMS provider is responsible for sending the messages from the message destination to the corresponding message consumer.

The receive() method is used to establish a synchronous communication among various entities in the application. The receive() method is an overloaded method which can be invoked with or without parameters. When the receive method is invoked without parameters, then the object invokes the method until a message from the sender is received. The receive() method also has an argument which is an integer parameter representing the timeout period after which the request has to exit.

Asynchronous communication is implemented through JMS message listeners by implementing the MessageListener interface. The onMessage() method of the listener is invoked when a message is received. A message listener is registered with a JMS consumer through the setMessageListener method as follows:

```
Listener L = new Listener();
consumer.setMessageListener(Listener);
```

where consumer is a message consumer object.

When a message-driven bean is used for asynchronous communication the message-driven bean also implements the MessageListener interface. The bean defines the onMessage() method to handle the message.

→ Consuming Messages from Topics

Asynchronous communication and synchronous communication is applicable to point-to-point messaging model where messages are consumed by accessing the queues. In case of publish-subscribe model, the messages have to be consumed from topics. Messages are consumed by an application by creating a subscription on the topic and by creating a consumer for the subscription.

The consumer object is part of the JMS client and subscription object is part of the JMS provider. The subscription object is responsible for delivering the message to all those JMS client consumers who have subscribed for a topic. Subscription object receives a copy of message in the topic, as it is generated. This subscription may in turn have one or more consumers of the client consuming these messages.

Subscriptions can further be durable or non-durable. Non-durable subscriptions are those which exist only if there is an active consumer associated with it.

A durable subscription will exist even if there is no active consumer for the subscription. There are two variants for both durable and non-durable subscriptions as follows:

- **Shared subscriptions**

A shared subscription is shared by multiple consumer objects. If the subscription is a shared non-durable subscription, then the subscription object is created as soon as the first consumer is created.

- **Unshared subscriptions**

An unshared subscription object may have only a single consumer associated with it. They are identified by a subscription name and an optional client identifier. In case of non-durable subscription the subscription object is deleted when the consumer object is deleted.

Durable subscriptions are associated with a unique identity. All the consumer objects accessing the subscription access this identity to consume messages from the subscription. However, if there is no active consumer for a subscription, the JMS API provider retains the messages till a consumer for the message is available.

→ JMS Messages

JMS allows creating messages which can be sent to and received from non-JMS clients. It has a simple and flexible format. A JMS message comprises the following three parts:

- Message header
- Properties
- Message body

Message header is a set of administrative fields which identify the source and destination of the message and other administrative aspects of the message. Table 13.1 shows the header fields and their description.

Header Field	Description
JMSDestination	Set by the send method of JMSProvider. It identifies the destination to which the current JMS message is sent.
JMSDeliveryMode	Set by the send method of the JMSProvider. Defines whether the message should be delivered in a persistent mode or non-persistent mode.
JMSDeliveryTime	Specifies any time delay associated with the delivery of the message. This implies the tolerable time delay between the message generation and message consumption.
JMSExpiration	Specifies the validity period of the message.
JMSPriority	Defines a priority of the message set by the send method of the JMS provider.
JMSMessageID	An identifier used to identify the message which is set by the send method of the JMS provider.
JMSTimeStamp	Records the time when the message was generated and sent for delivery. It is set by the send method of the JMS provider.
JMSCorrelationID	Set by the client application to link a message to another.
JMSReplyTo	Set by the client application. It specifies the identifier of the destination to which reply has to be sent.
JMSType	Identifies the structure of the message and the type of data in the body of the message. This is also set by the client application.
JMSRedelivered	Specifies whether the message is being resent.

Table 13.1: JMS Message Headers

Message properties – In addition to the message header fields, message properties are also defined for the JMS messages. The purpose of these properties is to provide compatibility with other messaging systems. All the JMS API property names begin with JMSX.

Message body – JMS API supports six types of messages. Following are the message types supported by JMS:

- **TextMessage** – This comprises data as Java strings.
- **MapMessage** – The data has map objects which are key-value pairs. The entities of a map object can be sequentially accessed through enumerator.
- **BytesMessage** – A stream of un-interpreted bytes that match a data format such as JPEG and so on.
- **StreamMessage** – A stream of values which are of primitive types.
- **ObjectMessage** - 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 these message types. At the consumer end, the message is received as a generic Message object, which is later type casted to appropriate message type.

→ JMS Queue Browsers

As JMS supports asynchronous exchange of messages, when a message is received at the destination, it may not be immediately consumed as the message consumer may not be available. In this case, the message is placed in a message queue. The QueueBrowser object can browse through the messages in the queue and interpret the header values of all the messages in the queue. A QueueBrowser object, like the message producer and consumer objects is created with a JMS context.

→ JMS Topic

JMS Topic is associated with the publish-subscribe messaging model. It also stores the messages received from queues as in case of point-to-point communication. These queues from which messages are received into the topic are managed through the queue browsers and JMS providers. The only difference lies in the way the messages from a JMS Topic are consumed.

13.5 Using Advanced JMS Features

In order to improve the reliability of the application, PERSISTENT messages can be used. This is the default mode of the messages.

Messages can be sent and received along with the operations of transaction in an application. The reliable way of consuming messages from a transaction is either through durable transaction or by using a persistent message. Following are some of the advanced features of JMS that are used to improve the reliability and performance of the Java EE application.

13.5.1 Controlling Message Acknowledgement

For every JMS message sent, an acknowledgement is expected. A message is considered to be successfully consumed by the application only when the acknowledgement is received. When the client receives a message, it places the message in the queue, which refers to the buffer for messages at the client end. The client consumes and processes the message. An acknowledgement to the message is sent after the message processing is complete.

When the messages are sent as part of a transaction, then the messages are acknowledged once the transaction is committed.

The method of acknowledgement can be defined in the `createContext()` method by passing any of the following values as arguments to the `createContext()` method. Following are the different values and their respective interpretations:

- **JMSContext.AUTO_ACKNOWLEDGE** – When the context of the messages are set with auto acknowledge value, then the client sends an acknowledgement immediately after the `receive()` method returns in case of synchronous communication and in case of asynchronous communication the acknowledgement is sent after the `MessageListener` returns.
- **JMSContext.CLIENT_ACKNOWLEDGE** – An acknowledgement is sent by the client through an `acknowledge()` method.
- **JMSContext.DUPS_OK_ACKNOWLEDGE** – The delivery of the acknowledgements is not done immediately as the message is consumed. Acknowledgements of multiple messages are aggregated together and then sent. This mode of acknowledgement delivery is used in situations where the message consumers are capable of receiving duplicate messages.

At times, the `JMSContext` may close before delivering the acknowledgements for the messages received. In such cases, the acknowledgements are retained by the JMS provider and resent when required.

13.5.2 Specifying Options for Sending Messages

Following options can be set while sending messages to achieve various performance levels of the application:

- Define the persistence of the messages which will ensure that the messages are not lost in case of a JMS provider failure.
- Define priority levels for the messages which will define the order of delivery of the messages. The messages of higher priority are delivered first.
- Define an expiry time for the messages. If the expiry time of a message is lapsed, then the message is discarded and not delivered to the destination.

- Specify the delivery delay for the messages so that they can be scheduled for delivery at the right time.

13.5.3 Creating Temporary Destinations

JMS allows the programmers to create temporary destinations for the JMS clients, which will last only as long as the connection exists. These are `TemporaryQueue` and `TemporaryTopic` objects which are created through `createTemporaryQueue` and `createTemporaryTopic` methods respectively.

The message consumers of these temporary destinations are those which are created during the connection. These temporary destinations can receive messages from any message producers. These temporary destinations are closed as the connections to which they belong close too and all the messages in these destinations are lost. These queues are used for situations where there are several connection specific messages to be exchanged.

13.5.4 Using JMS Local Transactions

A local transaction can be used to group multiple send and receive messages. The messages sent through transactions are not added to the queue or topic until the transactions are committed and the messages are acknowledged only after the transactions are committed.

When a transaction is rolled back, then all the messages sent and received by the transaction are destroyed.

A local transaction can be used to group multiple messages to be sent and received, if all the messages belong to a single `JMSContext`. Synchronous message exchanges cannot be implemented through local transactions as the messages are added to the destination queue only after the transaction commits. This may result in indefinite blocking of both transactions and JMS clients.

13.6 Using JMS API in Java EE Applications

JMS API allows definition of an independent messaging system with different application clients or can be incorporated as a part of Java EE application. When a JMS API is used as a part of Java EE application, then the EJB container or the Web container should not attempt to create more than one active session per each connection object. In case of JMS application clients, multiple sessions can be associated with each connection object.

→ Creating Resources for Java EE Applications

JMS resources can be added to Java EE applications either through resource injection or through deployment descriptor elements. While specifying the resource in the deployment descriptor, any of the following JNDI namespaces can be used:

- When the resource is declared with `java:global` namespace, then the resource is available across all the applications deployed.
- The namespace `java:app` implies that the resource is available across all the modules of the application.

- The namespace `java:module` makes the resource available to all components within a given module.
- When a resource is declared with namespace `java:component` then the resource is available only to that component.

→ Using Java EE Components to Produce and Synchronously Receive Messages

Though using synchronous messages is not a preferred design of an application, they can be used with definite timeout periods where reliability of messages is required. A Java Web or Java EE component can be used to synchronously produce and receive messages. A programmer should observe the following practices while managing the JMS resources from the Java EE components or Web components:

- When a JMS resource is required for the span of a business method, then the `JMSContext` resource has to be created in a `try-with-resources` block. Creating the resource with `try-with-resources` block ensures that the resource is closed at the end of the `try` block.
- The JMS resource has to be injected if the JMS resource has to be maintained for the duration of the transaction or request.

→ Using Message-Driven Beans to Receive Messages Asynchronously

A message-driven bean is an enterprise bean in a Java application which is capable of processing JMS messages asynchronously. It acts as a message listener for JMS messages. Following are the requirements of a message-driven bean:

- A message-driven bean can be declared in the deployment descriptor or can be injected through resource injection.
- The bean class should be public. It can be inherited by other classes hence, it cannot be abstract or final.
- It should have a public default constructor.

It is recommended that a message-driven bean class implements `javax.jms.MessageListener` interface and defines the `onMessage()` method. The `onMessage()` method is invoked when a message is received. This message holds the business logic for processing the received message.

13.7 Creating a Message-Driven Bean (MDB)

To create a Message-Driven Bean, create an EJB project and right-click the project name. Select `New → Other → Enterprise JavaBeans → Message-Driven Bean` from the `New File` dialog box as shown in figure 13.6.

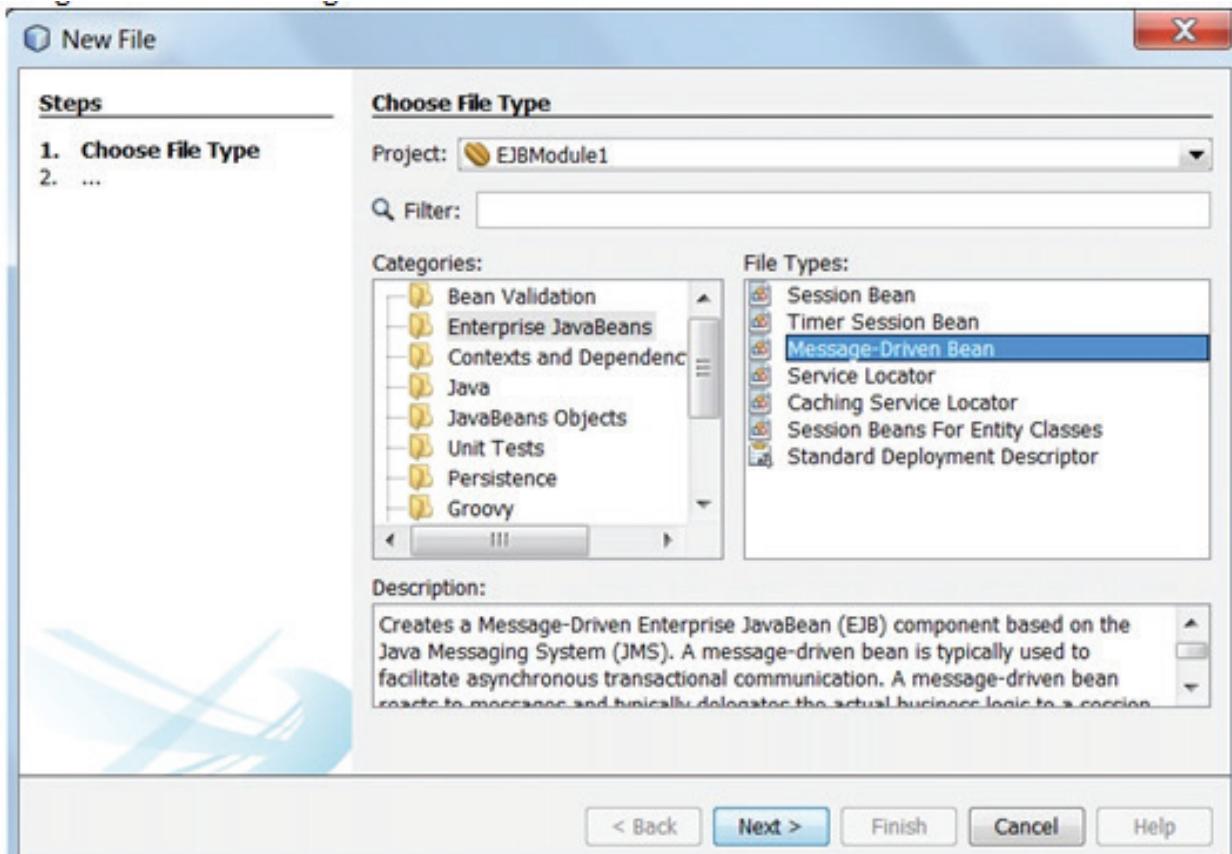


Figure 13.6: Creating a Message-Driven Bean

The New Message-Driven Bean dialog box is displayed which prompts for the EJB Name and Package of the message-driven bean, and destination of messages as shown in figure 13.7.

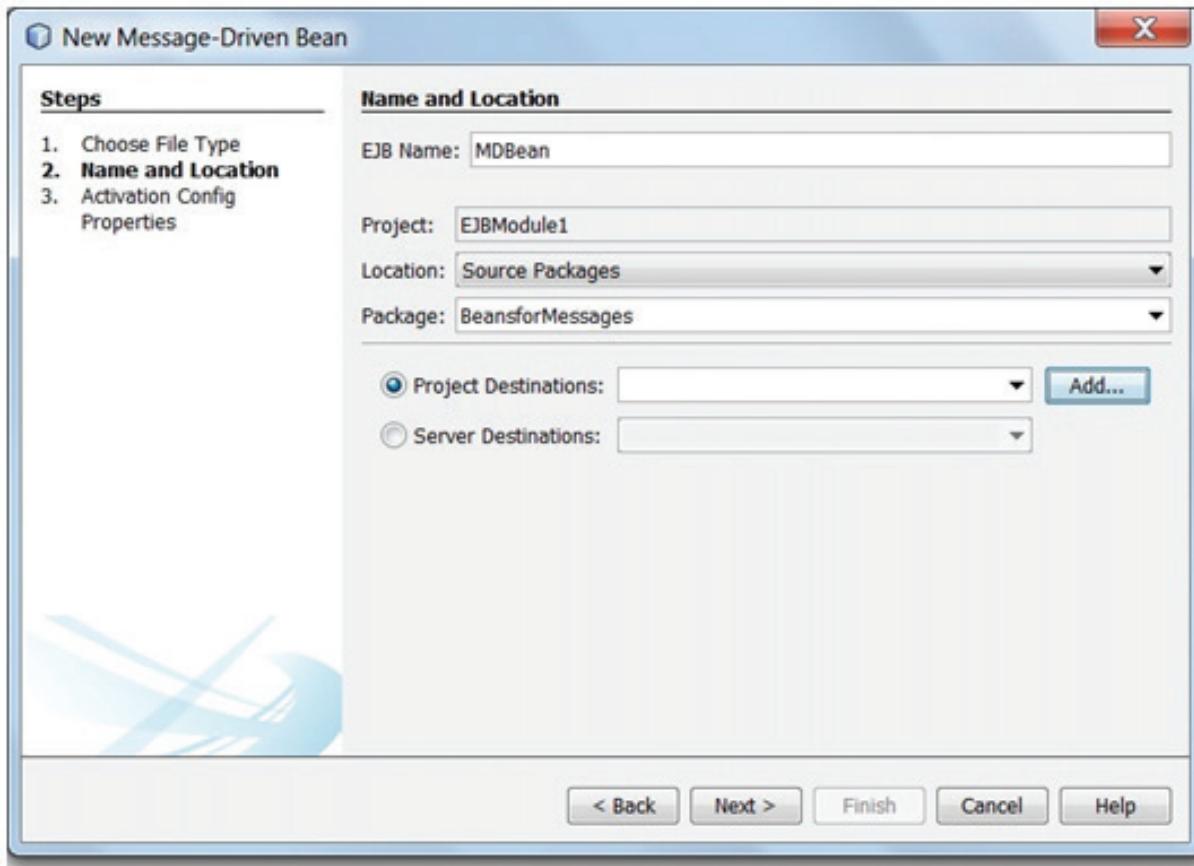


Figure 13.7: Location and Destination of Message-Driven Bean

The destination of the message can be a topic or a queue which has to be selected by clicking Add. The Add Message Destination dialog box is displayed as shown in figure 13.8.

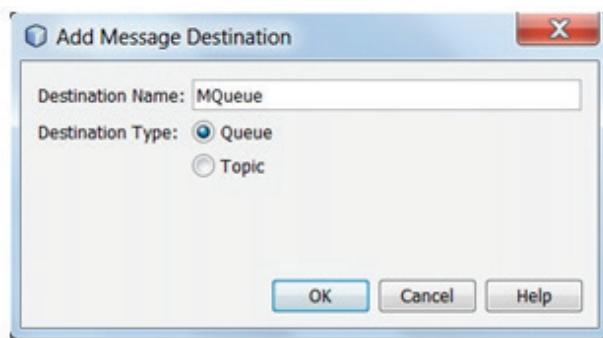


Figure 13.8: Selecting the Destination

Select Queue. Provide the Destination Name as MQueue, and click OK. The name, MQueue will be added to the Project Destinations drop-down.

Click Next. The Activation Config Properties screen is displayed.

The configuration options for the destination are specified in this screen of the wizard as shown in figure 13.9.

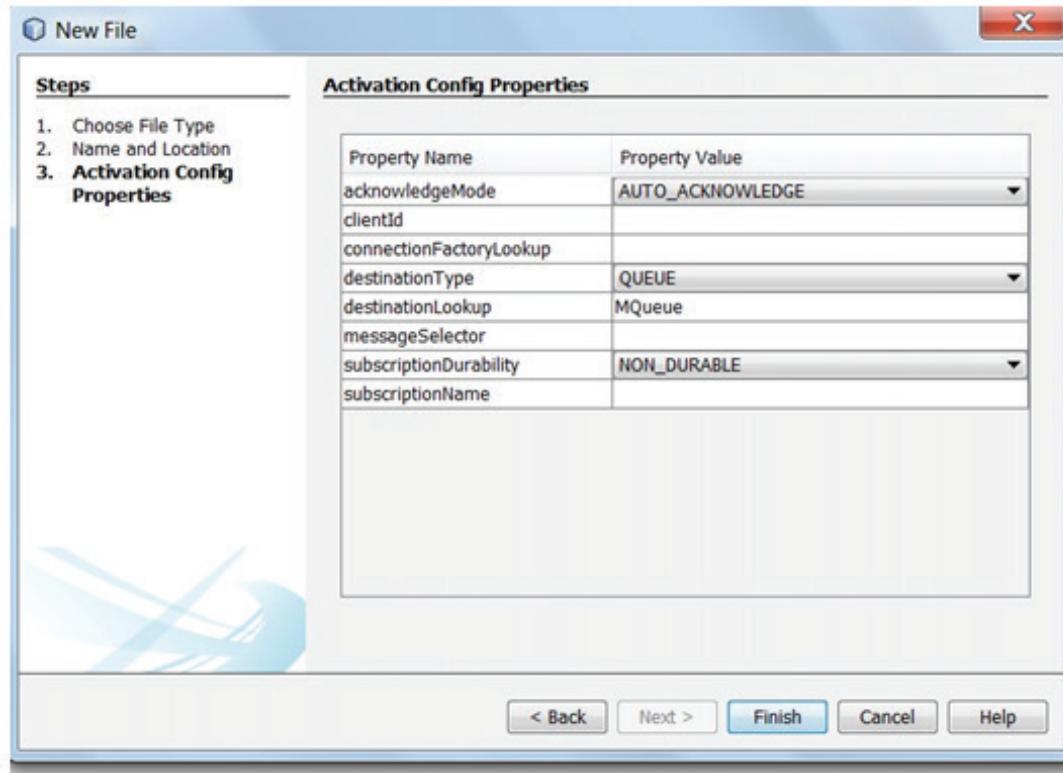


Figure 13.9: Configuring the Message Destination

A message-driven bean is created with the `onMessage()` method where the developer writes the code to be executed when this method is invoked. The messages generated by the Message-Driven Bean are sent to the destination MQueue. The Message-Driven Bean can receive messages from any of the producers in its JMSContext and scope.

The Message producer and receiver can be created using a JSF page and a Message-driven bean. When a message producer sends a message, the message is sent to a message queue if a point-to-point messaging model is used.

13.8 Creating a Simple JMS Application

Create a Web application in Netbeans IDE, which can be used as the message producer. Here, the MessageProducer Web application has been created using the JSF framework as shown in figure 13.10.

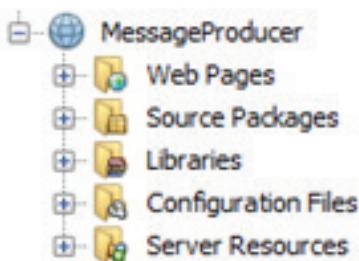


Figure 13.10: MessageProducer Web Application

A typical JMS application will have a Producer entity, which produces the messages. These messages can be stored on a queue and the messages can be accessed by a message receiver from the queue. This pattern is followed if the messaging model is point-to-point messaging model. If the application uses publish-subscribe model, then the messages are written to a topic and subscribed from the topic.

Following is the overview of steps for creating a message producer application:

1. Create a Message Producer Web application.
2. Create JMS admin objects of Queue and ConnectionFactory on the server.
3. Create a JSFManagedBean to send/produce the message to the Queue.
4. Create the Message-Driven bean to accept/consume message.

In order to post a message to the queue on the Web Server, create these objects on the server.

Here, the producer has been mapped with the Queue named jms/myQueue.

→ Creating the Queue and Connector Resources

The message queue is created on the server. In this example, consider the instance of GlassFish server to which the JMS resources are added. Right-click the MessageProducer project and select New → Other → GlassFish → JMS Resource from the New File dialog box.

Figure 13.11 shows adding JMS resource to the application.

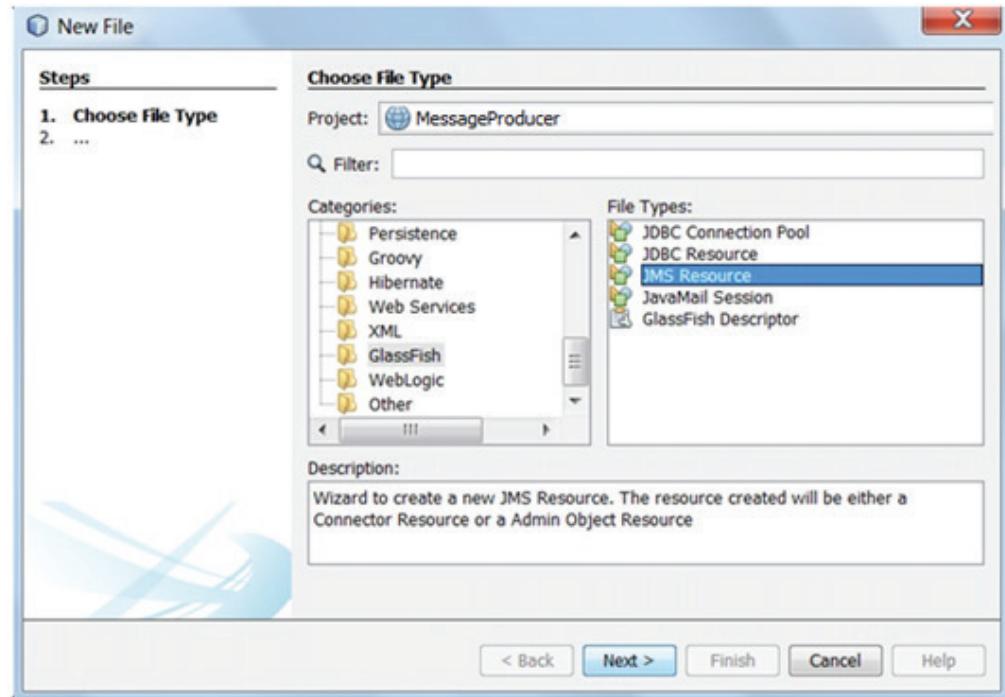


Figure 13.11: Adding JMS Resources to the Application

Click Next. The New JMS Resource dialog box is displayed.

The developer can 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). Figure 13.12 shows the wizard.

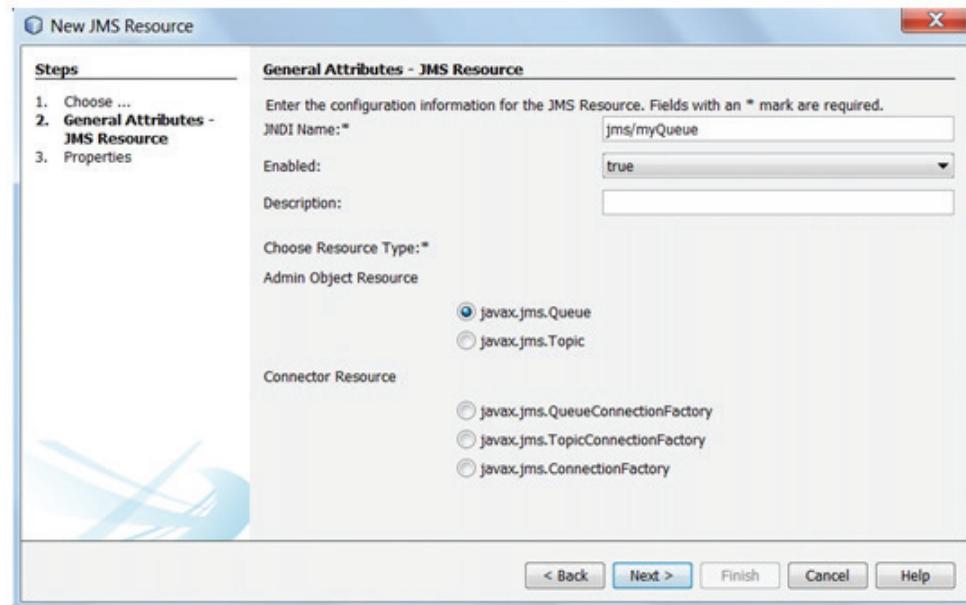


Figure 13.12: Creating a JMSQueue

Click Next. On the next screen, provide myQueue in the Value field of the Name property. Click Finish.

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

Now, create a Connection Factory resource. Right-click the project and select New → Other → GlassFish → JMS Resource from the New File dialog box.

Click Next and specify the options as shown in figure 13.13 to create a QueueConnectionFactory object.

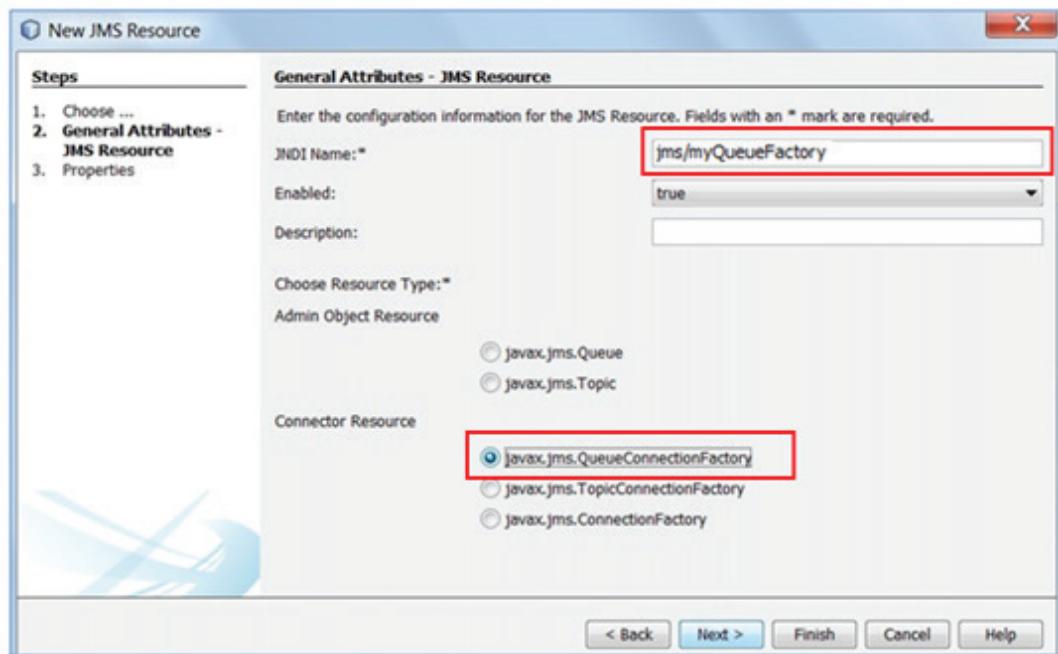


Figure 13.13: Creating a QueueConnectionFactory Object on the Server

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.

Specify the JNDI Name as jms/myQueueFactory and select the Connector Resource as javax.jms.QueueConnectionFactory.

Click Next. The JMS Properties screen is displayed. Here, you can specify additional configuration information.

Click Finish. The information about myQueueFactory is added to the GlassFish-resources.xml file in the Server Resources folder.

Deploy the application. Open the Services tab. Right-click the Applications folder and select Refresh to see that the MessageProducer application is deployed.

Expand the Resources folder and then, expand the Connectors folder.

Right-click Admin Object Resources and select Refresh. Similarly, 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 13.14.

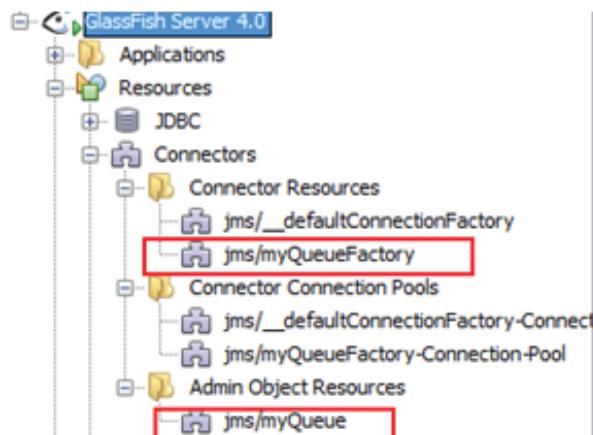


Figure 13.14: GlassFish Server with JMS Resource Objects

→ Creating the Message Producer Bean

To create a JSF ManagedBean that will act as the message producer, right-click the project name and select New → Other → JavaServer Faces → JSF ManagedBean from the New File dialog box.

Click Next. The New JSF ManagedBean dialog box is displayed.

Specify the Class Name of the bean as MessageProducerBean and the Package as mes.

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.

→ Add the JMS code to the ManagedBean

In the editor area of MessageProducerBean, press Alt+Insert to open the NetBeans Code Generator feature and select Send JMS Message.

The Send JMS Message dialog box is displayed.

Ensure that Project Destinations is set to jms/myQueue and Connection Factory is set to jms/myQueueFactory.

Click OK. NetBeans adds the proper resource declarations to your 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 () {
    }
}

```

Once the JMS resource is configured on the server, add a new send() method to the MessageProducerBean.

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

1. Initial context of the application is invoked.
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. The messages are sent to the queue after the queue is configured and a connection to the queue on the server is created. These objects will appear on the server. In order to see these resources go to Services tab, Servers → GlassFish.

→ **Creating the User Interface**

Code Snippet 4 shows the code of the JSF page sending the message

Code Snippet 4:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
  <h:head>
    <title>SendMessage</title>
  </h:head>
  <h:body>
    <h:form>
      <h:outputLabel value="Message :"/>
      <h:inputText id="message" value="#{messageProducerBean.message}"
                   size="30" />
      <h:commandButton id="Send" value="SendMessage"
                       action="#{messageProducerBean.send() }"/>
      <h:messages globalOnly="true"></h:messages>
    </h:form>
  </h:body>
</html>
```

Code Snippet 4 shows the value entered in the text area is bound to the message attribute of the MessageProducerBean. The send() method of the MessageProducerBean is invoked on clicking the Send Message button.

Deploy and run the application.

Figure 13.15 shows the sender page of the MessageProducer.

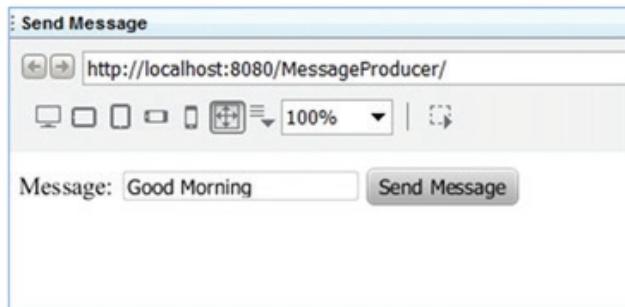


Figure 13.15: Message Producer

Figure 13.15 shows the message producer page where the user can type a message and on clicking the button, the send() method is invoked which sends the message to the message queue.

Click Send Message.

Figure 13.16 shows the output when the message is sent.

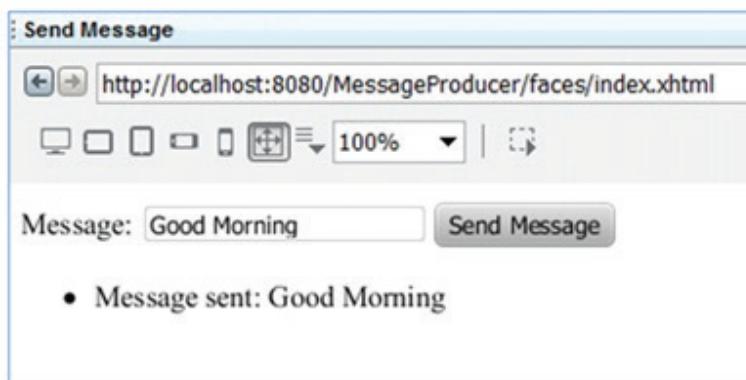


Figure 13.16: Displaying the Sent Message

→ Viewing the Queue on the Admin Console

To view the Queue resource created on the server, right-click the GlassFish server in the Services tab and select View Admin Console.

The Server Admin Console is displayed in the browser. Select Server (Admin) from the left pane.

Click JMS Physical Destinations tab. The list of available destinations is displayed as shown in figure 13.17.

The screenshot shows a software interface for managing JMS physical destinations. At the top, there are tabs: General, Resources, Properties, Monitor, Batch, and JMS Physical Destinations. The JMS Physical Destinations tab is selected. Below the tabs, the title "JMS Physical Destinations" is displayed. A descriptive text states: "Java Message Service (JMS) physical destination objects are maintained by Message Queue brokers. The queue named mq.sys.dmq is the system destination, to which expired and undeliverable messages are redirected. Click New to create a new physical destination." An "Instance Name: server" is specified. A table titled "Destinations (2)" lists the physical destinations:

Select	Name	Type	Statistics
<input type="checkbox"/>	mq.sys.dmq	queue	View
<input type="checkbox"/>	myQueue	queue	View

Figure 13.17: JMS Physical Destinations on the Server

Click View under the Statistics column for myQueue. This opens the View JMS Physical Destination Statistics page. Scroll down to see the value of Number of Messages, Number of Messages Received, and Number of Messages Sent.

→ Creating the Message Consumer

Similarly, a consumer can be configured by creating a Message-Driven Bean for the MessageProducer application.

Create an new EJB project by clicking File → New Project → Java EE → EJB Module. Name the project as MDBConsumer. Click Next and then, click Finish.

For the MessageProducer application, a Message-Driven Bean MesReceiver can be created by right-clicking the MDBConsumer project and selecting New → Other → Enterprise JavaBeans → Message-Driven Bean.

Click Next and specify the values for Name and Location as shown in figure 13.18.

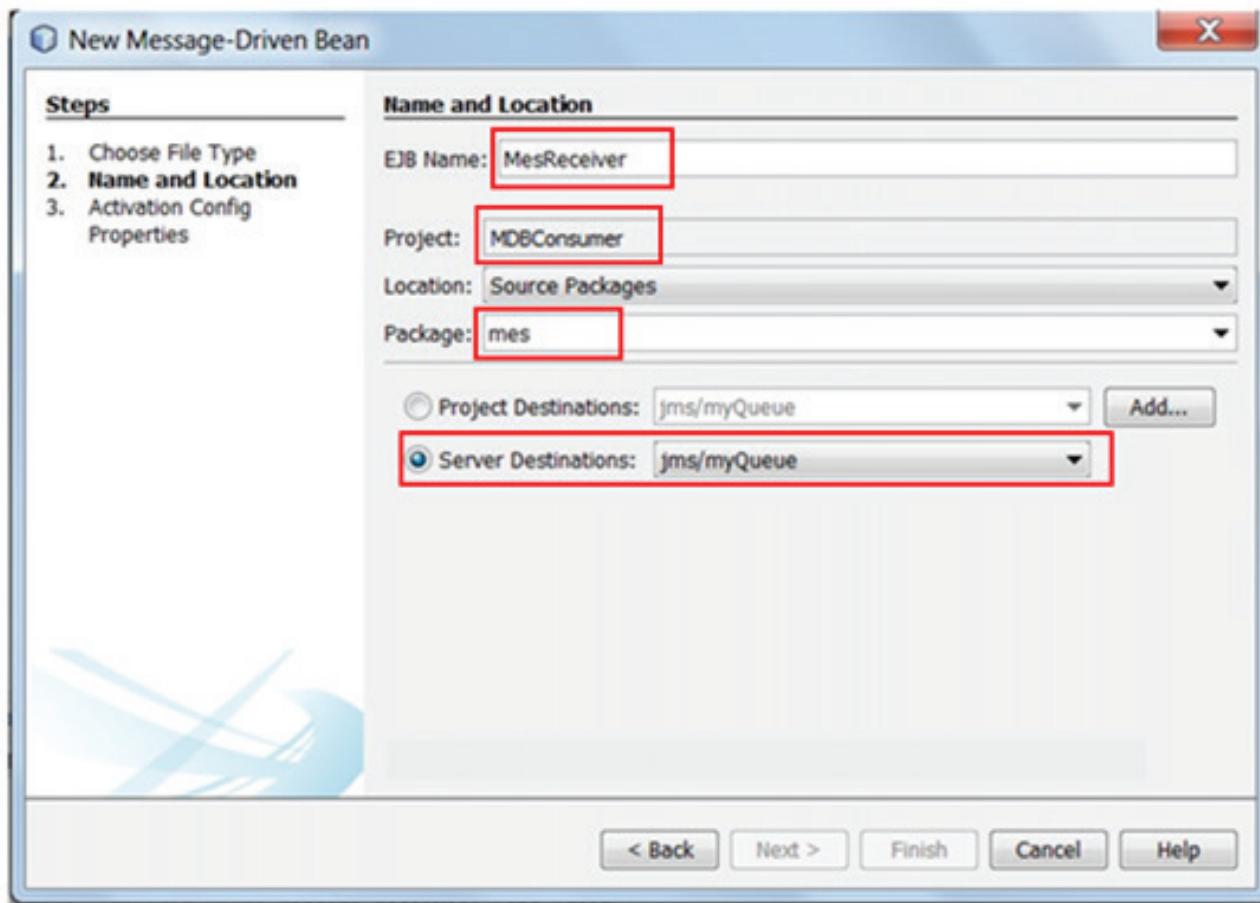


Figure 13.18: Creating a Message-Driven Bean to Receive Message from the Queue

MessageProducerBean sends a message to a Queue object on the server, in this case, a GlassFish server has been used. The Message-Driven Bean is configured to receive messages from the Queue on the destination whose jndi name is jms/myQueue. Figure 13.19 shows the Activation Config properties of the Message-Driven Bean used.

Click Next.

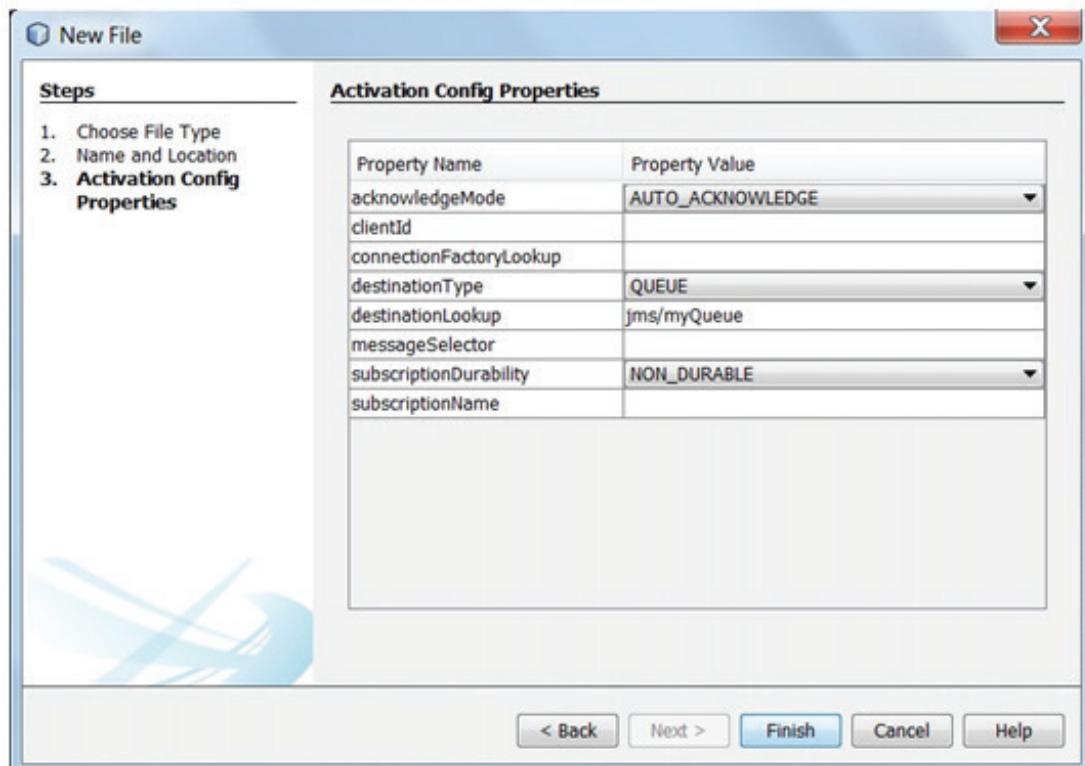


Figure 13.19: Activation Config Properties of the Message-Driven Bean

Click Finish. The Message-Driven bean is created with annotations mapping it to the jms/Queue resource on the server as depicted in Code Snippet 5.

Code Snippet 5:

```
package mes;

import java.util.logging.Level;
import java.util.logging.Logger;
import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;
```

```

@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationType",
    propertyValue = "javax.jms.Queue"),
    @ActivationConfigProperty(propertyName = "destinationLookup",
    propertyValue = "jms/myQueue")
})

public class MesReceiver implements MessageListener {

    public MesReceiver() {
    }

    @Override
    public void onMessage(Message message) {
        if (message instanceof TextMessage) {
            TextMessage msg = (TextMessage) message;
            try {
                System.out.println("Consumed message: " + msg.getText());
            } catch (JMSException ex) {
                Logger.getLogger(MesReceiver.class.getName()).log(Level.SEVERE,
null, ex);
            }
        } else {
            System.out.println("Message of wrong type: " + message.getClass().
getName());
        }
    }
}

```

In Code Snippet 5, when a Message-Driven Bean is created, the resources of the application are injected into the Message-Driven Bean. It implements `MessageListener` interface. The `onMessage()` method of the Message-Driven Bean is invoked whenever a message is received in the queue.

Deploy the `MDBConsumer` project. In the Output window, click the GlassFish Server tab. The consumed message is visible as shown in figure 13.20.

The screenshot shows the GlassFish Server 4.0 Output window with three tabs: Java DB Database Process, GlassFish Server 4.0, and MDBConsumer (run). The MDBConsumer (run) tab is active and displays the following log entries:

```
INFO: Loading application [MessageApplication2] at [/MessageApplication2]
INFO: MessageApplication2 was successfully deployed in 394 milliseconds.
INFO: Consumed message:Good Morning
INFO: visiting unvisited references
INFO: visiting unvisited references
INFO: MDBConsumer was successfully deployed in 341 milliseconds.
```

The line "INFO: Consumed message:Good Morning" is highlighted with a red box.

Figure 13.20: Consumer Message-Driven Bean

13.9 Check Your Progress

1. Which of the following statements about JMS API are true?

a.	The destination of a JMS message can be a queue.
b.	The destination of a JMS message can be a topic.
c.	JMS API can only be used with Java EE applications.

(A)	a, c	(C)	b, c
(B)	a, b	(D)	None of these

2. Which of the following is not an administered object?

(A)	Destination objects	(C)	Connection factory
(B)	JMS Provider	(D)	Message topic

3. Which of the following methods are used to implement synchronous communication through JMS API?

(A)	receive()	(C)	onMessage()
(B)	Either a or b	(D)	createContext()

4. Which of the following can be clients of JMS providers?

(A)	Webpages	(C)	Session beans
(B)	Servlets	(D)	All of these

5. Which of the following objects are used to inspect the headers of the messages?

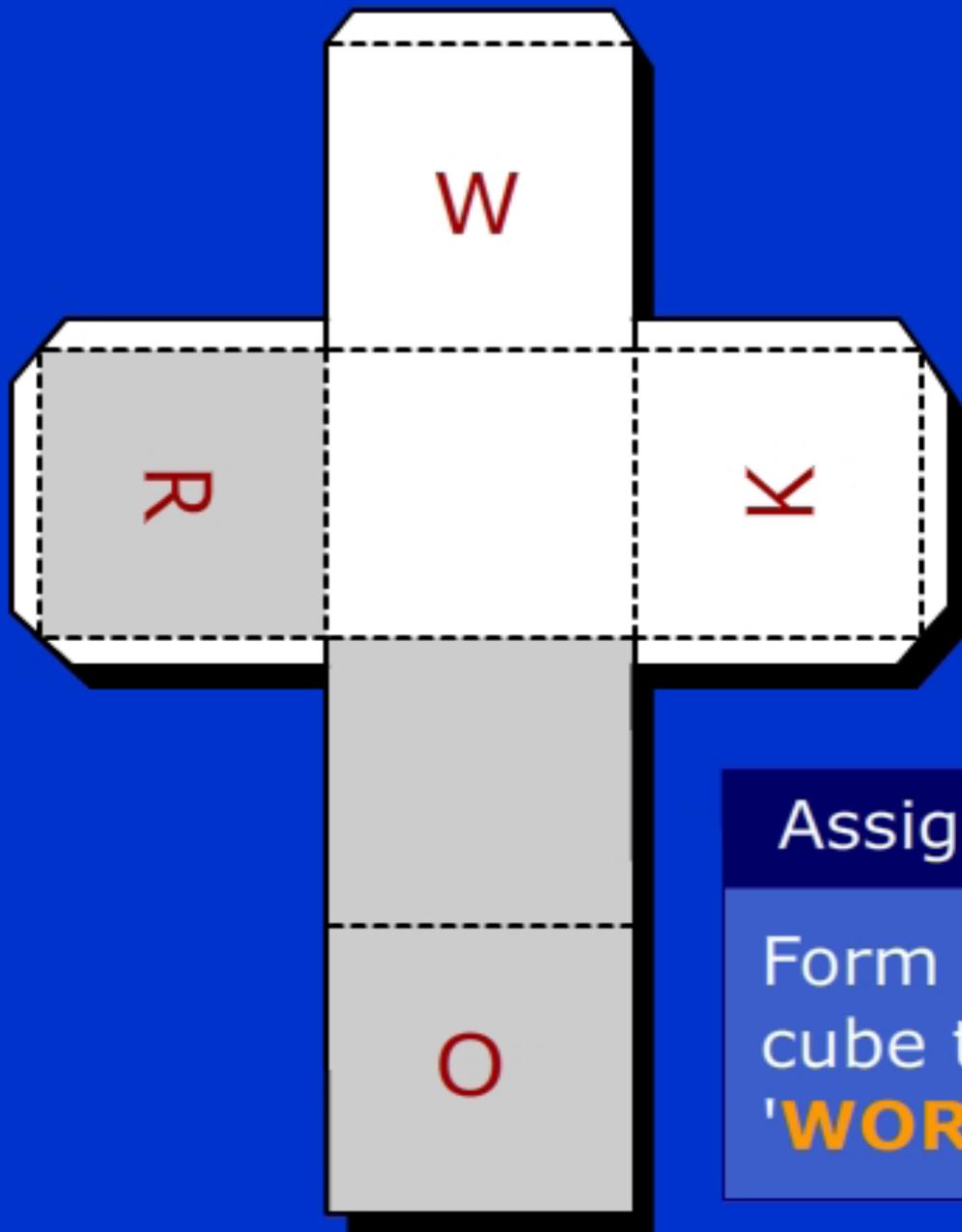
(A)	Topics	(C)	Message consumers
(B)	Queue browsers	(D)	Message producers

13.9.1 Answers

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

Summary

- JMS API is used to set up the messaging infrastructure for a Java EE application. It can also be used to develop an independent messaging system.
- JMS follows two messaging models – point-to-point messaging and publish-subscribe model.
- The message exchange between JMS message producer and receiver is primarily asynchronous.
- JMS API provides for both synchronous and asynchronous communication.
- Synchronous communication is accomplished through queue objects and asynchronous communication through topic objects.
- JMS communication is initiated after the communicating message producer and receiver are bound through JNDI namespace.
- JMS messaging system includes various administrative objects such as message producers, message consumers, connections, sessions, and so on.
- JMS Messages can be handled through Message-Driven Beans. Message-Driven Beans are invoked when a message is received on the destination.



Assignment

Form the
cube to read
'WORK'.

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

For perfection, solve the assignments @

www.onlinevarsity.com

Session - 14

Building Web Services with JAX-WS and JAX-RS

Welcome to the Session, **Building Web Services with JAX-WS and JAX-RS**.

This session describes Web services and Java APIs used for their development. It also describes the types of Web services and the standards used for ensuring interoperability. The session describes in detail how to build Web services using JAX- WS and JAX-RS.

In this Session, you will learn to:

- ➔ Describe Web services
- ➔ Describe Java APIs used in development of Web services
- ➔ Describe types of Web services
- ➔ Understand standards used to ensure interoperability of Web services
- ➔ Understand how to build Web services using JAX-RS and JAX-WS



14.1 Web Services

Web services are applications whose components are distributed over the Internet. The components of the application are mutually invoked through XML messages. These components may be running over different hardware and software platforms. The service provider and the service consumer interact with the help of standard protocols and technologies used for their development.

A Web service is invoked through XML messages such as Simple Object Access Protocol (SOAP). The pattern of message exchange is defined according to the semantics of the application. This pattern is defined through Web Services Description Language (WSDL) document.

The message exchange between the components of the Web service can be synchronous or asynchronous. In a synchronous messaging model, the message sender expects an immediate response from the message receiver. In case of asynchronous model, the message sender does not expect an immediate response from the receiver.

Following is a brief description of standard protocols and technologies used in Web services:

- **SOAP** – It is used for exchange of information. It provides a standard to package the XML documents and transport them through Internet protocols such as HTTP, SMTP, and so on.
- **WSDL** - It is used to describe the Web service and define a set of communication end points or ports on which the messages are exchanged. A collection of ports together define a Web service where each port is bound with a certain network address.
- **Universal Description, Discovery, and Integration (UDDI)** – It is used to describe the type of service provided. All the services provided over the Internet have to be registered with the UDDI registry. This kind of registration enables other applications to search and discover existing services and use them.
- **Electronic Business Using eXtensible Markup Language (ebXML)** - It is used to achieve business interoperability among multiple business services.

14.1.1 Java APIs for XML and Web Services

Following are the APIs used for XML document processing and implementation of Web services:

- Java API for XML Processing (JAXP) is used for processing XML documents based on Simple API for XML (SAX), Document Object Model (DOM), and XML Stylesheet Language Transformation (XSLT) engine. This API is used to parse and transform XML documents.
- Java API for XML-based RPC (JAX-RPC) is used to develop SOAP based synchronous Web services. To invoke a heterogeneous Web services application, JAX-RPC can use stubs-based, dynamic proxy, or dynamic invocation interface. The API allows carrying MIME content on SOAP messages. This API also supports HTTP based session management and SSL based security mechanism.

- Java API for XML Registries (JAXR) can operate on different registries for service discovery. An XML registry is an infrastructure entity which enables deployment and discovery of the available Web services. There are different types of XML registries such as ebXML, UDDI specification, and so on.
- SOAP with Attachments API for Java (SAAJ) is an API which supports transmission of SOAP message that may have various attachments such as facsimile images, GIF, and JPEG images, and so on.
- Java API for XML Messaging (JAXM) is used for XML messaging using a message provider. It allows transmission and receipt of document oriented XML messages.

14.1.2 Types of Web Services

Following are the two ways in which Web services can be technically implemented:

- Java API for XML Web Services (JAX-WS) Web services
- Representational State Transfer (REST) or RESTful Web services

JAX-WS Web services are built by using XML messages that follow SOAP standard and XML-based message architecture and formats to build the services. They use Web Services Description Language (WSDL) for defining interfaces of the service.

Following are the essential elements of SOAP based design:

- A formal contract which describes the service interface. This contract can be defined through WSDL. This contract includes information such as messages, operations, bindings, and location of the Web service.
- A Web service specification must be defined to address complex, non-functional requirements of the application. These requirements may include trust coordination, security, addressing, and so on.
- The architecture of the service should also handle asynchronous processing and invocation. Standards such as Web Services Reliable Messaging (WSRM) and APIs such as JAX-WS can be used to support asynchronous invocation of services.

RESTful Web services are developed through the Java API for RESTful Web Services (JAX-RS) API. These Web services implement Representational State Transfer (REST). The Web services are well suited for basic and ad hoc integration scenarios. The integration of RESTful Web services with HTTP is simpler than those of SOAP based Web services. This category of services does not require XML message definitions or WSDL service API definitions.

Note - Representational State Transfer (REST) is an architectural style for Web service development with a coordinated set of architectural constraints applied to components, connectors, and data elements, within a distributed system.

A RESTful design is used if the requirement of the service meets the following conditions:

- The Web service must be completely stateless.
- It will use caching mechanisms to improve the performance of the service.
- The service provider and service consumer are allowed to mutually agree upon the dynamic content and context of the application. As there is no formal mechanism which describes the Web services interface, the provider and consumer have to explicitly agree upon these aspects.
- The device accessing the Web service is a hand held device where the resources and bandwidth are limited.

JAX-WS services are useful in scenarios where there are strict Quality of Service (QoS) requirements for services which specify the requirements in terms of end user quality experience and which require stringent security mechanisms. JAX-RS is characterised by loosely coupled application components, scalability, and architectural simplicity. JAX-RS applications, being scalable, are implemented when the developer intends to scale the application over multiple platforms and different types of clients.

14.2 Creating a Simple Web Service Using JAX-WS

JAX-WS allows building Web services which are loosely coupled implementations where the communication among the service provider and consumer is message oriented. It also allows development of services which are tightly coupled where the communication among the entities is through Remote Procedure Call (RPC).

The Web service is invoked through XML messages based on SOAP protocol. The protocol defines the envelope structure, encoding rules, and conventions to be followed for message exchange. The Web service operations on the server are defined through methods in an interface. These methods are written in Java. On the client side, the client creates a proxy and invokes methods on the proxy. The JAX-WS runtime system converts these method invocations into SOAP messages and accesses the server. The client is unaware of this process.

Figure 14.1 shows how the Web service client and server communicate with each other.

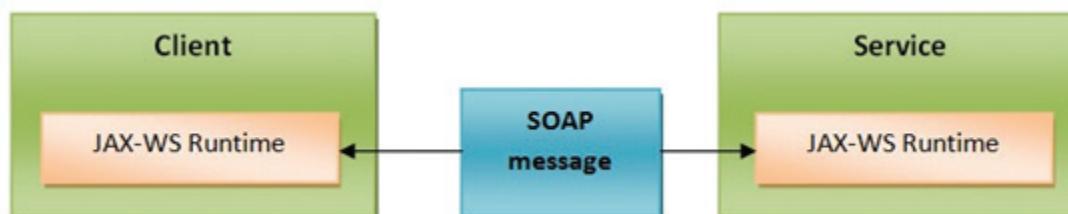


Figure 14.1: JAX-WS Web Service

Following are the steps involved in creating a Web service through NetBeans IDE:

1. A Web service has to be deployed in a container. This container can be a Web container or an EJB container. Begin the Web service creation by creating a container.
2. Define the Web service by adding operations to the Web service. Any number of operations can be added to the Web service.
3. After completing the addition of operations, deploy the Web service in the container and test it.
4. Create a service end point to the Web service. A service end point is a class or interface which comprises all the methods a client might invoke on the service. A Web service can have different types of service end points such as a JSP page, Servlet, Java application, and so on.

→ Creating a Web Service

Create a Web service which accepts the Principle, Rate of interest, and Duration parameters and returns the simple interest. In order to implement this Web service, create a Web container first.

In NetBeans IDE, a Web container can be created by creating a File → New project → Java Web → Web Application. Here, the Web application is named as InterestCalculatorApplication.

To create a new Web service in this Web container, right-click the Project InterestCalculatorApplication and select New → Other → Web Services → Web Service as shown in figure 14.2.

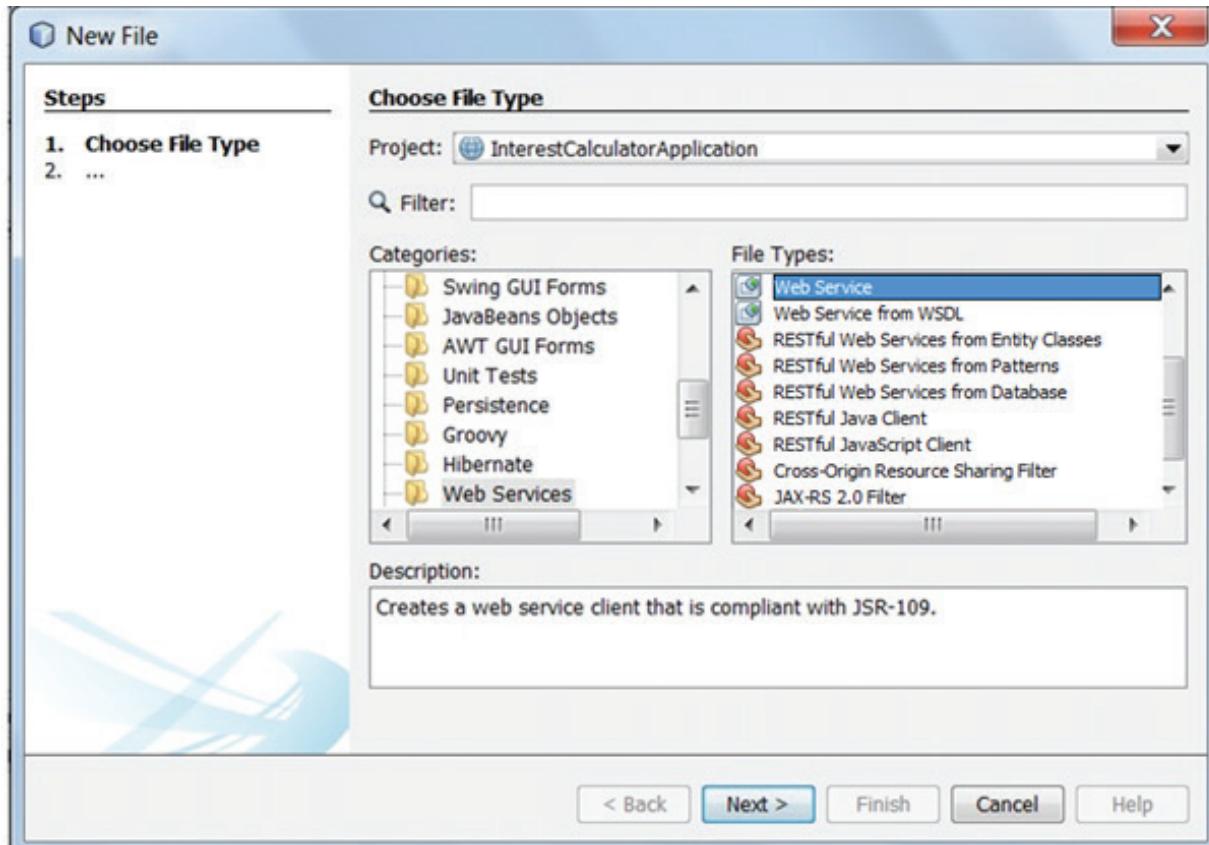


Figure 14.2: Creating a Web Service

Click Next. The New Web Service wizard is displayed. Provide the name of the Web service and the package in which the Web service must be created as shown in figure 14.3.

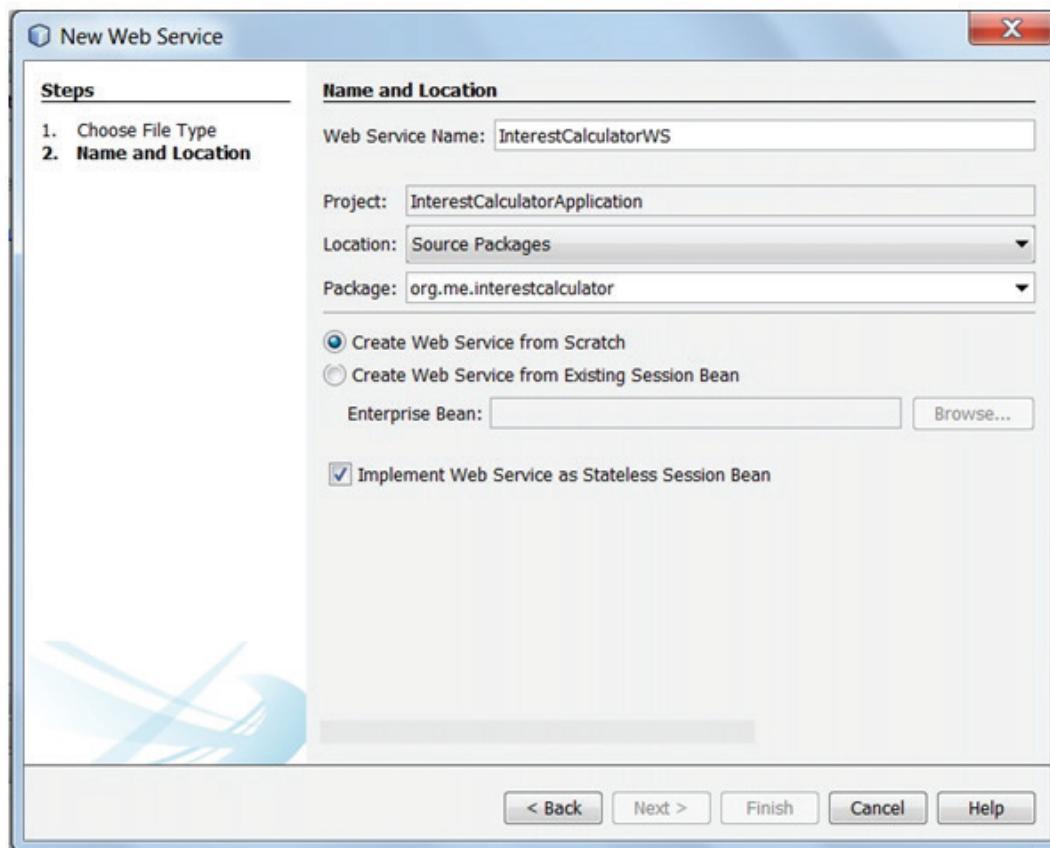


Figure 14.3: New Web Service

Select the Creating Web Service from Scratch option and select the Implement Web Service as Stateless Session Bean checkbox. This implies that the session state is non-persistent.

Click Finish.

Once the Web service named InterestCalculatorWS.java is created, select the Design view of the Web service to add operations to the Web service as shown in figure 14.4.



Figure 14.4: Adding Operations to the Web Service

Add an operation to the Web service by clicking Add Operation window. This will open a wizard to add operations to the Web service as shown in figure 14.5.

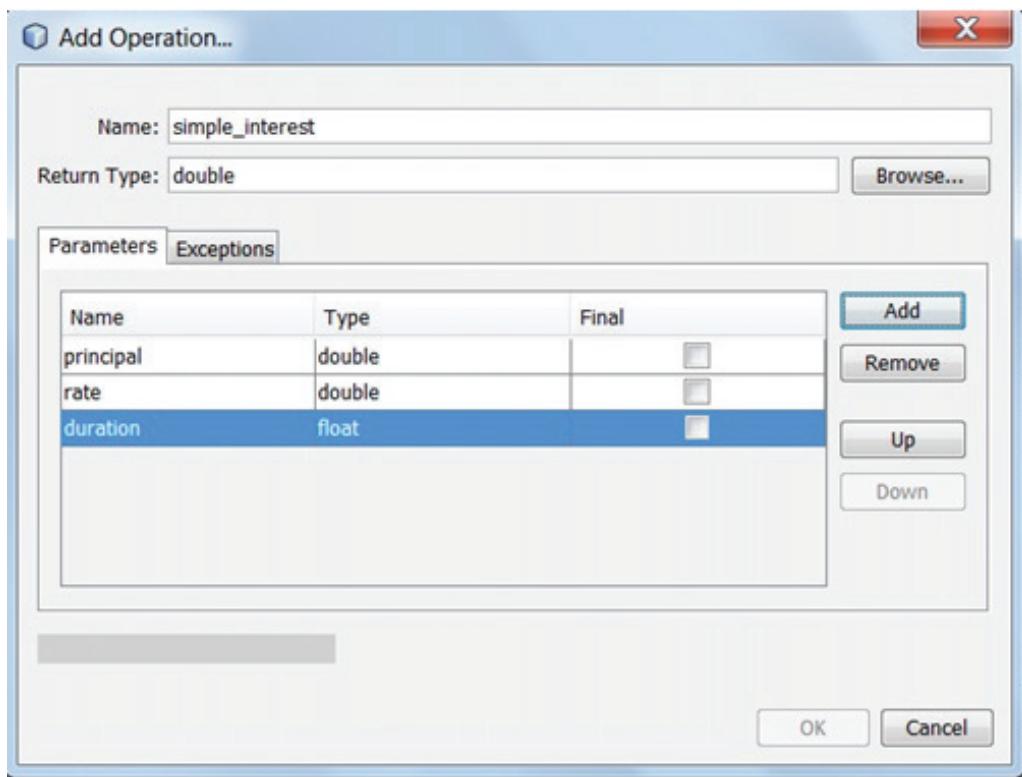


Figure 14.5: Add Operation

Fill in appropriate values as shown in figure 14.5 in the wizard to define the operation of the Web service and click OK. The Web method is added to the Web service class as shown in the Source view in figure 14.6.

```
@WebMethod(operationName = "simple_interest")
public double simple_interest(@WebParam(name = "principal") double principal,
    @WebParam(name = "rate") double rate, @WebParam(name = "duration") float duration) {
    //TODO write your implementation code here:
    double result = (principal*rate*duration)/100;
    return result;
}
```

Figure 14.6: Web Method Added to Web Service

Add the code shown in Code Snippet 1 inside the simple_interest Web method to implement the operation for calculation of simple interest.

Code Snippet 1:

```
.....
double result = (principal*rate*duration)/100;
return result;
.....
```

Once the Web method is created, deploy the Web service once all the operations are defined.

→ Creating a Web Service Client

A client can be JSP page, Servlet, or a Java application. For the current example, create a JSP page as client.

Create a Web application named InterestCalculator_Client. Add a Web Service Client to the project by right-clicking the project and selecting New → Other → Web Services → Web Service Client as shown in figure 14.7.

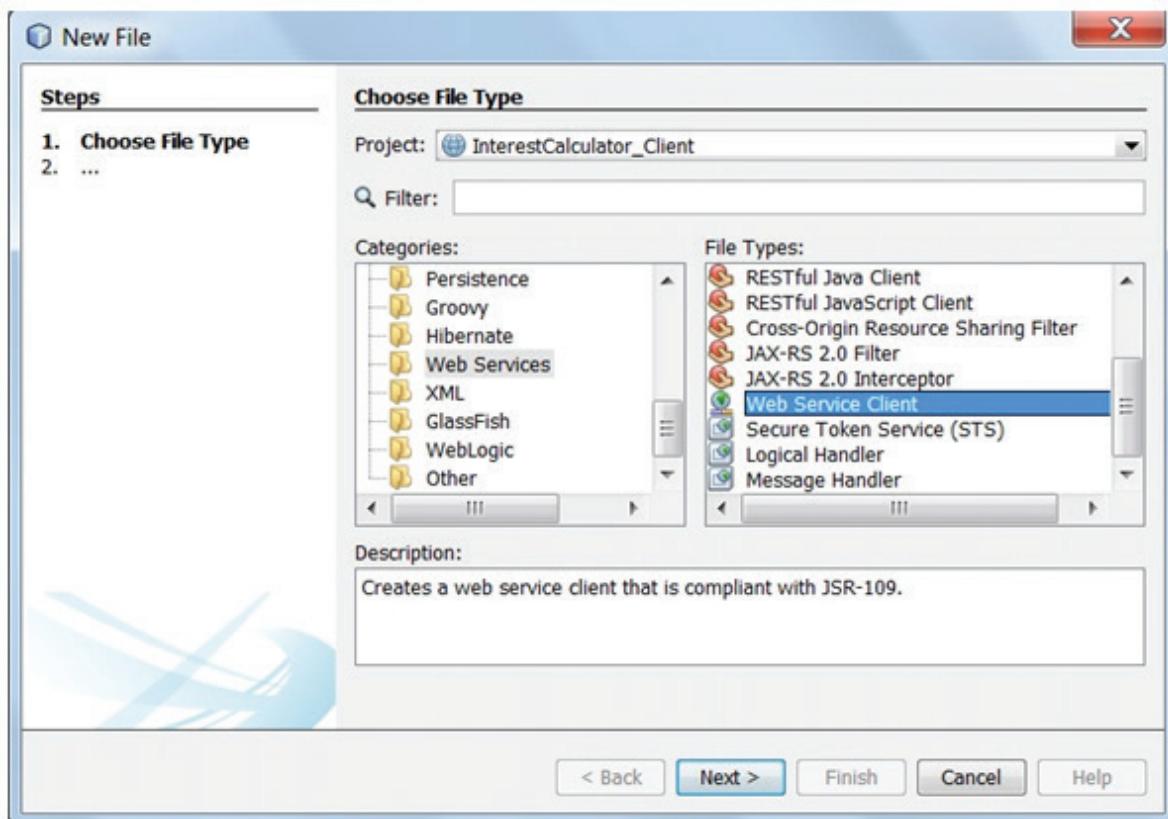


Figure 14.7: Adding Web Services Client

Selecting the Web Service Client will lead to the wizard in figure 14.8.

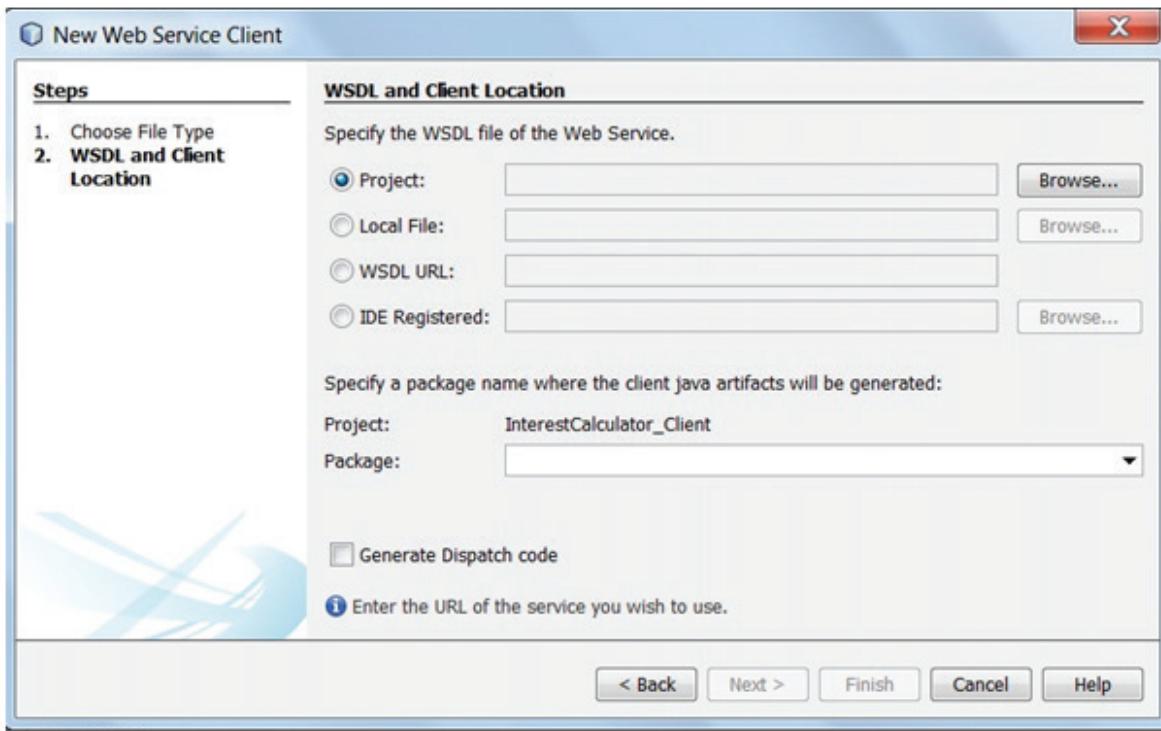


Figure 14.8: Selecting Web Service

To specify the Web Service location, click the Browse button. It displays all the available Web services as shown in figure 14.9.

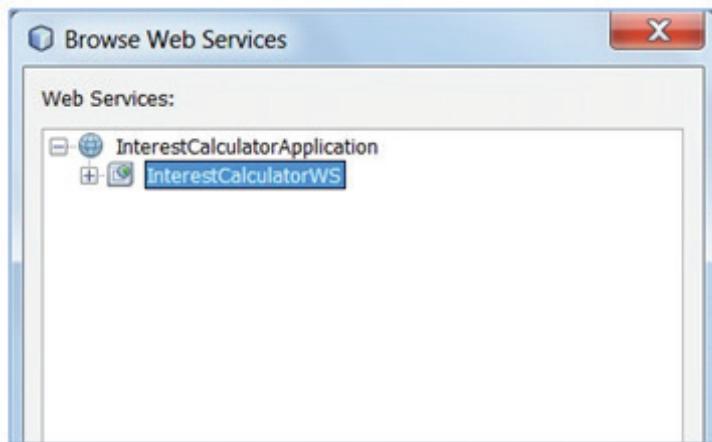


Figure 14.9: List of Web Services

Select the InterestCalculatorWS Web service and click Finish. A Web Service References directory is created in the client application folder with the reference to the Web service as shown in figure 14.10.

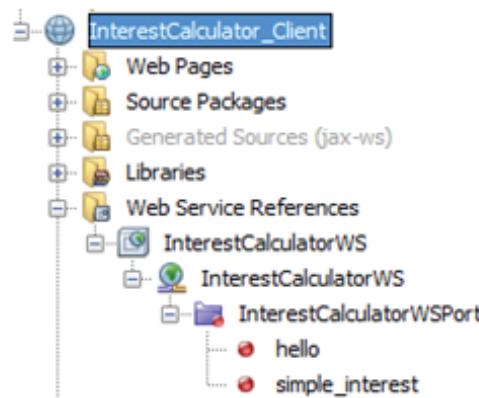


Figure 14.10: Web Service References in Client

Double-click InterestCalculatorWS and it will open the InterestCalculatorWS.wsdl file as shown in figure 14.11.

```

<?xml version='1.0' encoding='UTF-8'?><!-- Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is 1.1.4-b01-1432-1132-1132-1132 -->
<types>
  <xsd:schema>
    <xsd:import namespace="http://interestcalculator.me.org/" schemaLocation="http://localhost:13422/InterestCalculatorWS?xsd=1" />
  </xsd:schema>
</types>
<message name="simple_interest">
  <part name="parameters" element="tns:simple_interest"/>
</message>
<message name="simple_interestResponse">
  <part name="parameters" element="tns:simple_interestResponse"/>
</message>
<message name="hello">
  <part name="parameters" element="tns:hello"/>
</message>
<message name="helloResponse">
  <part name="parameters" element="tns:helloResponse"/>
</message>
<portType name="InterestCalculatorWS">
  <operation name="simple_interest">
    <input wsam:Action="http://interestcalculator.me.org/InterestCalculatorWS/simple_interestRequest" message="tns:simple_interestRequest"/>
    <output wsam:Action="http://interestcalculator.me.org/InterestCalculatorWS/simple_interestResponse" message="tns:simple_interestResponse"/>
  </operation>
</portType>
<binding name="InterestCalculatorWS_Binding" type="tns:InterestCalculatorWS">
  <operation name="simple_interest">
    <input wsam:Action="http://interestcalculator.me.org/InterestCalculatorWS/simple_interestRequest" message="tns:simple_interestRequest"/>
    <output wsam:Action="http://interestcalculator.me.org/InterestCalculatorWS/simple_interestResponse" message="tns:simple_interestResponse"/>
  </operation>
  <operation name="hello">
    <input wsam:Action="http://interestcalculator.me.org/InterestCalculatorWS/helloRequest" message="tns:helloRequest"/>
    <output wsam:Action="http://interestcalculator.me.org/InterestCalculatorWS/helloResponse" message="tns:helloResponse"/>
  </operation>
</binding>
<service name="InterestCalculatorWS_Service">
  <port name="InterestCalculatorWS_Port" binding="tns:InterestCalculatorWS_Binding">
    <soap:address location="http://localhost:13422/InterestCalculatorWS?wsdl=1" />
  </port>
</service>

```

Figure 14.11: InterestCalculatorWS.wsdl File

This is the Web Service Description Language (WSDL) file that holds all the information related to the Web Service whose reference was added to the client application.

→ Creating the JSP Page to Access the Web Service

Create a new JSP page named index.jsp. Modify the JSP page as shown in Code Snippet 2.

Code Snippet 2:

```

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <h1> <%-- start web service invocation --%><hr/>
        <%
        try {
            org.me.interestcalculator.InterestCalculatorWS_Service service = new
            org.me.interestcalculator.InterestCalculatorWS_Service();
            org.me.interestcalculator.InterestCalculatorWS port = service.
            getInterestCalculatorWSPort();
            // TODO initialize WS operation arguments here
            double principal = 1000.0d;
            double rate = 12.0d;
            float duration = 3.0f;
            // TODO process result here
            double result = port.simpleInterest(principal, rate, duration);
            out.println("Simple Interest is "+result);
        } catch (Exception ex) {
            // TODO handle custom exceptions here
        }
        %>
        <%-- end web service invocation --%><hr/></h1>
    </body>
</html>

```

Code Snippet 2 depicts the call to the Web service from the client application. The method `simpleInterest` is invoked from the Web Service using the references to the Web service added earlier in the application.

A Web Service reference object is created and it is bound to the port through which the client can access the Web service. Once the binding is successful, the required operation of the Web service is invoked through the port. The Web service method, simpleInterest(), is invoked to calculate the Simple Interest for the values passed for principal, rate, and duration

Add the web.xml deployment descriptor and set index.jsp as the start up page in the Welcome Files box.

Build and run the client application. The result of running the Web service client application is shown in figure 14.12.

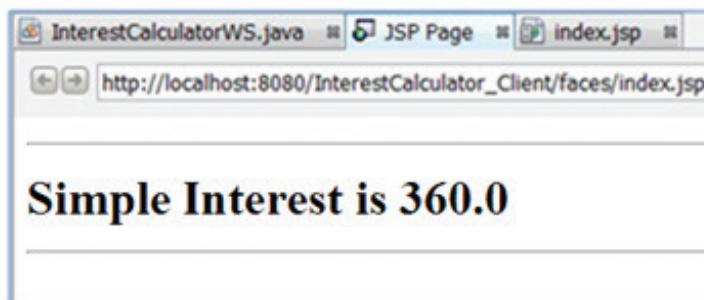


Figure 14.12: Execution of the Client Service

14.3 Types Supported by JAX-WS

Java platform and XML together are ideal combination for building portable Web services and Web service application clients. This is because, XML enables exchange of data across platforms and Java programs can run across different platforms. The Java Architecture for XML Binding (JAXB) API is used for accessing XML documents from Java applications. It allows Java developers to access and process XML data without actually knowing XML and how to process it. JAXB also provides a way to generate XML documents from Java applications. In short, JAXB handles the access of XML documents and their processing, and also handles the generation of XML documents from Java applications.

An XML schema can be mapped to a Java class based on some standard conversions specified by JAXB. Similarly, a Java class can also be mapped to an XML schema according to the standard conversions provided by JAXB. Table 14.1 shows the mapping of XML schema to Java data types.

XML Schema Type	Java Data Type
xsd:string	java.lang.String
xsd:integer	java.math.BigInteger
xsd:int	Int
xsd:long	Long
xsd:short	Short
xsd:decimal	java.math.BigDecimal
xsd:float	Float
xsd:double	Double

XML Schema Type	Java Data Type
xsd:boolean	Boolean
xsd:byte	Byte
xsd:QName	javax.xml.namespace.QName
xsd:dateTime	javax.xml.datatype.XMLGregorianCalendar
xsd:base64Binary	byte[]
xsd:hexBinary	byte[]
xsd:unsignedInt	Long
xsd:unsignedShort	Int
xsd:unsignedByte	Short
xsd:time	javax.xml.datatype.XMLGregorianCalendar
xsd:date	javax.xml.datatype.XMLGregorianCalendar
xsd:g	javax.xml.datatype.XMLGregorianCalendar
xsd:anySimpleType	java.lang.Object
xsd:anySimpleType	java.lang.String
xsd:duration	javax.xml.datatype.Duration
xsd:NOTATION	javax.xml.namespace.QName

Table 14.1: Mapping XML Schema to Java Data Types

Table 14.2 shows the mapping of primitive Java data types to XML schema entities.

Java Class	XML Data Type
java.lang.String	xs:string
java.math.BigInteger	xs:integer
java.math.BigDecimal	xs:decimal
java.util.Calendar	xs:dateTime
java.util.Date	xs:dateTime
javax.xml.namespace.QName	xs:QName
java.net.URI	xs:string
javax.xml.datatype.XMLGregorianCalendar	xs:anySimpleType
javax.xml.datatype.Duration	xs:duration
java.lang.Object	xs:anyType
java.awt.Image	xs:base64Binary
javax.activation.DataHandler	xs:base64Binary
javax.xml.transform.Source	xs:base64Binary
java.util.UUID	xs:string

Table 14.2: Mapping of Primitive Java Types to XML Schema Elements

14.4 Web Services Interoperability and JAX-WS

Interoperability of Web services is achieved through standard protocols and standard methods of communication. The industry consortium Web Services Interoperability (WS-I) has defined a WS-I Basic Profile, which provides guidelines and tests to ensure interoperability of the Web services. It defines an interoperable subset of core Web service specifications which include XML schema, SOAP, WSDL, and UDDI.

A developer can achieve interoperability of Web services by following these guidelines.

14.5 RESTful Services

Representational State Transfer or REST is an architectural style for designing Web services. It emphasizes stateless client-server architecture where Web services are seen as resources and are accessed through Universal Resource Identifiers (URIs).

Following are the important features of RESTful services:

- **Using URIs for resource identification** – All the resources in RESTful services are identified through URIs, through a global address space for resource and service discovery. These URIs expose the directory structure and each resource is identified through a global addressing space. This global address space provides for the identification of resources and services.
- **Using HTTP methods for resource manipulation** – All the resources are created through HTTP methods GET, PUT, POST, and DELETE. These methods are mapped to the SQL statements for database manipulation to maintain a uniform interface for resource management. The RESTful design defines a one-to-one mapping between the database operations and the HTTP methods to provide a uniform interface as follows:
 - READ operation, which is usually implemented through select statements in databases, is mapped to the GET method of HTTP.
 - CREATE operation on databases is mapped to the POST method where it is used to create resources.
 - Update operation on the databases is mapped to the PUT method of HTTP.
 - DELETE operation on databases is mapped to DELETE method of HTTP.
- **The Service provider and Service consumer are stateless** – All the client requests and server responses in restful services are stateless. The service provider need not maintain any application context for requests. This design uses techniques such as URL rewriting, cookies, and hidden form fields to transfer the parameters associated with a request. This provides improved Web service performance and simplifies the implementation of the server-side components as they need not maintain the state of any sessions.

- **Self descriptive messages** – Resources and their representation are separated from each other. This enables accessing the resources in different formats such as JPEG, PDF, and so on. The metadata of each resource is available and used as and when required by the service components.

14.6 Creating a RESTful Root Resource Class

Root resource classes are the entry point for a JAX-RS Web service. These are Plain Old Java Objects (POJO) which are annotated with @Path or at least one method of the class is annotated with @Path.

Following are the requirements for a Java class to be a root resource class:

- The class must be annotated with @Path annotation, where the path given along with the annotation is the root URI for all the resources of the Web service. Following is an example for usage of @Path annotation:

```
@Path("/admissionservice/")
```

- A public constructor which can be invoked at the runtime. If the public constructor is a parameterized constructor, then the runtime environment should provide all the required parameters for the constructor.
- Any one of the root resource class methods should be mapped to the HTTP methods.

Code Snippet 3 shows a root resource class for admission service of a university.

Code Snippet 3:

```
package demo.jaxrs.server;

import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.Response;
@Path("/admissionservice/")
public class AdmissionService{
    public AdmissionService() {
        ...
    }
}
```

```

@GET
@Path("/studentname/")
@Produces("MediaType.TEXT_PLAIN")
public String getStudName (@QueryParam("Roll_no") String Roll_no) {
    ...
}

@GET
public Student getStudent (@QueryParam("Roll_no") String Roll_no) {
    ...
}

@DELETE
public Response deleteStudent (@QueryParam("Roll_no") String Roll_no) {
    ...
}

@PUT
public Response updateStudent (Student stud) {
    ...
}

@POST
public Response addStudent (Student stud) {
    ...
}

```

Resource methods are those which are annotated with any of the HTTP methods. In Code Snippet 3, there are four root resource methods. A resource method can return a void, Entity object, or Response object.

Annotations are a form of metadata used in Java applications for resource injection. They are helpful for compile time and runtime processing to make the application execution more efficient.

Table 14.3 shows all the annotations used in JAX-RS Web services.

Annotation	Description
@Path	The method marked with this annotation specifies the location of the root resource class based on which the URI can be created for all the data to be accessed on the Web service.
@GET	The method marked with this annotation maps the function implemented in the method to a HTTP GET request.
@POST	The method marked with this annotation maps to the HTTP POST method and the method definition acts according to the resource which is being posted to the Web server.
@PUT	The method marked with this annotation maps to the HTTP PUT method and performs operations according to the resource which is put to the Web service.
@DELETE	The method marked with this annotation maps to the DELETE HTTP method.
@HEAD	The method marked with this annotation is used to process HTTP headers.
@PathParam	The method marked with this annotation is used to extract parameters with respect to the path of the Web service from the URI for further processing.
@QueryParam	The method marked with this annotation extracts the parameters used for querying the Web service from the URI.
@Consumes	The method marked with this annotation represents the MIME types that can be consumed by a resource in the Web service.
@Produces	The method marked with this annotation specifies the MIME types that can be produced by the resource in the Web service.

Table 14.3: Annotations in Root Resource Class

14.7 Overview of Client API for Accessing RESTful Resources with JAX-RS API

Client API is used by developers to create clients for RESTful services. This API is defined as javax.ws.rs.client package.

Following are the steps for creating a basic client request using the Client API:

1. Import the javax.ws.rs.client package to the class which is supposed to be the client of the Web service and obtain an instance of javax.ws.rs.client.Client.
2. The Client instance must be configured with the target, where the target is the Web service.
3. Create a request for the Web service based on the configured target and invoke the request.

Following is an example of a request:

```
Client client = ClientBuilder.newClient();  
String name = client.target("http://College.com/admissionservice/stu-  
dentname")  
.request(MediaType.TEXT_PLAIN)  
.get(String.class);
```

The given example creates a client object and sets the target based on the URI passed as parameter. The request method specifies the media type which is being requested. In this case, the request is placed for plain text. The get method has the expected data as parameter and invokes the service to obtain a string object from the Web service.

14.8 Check Your Progress

1. Match the following APIs with their corresponding functions.

	APIs		Functions
a.	JAXB	1.	An API for XML messaging
b.	JAXM	2.	An API used for discovery of Web services
c.	JAXR	3.	API for accessing XML documents from Java applications

(A)	1-c, 2-a, 3-b	(C)	1-b, 2-a, 3-c
(B)	1-a, 2-c, 3-b	(D)	1-c, 2-b, 3-a

2. Which of the following is not a feature of SOAP based design?

(A)	Service contract is not defined.	(C)	The Web service requirements of the application include trust coordination and security specifications.
(B)	WSDL document has specification of messages, operations, bindings, and location of Web service.	(D)	Supports asynchronous invocation of services.

3. Which of the following statements about RESTful services are true?

(A)	It is not stateless.	(C)	They are suitable for devices with minimal resources.
(B)	The parameters for method invocation are not passed through the URI.	(D)	None of these

4. Given are the steps for creating a Web service using JAX-WS. Arrange them in the correct sequence.

(A)	Create a Web client
(B)	Create a Web service
(C)	Create a Web or EJB container

(A)	1, 2, 3	(C)	3, 1, 2
(B)	2, 1, 3	(D)	3, 2, 1

5. Which of the given annotations do not map to an HTTP method?

(A)	@Path	(C)	@Head
(B)	@Get	(D)	@Post

14.8.1 Answers

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

Summary

- Web services are Web applications whose components are distributed over the Internet and are mutually invoked through XML messages.
- JAX-WS and JAX-RS are the APIs in Java which are used to develop SOAP based and RESTful services respectively.
- SOAP based Web services have a security specification and WSDL contract which defines the service.
- Interoperability of Web services is achieved through standard protocols and standard methods of communication.
- RESTful services are stateless services and the request parameters for the service methods are passed through the URLs.
- Annotations are used in Web services for resource injection and to generate metadata.

Technowise



Are you a
TECHNO GEEK
looking for updates?

Login to

www.onlinevarsity.com

Session - 15

Java Security

Welcome to the Session, **Java Security**.

This session describes the security mechanisms in Java. It explains how application containers are secured. It also explains how to define user roles and groups. It describes tasks associated with securing enterprise and Web applications.

In this Session, you will learn to:

- Describe the security mechanisms in Java
- Describe how to secure application containers
- Explain how to define user roles and groups
- Explain tasks associated with securing enterprise applications
- Explain how to secure Web applications
- Explain implementation of programmatic security



15.1 Introduction

Enterprise applications and Web applications are deployed in containers and these containers provide security to the application data. Containers provide two types of security: declarative security and programmatic security.

Declarative security for an application is provided through mechanisms such as deployment descriptors and annotations.

A deployment descriptor is an XML file which describes how the application has to be deployed with container options, security settings, and other configuration settings.

Annotations are metadata which are used in the class to specify the security settings. The specifications provided through the metadata can be overridden at run time by the data in the deployment descriptor.

Programmatic security for an application is provided by embedding the security mechanisms in the application. This is provided in addition to declarative security, when that is not sufficient for the application.

Following are the expectations from an efficiently implemented security mechanism:

- Prevents unauthorized access to application data
- Implements non-repudiation, that is, the users hold responsibility for the operations they perform
- Ensures service continuity of the application and protects the system from service interruptions due to security breaches

Java applications being portable across platforms, the security mechanism should be interoperable across enterprise and application boundaries. The security mechanism should be easy to administer and be transparent to the system users.

→ Characteristics of Security Mechanisms

The security mechanisms should ensure that only authorized users have access to the resources of the application. Authorization of a user to access resources includes identification and authentication. Every entity in an application is associated with an identity and this identity of the entity has to be established before giving access to the application resources. Identification process identifies the entity which is trying to access the application resources and the authentication process verifies the identity of the entity and provides access.

When the following characteristics of application security are properly addressed, it minimizes the security threats to the application data:

- **Authentication** – This mechanism is used to prove the identity of the user who is trying to access the application data. The administrative entity of the application defines different roles for users who can access the application. Whenever a user attempts to access the application data, the user has to present his/her identity. The identity of the user has to be verified and then allowed access. This process of verification is known as authentication.

- **Authorization** – This mechanism provides access to the resources of the application after the identity of the user is verified based on the permissions granted to the role to which the user belongs.
- **Data integrity** – The security mechanism of the application should ensure that the data is not modified by any unauthorised third party.
- **Confidentiality or data privacy** - This is a mechanism which should ensure that the application data is not revealed to unauthorized users.
- **Non- repudiation** – This mechanism ensures that the users cannot deny an action performed on the application data.
- **Quality of Service** – It defines the service characteristics expected from the security mechanism.
- **Auditing** – This mechanism analyses all the security related events of the application. The records are maintained through system logs and other record keeping mechanisms.

15.2 Security Mechanisms

Java EE provides security mechanisms through various packages and APIs. These APIs incorporate the desirable characteristics which can be used in the application. These APIs can be used individually or in combination with other APIs to provide the required security service.

Following are the security mechanisms provided by Java EE:

- ➔ **Java Authentication and Authorization Service (JAAS)** – JAAS provides an extensible and pluggable framework for defining the authorization and authentication mechanisms in the application.
- ➔ **Java Generic Security Services (Java GSS-API)** – Java GSS is an API used for secure message exchange among various components of the application. It provides cryptographic techniques to implement the security for message exchange in the application.
- ➔ **Java Cryptography Extension (JCE)** – JCE provides a spectrum of cryptographic techniques which can be used for key agreement, key generation, encryption, and message authentication algorithms. It supports generation of encrypted data as both blocks and streams.
- ➔ **Java Secure Socket Extension (JSSE)** – This API provides a java implementation of Secure Sockets Layer(SSL) and Transport Layer Security (TLS) protocols which include functionality for message integrity, data encryption, server authentication, and so on.
- ➔ **Simple Authentication and Security Layer (SASL)** – SASL is an Internet standard which defines protocol authentication and also defines the security layer between the client and server applications.

The application component container also provides certain security mechanisms which are robust and can be easily configured for authenticating and authorizing users. These security mechanisms define the security layers at different levels of the application. Following are the security mechanisms defined at different layers of the application:

- Application Layer Security
- Transport Layer Security
- Message Layer Security

Application Layer Security is provided by the component container. The container extends the security services based on the type of the application. Application firewalls can be used in the application layer, to protect the communication among different application components and also insulate the application access from malicious attacks. It is appropriate to use application layer security in case of enterprise applications but in case of Web applications the application data would traverse through several intermediate components. Hence, security mechanisms at the transport layer level and message layer level are required for a complete security solution.

Following are the advantages and disadvantages of application level security:

- ➔ When application layer security is applied, then the security mechanisms are fine grained with specific configuration settings and are defined in a way suitable for the application.
- ➔ The disadvantage of using application layer security is that the security attributes are not transferrable among multiple application types.
- ➔ When there are multiple protocols used in the application then the application level security is not sufficient.

Transport Layer Security is provided by the mechanisms or protocols used for transferring data. Commonly used protocols in transport layer are secure HTTP using Secure Sockets Layer (SSL). This is a point-to-point security mechanism which is used for authentication, message integrity, and confidentiality between two entities on the network.

When the client and server communicate over an SSL protected session, they authenticate each other, exchange keys used during communication, and also agree upon cryptographic algorithms. These mechanisms are applied only for two communicating entities in the network but not to the application data throughout its traversal over the network.

Following are the different phases in transport layer security:

- ➔ The communicating entities, that is, the client and server agree upon a cryptographic algorithm that can be used for communication.
- ➔ Agree upon a key that is to be used during communication.
- ➔ Transport layer security uses symmetric cipher during information exchange.

When HTTP is used along with SSL, the communication requires digital certificates.

Following are the advantages and disadvantages of using transport layer security:

- It is a relatively simple mechanism as compared to application layer security and is only applicable between two communicating entities in the entire network.
- The disadvantage, however, is that it is tightly coupled with the transport layer.
- The protection at transport layer is transient. The security mechanism is not applicable after it is received at the receiving end of the pair of nodes. A new security mechanism has to be applied for the following pair of nodes as transport layer security mechanism is a point-to-point security.

In case of **Message Layer Security**, the security information is sent along with the message. The message is a SOAP message, where the data is in message header. Message level security allows a part of the message to be encrypted. This encrypted message can traverse through several intermediary nodes without being accessed by them. The encrypted message can be accessed by the intended recipient only. Message layer security can be implemented as both end-to-end security and point-to-point security.

Following are the advantages and disadvantages of message layer security:

- Message level security mechanism stays with the message irrespective of the number of nodes through which it may traverse.
- It allows for the security mechanism to be applied only to a part of the message.
- It is independent of the transport protocol or application environment.
- The disadvantage of message level security is that it adds additional overhead of message processing at each node.

15.3 Container Security

Containers provide application security either declaratively or programmatically. Declarative security mechanisms use annotations and deployment descriptors to declare the security mechanisms.

→ Annotations for Specifying Security Mechanisms:

Annotations provide declarative security mechanisms in the class files. However, all the security information cannot be specified through annotations. Therefore, certain security mechanisms have to be specified through the deployment descriptor.

`@RolesAllowed`, `@PermitAll`, `@DenyAll` are some of the annotations used for implementing declarative security.

→ Deployment Descriptors used for Specifying Security Mechanisms

Declarative security mechanisms can be provided through deployment descriptors. Changing the security mechanisms through deployment descriptor removes the need of changing the application source code. Deployment descriptors can also be used to provide structural information for each component in the application.

NetBeans IDE provides different tools to create and modify deployment descriptors. web.xml is the deployment descriptor for Web applications and ejb-jar.xml is the deployment descriptor for EJB applications.

→ Programmatic Security

Programmatic security can be provided to the applications by using the methods of APIs provided by Java EE.

15.4 Working with Users, Groups, and Roles

Java applications define users, their respective roles, and user groups that are used to define access rights on application data. The users are authenticated and then given access to protected resources. Following are the steps involved in authentication:

1. An application developer would allow different users to access data by writing code which would prompt for username and password.
2. Application developer defines the security policy of the application through annotations or deployment descriptors.
3. The server administrator sets up groups of authorized users on the GlassFish server or any other Web server.
4. The application deployer maps different application security roles to users and user groups.

Any entity which tries to access application data can be termed as a user. Each user's identity is authenticated through the username and password. The authorized users in an application are assigned with roles such as administrator, user, and so on. Multiple users can be grouped together into a user group and the administrator can define access rights to this group of users at a time. A security policy domain defined for a Web application is known as a realm.

All the protected resources on an application server can be partitioned into different protection spaces, where each protection space can have its own authentication scheme.

This protection space can be termed as a realm. A realm can be defined as users and groups of users who are governed by the same authentication policy. The Java EE authentication service manages different realms on Web server, for example, GlassFish server has file realm, admin realm, and certificates realm preconfigured on it.

In case of file realm the user credentials are saved in a file known as keyfile. In this realm, the user identity is established by referring to the file. This realm is useful for enterprise applications but not very relevant for the Web applications.

In case of certificate realm, the server stores user credentials through a certificate database. The user identity in this case is established through X.509 certificate.

The admin realm is also a file realm where the user credentials are stored in a file named admin-keyfile.

A user refers to an individual or an application program which tries to access the application resources. The user has to establish identity to access the resources. Roles are associated with the identity of the user and the user can access the resources according to their respective roles.

A group of users in an application domain is a set of authenticated users who are defined by a set of common traits. Every group has a distinct set of access rights in the application. These user groups are defined on the Web server, for example a group of users is associated with entire GlassFish server instance whereas a role is associated with an application on the GlassFish server.

A Role defines access to a particular set of resources in an application. A role is characterised with the access rights associated with it, for instance, administrator is a role for an application.

Following are other terms associated with Java security mechanisms:

- **Principal** – Principal is an entity in the application which is identified through its name and authenticated using authentication data.
- **Security policy domain** – A security policy domain defines the scope on which the security policy is defined.
- **Security attributes** – Refer to properties associated with a principal in the security domain.
- **Credentials** – Are the security attributes which are used for authentication of a principal.

15.5 Basic Security Tasks for Enterprise Applications

Security of an enterprise application is implemented by system administrators, application developers, and deployers. Following are the tasks for defining the security of the application:

- Defining the database of users who can access the application data and assigning them to appropriate groups.
- Defining the method of identity propagation.
- Configuring the Web server for application execution.
- Annotating the classes and methods to identify the methods which need to have restricted access.

Figure 15.1 shows the enterprise application architecture

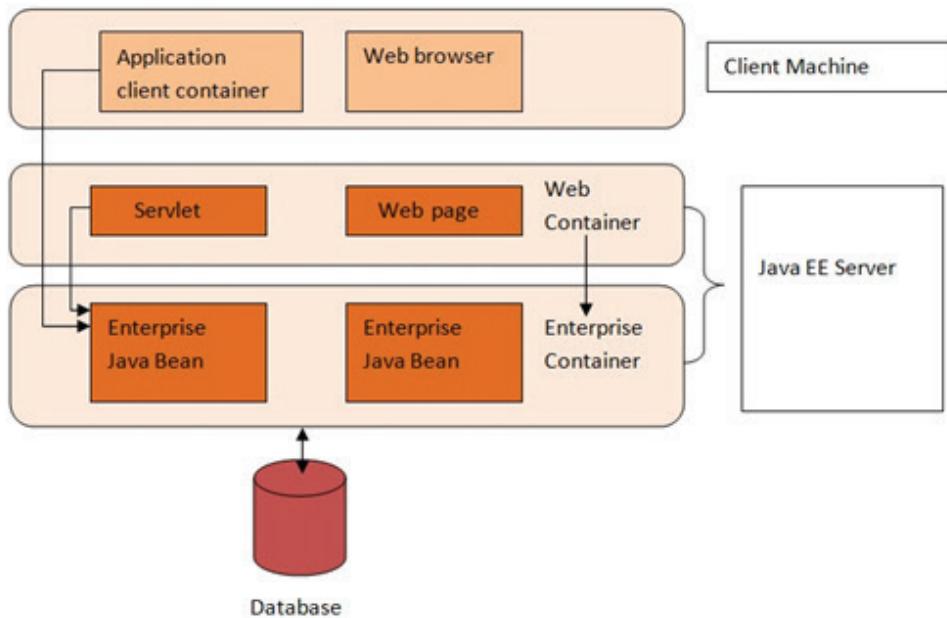


Figure 15.1: Layered Architecture for Java Enterprise Applications

There are three different containers in figure 15.1; the Application client container, Web container, and Enterprise container. The security of the components is independently provided by the respective containers.

15.6 Securing Enterprise Beans

The security mechanism of the enterprise beans can be specified declaratively or programmatically. If the security mechanism is specified declaratively, then it is done through annotations or through deployment descriptors. If the security mechanisms are specified programmatically, it is done through the Java APIs. Declarative definition of the security mechanisms is the preferred way of defining the security mechanism.

For simple deployment of the security mechanisms, the application developer can define various security roles. Each security role has a set of permissions that a subset of the application users should possess to use the application. The application developer will define different types of users who may access the application and also the methods which are to be accessed by these users. Based on these facts, the roles in the application are defined. The definition of these roles is defined through annotations and deployment descriptors.

When a user tries to access a bean method whose access is only given to a subset of users, it prompts for a user name and password. The username and password credentials provided by the user are verified by comparing against the existing database of users. If the user exists in the database, then the corresponding role is granted. Based on the role, the user is allowed/denied access the current bean method according to the permissions granted to the role.

Programmatic security mechanisms are used when declarative security mechanisms are not sufficient.

15.6.1 Securing an Enterprise Bean Using Declarative Security

When an application developer uses declarative security to define permissions to access methods and mechanisms for authentication, then it implies that the developer is passing information to the application deployer. The deployer uses this information to define the method permissions to the security roles.

The deployer defines the security view of the application. Security view of the application implies a set of security roles and groups of permissions associated with each role to access the application. The security roles must represent a set of users who are intended to access the application.

→ Specifying Authorized Users by Declaring Security Roles

Permissions of access can be defined at the class level or at the method level. Apart from the annotations `@DeclareRoles` and `@RolesAllowed` which were discussed earlier, the `@PermitAll` and `@DenyAll` annotations are used to declare access to the bean methods or classes.

`@DeclareRoles` specifies the list of all the roles that will be used in the application. This annotation is specified on a bean class. Using this annotation is equivalent to adding a role in the Web application deployment descriptor.

`@RolesAllowed` specifies the roles that are allowed access to methods. This annotation can be specified on a class or on one or more methods.

For example,

```
@DeclareRoles({"Administrator", "Faculty", "Student"})
public class Calculator {

    @RolesAllowed("Administrator")
    public void setIncentive(int rate) {
        ...
    }
}
```

`@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. Following code demonstrates the usage of `@PermitAll` annotation.

```
public class Hello {
    @PermitAll
    public void greeting(String name) {
        System.out.println("Hello, " + name);
    }
}
```

The method greeting() can be accessed by all the users, using the class Hello.

@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. Following code demonstrates the usage of @DenyAll annotation.

```
public class Hello {
    @DenyAll
    public void getUserData() {
        ....
    }
}
```

The usage of getUserData() method is denied for all the users.

15.6.2 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.

→ **Accessing an Enterprise Bean Caller's Security Context:**

Defining the security mechanisms declaratively is a preferred way of defining the security mechanisms. 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:

- **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 a certain role assigned 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.

Following code demonstrates the usage of isCallerRole() method:

```
@Resource Session Context x
if(x.isCallerRole(admin)==true) {
    System.out.println("Admin rights assigned");
}
else{
    System.out.println(" No Admin rights");
```

{}

15.6.3 Propagating a Security Identity

When an entity is allowed access to a bean class or method, the entity during the program flow may in turn access other methods in the application. The security identity for the second method call or second level of access can be defined according to 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, configure a run-as identity 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.

`@RunAs` annotation can be used to configure the ‘run as’ identity or the propagated identity of an entity.

15.7 Securing Web Applications

Web applications are accessed using Web browser and contain resources which are accessible by many users. These resources traverse over unprotected on open networks. Therefore, appropriate security mechanisms should be defined for Web applications.

Figure 15.2 shows the layered architecture of Web applications.

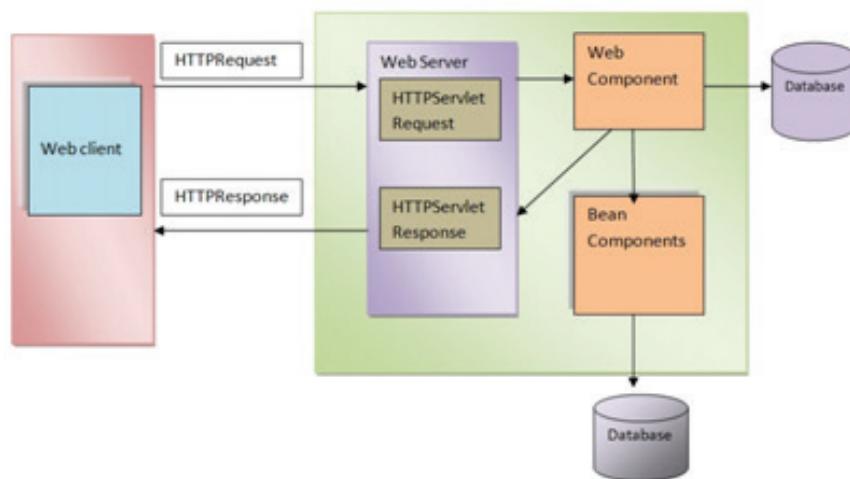


Figure 15.2: Layered Architecture of Web Applications

Figure 15.2 shows the information flow among different components in Web applications. Similar to enterprise applications, Web application security is also implemented both declaratively and

programmatically. Apart from these two techniques, Web application security is also implemented as message security. Following is the brief description about the implementation of these mechanisms:

- **Declarative security** - Declarative security mechanisms declare the requirements through annotations in the class file and deployment descriptors of the application. The values specified in the deployment descriptor override the values in the annotations. The deployer of the application defines the security policy of the application based on the annotations and deployment descriptors.
- **Programmatic security** – Declarative security may not be sufficient if the application has to implement conditional login and access resources based on some conditions which arise during the program flow of the application.
HttpServletRequest interface provides methods such as authenticate, login, and logout which can be used to implement security mechanisms.
- **Message Security** – Message security is implemented through security features such as digital signatures and encryption, which are transmitted as a part of the message in the SOAP message header. Message security is not part of Java EE, it is implemented through an API and works in the application layer.

15.7.1 Specifying Security Constraints

A security constraint is defined in the deployment descriptor to define access privileges for a collection of resources.

These constraints are defined in the deployment descriptor using security-constraint element. If the application uses an authentication mechanism other than the basic one, then it has to be defined through the security-constraint element in the deployment descriptor.

The security-constraint element can have the following sub-elements:

- **web-resource-collection** – This element comprises a list of URL patterns which represent the resources of the application and HTTP operations to be performed on these resources which need to be constrained.
- **auth-constraint** – This element is used to define the authorization, where the names and roles of the users who are authorized to perform operations on the resource are defined.
- **user-data-constraint** – This element is used to determine the security mechanism that can be used to define how user data is protected during transmission over the network.

If the Web application uses servlets, @HttpConstraint and @HttpMethodConstraint annotations can be used within the @ServletSecurity annotation to define the security constraint.

→ Specifying a Web Resource Collection

In order to specify a Web resource collection to be protected, following are the sub-elements used along with the element web-resource-collection.

- **web-resource-name** – This is an optional element which specifies the name used for the Web resource.
- **url-pattern** – This element represents the list of URLs that require protection in the application. A configuration is required in the deployment descriptor only when there is an access constraint to be defined.
- **http-method** – This element defines the http methods that are supposed to implement the protection mechanism on the resource. If there are no HTTP methods mentioned with this element, it implies that all the HTTP methods are executed under the security constraints.
- **http-method omission** – This element specifies the HTTP methods which can be exempted from the protection mechanism.

→ Specifying the Authorization Constraint

The auth-constraint element has a sub element role-name which is used to specify the application roles that are authorized to access certain application resources.

This constraint is specified when access to a certain resource requires authentication. The roles authenticated to access the resource are specified through the role-name element. If there is an authorization constraint and none of the role names are specified along with the authorization constraint, then the resource cannot be accessed by any of the entities in the application. Role names are case sensitive.

→ Specifying a Secure Connection

The user-data-constraint has a transport-guarantee sub element. This element is used for specifying the security constraints for the transport layer connection. There are three levels of transport layer guarantees which are specified as values for transport-guarantee element. They are: CONFIDENTIAL, INTEGRAL, and NONE.

The value CONFIDENTIAL implies that the data in the message is not revealed to any other entity on the network.

The value INTEGRAL implies that the data does not change during the network transit.

The value NONE implies that the data can be transmitted over an unprotected connection.

The user-data-constraint element is used along with the basic and form-based user authentication.

Code Snippet 1 shows definition of an authorization constraint.

Code Snippet 1:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Results</web-resource-name>
        <url-pattern>/University/Results/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>Professor</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>
```

The code given in Code Snippet 1 defines access to the resource Results. The access of this resource is only allowed for the users with the role Professor.

15.7.2 Specifying Authentication Mechanisms

The authentication mechanism defines the access to the resources. Java EE platform supports the following authentication mechanisms:

- Basic authentication
- Form-based authentication
- Digest authentication
- Client authentication
- Mutual authentication

Basic authentication mechanism sends usernames and passwords over the Internet as base64-encoded text and the target server is not authenticated, that is, the data may reach an unintended server. In case of form-based authentication, the username and password information is sent in plain text and the target server is not authenticated, that is, the message can reach an unintended destination. Basic authentication and form-based authentication mechanisms are not considered very strong authentication mechanisms.

→ Basic Authentication Mechanism

Following is the sequence of operations in a basic authentication mechanism:

1. The client tries to access a protected resource of the application.
2. The security mechanism prompts for username and password through a dialog box.
3. The client responds with the username and password in the dialog box.
4. The server verifies the username and password. If the credentials provided have access to the requested resource then the client is given access.

→ Form-based Authentication Mechanism

In case of form-based authentication, the client trying to access a protected resource is redirected to a login page. The login page has an elaborate security mechanism other than the username and password credentials.

Following is the sequence of operations involved in form-based authentication:

1. The client tries to access a protected resource.
2. The server redirects the client to a login page/form requesting for the client details.
3. Based on the details filled in the login page the client is granted access to the resource.

When a form-based login is created, then the application should maintain the sessions of the communication through cookies or through SSL session information.

→ Digest Authentication

The authentication mechanism for the digest authentication also follows the same set of steps as in case of basic authentication. However, the exchange of username and password is not as plain text but as a cryptographic hash. The client requesting the protected resource sends the password as a one-way cryptographic hash and additional data. The receiver end should also possess the password in plain text format. The receiver will generate the hash of the user password and compare it with the received password hash.

→ Client Authentication

In client authentication, the client is authenticated by the server through the public key of the client. In case of public key cryptography, every entity in the network has a pair of public and private keys. The data encrypted by the public key can be decrypted by the private key. The public keys of all the entities are stored in a public repository which can be accessed by all the entities in the network. The server accesses the public key of the client from the repository and encrypts the message with it. Only the recipient who possesses the private key can decrypt the message and read it.

→ **Mutual Authentication**

There are two variants of mutual authentication as follows:

- Certificate based authentication
- Username-password based authentication

When the client accesses a protected resource, the Web server presents its certificate to the client. The client verifies the certificate received and if the certificate from the server is found to be authentic, then the client sends its credentials. If the client's credentials are verified by the server, the server gives access to the protected resource.

For mutual authentication with username/password credentials, the process is similar as in case of certificate, except the fact that instead of certificate, username and password credentials of both the client and server are exchanged.

15.8 Using Programmatic Security with Web Applications

When security of the Web applications is implemented programmatically, it involves the following tasks to be accomplished through code:

- Authenticating users programmatically
- Checking caller identity programmatically
- **Authenticating Users Programmatically**

In order to authenticate users programmatically, HttpServletRequest interface provides the following methods:

- **authenticate** – This method invokes the authentication of the client who is trying to access protected resource. This method provides a dialog box which prompts for username and password.
- **login** – This method collects the username and password credentials from the client and authenticates the client by verifying the username against the database of known users.
- **logout** – This method is used to reset the caller identity of a client request.

→ **Checking Caller Identity Programmatically**

Following are the methods provided in Servlets3.0 to programmatically check the identity of the client who is trying to access the resource:

- **getRemoteUser** – This method fetches the user name who is trying to access the protected resource. The method returns the user name of the client.

- **isUserInRole** – This method returns the role associated with the username of a client. If the user name is associated with a role then the method returns the role name otherwise the method returns false.
- **getUserPrincipal** - This method returns the principal name associated with the current user name. This method returns a javax.security.Principal object.

15.9 Configuring Declarative Security for Web Applications in NetBeans7.4

A Web application can be configured with different security options. The developer can give access to certain Web pages to a specific role of the application. In this example, two types of users have been defined – admin and user. The Web server (GlassFish) is configured with appropriate access rights.

To begin with, create a Web application named SecureWebApplication in the NetBeans IDE. Select JSF from the frameworks while creating the application.

Once the Web application is created, define two different security domains in the Web Pages folder of the application by creating new folders in the Web pages folder of the application. (These are normal folders and not Java Package). The hierarchy of the folders in the application is as shown in figure 15.3.

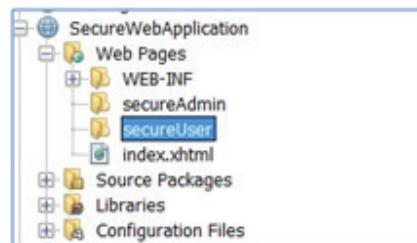


Figure 15.3: Directory Structure of Web Application

Note: To create the folders, right-click Web Pages and select New → Other → Other → Folder.

Create Web pages within the secureAdmin and secureUser directories. These two directories define the domain of the admin and user roles. The Web pages created are named secureAdminPage.html and secureUserPage.html respectively.

Code Snippet 2 shows the html code of the secureAdminPage.

Code Snippet 2:

```

<!DOCTYPE html>
<html>
  <head>
    <title>Admin page</title>
    <meta charset="UTF-8">
  
```

```
<meta name="viewport" content="width=device-width">
</head>
<body>
<h1>Secure admin page</h1>
</body>
</html>
```

The code is the default code generated by the NetBeans IDE, except the title of the Web page and the text added in the body section of the HTML.

Code Snippet 3 shows the HTML code of the secureUserPage.

Code Snippet 3:

```
<html>
<head>
<title>Secure User page</title>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width">
</head>
<body>
<h1>Secure user page</h1>
</body>
</html>
```

Add the code given in Code Snippet 4 to the body section of index.xhtml

Code Snippet 4:

```
<p>Access to secure Admin page <a href="secureAdmin/secureAdminPage.html">here!</a></p>
<p>Access to secure User page <a href="secureUser/secureUserPage.html">here!</a></p>
```

The roles of the application should be defined on the Web server. In order to define the users and roles on the application access the Domain Admin Console as shown in figure 15.4.

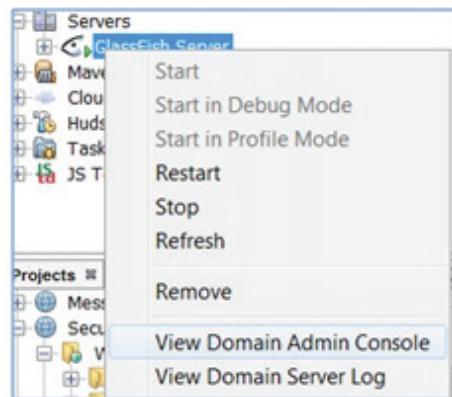


Figure 15.4: Accessing the Domain Admin Console

In order to define the roles for the intended files in the application, follow the given path.

Configurations → server-config → Security → Realms → file. ‘File’ is selected as security is defined at the file level here in the application. The selection appears as shown in figure 15.5.



Figure 15.5: Selecting the Component for Security Configuration

Click the Manage Users button in the server configuration screen. This will lead to an interface where users for the application can be defined as shown in figure 15.6.

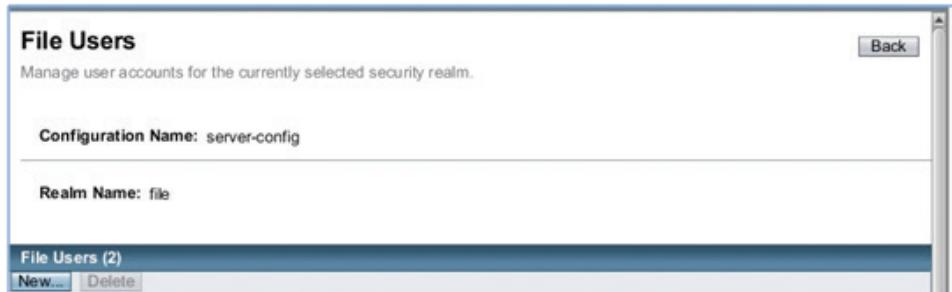


Figure 15.6: Adding New Users

Click New. The New File Realm User screen is displayed as shown in figure 15.7. Here, a new user and the validation credentials can be defined.

The screenshot shows a dialog box titled 'New File Realm User'. It says 'Create new user accounts for the currently selected security realm.' An asterisk indicates required fields. The form contains fields for 'Configuration Name: server-config', 'Realm Name: file', 'User ID: *' (with a note about character restrictions), 'Group List:' (with a note about separator), 'New Password:', and 'Confirm New Password:'. There are 'OK' and 'Cancel' buttons at the top right.

Figure 15.7: Defining User Credentials

Create two users, admin and user (specify the names in the User ID box) as per the application requirement. Provide password as admin123 for admin and user123 for user.

Figure 15.8 shows creation of the admin user.

New File Realm User

Create new user accounts for the currently selected security realm.

* Indicates required field

Configuration Name: server-config

Realm Name: file

User ID: * admin
Name can be up to 255 characters, must contain only letters, digits, underscore, dash, or dot characters

Group List:

New Password:
Confirm New Password:

OK Cancel

Figure 15.8: Creating the admin User

Once the users are created they will appear in the user table as shown in figure 15.9.

File Users

Manage user accounts for the currently selected security realm.

Configuration Name: server-config

Realm Name: file

File Users (2)

Select	User ID	Group List:
<input type="checkbox"/>	admin	[empty]
<input type="checkbox"/>	user	[empty]

New... Delete Back

Figure 15.9: Users in the File Users Table

The File Users table contains all the users defined on the Web server.

Once the users are defined, the developer has to define the authentication mechanism to login and access the resources. This is done in the application deployment descriptor web.xml.

Click the Security tab in the deployment descriptor as shown in figure 15.10.

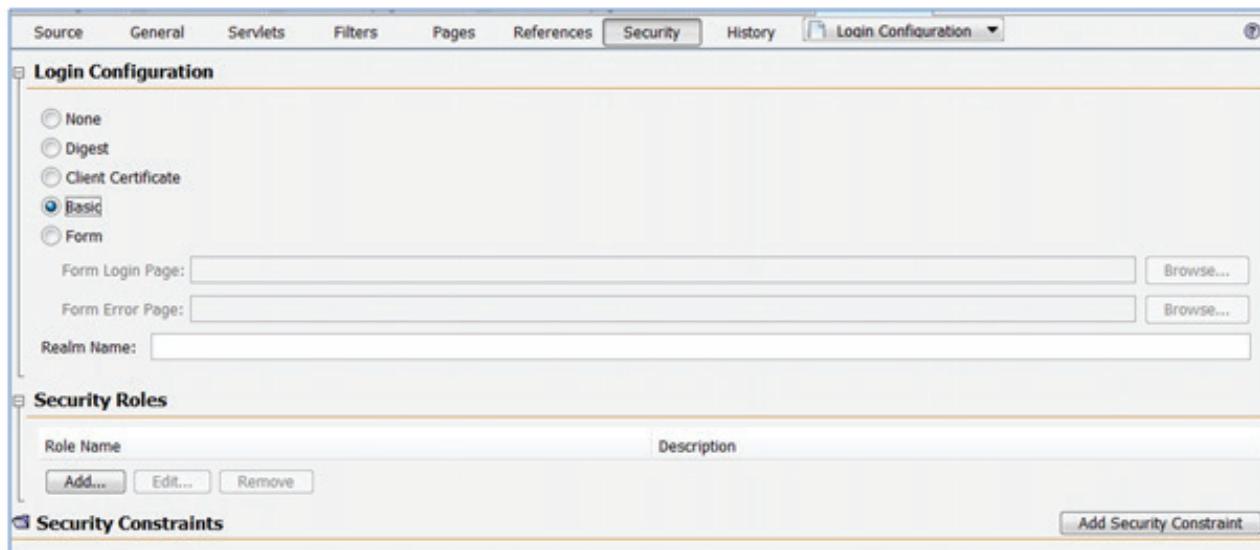


Figure 15.10: Selecting the Login Mechanism

Define the login mechanism in the Login Configuration section; here the Basic login mechanism is selected.

Add security roles to the application by clicking Add in the Security Roles section. It will lead to the screen as shown in figure 15.11 where the security roles can be added to the application.

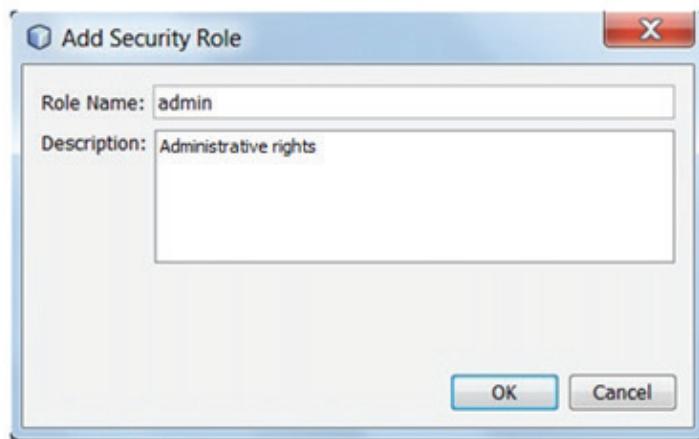


Figure 15.11: Adding Security Roles

Figure 15.12 shows the state of the security roles table after the roles are added with corresponding description.

Security Roles	
Role Name	Description
admin	Administrative rights
user	User access rights
Add...	Edit...
	Remove

Figure 15.12: Roles Added to the Deployment Descriptor

Once the roles are added, define the security constraints in the deployment descriptor by clicking Add Security Constraint. It leads to a screen as shown in figure 15.13.

The screenshot shows the 'UserConstraint' configuration dialog. It includes a 'Display Name' field set to 'UserConstraint', a 'Web Resource Collection' section, and two constraint sections: 'Enable Authentication Constraint' and 'Enable User Data Constraint'. Each constraint section has a 'Description' field and a 'Role Name(s)' field. Below these is a 'Transport Guarantee' dropdown set to 'NONE'.

Figure 15.13: Defining Security Constraint

Specify UserConstraint in the Display Name box and click Add. This will lead to screen as shown in figure 15.14.

The screenshot shows the 'Add Web Resource' dialog. It has fields for 'Resource Name' (set to 'user'), 'Description', and 'URL Pattern(s)' (set to '/secureUser/*'). Below these is a note about separating patterns with commas. Under 'HTTP Method(s)', the 'All HTTP Methods' radio button is selected, while 'Selected HTTP Methods' is unselected. A list of methods (GET, POST, HEAD, PUT, OPTIONS, TRACE, DELETE) is shown with checkboxes, all of which are unchecked.

Figure 15.14: Mapping the User to the Access Domain

In this wizard, map the user to the folder which the user can access. The role ‘user’ can access all the files in the folder secureUser according to the given URL pattern.

Click OK. The user to resource mapping will be displayed in the Web Resource Collection section.

Similarly, add the AdminConstraint for admin with respect to the secureAdmin folder.

Select the checkbox ‘Enable Authentication Constraint’ while defining the UserConstraint as well as the AdminConstraint.

Click Edit to set the Role Name as user for UserConstraint and admin for AdminConstraint.

Figure 15.15 shows the final state of the User Constraint after all the values are configured.

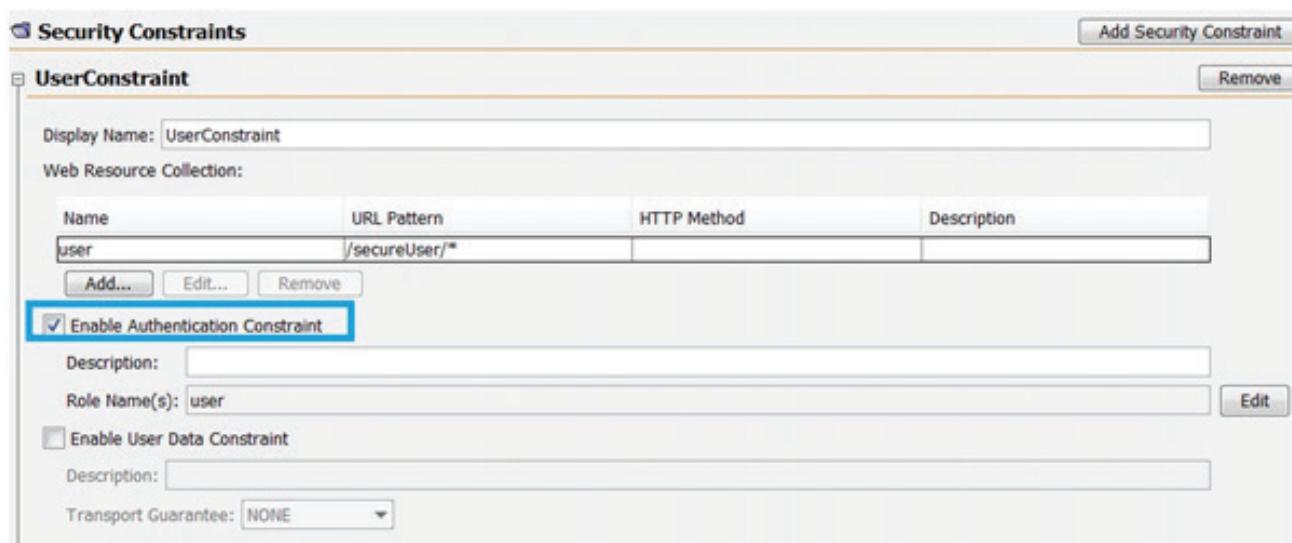


Figure 15.15: User Constraint Created

It is essential to select ‘Enable Authentication Constraint’ while defining the Security Constraint. This choice will prompt for the username and password credentials while accessing the resource. The access to these resources is allowed only for the value in the ‘Role Name(s)’ field.

Once the application's deployment descriptor is configured, configure the Web server deployment descriptor. In case of GlassFish server it is GlassFish-web.xml. If the deployment descriptor is not already present in the application, right-click the project and select New → Other → GlassFish → GlassFish Descriptor as shown in figure 15.16.

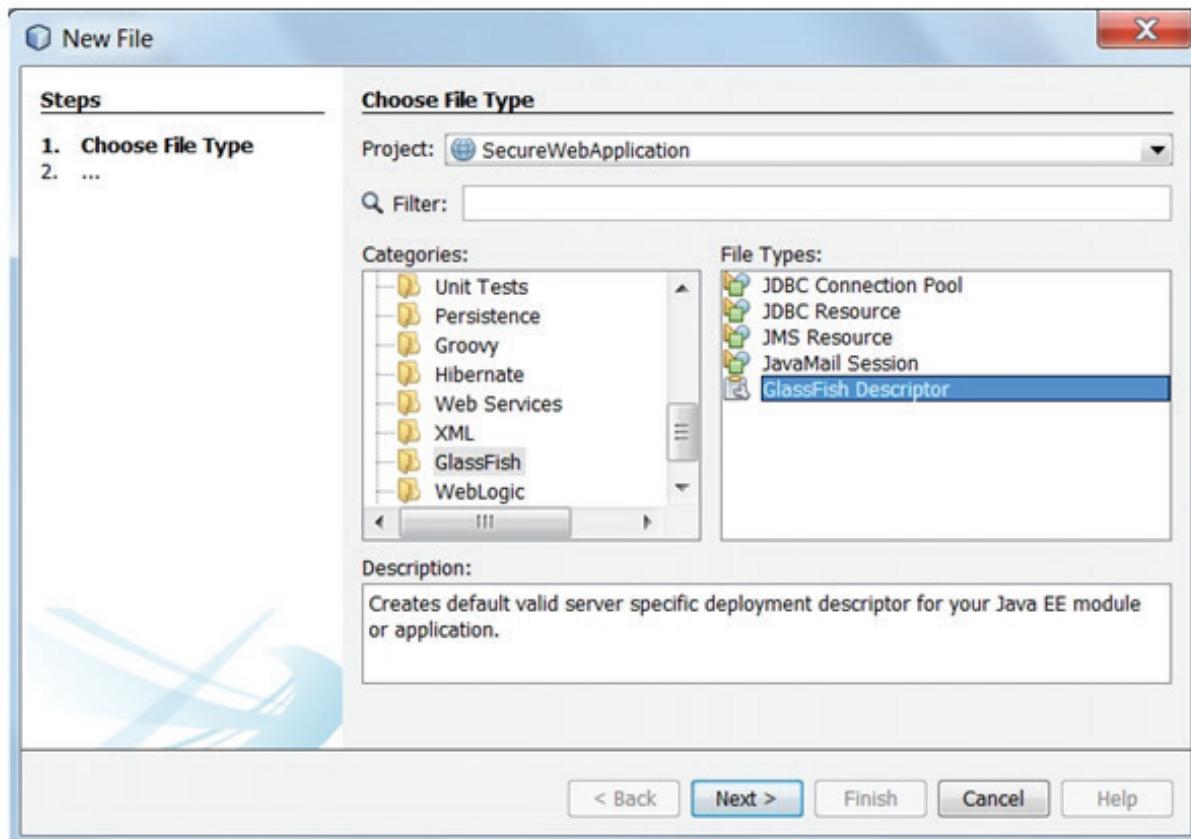


Figure 15.16: Adding GlassFish Deployment Descriptor

Once the Web server deployment descriptor is created, open it and click the Security tab. This will lead to the screen as shown in figure 15.17. The roles created earlier in the server's Admin Console are seen listed here.



Figure 15.17: Mapping the Security Roles

On expanding the security roles, the screen shown in figure 15.18 appears. Click Add Principal to add users to the role. For admin role, add 'admin' as the Principal. Similarly, add 'user' as Principal for the user role.

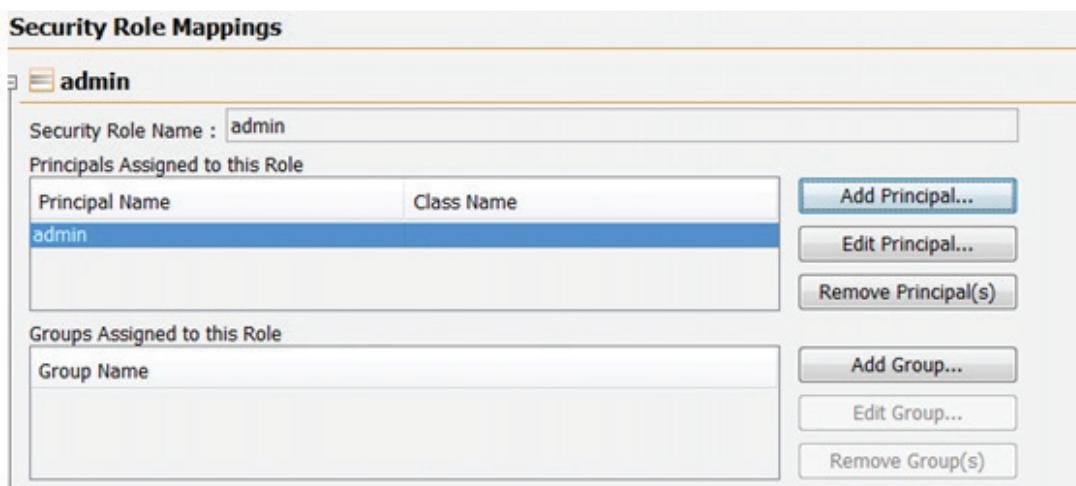


Figure 15.18: Adding Principals

After all the configurations are completed, deploy and run the application. The application execution will lead to the index page as shown in figure 15.19.

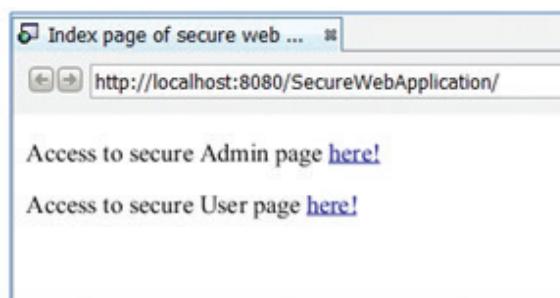


Figure 15.19: Executing the Secure Web Application

Click the hyperlink. It will prompt for user name and password as shown in figure 15.20.

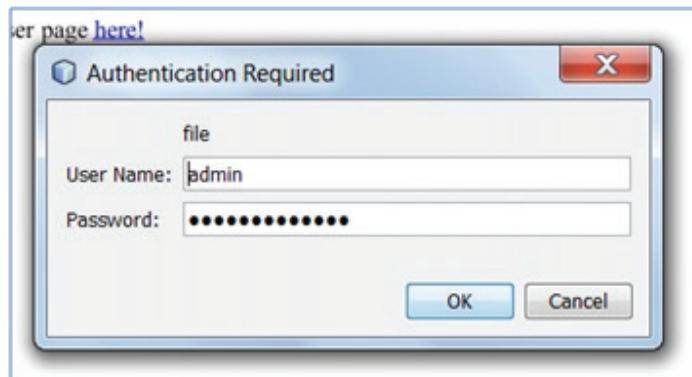


Figure 15.20: Prompting for User Name and Password

On providing the appropriate credentials, it will lead to the Web page as shown in figure 15.21.

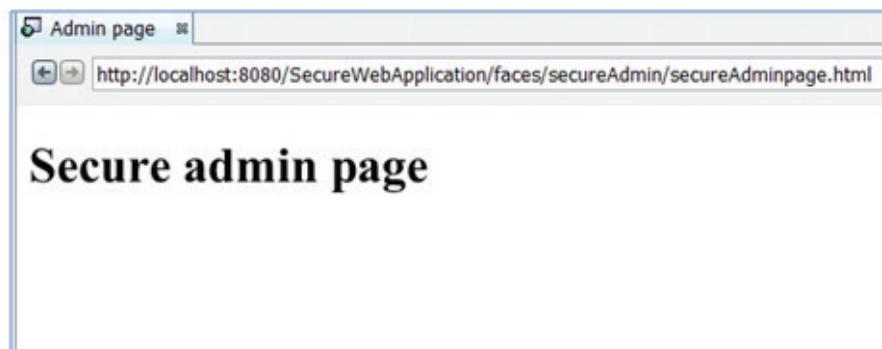


Figure 15.21: Accessing the Web Page after Authentication

Note: The application may not behave as expected at the first execution. It may give a '403:Forbidden page' error. The roles and respective credentials have to be properly deployed onto the application/Web server. This may require refreshing/restarting the Web server.

15.10 Check Your Progress

1. Which of the following is used for defining the security mechanism programmatically for Web applications?

(A)	Deployment descriptor	(C)	HttpServletRequest interface
(B)	Annotations	(D)	None of these

2. Which of the following authentication mechanisms uses public key cryptography?

(A)	Basic authentication	(C)	Digest authentication
(B)	Form authentication	(D)	Client authentication

3. Which of the following can be a sub element of security-constraint element?

(A)	web-resource-constraint	(C)	user-data constraint
(B)	auth-constraint	(D)	All of these

4. Which of the following annotations are not used for security mechanisms in Java EE applications?

(A)	@Stateless	(C)	@RolesAllowed
(B)	@DenyAll	(D)	None of these

5. Identify the method which is used to define the security mechanism programmatically in Web applications.

(A)	authenticate	(C)	getUserName
(B)	logout	(D)	All of these

15.10.1 Answers

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

Summary

- Security mechanisms in both enterprise and Web applications are specified both declaratively and programmatically.
- Security mechanisms are declaratively specified through annotations and deployment descriptors.
- Programmatically security mechanisms are specified through Java security APIs such as JAAS.
- Security mechanisms in Web applications are implemented at three levels: application level security, transport level security, and message level security.
- Methods of HttpServletRequest interface are used to programmatically define security mechanisms for Web applications.
- The security roles and mapping can be done on the application deployment descriptor and Web server deployment descriptor.

**Get
WORD WISE**



Visit
the **Glossary** section
@

www.onlinevarsity.com