18CS503

Hall Ticket Number:

III/IV B.Tech (Regular) DEGREE EXAMINATION

FEBRUARY, 2021 — **Computer Science and Engineering**

First Semester — **Enterprise Programming**

Time: Three Hours — **Maximum :** 50 Marks

Answer Question No.1 compulsorily. (1X10 = 10 Marks)

Answer ONE question from each unit. (4X10=40 Marks)

1. Answer all questions (1X10=10 Marks)

a Differentiate between PUT and POST operations in REST API

POST means "create new" as in "Here is the input for creating a user, create it for me".

PUT means "insert, replace if already exists"

b Name the types of JDBC drivers?

Type 1: JDBC-ODBC bridge.

Type 2: partial Java driver.

Type 3: pure Java driver for database middleware.

Type 4: pure Java driver for direct-to-database.

Type 5: highly-functional drivers with superior performance

c Write the syntax of JSP expression tag.

<%= statement %>

d List any two implicit objects in JSP?

Out, request, response, config, application, session, pageContext, page, exception

e Write the syntax for <c:if> tag.

<c:if test="${condition}">

...

..

</c:if>

f Name the annotation with which url mapping of a servlet can be done.

@WebServlet

g What does JSTL stand for?

JavaServer Pages Standard Tag Library

h Name two types of Session beans

stateless and stateful

i What is the protocol used by web sockets?

Ws-wss/ http-upgrade

j What is the header sent by a client socket in the initial handshake?

Ws-wss/ http-upgrade

UNIT – I

2.a Name the different APIs for the development of JAVA EE Applications and explain about Java EE platform architecture with a neat sketch. M

Java EE Architecture

The Java EE architecture is a component-based and platform-independent that makes Java EE applications easy to write because business logic is organized into reusable components and the Java EE server provides underlying services in the form of a container for every component type.

Containers and Services

Containers are the place holders for the components installed during deployment and are the interface between a component and the low-level platform-specific functionality that supports the component. Before a web, enterprise bean, or application client component can be executed, it must be assembled into a Java EE application and deployed into its container.

The container also manages non-configurable services such as enterprise bean and servlet life cycles, database connection resource pooling, data persistence, and access to the Java EE platform APIs described in Java EE APIs.

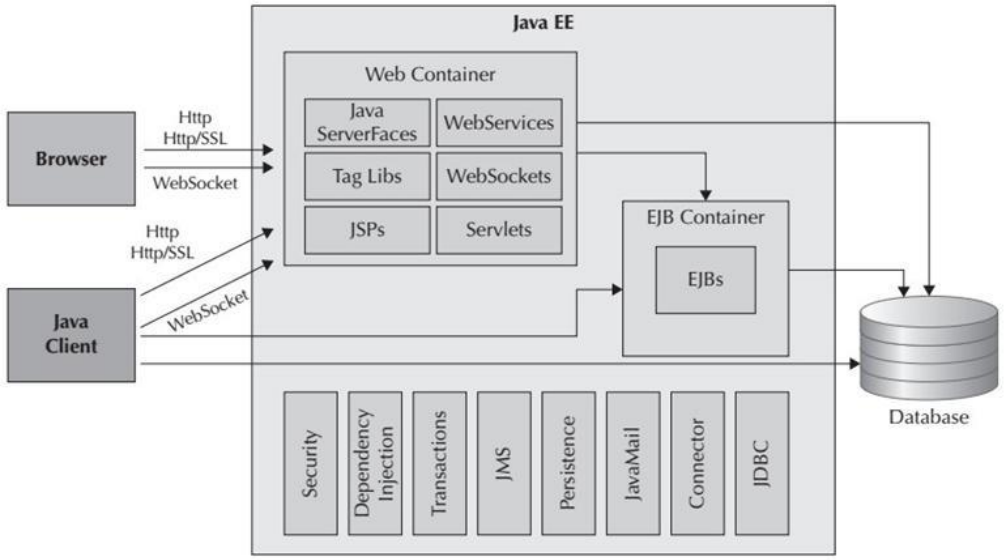


FIGURE 1-1. *Architecture of the Java EE platform*

Container Types

The deployment process installs Java EE application components in the following types of Java EE containers. The Java EE components and container addressed in the figure1below.

An Enterprise JavaBeans (EJB) container manages the execution of all enterprise beans for one Java EE application. Enterprise beans and their container run on the Java EE server.

A web container manages the execution of all JSP page and servlet components for one Java EE application. Web components and their container run on the Java EE server.

An application client container manages the execution of all application client components for one Java EE application. Application clients and their container run on the client machine.

An applet container is the web browser and Java Plug-in combination running on the client machine.

java EE APIs

The Java 2 Platform, Standard Edition (J2SE) SDK is required to run the Java EE SDK and provides core APIs for writing Java EE components, core development tools, and the Java virtual machine1. The Java EE SDK provides the following APIs to be used in Java EE applications.

Enterprise JavaBeans Technology 2.0

Enterprise bean as a building block that can be used alone or with other enterprise beans to execute business logic on the Java EE server. An enterprise bean is a body of code with fields and methods to implement modules of business logic.

JDBC 2.0 API

The JDBC API lets you invoke SQL commands from Java programming language methods.

Java Servlet Technology 2.3

Java Servlet technology lets you define HTTP-specific servlet classes. A servlet class extends the capabilities of servers that host applications accessed by way of a request-response programming model.

JavaServer Pages (JSP)

JSP pages technology lets you put combine snippets of Java programming language code with static markup in a text-based document.

Java Message Service (JMS) 1.0

The JMS API is a messaging standard that allows Java EE application components to create, send, receive, and read messages. It enables distributed communication that is loosely coupled, reliable, and asynchronous.

Java Transaction API (JTA) 1.0

The JTA API provides a standard demarcation interface for demarcating transactions. The Java EE architecture provides a default auto commit to handle transaction commits and roll backs.

JavaMail

Many Internet applications need to send email notifications so the Java EE platform includes the JavaMail API with a JavaMail service provider that application components can use to send Internet mail.

2.b Code a servlet that takes Regd.No. and the Name of a student and inserts in the ***rolls*** table of ***student*** database. M

Index.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>

<h1>Add New student</h1>
<form action="InsertServlet" method="post">
<table>
<tr><td>Name:</td><td><input type="text" name="name"/></td></tr>

<tr><td>Registered
No:</td><td><input type="text" name="regdno"/></td></tr>
</td></tr>
<tr><td colspan="2"><input type="submit" value="Save Employee"/></td></tr>

</table>
</form>
```

```
<br/>
</body>
</html>
```

InsertServlet.java

```
import java.io.IOException;
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("/InsertServlet")
public class SaveServlet extends HttpServlet {
protected void doPost(HttpServletRequest request, HttpServletResponse res ponse) throws
ServletException, IOException {
String dbURL = "jdbc:mysql://localhost:3306/student"; String username = "root";
String password = "secret"; try
{
Connection conn = DriverManager.getConnection(dbURL, username, password);
if (conn != null)
{
System.out.println("Connected");
}
}
catch (SQLException ex)

{
ex.printStackTrace();
}
response.setContentType("text/html"); PrintWriter out=response.getWriter();

String rno=request.getParameter("regdno"); String name=request.getParameter("name");
String sql = "INSERT INTO Users (regno, name) VALUES (?, ?, ?, ?)";
PreparedStatement statement = conn.prepareStatement(sql); statement.setString(1, rno);
statement.setString(2, name);
if(status>0){
out.print("<p>Record inserted successfully!</p>");
request.getRequestDispatcher("index.html").include(request, response);

}else{
out.println("Sorry! unable to save record");
}

out.close();
}
}
```

(OR)

3.a Explain the classes and interfaces needed to code a JDBC application. M

JDBC API is available in two packages java.sql, core API and javax.sql JDBC optional packages. Following are the important classes and interfaces of JDBC.

Class/interface	Description
DriverManager	This class manages the JDBC drivers. You need to register your drivers to this. It provides methods such as registerDriver() and getConnection().
Driver	This interface is the Base interface for every driver class i.e. If you want to create a JDBC Driver of your own you need to implement this interface. If you load a Driver class (implementation of this interface), it will create an instance of itself and register with the driver manager.
Statement	This interface represents a static SQL statement. Using the Statement object and its methods, you can execute an SQL statement and get the results of it. It provides methods such as execute(), executeBatch(), executeUpdate() etc. To execute the statements.
PreparedStatement	This represents a precompiled SQL statement. An SQL statement is compiled and stored in a prepared statement and you can later execute this multiple times. You can get an object of this interface using the method of the Connection interface named prepareStatement(). This provides methods such as executeQuery(), executeUpdate(), and execute() to execute the prepared statements and getXXX(), setXXX() (where XXX is the datatypes such as long int float etc..) methods to set and get the values of the bind variables of the prepared statement.
CallableStatement	Using an object of this interface you can execute the stored procedures. This returns single or multiple results. It will accept input parameters too. You can create a CallableStatement using the prepareCall() method of the Connection interface. Just like Prepared statement, this will also provide setXXX() and getXXX() methods to pass the input parameters and to get the output parameters of the procedures.
Connection	This interface represents the connection with a specific database. SQL statements are executed in the context of a connection. This interface provides methods such as close(), commit(), rollback(), createStatement(), prepareCall(),

	prepareStatement(), setAutoCommit() setSavepoint() etc.
ResultSet	This interface represents the database result set, a table which is generated by executing statements. This interface provides getter and update methods to retrieve and update its contents respectively.
ResultSetMetaData	This interface is used to get the information about the result set such as, number of columns, name of the column, data type of the column, schema of the result set, table name, etc It provides methods such as getColumnCount(), getColumnName(), getColumnType(), getTableName(), getSchemaName() etc.

3.b Design a JDBC program that connects to a ***dept*** table of ***emp*** database and prints the details of all the departments. M

```
public class DisplayTable
{
public static void main(String[] args) throws Exception
{
String driver="org.apache.derby.jdbc.ClientDriver"; String
url="jdbc:derby://localhost:1527/emp"; String user="ram";
String psw="ram"; Class.forName(driver);
Connection con=DriverManager.getConnection(url,user,psw); Statement
st=con.createStatement();
String sql = "select * from dept"; ResultSet rs = stmt.executeQuery(sql);
while(rs.next()){
//Retrieve by column name int id = rs.getInt("id"); int age = rs.getInt("age");
String name = rs.getString("name");
//Display values System.out.print("ID: " + id); System.out.print(", Age: " + age);
System.out.print(", name: " + name); System.out.println();
}
rs.close();
}
}
```

UNIT – II

4.a Design a JSP to find the factorial of a given number and sends it as reponse. M

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<html>
<head>
<title>Factorial</title>
</head>
<body>
<center>

<form>
<b>Value of n</b><input type="text" name="n"><br/><br/>
```

```
<input type="submit" name="sbtn" value="Submit">
</form>
<%
if(request.getParameter("sbtn")!=null){
int n = Integer.parseInt(request.getParameter("n")); int fact=1,i;
for(i=2;i<n;i++) fact*=i;
out.println("<h2>The factorial is :" + fact +"</h2>");
}
%>
<%
}
%>
<%-- using JSTLand Expression Language--%>
<c:if test="${param.sbtn != null}">
<c:set var="n" value="${param.n}"/>
<c:set var="fact" value="1"/>
<c:forEach var="i" begin="2" end="${n}">
<c:set var="fact" value="${fact*i}"/>
</c:forEach>
<h2>The factorial is ${fact}</h2>
</c:if>
</center>
</body>
</html>
```

4.b Explain all the implicit objects available in a JSP. M

These Objects are the Java objects that the JSP Container makes available to the developers in each page and the developer can call them directly without being explicitly declared. JSP Implicit Objects are also called pre-defined variables.

request :This is the HttpServletRequest object associated with the request.
<%=request.getLocale().toString() %>
response : This is the HttpServletResponse object associated with the response to the client.
<%
PrintWriter pw = response.getWriter(); pw.print("Response writer example.");
%>
out : This is the PrintWriter object used to send output to the client.
<% out.print("This is an example");%>
session : This is the HttpSession object associated with the request.
<%=session.getCreationTime()%>
application :This is the ServletContext object associated with the application context.
<%=application.getServerInfo()%>
config : This is the ServletConfig object associated with the page.
<%config.getServletName()%>
pageContext : This encapsulates use of server-specific features like higher performance JspWriters.
<%pageContext.setAttribute("regdno", "Y10500");%>
page : This is simply a synonym for this, and is used to call the methods defined by the translated servlet class.
<%=page.getClass().getName()%>
exception : The Exception object allows the exception data to be accessed by designated JSP.
<%= exception.getMessage() %>

(OR)

5.a Code a JSF application which registers a user with a name (required) and an email id (to be validated). Design the backing bean with all the needed methods. M

1. User form

```
<h:head>
<title>JSF Application</title>
</h:head>
<h:body>
<center>
<h1>Bapatla Engineering College</h1>
<h:form>
<h:outputLabel>Name :</h:outputLabel>
<h:inputText value="#{student.name}">
<f:validateRequired/>
</h:inputText>
<h:outputLabel>Email :</h:outputLabel>
<h:inputText value="#{student.email}">
<f:validator validatorId="emailValidator"/>
</h:inputText>
<!--
<h:commandButton value="Submit" action="display"/>
-->
<h:commandButton value="Submit" action="#{student.saveStudentRecord()}"/>
<h:dataTable value="#{student.studentList}" var="s" border="2"
rendered="#{student.studentList.size()>0}">
<h:column>
<f:facet name="header">Name</f:facet>
<h:outputText value="#{s.name}"/>
</h:column>
<h:column>
<f:facet name="header">Email</f:facet>
<h:outputText value="#{s.email}"/>
</h:column>
</h:dataTable>
</h:form>
</center>


</h:body>
</html>
```

Display Form

```
<h:head>
<title>student Info</title>
</h:head>
<h:body>
<center>
<h2><h:outputText value="The name is ${student.name}"/></h2>
<h2><h:outputText value="The name is ${student.email}"/></h2>
</center>
```

```
</h:body>
</html>
```

Managed Bean:

```
package jsf;


import java.util.ArrayList;
 import java.util.Date;
import javax.enterprise.context.Dependent;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;
import javax.faces.event.AjaxBehaviorEvent;
@ManagedBean
@RequestScoped
public class Student
 {
private String name; private Date dob; private String country; private String email;
private String phone;
private static ArrayList<Student> studentList = new ArrayList<Student>();
public ArrayList<Student> getStudentList()
{
        return studentList;
}
public String getName()
{
         return name;
}


public void setName(String name)
{
         this.name = name;
}


public String getEmail()
{
return email;
}


public void setEmail(String email)
{
        this.email = email;
}


public Student()
{

}
```

```
public Student(String name, Date dob, String country, String email, String phone)
{
this.name = name;
this.email = email;
}

public void saveStudentRecord()
{
 studentList.add(this);
}
}
```

Email Validator

```
package jsf;
import javax.faces.application.FacesMessage; import javax.faces.component.UIComponent; import javax.faces.context.FacesContext; import javax.faces.validator.FacesValidator; import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;
@FacesValidator("emailValidator")
public class EmailValidator implements Validator{
@Override
public void validate(FacesContext fc, UIComponent comp, Object val)throws ValidatorException{ String email = (String)val;
if(!email.matches("[A-Za-z][A-Za-z0-9]+@[A-Za-z]+\\.[A-Za-z]+")){ FacesMessage msg = new FacesMessage("Input a valid email"); throw new ValidatorException(msg);
}
}
}
```

5.b Explain any three managed bean scopes. M

The Java EE managed bean scopes facilitate to manage the lifecycle of a JavaBean that you want to use in an application.

They specify when and how many instances of a JavaBean are created. javax.enterprise.context contains several scopes.

The different managed bean scopes are:

@RequestScoped: A managed bean in request scope is instantiated once for every new HTTP request/response interaction. Created as the HTTP request arrives, and destroyed as the HTTP response leaves.

@SessionScoped : A managed bean in session scope has the same lifecycle as the HttpSession. The bean is instantiated once for each new HttpSession that is created for that web application. It is destroyed when the HttpSession to which the Java EE web container is associated either times out or is invalidated.

@ApplicationScoped : The Java EE container instantiates the bean once prior to any users accessing the application and destroys it prior to shutting down the application.

@ConversationScoped : The conversation scope is a scope that is contained within the session scope. It is instantiated once for every new HttpSession. It is active during a session. It is destroyed only either when the developer explicitly ends the conversation scope, or when the HTTP session with which it is associated times out or is invalidated.

@FlowScoped : The flow scope is another managed bean scope that is contained with the session scope. The goal of the scope is for a new one to be active during a preset sequence of interactions with the web application from a given user. The boundaries of the flow scope are managed by a mechanism in JavaServer Faces called faces flows.

@Dependent : The dependent scope is a scope that says “scope depends on where it is used.” If a

managed bean is marked @Dependent, its lifecycle and cardinality are governed by the component that uses it.

@ViewScoped : The view scope is a managed bean scope that extends the request scope by starting while a particular Java Server Faces page is being executed. It remains active while the client continues to interact with that same page. The scope becomes inactive once the client navigates to a different page.

UNIT – III

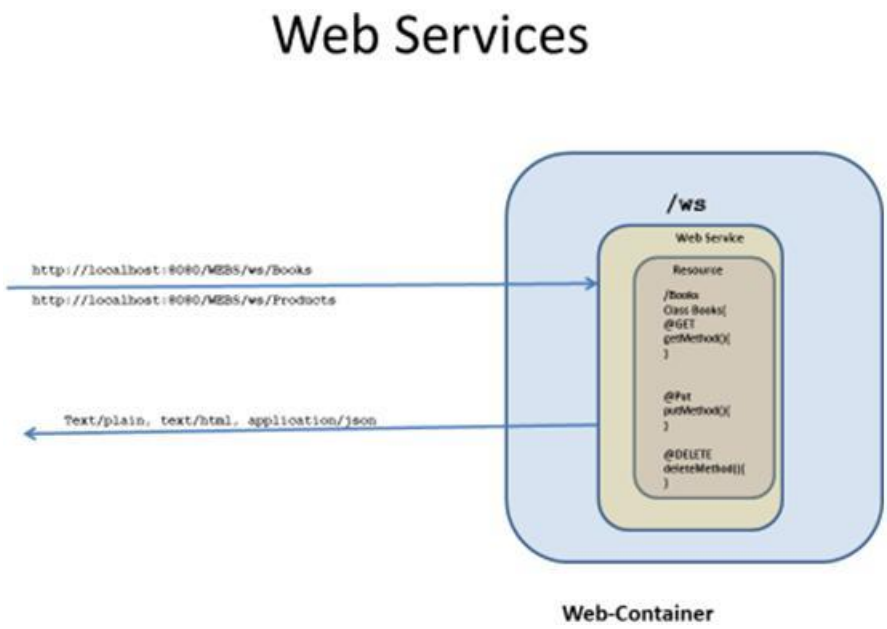
6.a Explain the web services and REST API. M

A Web application/service is the one that does not deliver a Web Page but delivers data to several kinds of client applications.

REST (Representational State Transfer) is a style of software architecture for distributed hypermedia systems such as the World Wide Web. It was introduced by Roy Fielding. REST is a collection of network architecture principles which outline how resources are defined and addressed. The key abstraction of information in REST is a resource. A resource is a conceptual mapping to an entity or set of entities. Any information that can be named can be a resource. For example, a document or image, a temporal service (e.g. "today's weather"),

The resource or collection of resources is represented with a global identifier (URI in HTTP). REST then defines different HTTP verbs- GET (single or all), PUT, POST and DELETE operations. A webservice in REST model is a class annotated with a path that becomes part of an URI. GET (single or all), PUT, POST and DELETE operations map to methods in the class. The respective annotations help in defining the methods. The following diagram shows the URIs and the class design

- http://www.bec.ac.in/books
- http://www.bec.ac.in/books/2048



JAX-RS (Java API for RESTful Web Services) is a Java programming language API. It provides support in creating web services according to REST architectural pattern. JAX- RS uses annotations to simplify the development and deployment of web service endpoints and clients. Some of the popular JAX-RS implementations available are: Jersey and RESTEasy.The annotations use the Java package javax.ws.rs. @Path specifies the relative path for a resource class or method. @GET, @PUT, @POST, @DELETE and @HEAD specify the HTTP request type of a resource. @PathParam binds the method parameter to a path segment. Two main formats of data representation are: JavaScript Object Notation (JSON) and XML. It is common to have multiple representations of the same data

6.b Code of a web service providing GET and POST operations on a collection of resources. Design the resource class. M

The class containing the methods is annotated with the path annotation.

```
@Path(“ws”)
class Library {
@GET
```

```
@Produces(MediaType.APPLICATION_JSON) public ArrayList<Product> getAll() {
return Product.getProducts();
}


@GET
@Path("{id}") @Produces(MediaType.APPLICATION_JSON)
public Product getProduct(@PathParam("id") int id) {

ArrayList<Product> products = Product.getProducts(); int i;
for(i=0;i<products.size();i++) if(products.get(i).id == id)
break;
return products.get(i);
}


@POST
@Consumes(MediaType.APPLICATION_JSON) @Produces(MediaType.APPLICATION_JSON)

Public Product addProduct(Product b)
{
Product.getProducts.add(b); Return b;
}
}
```

(OR)

7.a Explain the working model of websockets and its life cycle events. M

HTTP and WebSocket both are communication protocols used in client-server communication. HTTP is unidirectional where the client sends the request and the server sends the response. A user sends a request to the server in the form of HTTP or HTTPS protocol. After receiving a request, the server sends the response to the client. Each request establishes a new connection to the server. Each request is associated with a corresponding response. The connection is closed after the response is delivered/received. HTTP is stateless protocol runs on the top of TCP which is a connection-oriented protocol.

WebSocket is bidirectional, a full-duplex protocol that is used in the same scenario of client- server communication. The URI starts with ws:// or wss://. It is a stateful protocol. The connection between client and server will keep alive until it is terminated by either party (client or server). Then the connection is terminated from the other end.

WebSocket communication works in two parts: – A handshake and – the data transfer. When a WebSocket object is created by client, a handshake is exchanged between client and server. Client first sends a HTTP GET Upgrade request to the server. Client sends “Upgrade: websocket” in the request header which indicates a protocol upgrade request. Server then upgrades protocol to WebSocket protocol which is a TCP based protocol. Once the handshake is done, client and server can start sending messages over a single TCP connection.

A Server Websocket container listens for connection requests.

- A Client X creates a socket in its container.
- It connects to server on port 80 with HTTP request to initiate a webSocket connection.
- Server responds yes, to the upgrade to webSocket request,
- Server container creates a dedicated socket to the client.
- Client and server switch the protocol.
- Client and Server start exchanging packets using the webSocket protocol.

LIFE CYCLE EVENT:

The Java API for WebSocket (JSR-356) simplifies the integration of WebSocket into Java EE 7

applications. Annotation-driven programming allows developers to use POJOs to interact with WebSocket lifecycle events. Interface-driven programming allows developers to implement interfaces and methods to interact with WebSocket lifecycle events. Tyrus is the open source reference implementation for easy development of WebSocket applications. Server endpoint classes are POJOs (Plain Old Java Objects) that are annotated with

javax.websocket.server.ServerEndpoint. Client endpoint classes are POJOs annotated with javax.websocket.ClientEndpoint.

@OnOpen : This annotation may be used on methods of @ServerEndpoint or @ClientEndpoint. It is used to decorate a method which is called once new connection is established. The connection is represented by the optional Session parameter.

```
@OnOpen
public void onOpen( Session user) { //To do }
```

@OnMessage : This annotation may be used to decorate a method which is called once new message is received.

```
@OnMessage
public void onMessage(String message, Session user) { //To do }
```

@OnError : It is used to decorate a method which is called once Exception is being thrown by any method annotated with @OnOpen, @OnMessage and @OnClose.

```
@OnError
public void onError(Session user, Throwable t) { //To do }
```

@OnClose: It is used to decorate a method which is called once the connection is being closed. The method may have one Session parameter.

```
@OnClose
public void onClose(Session user) { //To do }
```

7.b Code and explain a web socket client. M

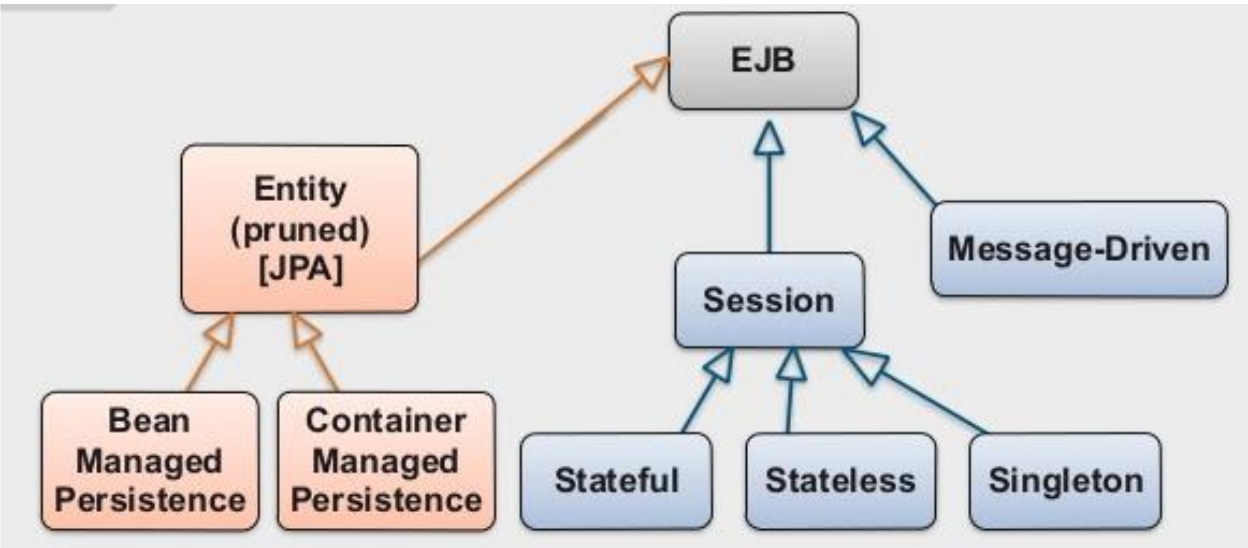
```
import javax.websocket.ClientEndpoint;
import javax.websocket.WebSocketContainer;
import javax.websocket.ContainerProvider;
import java.net.URI;
import javax.websocket.Session;
import javax.websocket.OnMessage;
import javax.websocket.RemoteEndpoint.Basic;
import java.util.Scanner;

@ClientEndpoint
public class EchoClient{

@OnMessage
public void onMessage(String msg){ System.out.println(msg);
}
public static void main(String[] args){

try{


//1. Create a WebSocketContainer WebSocketContainer container =
ContainerProvider.getWebSocketContainer();
//2. Connect to the Remote WebSocket Server URI uri =
URI.create("ws://localhost:8080/BroadcastServer
/broadcast"); Session ses =
container.connectToServer(EchoClient.class,uri);

//3.get the remote socket endpoint
Basic remoteEndPoint = ses.getBasicRemote();
//4. send the messages
Scanner sc = new Scanner(System.in); System.out.println("Enter Messages");

while(true){
```

```
String line = sc.nextLine(); remoteEndPoint.sendText(line);

}
}catch(Exception e){

System.out.println(e.getMessage());
}
}

}
```

UNIT – IV

8.a Explain the features of EJB and different types of beans and interfaces. M

Types of beans:



Session Bean: Session bean stores data of a particular user for a single session. It can be stateful or stateless. It is less resource intensive as compared to entity bean. Session bean gets destroyed as soon as user session terminates.
Stateless Session Bean: It doesn't maintain state of a client between multiple method calls. Stateful Session Bean: It maintains state of a client across multiple requests.

Singleton Session Bean: One instance per application, it is shared between clients and supports concurrent access.
Entity Bean: Entity beans represent persistent data storage. User data can be saved to database via entity beans and later on can be retrieved from the database in the entity bean.
Message Driven Bean: Message driven beans are used in context of JMS (Java Messaging Service). Message Driven Beans can consumes JMS messages from external entities and act accordingly.
An EJB component can have remote and local interfaces. Clients not located in the same application server instance as the bean use the remote interface to access the bean. Calls to the remote interface require marshalling and transportation of the arguments over the network, un-marshaling the arguments at the receiving end. Thus, using the remote interface entails significant overhead.

8.b Code a session bean with local interface and a web client. M

```
//Session Bean package ejb;
import javax.ejb.Stateless;
 @Stateless
public class ConvertBean implements ConvertBeanLocal
{
public float convert(float f)
```

```
{
return ((f-32)*(5/9));
}
}
```

```
package ejb;
import javax.ejb.Local;
@Local
public interface ConvertBeanLocal
{
public float convert(float );
}
//Servlet in webapp package web;
import ejb.ConvertBeanLocal;
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 HelloServlet extends HttpServlet
{

@EJB
private ConvetBeanLocal convBean;
@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{

response.setContentType("text/html");
PrintWriter pw = response.getWriter();
float f=request.getParameter(“f”);
pw.println("<h1>"+convBean.convert(f)+"</h1>");

}
}
```

(OR)

9.a Explain the annotations and the life cycle of a stateful session bean. M

EJB Lifecycle:

EJB Container creates and maintains a pool of session bean first. It injects the dependency if then calls the @PostConstruct method if any. Now actual business logic method is invoked by the client. Then, container calls @PreDestory method if any. Now bean is ready for garbage collection. There are 5 important annotations used in stateful session bean:

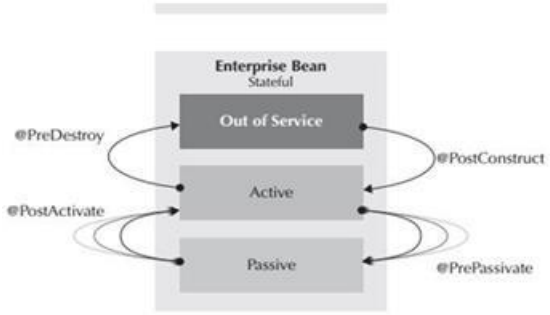
@Stateful : The class decorated with this annotation facilitates the creation of a stateful bean.

@PostConstruct : The method decorated with this annotation is executed before the bean becomes active.

@PreDestroy: The method decorated with this annotation is executed before the bean is destroyed.

When a user is not actively communicating with the bean then the bean is passivated, for example

when the user is entering the data in a html form. After the user enters the data and submits, the bean is activated once again.
@PrePassivate : Designates a method to receive a callback before a stateful session bean is passivated.
@PostActivate: Designates a method to receive a callback after a stateful session bean has been activated.



Statefull Bean Local Interface

9.b Code a session bean that takes temperature in Fahrenheit and returns its Celcius equivalent. M

```
//Session Bean package ejb;
import javax.ejb.Stateless;
 @Stateless
public class ConvertBean implements ConvertBeanLocal
{
public float convert(float f)
{
return ((f-32)*(5/9));
}
}
```

```
package ejb;
import javax.ejb.Local;
@Local
public interface ConvertBeanLocal
{
public float convert(float );
}
//Servlet in webapp package web;
import ejb.ConvertBeanLocal;
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 HelloServlet extends HttpServlet
{

@EJB
private ConvetBeanLocal convBean;
@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
```

```
throws ServletException, IOException
{

response.setContentType("text/html");
PrintWriter pw = response.getWriter();
float f=request.getParameter(“f”);
pw.println("<h1>"+convBean.convert(f)+"</h1>");

}
}
```