GUJARAT TECHNOLOGICAL UNIVERSITY (GTU)

Competency-focused Outcome-based Green Curriculum-2021 (COGC-2021)
Semester -V

Course Title: Advanced Java Programming

(Course Code: 4351603)

Diploma programme in which this course is offered	Semester in which offered
Information Technology	5th semester

1. RATIONALE

This course provides the knowledge necessary to develop dynamic web pages using Servlet, JSP, MVC web frameworks and hibernate. It covers the basic underlying concepts and techniques recently used in the IT industry. After going through this course students will be able to design MVC based Web applications.

2. COMPETENCY

The purpose of this course is to help the student to attain the following industry identified competency through various teaching-learning experiences:

- **Develop MVC based web applications using Java web framework.**

3. COURSE OUTCOMES (COs)

The practical exercises, the underpinning knowledge, and the relevant soft skills associated with this competency are to be developed in the student to display the following COs:

The student will develop underpinning knowledge, adequate programming skills of competency for implementing various applications using java programming language to attain the following course outcomes.

a) Develop a GUI application using swing components.
b) Apply ORM based Methodology for Application Development.
c) Develop Web Applications using Servlets and deploy in popular servers like Tomcat
d) Develop JSP based applications with database connectivity.
e) Apply MVC architecture using Spring framework.

4. TEACHING AND EXAMINATION SCHEME

Teaching Scheme (In Hours)			Total Credits (L+T/2+P/2)	Examination Scheme				
				Theory Marks		Practical Marks		Total Marks
L	T	P	C	CA	ESE	CA	ESE	
3	-	2	4	30	70	25	25	150

(*): Out of 30 marks under the theory CA, 10 marks are for assessment of the micro-project to facilitate integration of COs and the remaining 20 marks is the average of 2 tests to be taken

during the semester for the assessing the attainment of the cognitive domain UOs required for the attainment of the COs.
Legends: L*-Lecture;* ***T*** *– Tutorial/Teacher Guided Theory Practice;* ***P*** *-Practical;* ***C*** *– Credit,* ***CA*** *- Continuous Assessment;* ***ESE*** *-End Semester Examination.*

5. SUGGESTED PRACTICAL EXERCISES

The following practical outcomes (PrOs) are the subcomponents of the COs. These PrOs need to be attained to achieve the COs.

S. No.	Practical Outcomes (PrOs)	Unit No.	Approx. Hrs. required
1	Develop a GUI program by using one swing button and adding it on the JFrame object	I	02
2	Develop a program to create checkboxes for different courses belongingto a university such that the course selected would be displayed.	I	02
3	Develop a program to Implement Traffic signal(Red, Green and Yellow) by using Swing components (Using JFrame, JRadioButton, ItemListener etc.)	I	04
4	Develop a program using JDBC to display student's record (Enroll No, Name, Address, Mobile No,and Email-ID) into table 'StuRec'	II	02
5	Develop a program using JDBC to edit (insert, update, delete) Student's profile stored in the database	II	02
6	Develop an application to store, update, fetch and delete data of Employee (NAME, AGE, SALARY and DEPARTMENT) using Hibernate CRUD operations.	II	02
7	Develop a simple servlet program which maintains a counter for the number of times it has been accessed since its loading; initialize the counter using deployment descriptor.	III	02
8	Create a web form which processes servlet and demonstrates use of cookies and sessions.	III	02
9	Develop a web form which processes servlet for user login functionality	III	02
10	Develop a JSP web application to display student monthly attendance in each subject of current semester via enrollment number.	IV	02
11	Develop a JSP web application to select shopping products, buy any products and product billing page using session management.	IV	02
12	Develop a JSP program for student registration for new admission in college and display stored data it into admin dashboard.	IV	02

13	Develop a Web MVC Architecture based application to provide user Login and Register using Spring Boot.	V	02
	Total		**28**

Note

i. *More **Practical Exercises** can be designed and offered by the respective course teacher to develop the industry relevant skills/outcomes to match the COs. The above table is only a suggestive list.*

*ii. The following are some **sample** 'Process' and 'Product' related skills (more may be added/deleted depending on the course) that occur in the above listed **Practical Exercises** of this course required which are embedded in the COs and ultimately the competency..*

S. No.	Sample Performance Indicators for the PrOs	Weightage in %
1	Use of appropriate technology/software/tools.	10
2	Coding methodology.	30
3	Testing and Debugging of the program.	20
4	Correctness of Program.	20
5	Submission in time.	20
Total		**100**

6. MAJOR EQUIPMENT/ INSTRUMENTS REQUIRED

This major equipment with broad specifications for the PrOs is a guide to procure them by the administrators to usher in uniformity of practical in all institutions across the state.

S. No.	Equipment Name with Broad Specifications	PrO. No.
1	Computer system with operating system: Windows 7 or higher Ver., macOS, and Linux, with 4GB or higher RAM, Java, mysql	All
2	JDK and IDE such as eclipse, netbeans & spring framework	

7. AFFECTIVE DOMAIN OUTCOMES

The following ***sample*** Affective Domain Outcomes (ADOs) are embedded in many of the above-mentioned COs and PrOs. More could be added to fulfill the development of this competency.

a) Work as a Java developer.
b) Follow ethical practices.

The ADOs are best developed through the laboratory/field based exercises. Moreover, the level of achievement of the ADOs according to Krathwohl's 'Affective Domain Taxonomy' should gradually increase as planned below:

i. 'Valuing Level' in 1st year
ii. 'Organization Level' in 2nd year.
iii. 'Characterization Level' in 3rd year.

9. UNDERPINNING THEORY

Only the major Underpinning Theory is formulated as higher-level UOs of *Revised Bloom's taxonomy* in order development of the COs and competency is not missed out by the students and teachers. If required, more such higher-level UOs could be included by the course teacher to focus on the attainment of COs and competency.

Unit	**Unit Outcomes (UOs)** (4 to 6 UOs at Application and above level)	Topics and Sub-topics
Unit – I JAVA SWING	1a. Differentiate between AWT and Swing 1b. Develop GUI programs using Swing Components 1c. Develop simple event driven programs using event class and event listener interface	1.1 Introduction to JFC and Swing,Difference between AWT and Swing 1.2 Features of the Java Foundation Classes 1.3 Swing Components: Swing Classes Hierarchy, Commonly used Methods of Component class (add(), setSize(), setLayout(), and setVisible()), JApplet, JFrame, JLabel, JTextField, JTextArea, JButton, JCheckBox, JRadioButton, JComboBox, JMenu 1.4 Layout Management: Flow Layout, Border Layout, Card Layout, Box Layout, Grid Layout, Gridbag Layout, Group Layout, Spring Layout 1.5 Event Handling : Introduction, Action Events, Key Events, Focus Events, Window Event, Mouse Event, Item Events 1.6 EventListner Interface : ActionListener, KeyListener, FocusListener, WindowListener, MouseListener, MouseMotionListener, ItemListener

Unit – II Java Database Connectivity	2a. Describe the basics of JDBC and its connectivity 2b. Explain different types of JDBC drivers and their advantages and disadvantages 2c. Develop program using JDBC to query a database and modify it 2d. Explain Object Relational Mapping and Advantages 2e. Describe Hibernate Architecture and Environment setup	2.1 Two-Tier Database Design, Three-Tier Database Design 2.2 The JDBC API: The API components, database operations like creating tables, CRUD(Create, Read, Update, Delete) operations using SQL 2.3 JDBC- advantages and disadvantages 2.4 JDBC drivers 2.5 JDBC-ODBC bridge 2.6 Develop java program using JDBC 2.7 ORM working model, advantagesand ORM tools 2.8 Architecture of hibernate and installation steps for IDE 2.9 Hibernate Properties and supported database
Unit– III Servlets	3a. Describe life cycle of servlet 3b. Develop web application usingjavax.servlet package	3.1 The life cycle of a servlet 3.2 The Java Servlet Development Kit 3.3 The Simple Servlet: create and compile servlet source code, start a web browser and request the servlet,example of echo servlet and deployment in tomcat server 3.4 The javax.servlet Package: reading database/table records and displaying them using servlet
Unit– IV Java Server Pages	4a. Explain JSP with syntax and Semantics 4b. Implement web application using JSP Form input elements and validation 4c. Implement web application using JSP Cookies and Session tracking 4d. Describe JSTL- JSP STANDARD TAG LIBRARY 4e. Implement web application using JSP database connection	4.1 Advantages of JSP and lifecycle ofJSP 4.2 Components of JSP page: Directives, Comments, Expression, Scriptlets, Declarations, Implicit Objects, Standard Actions and Tag Extensions 4.3 Elements Created with the INPUT Tag, Elements Created with select and option, textarea Element 4.4 JSP form validation 4.5 Read and Delete data from cookies, maintain session and track session id, Core tags, SQL tags, XML tags, JSTL functions 4.6 Create Table using JSP, SELECT, INSERT, DELETE and UPDATE
Unit– V	5a. Explain Web ApplicationFramework	5.1 Importance of MVC architecture, Advantages of MVC Architecture

Web MVC framework	5b. Describe MVC Architecture Layers 5c. Implement web application using Spring Framework 5d. Describe Spring Boot and applications of Spring Boot	5.2 Model layer, View layer and Controller layer 5.3 Aspect-oriented programming (AOP), Dependency injection (DI) and Plain Old Java Object (POJO), Spring Framework Architecture 5.4 Features in Spring Boot, Comparison Between Spring and Spring Boot

***Note**: The UOs need to be formulated at the 'Application Level' and above of Revised Bloom's Taxonomy' to accelerate the attainment of the COs and the competency.*

10. SUGGESTED SPECIFICATION TABLE FOR QUESTION PAPER DESIGN

Unit No.	**Unit Title**	**Teaching Hours**	**Distribution of Theory Marks**			
			R Level	**U Level**	**A Level**	**Total Marks**
I	Java Swing	12	04	06	08	18
II	Java Database Connectivity	06	04	03	03	10
III	Servlets	09	04	06	06	16
IV	Java Server Pages	09	04	06	06	16
V	Web MVC framework	06	02	04	04	10
Total		**42**				**70**

***Legends:** R=Remember, U=Understand, A=Apply and above (Revised Bloom's taxonomy)*
__Note__: This specification table provides general guidelines to assist students for their learning and to teachers to teach and question paper designers/setters to formulate test items/questions assess the attainment of the UOs. The actual distribution of marks at different taxonomy levels (of R, U and A) in the question paper may vary slightly from the above table.

11. SUGGESTED STUDENT ACTIVITIES

Other than the classroom and laboratory learning, following are the suggested student-related ***co-curricular*** activities which can be undertaken to accelerate the attainment of the various outcomes in this course: Students should conduct following activities in group and prepare reports of about 5 pages for each activity, also collect/record physical evidences for their (student's) portfolio which will be useful for their placement interviews:

a) Explore different application development using java web application technologies/ tools and frameworks.
b) Undertake micro-projects in teams
c) Give a seminar on any relevant topics.

12. SUGGESTED SPECIAL INSTRUCTIONAL STRATEGIES (if any)

These are sample strategies, which the teacher can use to accelerate the attainment of the various outcomes in this course:

a) Massive open online courses (***MOOCs***) may be used to teach various topics/subtopics.
b) Guide student(s) in undertaking micro-projects.
c) ***'L' in section No. 4*** means different types of teaching methods that are to be employed by teachers to develop the outcomes.
d) About ***20% of the topics/sub-topics*** which are relatively simpler or descriptive in nature is to be given to the students for ***self-learning***, but to be assessed using different assessment methods.
e) With respect to ***section No.11***, teachers need to ensure to create opportunities and provisions for ***co-curricular activities***.
f) Guide students for open source java editors.

13. SUGGESTED MICRO-PROJECTS

Only one micro-project is planned to be undertaken by a student that needs to be assigned to him/her in the beginning of the semester. In the first four semesters, the micro-project are group-based. However, in the fifth and sixth semesters, it should be preferably be ***individually*** undertaken to build up the skill and confidence in every student to become problem solver so that he/she contributes to the projects of the industry. In special situations where groups have to be formed for micro-projects, the number of students in the group should ***not exceed three.***

The micro-project could be industry application based, internet-based, workshop-based, laboratory-based or field-based. Each micro-project should encompass two or more COs which are in fact, an integration of PrOs, UOs and ADOs. Each student will have to maintain a dated work diary consisting of individual contributions in the project work and give a seminar presentation of it before submission. The total duration of the micro-project should not be less than ***16 (sixteen) student engagement hours*** during the course. The student ought to submit a micro-project by the end of the semester to develop the industry-oriented COs.

A suggestive list of micro-projects is given here. This has to match the competency and the COs. Similar micro-projects could be added by the concerned course teacher:

- **Project idea 1:** Develop Degree/Diploma Admission web application using java web application technologies to get student registration data, process and filter data.
  - verification of submitted data
  - support files/documents uploading features.
- **Project idea 2:** Develop Library Management System
  - to update the record, monitor and add books,
  - search for the required ones, taking care of the issue date and return date
  - It comes with basic features like creating a new record and updating and deleting it.
- **Project idea 3:** Develop MVC based web application for college placement record keeping.
  - student registration, apply for company.
  - find the result of the interview.

14. SUGGESTED LEARNING RESOURCES

S. No.	Title of Book	Author	Publication with place, year and ISBN
1	Black Book " Java server programming" J2EE	Kathy walrath	Dream Tech Publishers
2	Complete Reference J2EE	James Keogh	McGraw Publication
3	Java: The Complete Reference	Herbert Schildt	McGraw-Hill ISBN: 9781260440249
4	JSP: The Complete Reference	Phillip Hanna	McGraw Hill Education ISBN: 0072224371
5	Beginning Hibernate 6	Joseph B. Ottinger, Jeff Linwood, Dave Minter	Springer India ISBN: 1484284135

15. SOFTWARE/LEARNING WEBSITES

a. https://www.javatpoint.com/java-ee
b. https://www.geeksforgeeks.org/introduction-to-java-swing/
c. https://www.tutorialspoint.com/
d. https://nptel.ac.in/
e. https://docs.jboss.org/hibernate/orm/3.6/reference/en-US/pdf/hibernate_reference.pdf
f. https://www.javatpoint.com/what-is-advance-java
g. https://docs.oracle.com/cd/E13222_01/wls/docs81/jsp/intro.html
h. https://docs.spring.io/spring-framework/docs/6.0.7/reference/pdf/spring-framework.pdf

16. PO-COMPETENCY-CO MAPPING

Semester V	Advanced JAVA Programming(Course Code: 4351603)						
	POs and PSOs						
Competency & Course Outcomes	PO 1 Basic & Discipline specific knowledge	PO 2 Problem Analysis	PO 3 Design/ development of solutions	PO 4 Engineering Tools, Experimentation & Testing	PO 5 Engineering practices for society, sustainability & environment	PO 6 Project Management	PO 7 Life-long learning
Competency Develop MVC based web applications using Java web framework.							
Course Outcomes CO a)Develop a GUI application using swing components.	2	3	3	2	2	2	2

CO b) Apply ORM based Methodology for Application Development.	3	2	3	2	-	2	2
CO c) Develop Web Applications using Servlets and deploy in popular servers like Tomcat	3	3	3	3	1	2	2
CO d) Develop JSP based applications with database connectivity.	3	3	3	3	-	2	2
CO e)Apply MVC architecture using Spring framework.	3	2	3	3	-	3	2

Legend: '**3**' for high, '**2**' for medium, **'1'** for low or **'-'** for the relevant correlation of each competency, CO, with PO/ PSO

17. COURSE CURRICULUM DEVELOPMENT COMMITTEE

GTU Resource Persons

Sr. No.	Name and Designation	Institute	Email
1	Miss. PRITI N. PARIKH (H.O.D I.T)	GP,Himatnagar	pritiparikhdit@gmail.com
2	Mr. AJAYKUMAR J. BAROT (L.I.T)	GP,Himatnagar	ajay2185@gmail.com
3	Mr. UMANG SHUKLA (L.I.T)	SSGP, Surat	umang.shuklait@gmail.com

CH – 2

JDBC

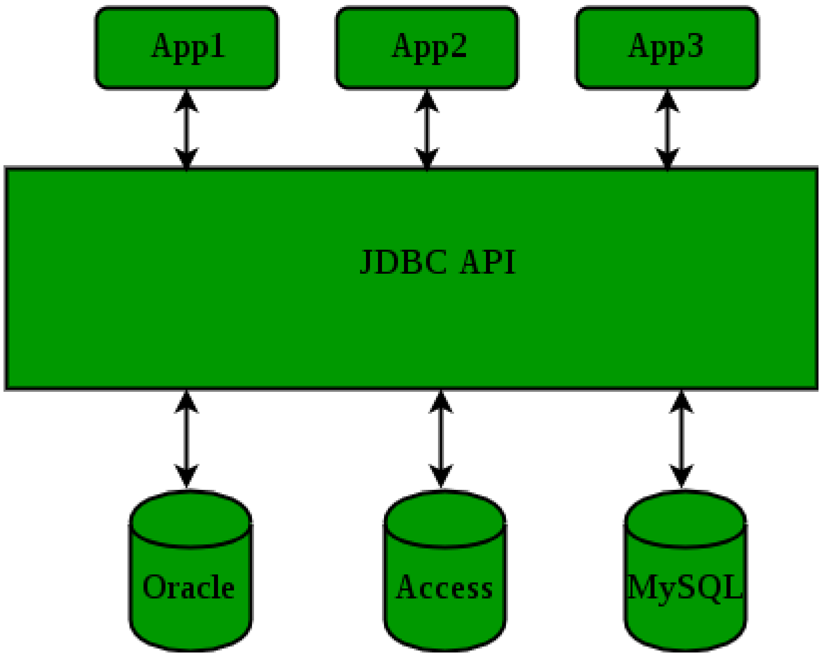
Java Database Connectivity (JDBC) - is an application programming interface (API) for the programming language Java, which defines how a client may access any kind of tabular data, especially a relational database. JDBC Drivers uses JDBC APIs which was developed by Sun Microsystem, but now this is a part of Oracle. There are 4 types of JDBC drivers. It is part of the Java Standard Edition platform, from Oracle Corporation. It acts as a middle-layer interface between Java applications and databases.

The JDBC classes are contained in the Java Package java.sql and javax.sql.

JDBC helps you to write Java applications that manage these three programming activities:

1. Connect to a data source, like a database.
2. Send queries and update statements to the database
3. Retrieve and process the results received from the database in answer to your query

Jdbc_architecture



2.1 Two-Tier Database Design:

In the realm of database design, the two-tier architecture stands as a foundational structure. Here, we have two essential components: the client and the server. The client, typically a user interface or application, interacts directly with the server, which houses the database. The simplicity of this model is its strength, making it easy to comprehend and implement.

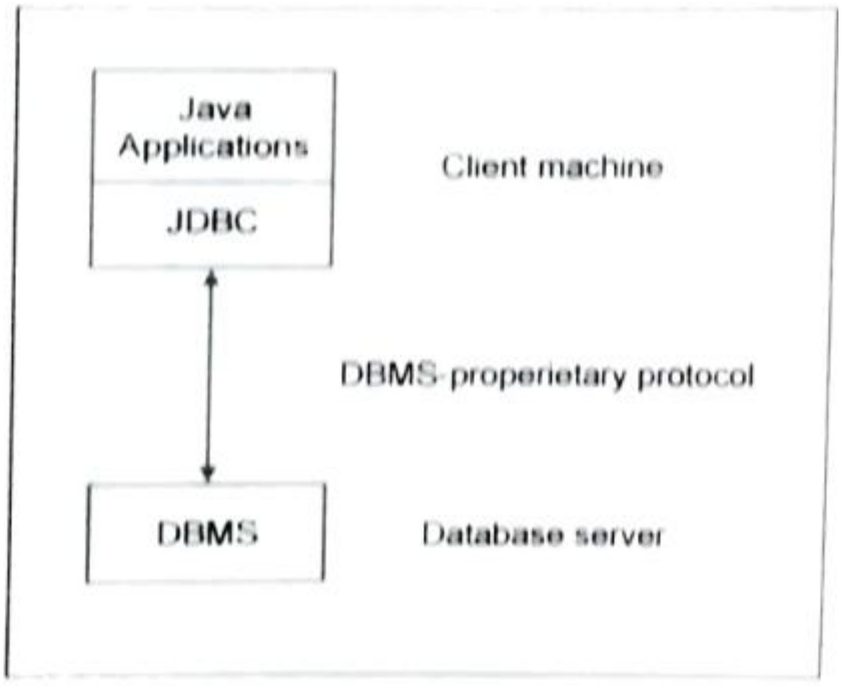
However, it does bear the drawback of potential scalability challenges, as both the application logic and data management reside on the client side and may strain resources as the system grows.

Advantages:

- Simplicity in design and implementation.
- Efficient for small-scale applications.

Disadvantages:

- Limited scalability.
- Security concerns as the client holds application logic.



Three-Tier Database Design:

As we progress to a more robust structure, the three-tier architecture emerges. This design comprises three integral layers: the presentation layer (client interface), the application layer (business logic), and the data layer (database). This modular approach enhances scalability and maintainability. The presentation layer focuses solely on the user interface, the application layer handles logic and processing, and the data layer manages the storage and retrieval of information. This separation of concerns not only facilitates growth but also improves security by isolating the database.

Components:

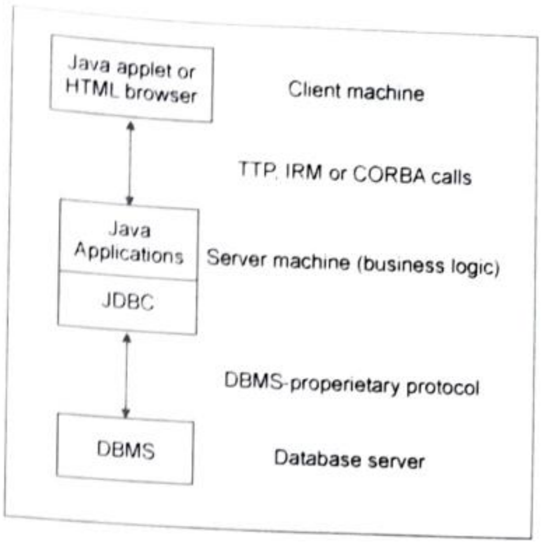
1. **Presentation Layer:**
    - User Interface (UI)
    - Client-Side Logic
2. **Application Layer:**
    - Business Logic
    - Server-Side Logic
3. **Data Layer:**
    - Database
    - Data Storage and Retrieval

Advantages:

- Improved scalability.
- Enhanced security through separation of layers.

Disadvantages:

- Increased complexity in design and implementation.


Java applet or HTML browser
Client machine
TTP, IRM or CORBA calls
Java Applications
JDBC
Server machine (business logic)
DBMS-properietary protocol
DBMS
Database server

1. Overview of JDBC API: The JDBC (Java Database Connectivity) API serves as a bridge between Java applications and relational databases. Its primary purpose is to enable communication and interaction with databases seamlessly.

- **Key Components:**
    - **DriverManager:** Manages a list of database drivers.

    - **Connection:** Establishes a connection to the database.
    - **Statement:** Executes SQL queries against the database.
    - **ResultSet:** Represents the result set of a database query.

2. DriverManager in JDBC: The DriverManager class plays a pivotal role in the JDBC API. It helps in managing a list of database drivers. These drivers act as intermediaries, facilitating communication between Java applications and various databases.

- **Steps for Connection:**
    1. **Loading Drivers:** Use **Class.forName("driverClass")** to load the desired database driver.
    2. **Establishing Connection:** Utilize **DriverManager.getConnection(url, username, password)** to establish a connection.

3. Connection Interface: The Connection interface is responsible for establishing and managing connections to the database. It forms the foundation for executing SQL statements and retrieving results.

- **Methods:**
    - **createStatement()**: Creates a Statement object for executing SQL queries.
    - **prepareStatement(sql)**: Prepares a SQL statement for execution, allowing parameterized queries.
    - **close()**: Closes the connection, releasing resources.

4. Statement Interface: The Statement interface executes SQL queries against the database. There are different types of statements, such as Statement, PreparedStatement, and CallableStatement, catering to various use cases.

- **Execution Methods:**
    - **executeQuery(sql)**: Executes a SELECT query and returns a ResultSet.
    - **executeUpdate(sql)**: Executes INSERT, UPDATE, or DELETE queries and returns the number of affected rows.

5. ResultSet Interface: The ResultSet interface represents the result set of a database query. It provides methods to traverse and retrieve data from the query result.

- **Navigation Methods:**
    - **next()**: Moves the cursor to the next row.
    - **getString(columnName)**: Retrieves the value of the specified column as a String.

6. Exception Handling in JDBC: Robust error handling is crucial in JDBC programming. Handling SQLExceptions ensures graceful degradation in case of database-related issues.

- **Try-Catch Blocks:**
    - Wrap JDBC code in try-catch blocks to catch and handle SQLExceptions.
    - Implementing proper error handling enhances the reliability of the application.

7. Best Practices in JDBC Programming: Adopting best practices ensures efficient and secure database interactions in Java applications.

- **Connection Pooling:** Use connection pooling to manage database connections effectively.
- **Prepared Statements:** Prefer PreparedStatement for enhanced security against SQL injection.

2.2

Database Operations:

In Java, database operations involve interacting with a database to perform tasks such as creating tables and manipulating data. The primary focus is on CRUD operations—Create, Read, Update, and Delete.

1. Creating Tables:

Creating tables is the foundation of database design. It involves defining the structure of your data. In Java, you can use the JDBC (Java Database Connectivity) API to execute SQL statements for table creation. Here's a simple example:

```java
import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.Statement;


public class CreateTableExample {

    public static void main(String[] args) {

        // JDBC URL, username, and password

        String url = "jdbc:mysql://localhost:3306/yourdatabase";

        String user = "yourusername";

        String password = "yourpassword";
        try {
            // Establish connection
            Connection connection = DriverManager.getConnection(url, user, password);

            // Create statement
            Statement statement = connection.createStatement();

            // SQL query for table creation
            String createTableQuery = "CREATE TABLE IF NOT EXISTS students (" +
                    "id INT PRIMARY KEY, " +
                    "name VARCHAR(255), " +
                    "age INT)";
            statement.execute(createTableQuery);

            // Close resources
            statement.close();
            connection.close();

            System.out.println("Table created successfully!");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

2. CRUD Operations:

CRUD operations are fundamental for interacting with data in a database.

a. Create (Insert):

Inserting data into a table is done with an SQL INSERT statement. Here's an example in Java:

```java
// Assuming the connection is already established
String insertDataQuery = "INSERT INTO students (id, name, age) VALUES (1, 'John', 20)";
statement.execute(insertDataQuery);
System.out.println("Data inserted successfully!");
```

b. Read (Select):

Reading data involves retrieving information from the database. Here's a basic read operation:

```java
String selectDataQuery = "SELECT * FROM students";
ResultSet resultSet = statement.executeQuery(selectDataQuery);

while (resultSet.next()) {
   int id = resultSet.getInt("id");
   String name = resultSet.getString("name");
   int age = resultSet.getInt("age");

   System.out.println("ID: " + id + ", Name: " + name + ", Age: " + age);
}
resultSet.close();
```

c. Update:

Updating data is accomplished with the SQL UPDATE statement. Here's a Java example:

```java
String updateDataQuery = "UPDATE students SET age = 21 WHERE name = 'John'";
statement.execute(updateDataQuery);
System.out.println("Data updated successfully!");
```

d. Delete:

Deleting data involves the SQL DELETE statement. Here's how you can do it in Java:

```java
String deleteDataQuery = "DELETE FROM students WHERE id = 1";
statement.execute(deleteDataQuery);
System.out.println("Data deleted successfully!");
```

Advantages of JDBC:

1. Platform Independence: JDBC is platform-independent, allowing Java applications to connect to various types of databases without concern for the underlying operating system.

2. Universal Database Access: JDBC provides a standardized interface, making it possible to access different database management systems (DBMS) using the same set of Java classes and methods.

3. Ease of Use: JDBC is designed to be user-friendly. Its API is straightforward, and developers can quickly grasp the concepts of connecting to databases, executing queries, and processing results.

4. Integration with Java EE: JDBC seamlessly integrates with Java EE (Enterprise Edition), making it a vital component for developing enterprise-level applications with database connectivity.

5. Dynamic SQL Support: JDBC allows the execution of dynamic SQL queries, enabling applications to construct SQL statements based on runtime conditions.

6. Robust Error Handling: JDBC provides robust error handling mechanisms, allowing developers to capture and handle exceptions effectively, ensuring the stability of database interactions.

Disadvantages of JDBC:

1. Manual Resource Management: JDBC requires manual management of database resources, such as opening and closing connections, statements, and result sets. Improper resource handling can lead to memory leaks and other issues.

2. Boilerplate Code: JDBC code often involves a significant amount of boilerplate code for tasks like database connection, exception handling, and result set processing. This can make the code verbose and less maintainable.

3. Limited Object-Relational Mapping (ORM) Features: JDBC primarily deals with low-level database interactions and lacks advanced Object-Relational Mapping features. Developers may need additional frameworks like Hibernate for comprehensive ORM capabilities.

4. SQL-Centric Approach: JDBC follows a SQL-centric approach, which means developers need to write and manage SQL queries explicitly. This can be a drawback for those accustomed to more abstract and object-oriented programming.

5. Database Vendor-Specific Code: Despite being a standard API, JDBC code can become database vendor-specific when dealing with certain features. This limits the portability of code across different database systems.

6. Lack of Connection Pooling: JDBC does not provide built-in connection pooling, a feature crucial for managing database connections efficiently in high-performance applications. Developers often need to implement custom connection pooling solutions.

2.2 jdbc drivers

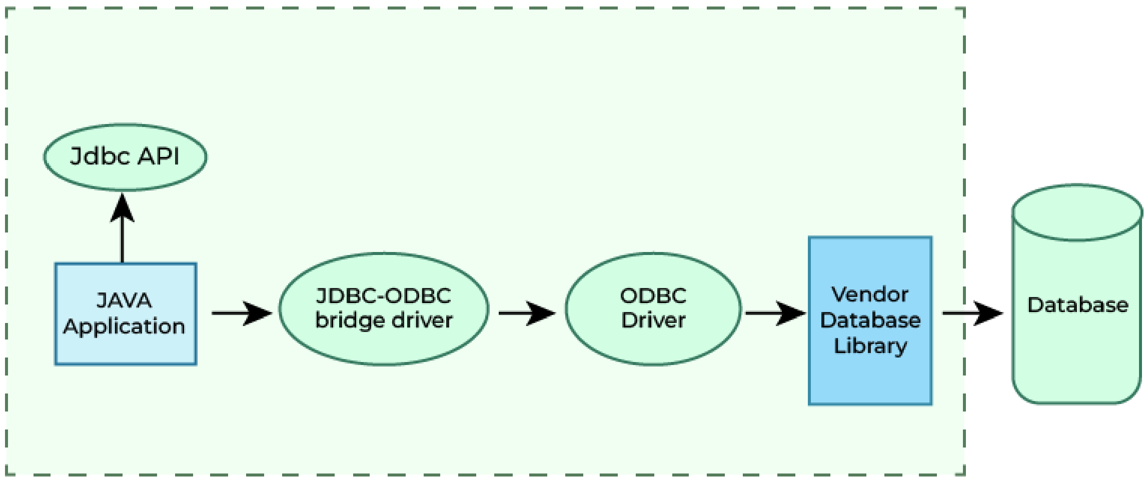
JDBC Drivers

JDBC drivers are client-side adapters (installed on the client machine, not on the server) that convert requests from Java programs to a protocol that the DBMS can understand. **JDBC drivers** are the software components which implements interfaces in JDBC APIs to enable java application to interact with the database. Now we will learn how many JDBC driver types does Sun defines? There are four types of JDBC drivers defined by Sun Microsystem that are mentioned below:

1. Type-1 driver or JDBC-ODBC bridge driver
2. Type-2 driver or Native-API driver
3. Type-3 driver or Network Protocol driver
4. Type-4 driver or Thin driver

1. JDBC-ODBC bridge driver – Type 1 driver

Type-1 driver or JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. Type-1 driver is also called Universal driver because it can be used to connect to any of the databases.



Advantages

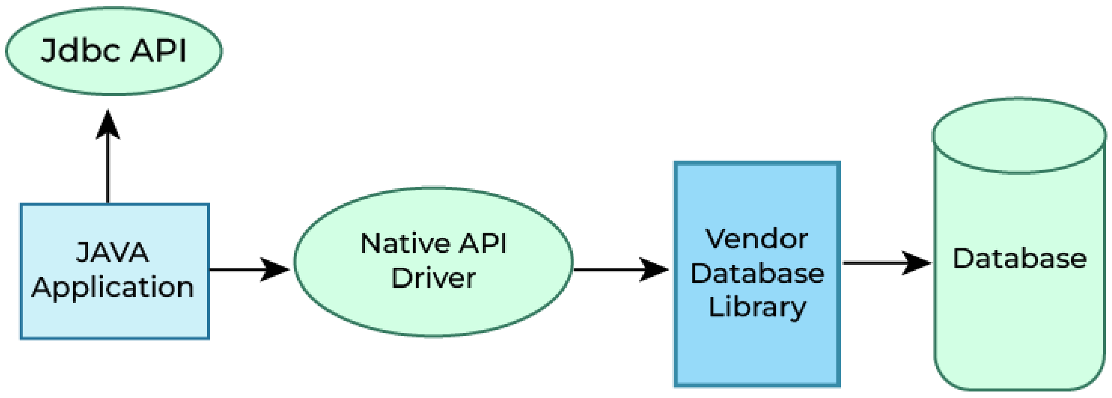
- This driver software is built-in with JDK so no need to install separately.
- It is a database independent driver.

Disadvantages

- As a common driver is used in order to interact with different databases, the data transferred through this driver is not so secured.
- The ODBC bridge driver is needed to be installed in individual client machines.
- Type-1 driver isn't written in java, that's why it isn't a portable driver.

2. Native-API driver – Type 2 driver (Partially Java driver)

The Native API driver uses the client -side libraries of the database. This driver converts JDBC method calls into native calls of the database API. In order to interact with different database, this driver needs their local API, that's why data transfer is much more secure as compared to type-1 driver. This driver is not fully written in Java that is why it is also called Partially Java driver.



Advantage

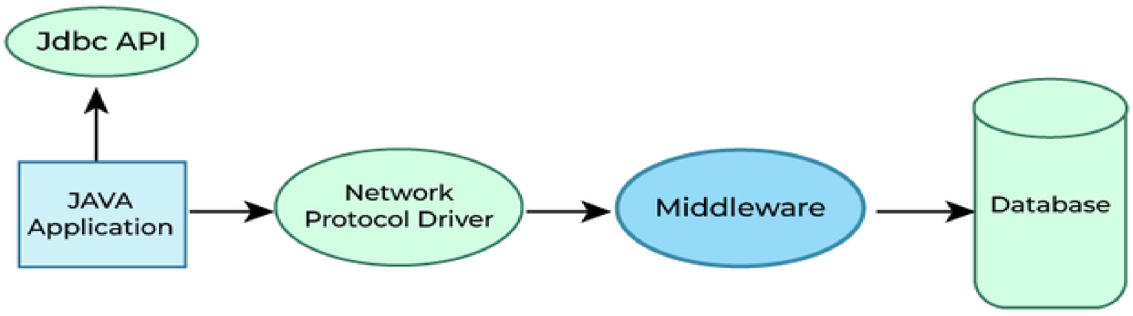
- Native-API driver gives better performance than JDBC-ODBC bridge driver.

Disadvantages

- Driver needs to be installed separately in individual client machines
- The Vendor client library needs to be installed on client machine.
- Type-2 driver isn't written in java, that's why it isn't a portable driver
- It is a database dependent driver.

3. Network Protocol driver – Type 3 driver (fully Java driver)

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. Here all the database connectivity drivers are present in a single server, hence no need of individual client-side installation.



Advantages

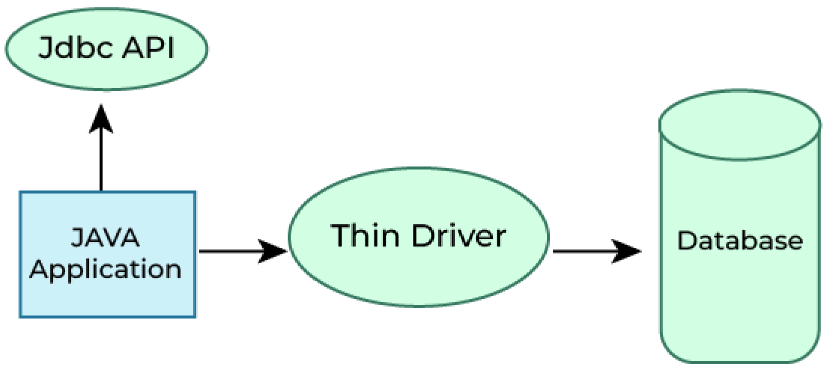
- Type-3 drivers are fully written in Java, hence they are portable drivers.
- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.
- Switch facility to switch over from one database to another database.

Disadvantages

- Network support is required on client machine.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

4. Thin driver – Type 4 driver (fully Java driver)

Type-4 driver is also called native protocol driver. This driver interact directly with database. It does not require any native database library, that is why it is also known as Thin Driver.



Advantages

- Does not require any native library and Middleware server, so no client-side or server-side installation.
- It is fully written in Java language, hence they are portable drivers.

Disadvantage

- If the database varies, then the driver will carry because it is database dependent.

2.6 Develop java program using JDBC with output

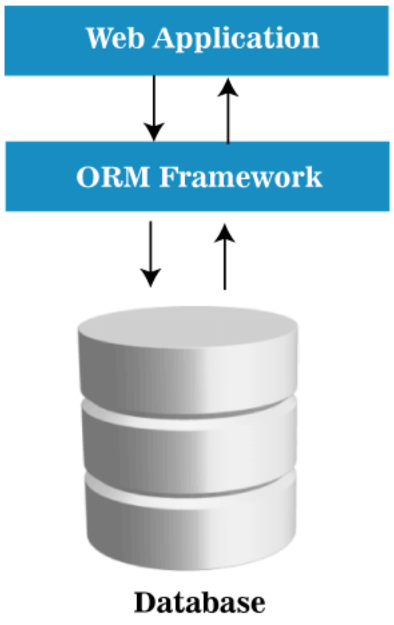
```
import java.sql.*;

public class JDBCExample {
   public static void main(String[] args) {
      // JDBC connection parameters
      String url = "jdbc:mysql://localhost:3306/yourdatabase";
      String username = "yourusername";
      String password = "yourpassword";

      // SQL query
      String sqlQuery = "SELECT * FROM yourtable";

      try {
         // Load JDBC driver
         Class.forName("com.mysql.cj.jdbc.Driver");

         // Establish database connection
         Connection connection = DriverManager.getConnection(url, username, password);

         // Create a statement
         Statement statement = connection.createStatement();

         // Execute SQL query
         ResultSet resultSet = statement.executeQuery(sqlQuery);

         // Process result set
         while (resultSet.next()) {
            int id = resultSet.getInt("id");
            String name = resultSet.getString("name");
            int age = resultSet.getInt("age");

            // Display results
            System.out.println("ID: " + id + ", Name: " + name + ", Age: " + age);
         }

         // Close connections
         resultSet.close();
         statement.close();
         connection.close();

      } catch (ClassNotFoundException | SQLException e) {
         e.printStackTrace();
      }
   }
}
```

Output

```
ID: 1, Name: John, Age: 25
ID: 2, Name: Jane, Age: 30
ID: 3, Name: Mike, Age: 22
```

2.7 ORM working model, advantagesand ORM tools

ORM is a technique for converting data between Java objects and relational databases (table). In simple words, we can say that the **ORM implements responsibility of mapping the object to relational model and vice-versa**. the ORM tool does mapping in such a way that model class becomes a table in the database and each instance becomes a row of the table.



Advantages of ORM Tool

- It saves time and efforts.
- It gives pace to development process.
- It reduces the development cost.
- It provides connectivity to the database.
- It makes development more object-oriented.
- Easy transaction management.
- No need to implement database manually.
- Modification in any model (object or relational model) does not affect each other.

ORM Tools

There are many ORM tools available but the following ORM tools are the most commonly used.

- Hibernate
- TopLink
- EclipseLink
- OpenJPA
- MyBatis (formally known as iBatis)

2. Hibernate: *Definition:* Hibernate is a powerful, open-source ORM tool for Java applications. It automates the mapping between Java objects and database tables, simplifying database interactions. Hibernate supports various relational databases.

Key Features:

- Object-Relational Mapping
- Automatic Table Creation
- HQL (Hibernate Query Language)

3. TopLink: *Definition:* TopLink is an ORM tool that originated from Oracle. It provides a comprehensive set of features for mapping Java objects to relational databases and vice versa.

Key Features:

- Mapping Support
- Caching Mechanisms
- Advanced Querying Options

4. EclipseLink: *Definition:* EclipseLink is an open-source ORM solution and the reference implementation for the Java Persistence API (JPA). It offers a wide range of features for mapping Java objects to databases.

Key Features:

- JPA Implementation

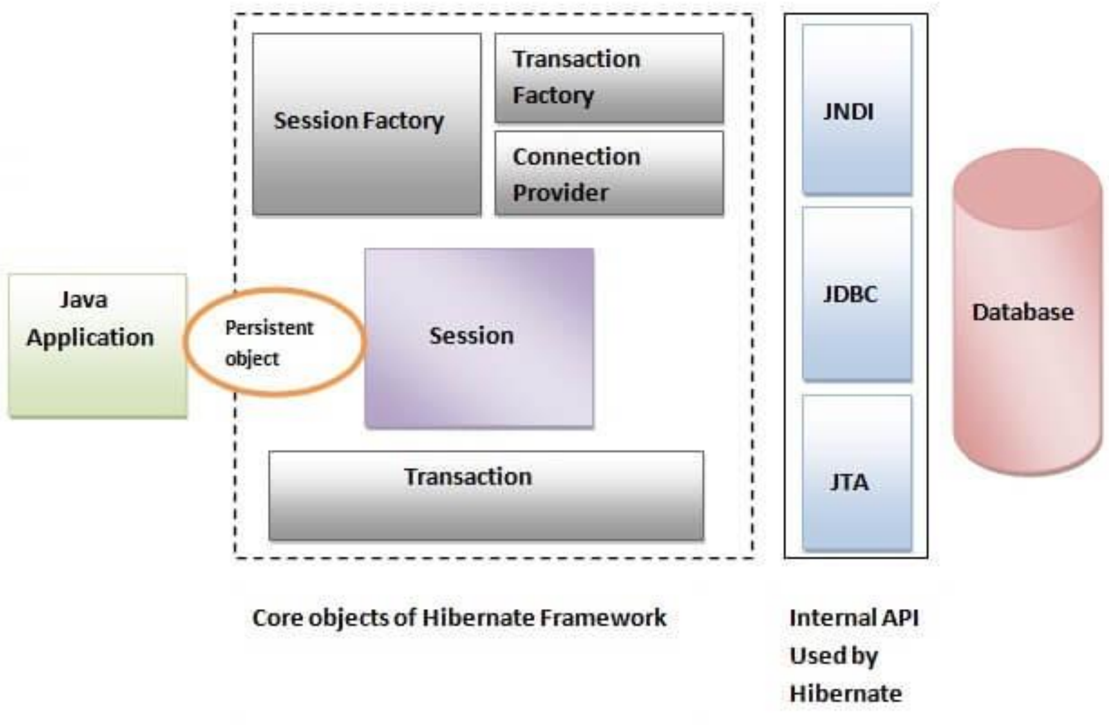
- Support for NoSQL Databases
- Caching Strategies

5. OpenJPA: *Definition:* Apache OpenJPA is another open-source implementation of the JPA specification. It provides features to manage the mapping between Java objects and relational databases.

Key Features:

- Transparent Persistence
- Dynamic Enhancements
- Query Optimization

2.8 Architecture of hibernate and installation steps for IDE



SessionFactory

The SessionFactory is a factory of session and client of ConnectionProvider. It holds second level cache (optional) of data. The org.hibernate.SessionFactory interface provides factory method to get the object of Session.

Session

The session object provides an interface between the application and data stored in the database. It is a short-lived object and wraps the JDBC connection. It is factory of Transaction, Query and Criteria. It holds a first-level cache (mandatory) of data. The org.hibernate.Session interface provides methods to insert, update and delete the object. It also provides factory methods for Transaction, Query and Criteria.

Transaction

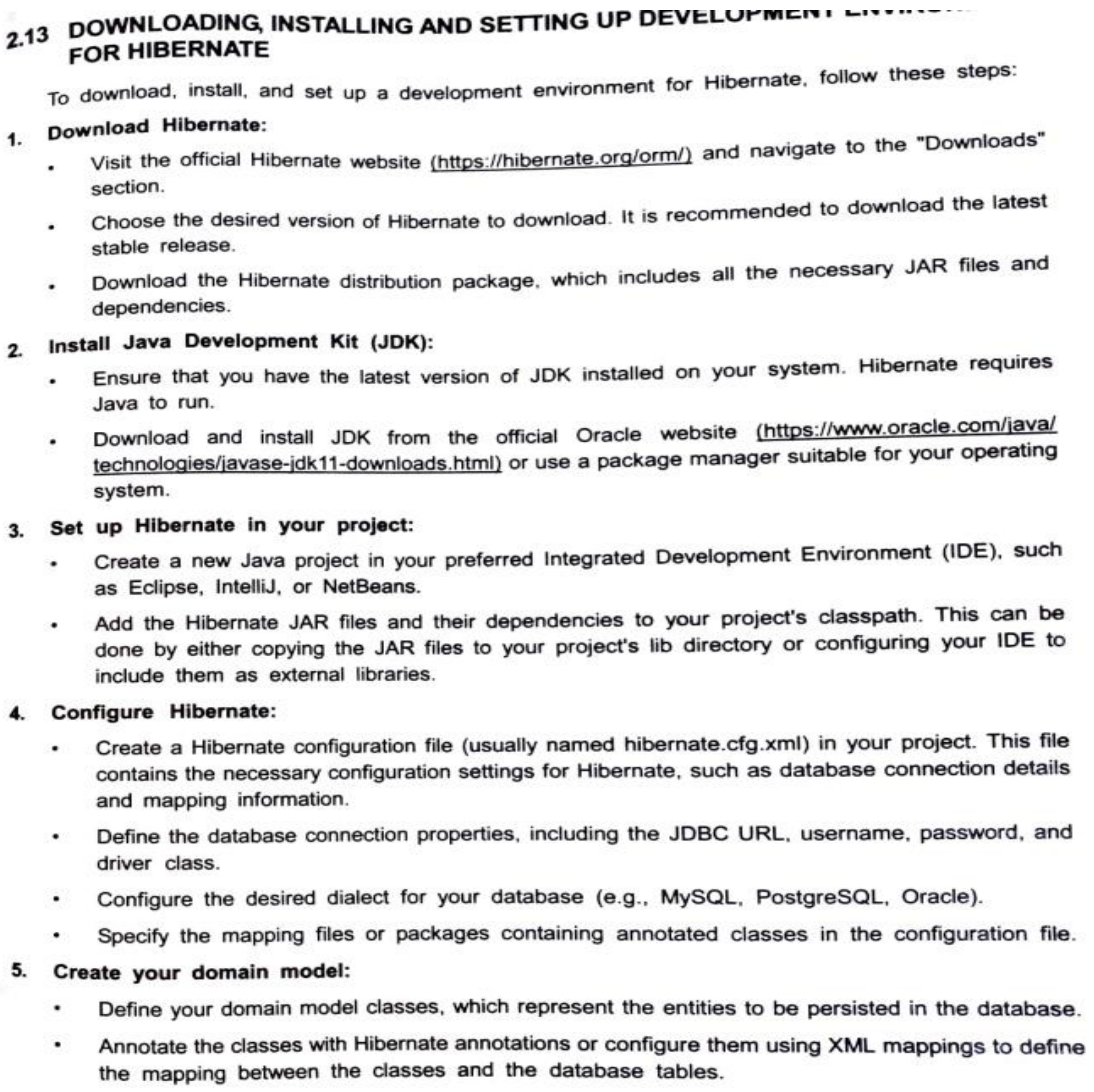
The transaction object specifies the atomic unit of work. It is optional. The org.hibernate.Transaction interface provides methods for transaction management.

ConnectionProvider

It is a factory of JDBC connections. It abstracts the application from DriverManager or DataSource. It is optional.

TransactionFactory

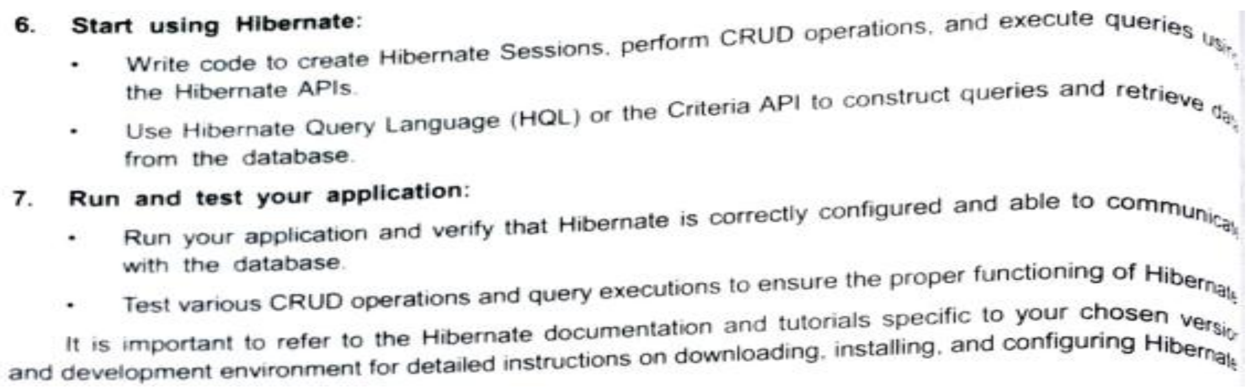
It is a factory of Transaction. It is optional.



2.13 DOWNLOADING, INSTALLING AND SETTING UP DEVELOPMENT FOR HIBERNATE

To download, install, and set up a development environment for Hibernate, follow these steps:

1. **Download Hibernate:**
   - Visit the official Hibernate website (https://hibernate.org/orm/) and navigate to the "Downloads" section.
   - Choose the desired version of Hibernate to download. It is recommended to download the latest stable release.
   - Download the Hibernate distribution package, which includes all the necessary JAR files and dependencies.
2. **Install Java Development Kit (JDK):**
   - Ensure that you have the latest version of JDK installed on your system. Hibernate requires Java to run.
   - Download and install JDK from the official Oracle website (https://www.oracle.com/java/technologies/javase-jdk11-downloads.html) or use a package manager suitable for your operating system.
3. **Set up Hibernate in your project:**
   - Create a new Java project in your preferred Integrated Development Environment (IDE), such as Eclipse, IntelliJ, or NetBeans.
   - Add the Hibernate JAR files and their dependencies to your project's classpath. This can be done by either copying the JAR files to your project's lib directory or configuring your IDE to include them as external libraries.
4. **Configure Hibernate:**
   - Create a Hibernate configuration file (usually named hibernate.cfg.xml) in your project. This file contains the necessary configuration settings for Hibernate, such as database connection details and mapping information.
   - Define the database connection properties, including the JDBC URL, username, password, and driver class.
   - Configure the desired dialect for your database (e.g., MySQL, PostgreSQL, Oracle).
   - Specify the mapping files or packages containing annotated classes in the configuration file.
5. **Create your domain model:**
   - Define your domain model classes, which represent the entities to be persisted in the database.
   - Annotate the classes with Hibernate annotations or configure them using XML mappings to define the mapping between the classes and the database tables.



6. **Start using Hibernate:**
   - Write code to create Hibernate Sessions, perform CRUD operations, and execute queries usi the Hibernate APIs.
   - Use Hibernate Query Language (HQL) or the Criteria API to construct queries and retrieve da from the database.
7. **Run and test your application:**
   - Run your application and verify that Hibernate is correctly configured and able to communica with the database.
   - Test various CRUD operations and query executions to ensure the proper functioning of Hiberna

It is important to refer to the Hibernate documentation and tutorials specific to your chosen versio and development environment for detailed instructions on downloading, installing, and configuring Hiberna

Hibernate Properties and Supported Database:

Hibernate is a powerful tool for Java developers, aiding in the management of database interactions seamlessly. Let's delve into the intricacies of Hibernate properties and the databases it supports.

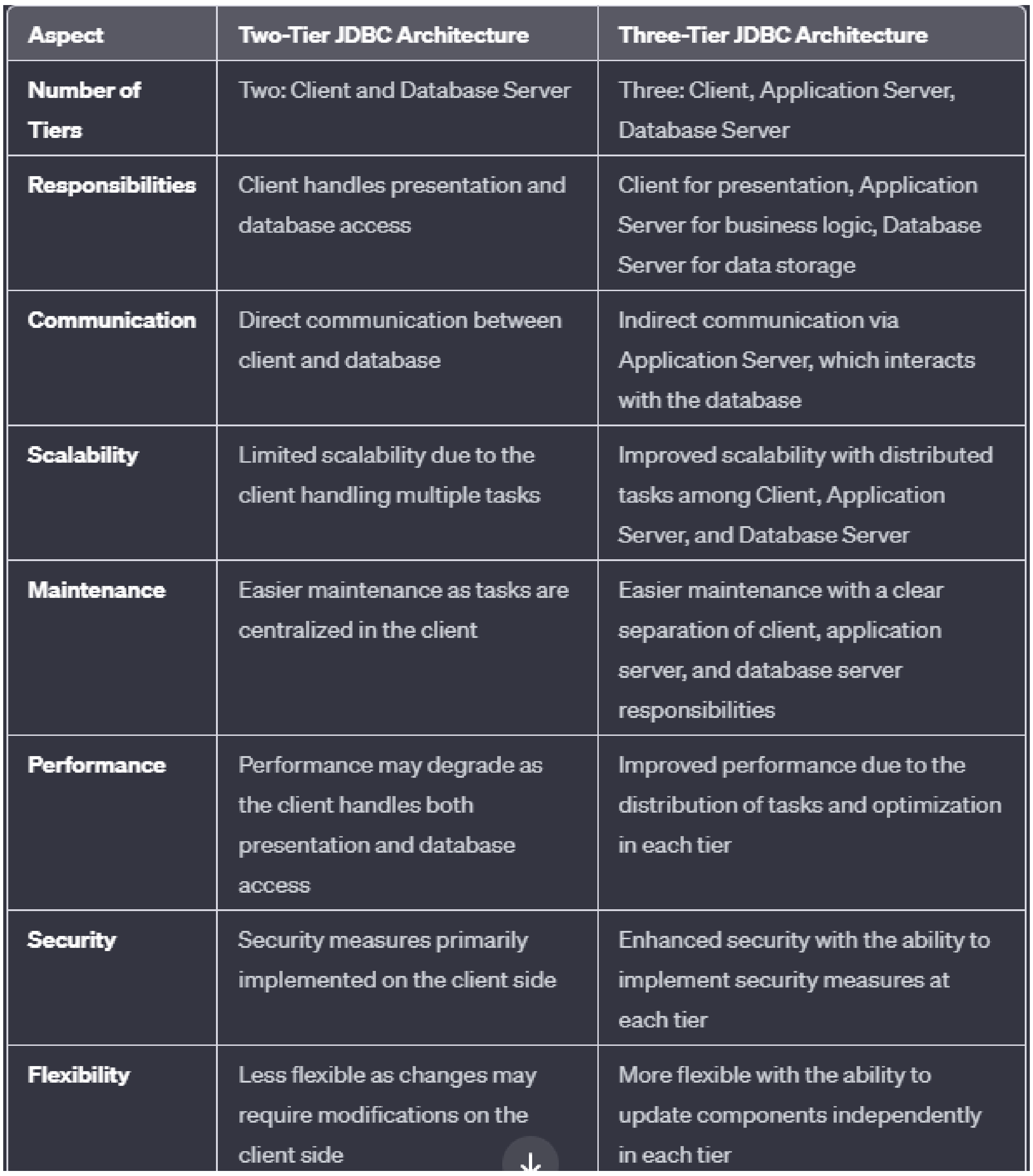
1. Hibernate Properties:

- **hibernate.cfg.xml Configuration:** Hibernate properties are often specified in the hibernate.cfg.xml file. This file acts as the configuration hub for Hibernate, housing essential settings.
- **Database Connection Properties:** Crucial properties include the database URL, username, and password. These details establish a connection between Hibernate and your database.
- **Dialect Configuration:** The dialect property defines the SQL variant for your database, ensuring compatibility and optimal query execution.
- **Show SQL Property:** Enabling the "hibernate.show_sql" property facilitates debugging by displaying executed SQL queries in the console.
- **Hibernate Caching:** Properties like "hibernate.cache.use_second_level_cache" and "hibernate.cache.use_query_cache" enhance performance by implementing caching strategies.

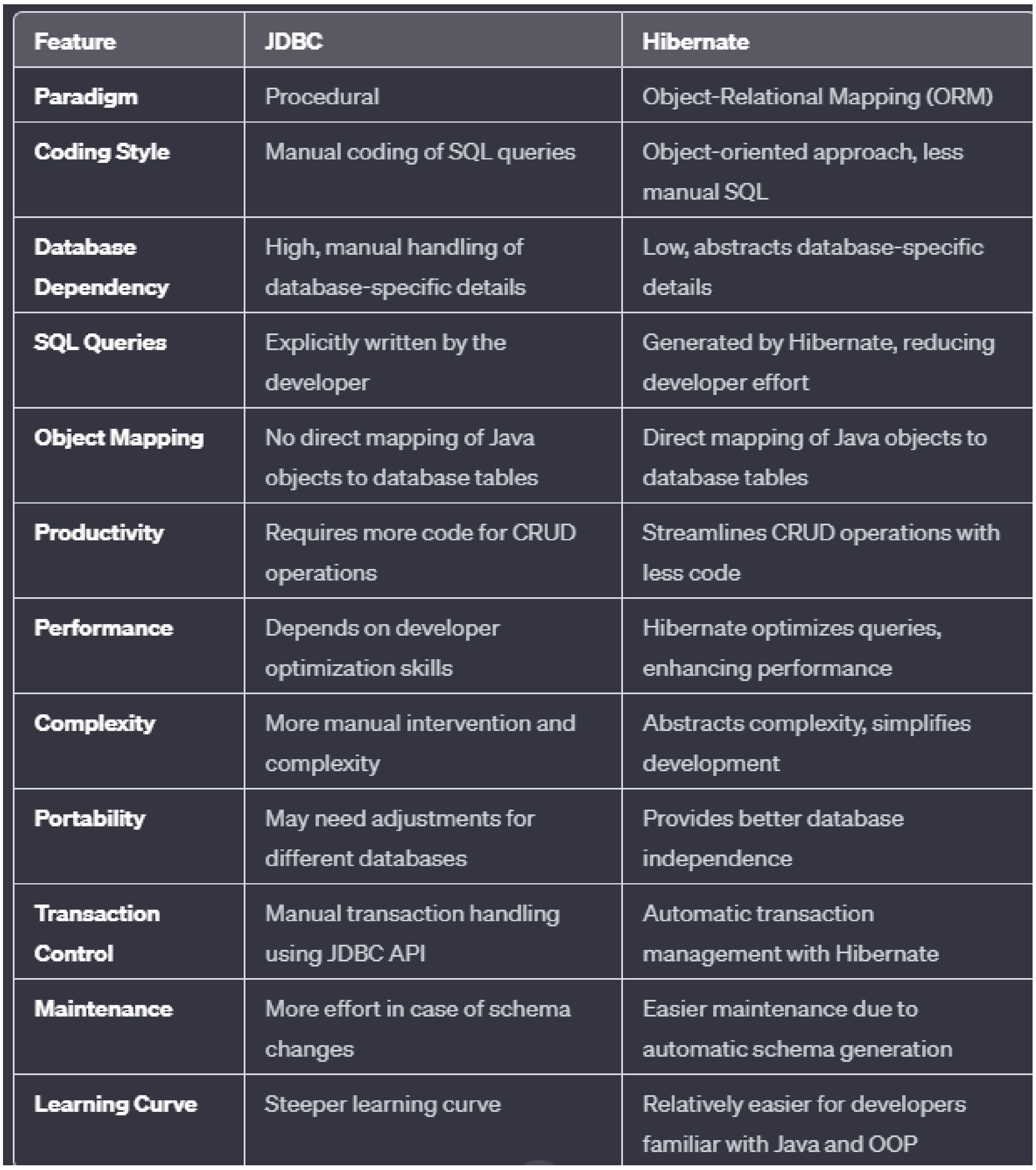
2. Supported Databases:

- **MySQL:** Hibernate seamlessly integrates with MySQL, a popular open-source relational database management system. Compatibility is achieved through appropriate dialect configuration.
- **PostgreSQL:** PostgreSQL, known for its extensibility, is another compatible database. Hibernate adapts to its unique features with ease.
- **Oracle Database:** For enterprises relying on Oracle Database, Hibernate offers robust support. Configuration parameters ensure smooth communication between Java applications and Oracle databases.
- **Microsoft SQL Server:** Hibernate caters to Microsoft SQL Server users, fostering efficient data management and retrieval through tailored configurations.

- **H2 Database:** H2, a lightweight Java-based database, is well-supported by Hibernate. This is particularly advantageous for development and testing scenarios.



Aspect	Two-Tier JDBC Architecture	Three-Tier JDBC Architecture
Number of Tiers	Two: Client and Database Server	Three: Client, Application Server, Database Server
Responsibilities	Client handles presentation and database access	Client for presentation, Application Server for business logic, Database Server for data storage
Communication	Direct communication between client and database	Indirect communication via Application Server, which interacts with the database
Scalability	Limited scalability due to the client handling multiple tasks	Improved scalability with distributed tasks among Client, Application Server, and Database Server
Maintenance	Easier maintenance as tasks are centralized in the client	Easier maintenance with a clear separation of client, application server, and database server responsibilities
Performance	Performance may degrade as the client handles both presentation and database access	Improved performance due to the distribution of tasks and optimization in each tier
Security	Security measures primarily implemented on the client side	Enhanced security with the ability to implement security measures at each tier
Flexibility	Less flexible as changes may require modifications on the client side	More flexible with the ability to update components independently in each tier



Feature	JDBC	Hibernate
Paradigm	Procedural	Object-Relational Mapping (ORM)
Coding Style	Manual coding of SQL queries	Object-oriented approach, less manual SQL
Database Dependency	High, manual handling of database-specific details	Low, abstracts database-specific details
SQL Queries	Explicitly written by the developer	Generated by Hibernate, reducing developer effort
Object Mapping	No direct mapping of Java objects to database tables	Direct mapping of Java objects to database tables
Productivity	Requires more code for CRUD operations	Streamlines CRUD operations with less code
Performance	Depends on developer optimization skills	Hibernate optimizes queries, enhancing performance
Complexity	More manual intervention and complexity	Abstracts complexity, simplifies development
Portability	May need adjustments for different databases	Provides better database independence
Transaction Control	Manual transaction handling using JDBC API	Automatic transaction management with Hibernate
Maintenance	More effort in case of schema changes	Easier maintenance due to automatic schema generation
Learning Curve	Steeper learning curve	Relatively easier for developers familiar with Java and OOP

1. Type 1: JDBC-ODBC Bridge Driver

- Bridges JDBC to ODBC.
- ODBC driver acts as an intermediary.
- Platform-independent.
- Relatively slower due to double translation.
- Requires ODBC driver installation.
- Limited functionality for advanced databases.

- Suitable for small-scale applications.

2. Type 2: Native-API Driver

- Utilizes database-specific API provided by vendors.
- Direct interaction with the database API.
- Faster performance than Type 1.
- Platform-dependent, requires native library.
- Still requires database-specific coding.
- Maintenance and updates depend on the database API.
- Improved performance over Type 1.

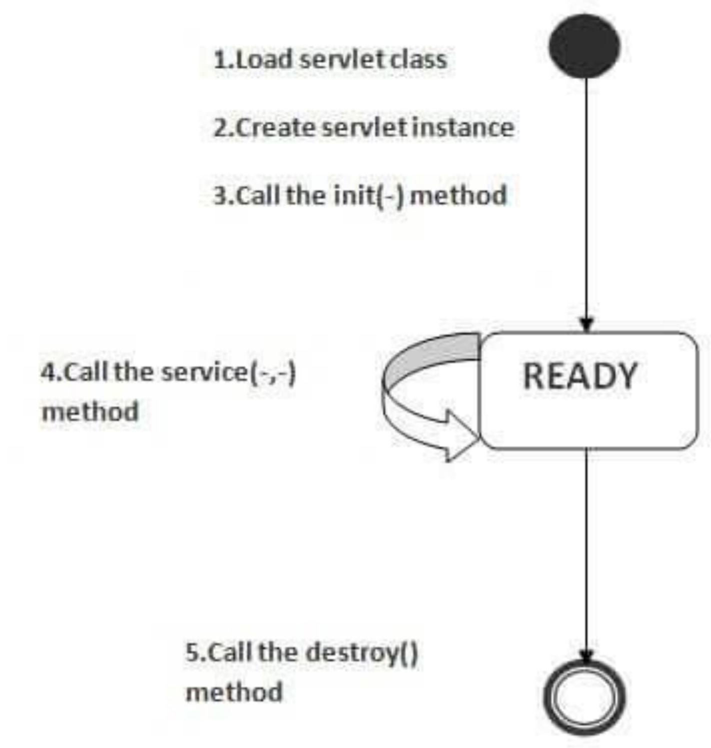
3. Type 3: Network Protocol Driver

- Converts JDBC calls to a database-independent network protocol.
- Provides a middle-tier server for translation.
- Database-independent.
- Slower than Type 2 due to an additional layer.
- Requires a middleware server.
- Suitable for large-scale applications.
- Supports multiple databases.

4. Type 4: Thin Driver (JDBC Net-Protocol Pure Java Driver)

- Pure Java driver that communicates directly with the database server.
- Suitable for internet applications.
- Platform-independent.
- High performance compared to other types.
- No client-side installation or configuration.
- Requires a specific driver for each database.
- Limited support for legacy databases.

Ch -3


1.Load servlet class
2.Create servlet instance
3.Call the init(-) method
4.Call the service(-,-) method
READY
5.Call the destroy() method

1. Servlet class is loaded.
2. Servlet instance is created.
3. init method is invoked.
4. service method is invoked.
5. destroy method is invoked

1) Servlet class is loaded

The classloader is responsible to load the servlet class. The servlet class is loaded when the first request for the servlet is received by the web container.

---

2) Servlet instance is created

The web container creates the instance of a servlet after loading the servlet class. The servlet instance is created only once in the servlet life cycle.

---

3) init method is invoked

The web container calls the init method only once after creating the servlet instance. The init method is servlet. It is the life cycle method of the javax.servlet.Servlet interface. Syntax of the init method is given belo

1. **public void** init(ServletConfig config) **throws** ServletException

---

4) service method is invoked

The web container calls the service method each time when request for the servlet is received. If servlet is not initialized, it follows the first three steps as described above then calls the service method. If servlet is initialized, it calls the service method. Notice that servlet is initialized only once. The syntax of the service method of the Servlet interface is given below:

1. **public void** service(ServletRequest request, ServletResponse response)
2. **throws** ServletException, IOException

---

5) destroy method is invoked

The web container calls the destroy method before removing the servlet instance from the service. It gives the servlet an opportunity to clean up any resource for example memory, thread etc. The syntax of the destroy method of the Servlet interface is given below:

1. **public void** destroy()

create and compile servlet source code

3. Writing Servlet Code:**

Create a new Java class that extends HttpServlet. Override methods like `doGet()` or `doPost()` to handle specific HTTP requests.

```java

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import java.io.IOException;

import java.io.PrintWriter;
```

```
  public class MyServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {
      // Your code here
      PrintWriter out = response.getWriter();
      out.println("Hello, Servlet!");
    }
  }
```

4. Compiling Servlet Code:

After writing your servlet code, compile it using the Java compiler (`javac`). Ensure that your servlet class file is in the correct directory structure.

```
  javac MyServlet.java
```

This generates the corresponding `.class` file.

5. Creating Web Deployment Descriptor (web.xml):

Define your servlet in the web deployment descriptor to let the server know about your servlet class.

```xml
  <servlet>
    <servlet-name>MyServlet</servlet-name>
    <servlet-class>com.example.MyServlet</servlet-class>
  </servlet>
  <servlet-mapping>
      <servlet-name>MyServlet</servlet-name>
      <url-pattern>/hello</url-pattern>
  </servlet-mapping>
```

6. Deploying to a Servlet Container:

Deploy your servlet to a servlet container like Apache Tomcat. Copy the compiled `.class` file and the `web.xml` to the appropriate directories.

7. Accessing the Servlet:

Open your web browser and navigate to the specified URL, for example, `http://localhost:8080/yourwebapp/hello`.

8. Output:

You should see the output generated by your servlet, in this case, "Hello, Servlet!"

start a web browser and request the servlet

Certainly! It seems like you're asking for an explanation of how to start a web browser, request a servlet, and include code and output. Let's break this down:

1. Starting a Web Browser:

To initiate a web browser, you need to click on its icon or search for it in your computer's applications. Common web browsers include Chrome, Firefox, Safari, or Edge. Once opened, you can type a web address (URL) into the address bar and press Enter to navigate to a website.

2. Requesting a Servlet:

Servlets are Java-based programs that extend the capabilities of a server. To request a servlet:

a. **Server Configuration:**

- Ensure that you have a web server (like Apache Tomcat) installed.

- Configure the server to recognize your servlet.

b. **Servlet Code:**

- Write your servlet code in Java. This code handles HTTP requests and generates responses.

- A simple example might involve extending the HttpServlet class and overriding the doGet() or doPost() method.

c. **Compile and Deploy:**

- Compile your servlet code to create a .class file.

- Deploy the compiled servlet to the web server.

d. **URL Mapping:**

- Map your servlet to a URL pattern in the web.xml file or using annotations.

e. **Request from Browser:**

- In the web browser, enter the URL mapped to your servlet. For example, if your servlet is mapped to "/MyServlet," type "http://localhost:8080/YourWebAppName/MyServlet" in the address bar.

3. Code and Output:

a. **Servlet Code Example:**

```java
import java.io.IOException;
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("/MyServlet")
public class MyServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        // Your servlet code here
        response.getWriter().write("Hello, this is my servlet!");
    }
}
```

b. **Output:**

- After requesting the servlet in the browser, you should see the output generated by your servlet, which in this case is "Hello, this is my servlet!"

example of echo servlet and deployment in tomcat server

I can certainly help explain topics in detail, but keep in mind that the depth of explanation may vary based on the complexity of the subject. Let's start with an example, and feel free to ask for more details on specific topics afterward.

Example: Echo Servlet and Deployment in Tomcat Server

1. Introduction to Servlets:

Servlets are Java programs that run on the server side to dynamically process and respond to requests from clients, usually web browsers. The Echo Servlet is a simple example that echoes back the client's request.

```java
   // EchoServlet.java
   import java.io.IOException;
   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("/EchoServlet")
   public class EchoServlet extends HttpServlet {
       protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
           // Retrieve client input
           String userInput = request.getParameter("input");

           // Echo back to the client
           response.getWriter().write("You entered: " + userInput);
       }
```

  }

2. Deployment in Tomcat Server:

  Tomcat is a popular open-source servlet container that implements the Java Servlet and JavaServer Pages (JSP) technologies. To deploy our Echo Servlet:

  - **Step 1: Compile Servlet:**
    ```bash
    javac -cp TOMCAT_HOME/lib/servlet-api.jar EchoServlet.java
    ```

  - **Step 2: Create WAR File:**
    ```bash
    jar cf EchoServlet.war EchoServlet.class
    ```

  - **Step 3: Deploy WAR File:**

    Copy the `EchoServlet.war` file to the `webapps` directory of your Tomcat installation.

  - **Step 4: Start Tomcat:**

    Navigate to the `bin` directory and run `catalina.bat run` (Windows) or `catalina.sh run` (Unix).

  - **Step 5: Access the Servlet:**

    Open a web browser and go to `http://localhost:8080/EchoServlet`.

3. Output:

  Once the server is running, and you access the servlet through the browser, you'll see a page prompting you to enter some text. After entering text and submitting, the page will display: "You entered: [your input]".

Alright, let's dive into the javax.servlet package and how to read database/table records and display them using a servlet.

javax.servlet Package: Reading Database/Table Records

The `javax.servlet` package provides classes and interfaces to create web-based applications in Java. In this context, we'll explore reading database/table records and displaying them using a servlet.

1. Establishing Database Connection:

  To begin, establish a connection to your database. Use JDBC (Java Database Connectivity) to interact with the database. Create a connection object, specifying the database URL, username, and password.

  ```java
  Connection connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/yourdatabase", "username", "password");
  ```

2. Executing SQL Queries:

  Once connected, use the `Statement` interface to execute SQL queries. Retrieve data from the database by executing a SELECT query.

  ```java
  Statement statement = connection.createStatement();
  ResultSet resultSet = statement.executeQuery("SELECT * FROM yourtable");
  ```

3. Processing Results:

  Process the results obtained from the database. Iterate through the `ResultSet` to access each record's data.

3.4 The javax.servlet Package: reading database/table records and displaying them using servlet

```java
while (resultSet.next()) {
    String name = resultSet.getString("name");
    int age = resultSet.getInt("age");
    // Process and store the data as needed
}
```

4. Creating Servlet:

Now, integrate this functionality into a servlet. Extend the `HttpServlet` class, and override the `doGet` method.

```java
public class DatabaseServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
        // Database operations as mentioned above
        // Display data or forward to a JSP page for rendering
    }
}
```

5. Output/Display:

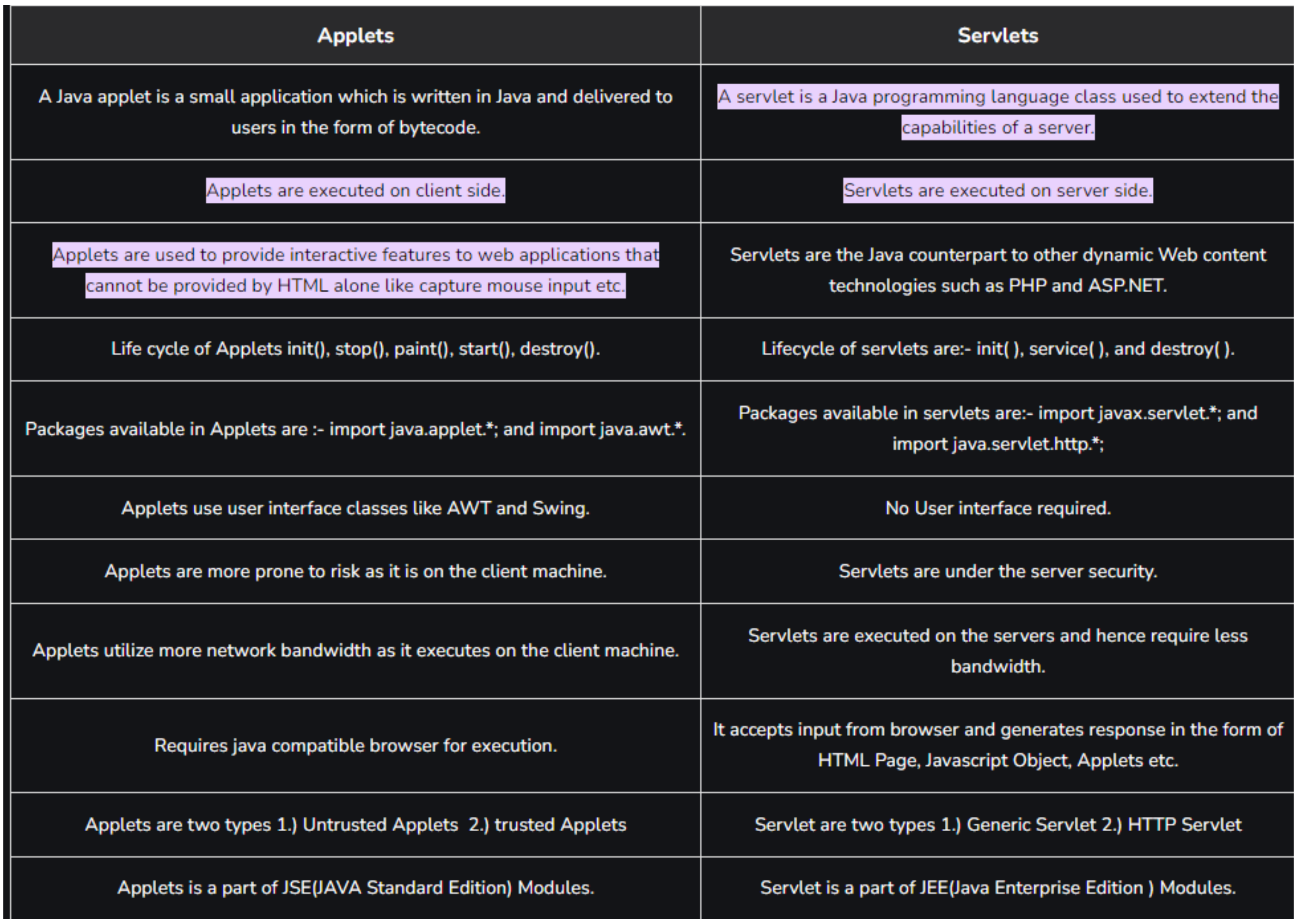
Depending on your application, you can choose to display the data in various ways. For simplicity, let's forward the data to a JSP (JavaServer Pages) for rendering.

```java
RequestDispatcher dispatcher = request.getRequestDispatcher("display.jsp");
request.setAttribute("data", dataList); // Assuming you have a List to store data
dispatcher.forward(request, response);
```

6. JSP Rendering:

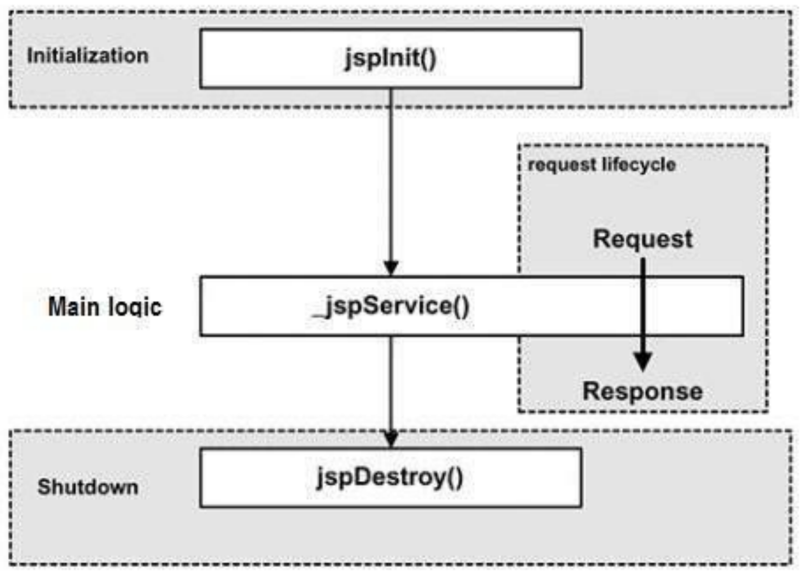
  Create a JSP page (`display.jsp`) to render the data received from the servlet.

  ```jsp
  <html>
  <body>
     <h2>User Data</h2>
     <%
        List<User> dataList = (List<User>) request.getAttribute("data");
        for (User user : dataList) {
     %>
     <p>Name: <%= user.getName() %></p>
     <p>Age: <%= user.getAge() %></p>
     <% } %>
  </body>
  </html>
  ```



Applets	Servlets
A Java applet is a small application which is written in Java and delivered to users in the form of bytecode.	A servlet is a Java programming language class used to extend the capabilities of a server.
Applets are executed on client side.	Servlets are executed on server side.
Applets are used to provide interactive features to web applications that cannot be provided by HTML alone like capture mouse input etc.	Servlets are the Java counterpart to other dynamic Web content technologies such as PHP and ASP.NET.
Life cycle of Applets init(), stop(), paint(), start(), destroy().	Lifecycle of servlets are:- init( ), service( ), and destroy( ).
Packages available in Applets are :- import java.applet.*; and import java.awt.*.	Packages available in servlets are:- import javax.servlet.*; and import java.servlet.http.*;
Applets use user interface classes like AWT and Swing.	No User interface required.
Applets are more prone to risk as it is on the client machine.	Servlets are under the server security.
Applets utilize more network bandwidth as it executes on the client machine.	Servlets are executed on the servers and hence require less bandwidth.
Requires java compatible browser for execution.	It accepts input from browser and generates response in the form of HTML Page, Javascript Object, Applets etc.
Applets are two types 1.) Untrusted Applets 2.) trusted Applets	Servlet are two types 1.) Generic Servlet 2.) HTTP Servlet
Applets is a part of JSE(JAVA Standard Edition) Modules.	Servlet is a part of JEE(Java Enterprise Edition) Modules.

CH -4



JSP Compilation

When a browser asks for a JSP, the JSP engine first checks to see whether it needs to compile the page. If the page has never been compiled, or if the JSP has been modified since it was last compiled, the JSP engine compiles the page.

The compilation process involves three steps −

- Parsing the JSP.
- Turning the JSP into a servlet.
- Compiling the servlet.

JSP Initialization

When a container loads a JSP it invokes the **jspInit()** method before servicing any requests. If you need to perform JSP-specific initialization, override the **jspInit()** method −

```
public void jspInit(){
   // Initialization code...
}
```

Typically, initialization is performed only once and as with the servlet init method, you generally initialize database connections, open files, and create lookup tables in the jspInit method.

JSP Execution

This phase of the JSP life cycle represents all interactions with requests until the JSP is destroyed.

Whenever a browser requests a JSP and the page has been loaded and initialized, the JSP engine invokes the **_jspService()** method in the JSP.

The _jspService() method takes an **HttpServletRequest** and an **HttpServletResponse** as its parameters as follows −

```
void _jspService(HttpServletRequest request,
HttpServletResponse response) {
   // Service handling code...
}
```

The **_jspService()** method of a JSP is invoked on request basis. This is responsible for generating the response for that request and this method is also responsible for generating responses to all seven of the HTTP methods, i.e, **GET, POST, DELETE**, etc.

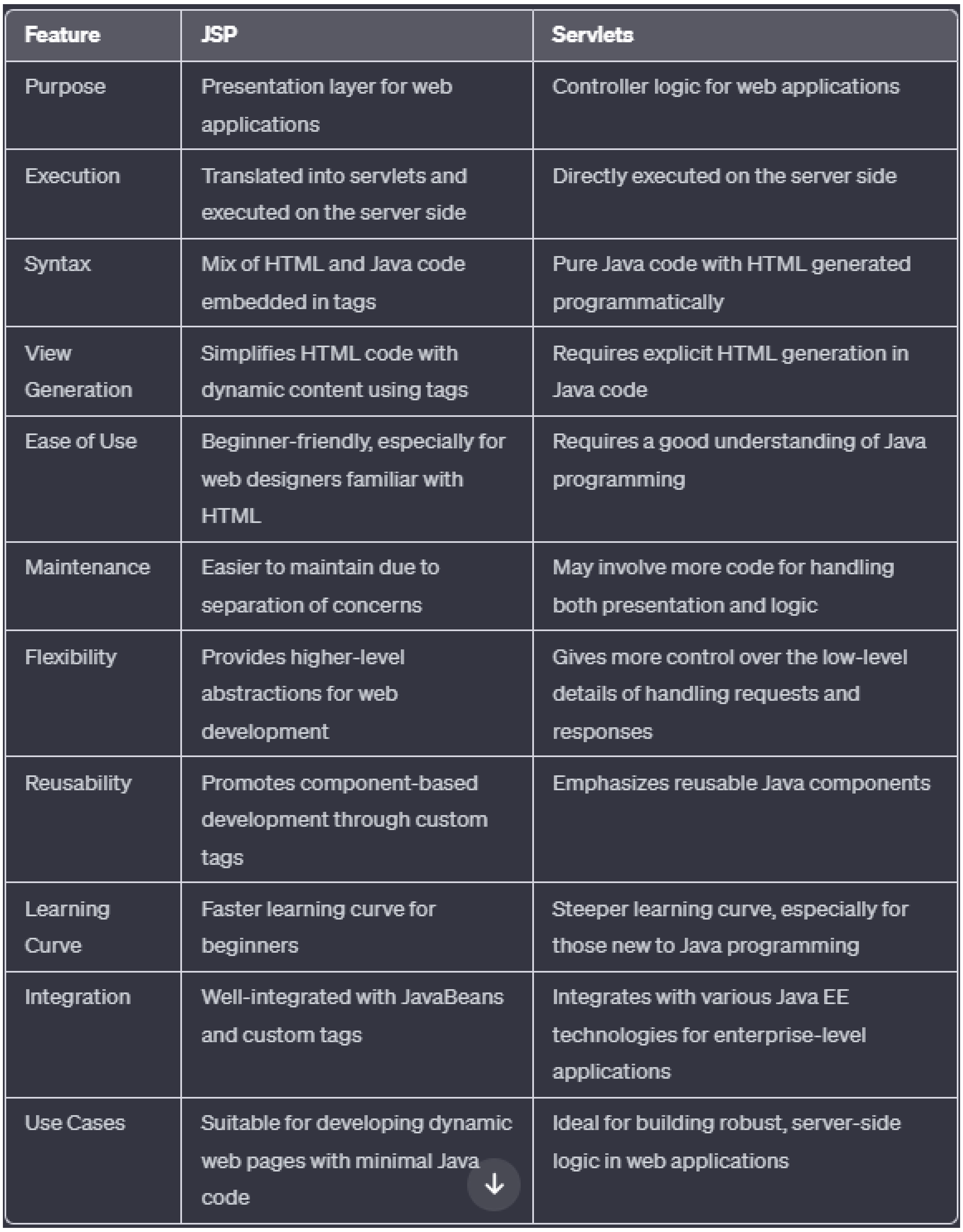
JSP Cleanup

The destruction phase of the JSP life cycle represents when a JSP is being removed from use by a container.

The **jspDestroy()** method is the JSP equivalent of the destroy method for servlets. Override jspDestroy when you need to perform any cleanup, such as releasing database connections or closing open files.

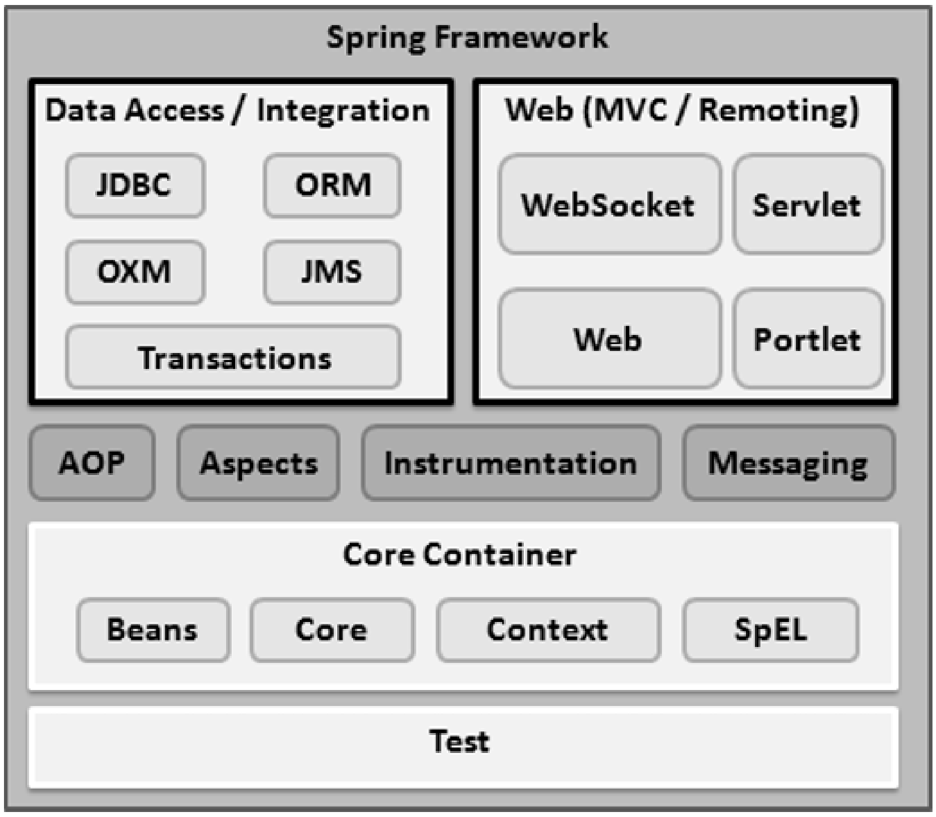
The jspDestroy() method has the following form −

```
public void jspDestroy() {
   // Your cleanup code goes here.
}
```



Feature	JSP	Servlets
Purpose	Presentation layer for web applications	Controller logic for web applications
Execution	Translated into servlets and executed on the server side	Directly executed on the server side
Syntax	Mix of HTML and Java code embedded in tags	Pure Java code with HTML generated programmatically
View Generation	Simplifies HTML code with dynamic content using tags	Requires explicit HTML generation in Java code
Ease of Use	Beginner-friendly, especially for web designers familiar with HTML	Requires a good understanding of Java programming
Maintenance	Easier to maintain due to separation of concerns	May involve more code for handling both presentation and logic
Flexibility	Provides higher-level abstractions for web development	Gives more control over the low-level details of handling requests and responses
Reusability	Promotes component-based development through custom tags	Emphasizes reusable Java components
Learning Curve	Faster learning curve for beginners	Steeper learning curve, especially for those new to Java programming
Integration	Well-integrated with JavaBeans and custom tags	Integrates with various Java EE technologies for enterprise-level applications
Use Cases	Suitable for developing dynamic web pages with minimal Java code	Ideal for building robust, server-side logic in web applications

Ch-5



Core Container

The Core Container consists of the Core, Beans, Context, and Expression Language modules the details of which are as follows –

- The **Core** module provides the fundamental parts of the framework, including the IoC and Dependency Injection features.
- The **Bean** module provides BeanFactory, which is a sophisticated implementation of the factory pattern.
- The **Context** module builds on the solid base provided by the Core and Beans modules and it is a medium to access any objects defined and configured. The ApplicationContext interface is the focal point of the Context module.
- The **SpEL** module provides a powerful expression language for querying and manipulating an object graph at runtime.

Data Access/Integration

The Data Access/Integration layer consists of the JDBC, ORM, OXM, JMS and Transaction modules whose detail is as follows −

- The **JDBC** module provides a JDBC-abstraction layer that removes the need for tedious JDBC related coding.
- The **ORM** module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis.
- The **OXM** module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.
- The Java Messaging Service **JMS** module contains features for producing and consuming messages.
- The **Transaction** module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs.

Web

The Web layer consists of the Web, Web-MVC, Web-Socket, and Web-Portlet modules the details of which are as follows −

- The **Web** module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context.
- The **Web-MVC** module contains Spring's Model-View-Controller (MVC) implementation for web applications.
- The **Web-Socket** module provides support for WebSocket-based, two-way communication between the client and the server in web applications.
- The **Web-Portlet** module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

Miscellaneous

There are few other important modules like AOP, Aspects, Instrumentation, Web and Test modules the details of which are as follows −

- The **AOP** module provides an aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.
- The **Aspects** module provides integration with AspectJ, which is again a powerful and mature AOP framework.
- The **Instrumentation** module provides class instrumentation support and class loader implementations to be used in certain application servers.
- The **Messaging** module provides support for STOMP as the WebSocket sub-protocol to use in applications. It also supports an annotation programming model for routing and processing STOMP messages from WebSocket clients.
- The **Test** module supports the testing of Spring components with JUnit or TestNG frameworks.

Features of sping boot

1. **Spring Boot Auto-Configuration:

  - *Automatic Setup:* Spring Boot simplifies configuration by automatically configuring application components based on project dependencies.

  - *Convention over Configuration:* Adheres to the principle of convention over configuration, reducing the need for extensive setup.

2. **Embeddable Web Server:

  - *Integrated Servers:* Spring Boot comes with embedded servers like Tomcat, Jetty, or Undertow, eliminating the need for external server setup.

  - *Easy Deployment:* Simplifies deployment with standalone JAR files, making it convenient for microservices architecture.

3. **Spring Boot Starters:

  - *Ready-Made Dependencies:* Starters provide pre-configured templates for common use cases (e.g., web, data, security), streamlining project setup.

  - *Saves Development Time:* Reduces development time by handling dependencies and configurations.

4. **Spring Boot Actuator:

  - *Production-Ready Features:* Actuator offers production-ready features like health checks, metrics, and monitoring out of the box.

  - *Easy Monitoring:* Enables monitoring and managing applications through built-in endpoints, enhancing operational visibility.

5. **Spring Boot DevTools:

  - *Automatic Restart:* DevTools facilitate automatic application restart during development, accelerating the testing and debugging process.

  - *Live Reload:* Supports live reload, allowing changes to be instantly reflected without restarting the entire application.

6. **Spring Boot CLI (Command Line Interface):

  - *Command-Line Development:* Offers a command-line interface for rapid development and testing.

  - *Groovy Support:* Allows writing application code in Groovy, promoting concise and expressive syntax.

7. **Spring Boot Data JPA:

  - *Data Access Simplified:* Integrates with Spring Data JPA for simplified data access and database operations.

  - *Annotation-Based Mapping:* Utilizes annotations to define data models and relationships, reducing boilerplate code.

8. **Spring Boot Security:

  - *Built-In Security Features:* Provides robust security configurations by default, including authentication and authorization.

  - *Customization Options:* Allows customization to meet specific security requirements of applications.

9. **Spring Boot Testing:

  - *Test Support:* Offers a comprehensive suite of testing tools and annotations for unit, integration, and end-to-end testing.

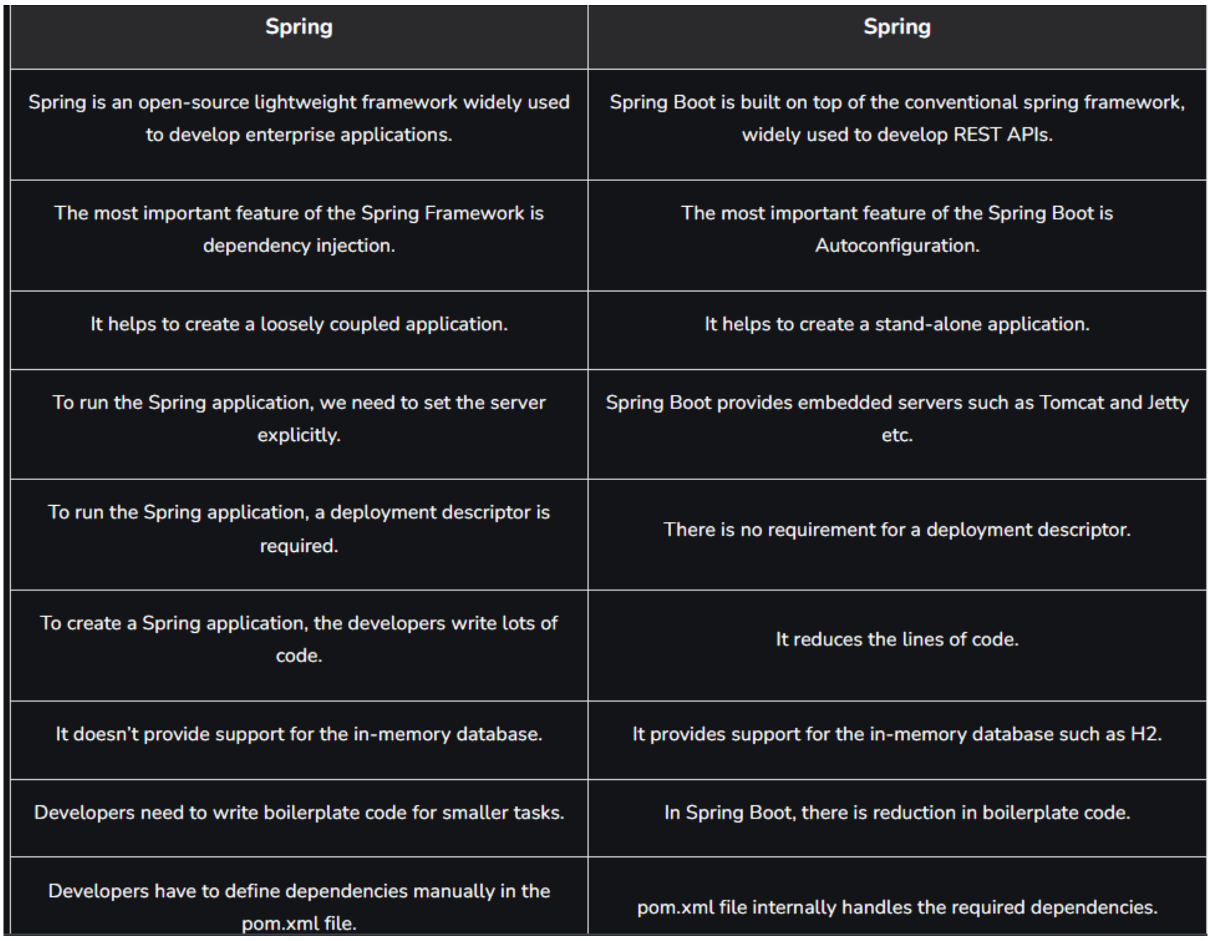
  - *Embedded Database Support:* Supports embedded databases for testing, ensuring a consistent and isolated testing environment.

10. **Spring Boot Externalized Configuration:

- *Configuration Management:* Externalizes configuration properties, allowing easy adjustment without code changes.

- *Profile Support:* Supports different profiles for various deployment environments (e.g., development, production).

Understanding these features empowers developers to leverage Spring Boot's capabilities effectively, fostering efficient and scalable application development.



Spring	Spring
Spring is an open-source lightweight framework widely used to develop enterprise applications.	Spring Boot is built on top of the conventional spring framework, widely used to develop REST APIs.
The most important feature of the Spring Framework is dependency injection.	The most important feature of the Spring Boot is Autoconfiguration.
It helps to create a loosely coupled application.	It helps to create a stand-alone application.
To run the Spring application, we need to set the server explicitly.	Spring Boot provides embedded servers such as Tomcat and Jetty etc.
To run the Spring application, a deployment descriptor is required.	There is no requirement for a deployment descriptor.
To create a Spring application, the developers write lots of code.	It reduces the lines of code.
It doesn't provide support for the in-memory database.	It provides support for the in-memory database such as H2.
Developers need to write boilerplate code for smaller tasks.	In Spring Boot, there is reduction in boilerplate code.
Developers have to define dependencies manually in the pom.xml file.	pom.xml file internally handles the required dependencies.

Ch 1

Sure, I can provide you with a brief explanation and example code for each of these components in Java Swing:

1. **JApplet:**

   - JApplet is a class that represents an applet, which is a small Java program that runs within a web browser.

   - Example code:

   ```java
   import javax.swing.JApplet;

   public class MyApplet extends JApplet {
       // Applet code goes here
   }
   ```

2. **JFrame:**

   - JFrame is a class for creating a top-level container (window) in a Swing application.

   - Example code:

   ```java
   import javax.swing.JFrame;

   public class MyFrame extends JFrame {
       public MyFrame() {
           // Frame initialization code goes here
       }
   }
   ```

3. **JLabel:**

- JLabel is a class for displaying text or an image.

- Example code:

```java
import javax.swing.JLabel;

JLabel myLabel = new JLabel("Hello, Swing!");
```

4. **JTextField:**

- JTextField is a class for creating a single-line text input field.

- Example code:

```java
import javax.swing.JTextField;

JTextField textField = new JTextField("Default Text");
```

5. **JTextArea:**

- JTextArea is a class for creating a multi-line text area.

- Example code:

```java
import javax.swing.JTextArea;

JTextArea textArea = new JTextArea("Default Text");
```

6. **JButton:**

- JButton is a class for creating a button.

   - Example code:
     ```java
     import javax.swing.JButton;

     JButton myButton = new JButton("Click me");
     ```

7. **JCheckBox:**
   - JCheckBox is a class for creating a checkbox.
   - Example code:
     ```java
     import javax.swing.JCheckBox;

     JCheckBox checkBox = new JCheckBox("Check me");
     ```

8. **JRadioButton:**
   - JRadioButton is a class for creating a radio button.
   - Example code:
     ```java
     import javax.swing.JRadioButton;

     JRadioButton radioButton = new JRadioButton("Select me");
     ```

9. **JComboBox:**
   - JComboBox is a class for creating a drop-down combo box.
   - Example code:
     ```java
     ```

import javax.swing.JComboBox;

String[] options = {"Option 1", "Option 2", "Option 3"};

JComboBox<String> comboBox = new JComboBox<>(options);

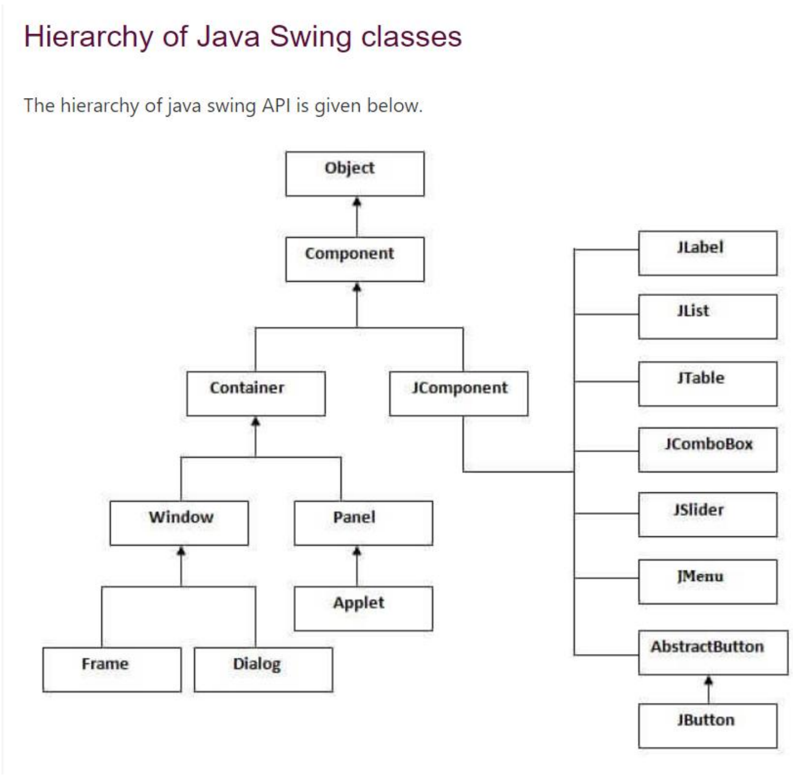
10. **JMenu:**

- JMenu is a class for creating a menu.

- Example code:

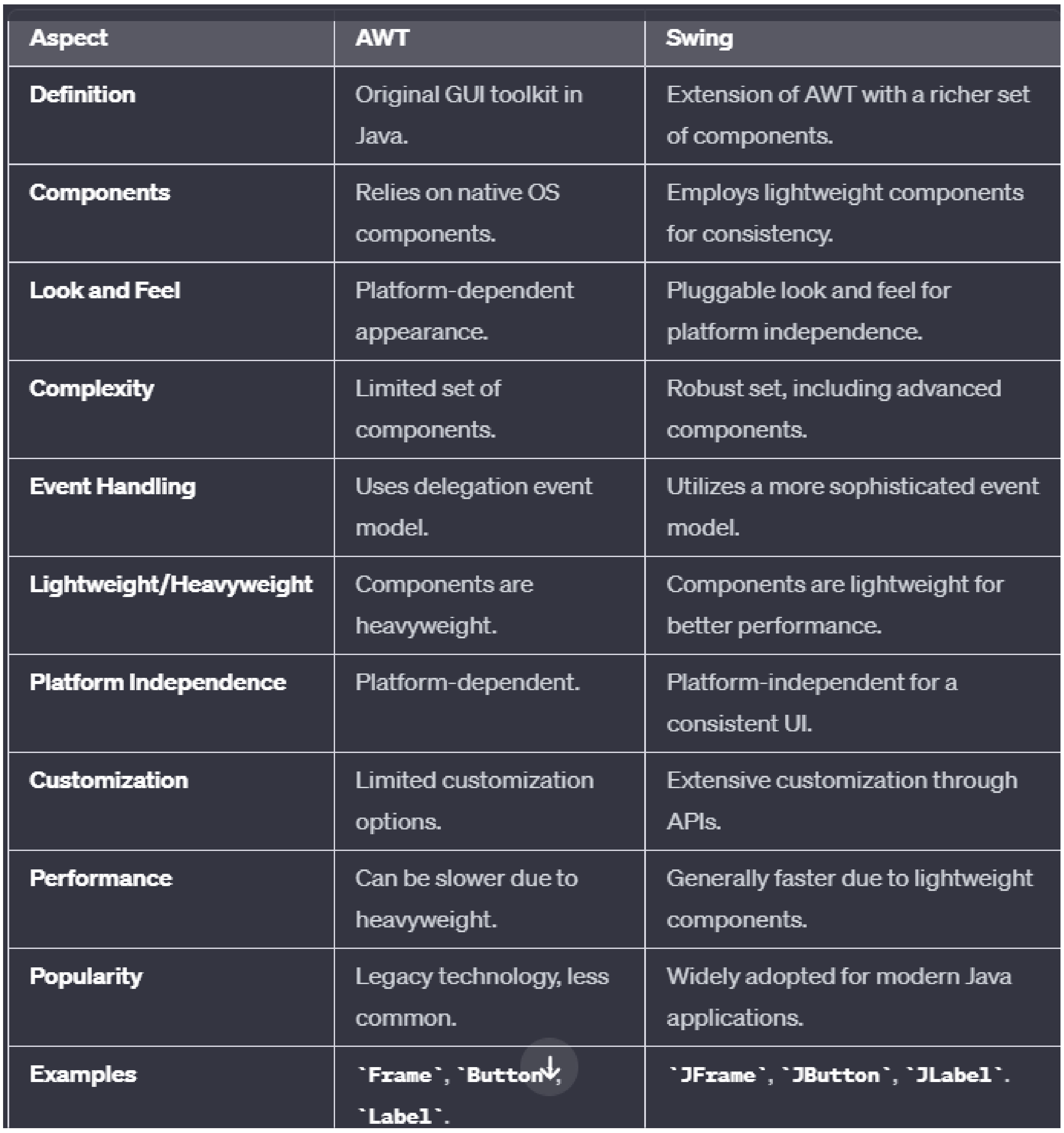
```java
import javax.swing.JMenu;

JMenu menu = new JMenu("File");
```

These examples provide a basic understanding of how to create and use each component. You can customize their properties and add them to your GUI as needed.



Hierarchy of Java Swing classes

The hierarchy of java swing API is given below.



Aspect	AWT	Swing
Definition	Original GUI toolkit in Java.	Extension of AWT with a richer set of components.
Components	Relies on native OS components.	Employs lightweight components for consistency.
Look and Feel	Platform-dependent appearance.	Pluggable look and feel for platform independence.
Complexity	Limited set of components.	Robust set, including advanced components.
Event Handling	Uses delegation event model.	Utilizes a more sophisticated event model.
Lightweight/Heavyweight	Components are heavyweight.	Components are lightweight for better performance.
Platform Independence	Platform-dependent.	Platform-independent for a consistent UI.
Customization	Limited customization options.	Extensive customization through APIs.
Performance	Can be slower due to heavyweight.	Generally faster due to lightweight components.
Popularity	Legacy technology, less common.	Widely adopted for modern Java applications.
Examples	`` `Frame` ``, `` `Button` ``, `` `Label` ``.	`` `JFrame` ``, `` `JButton` ``, `` `JLabel` ``.

Certainly! Let me provide you with a brief explanation along with code examples for each section.

3.2 The Java Servlet Development Kit:

Java Servlet Development Kit (JSDK) is a software development package for creating Java servlets. Servlets are Java programs that run on a web server and respond to client requests. To work with servlets, you need to set up your development environment with Java and a servlet container like Apache Tomcat.

File Structure:
```
- MyServletApp
  - WEB-INF
    - classes
      - MyServlet.class
    - lib
  - index.html
  - web.xml
```

3.3 The Simple Servlet:

Create a simple servlet named `MyServlet` that echoes the client's request back to the browser.

MyServlet.java:

```java
import java.io.IOException;
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("/MyServlet")
public class MyServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        String message = request.getParameter("message");
        response.getWriter().println("Echo from servlet: " + message);
    }
}
```

#### index.html:

```html
<!DOCTYPE html>
<html>
<head>
    <title>Servlet Example</title>
</head>
<body>
```

    <form action="MyServlet" method="get">
        <input type="text" name="message" placeholder="Enter your message">
        <input type="submit" value="Submit">
    </form>
</body>
</html>
~~~

3.4 The javax.servlet Package:

Extend your servlet to read database records and display them.

MyDatabaseServlet.java:
```java
import java.io.IOException;
import java.io.PrintWriter;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
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("/MyDatabaseServlet")

public class MyDatabaseServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {

        String jdbcUrl = "jdbc:mysql://localhost:3306/mydatabase";

        String user = "username";

        String password = "password";


        try {

            Connection connection = DriverManager.getConnection(jdbcUrl, user, password);

            Statement statement = connection.createStatement();

            ResultSet resultSet = statement.executeQuery("SELECT * FROM mytable");


            PrintWriter out = response.getWriter();

            out.println("<html><body>");

            while (resultSet.next()) {

                out.println("ID: " + resultSet.getInt("id") + ", Name: " + resultSet.getString("name") +
"<br>");

            }

            out.println("</body></html>");


            resultSet.close();

            statement.close();

            connection.close();

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

Ensure you have the appropriate JDBC driver in your `WEB-INF/lib` directory to connect to your database.

This example assumes you have a MySQL database named `mydatabase` with a table named `mytable` containing columns `id` and `name`.

Remember to configure your `web.xml` file to map servlets to their respective URLs.

I hope this helps! Let me know if you need further assistance!

4.1 Advantages of JSP (JavaServer Pages):

1. **Simplicity**: JSP allows embedding Java code directly into HTML, making it easier for web developers to work with both Java and HTML in the same file.

2. **Java Integration**: JSP pages are compiled into servlets, which means they can benefit from Java's robust features and libraries. Java code can be easily integrated into JSP pages, enabling the use of custom Java objects and methods.

3. **Reusable Components**: JavaBeans can be utilized within JSP, enabling the creation of reusable components that can be easily integrated into different pages.

4. **Separation of Concerns**: JSP promotes the separation of concerns by allowing web designers to focus on the presentation layer (HTML and CSS) while developers work on the business logic (Java code) separately.

5. **Easy to Maintain**: JSP allows for the creation of modular and maintainable code by breaking down the application into smaller components, which can be developed, tested, and debugged independently.

6. **Performance**: JSP pages are compiled into servlets at runtime, and subsequent requests are served by compiled servlets, which significantly improves performance compared to interpreting the JSP every time a request is made.

Lifecycle of JSP (JavaServer Pages):

1. **Translation**: When a JSP page is requested for the first time, the JSP container translates the JSP file into a servlet. This servlet is responsible for handling subsequent requests for that JSP page.

2. **Compilation**: The translated JSP servlet code is compiled into bytecode by the Java compiler. The resulting class file is loaded and executed by the Java Virtual Machine (JVM).

3. **Initialization**: During initialization, the JSP servlet's `init()` method is called. This method can be overridden to perform any necessary setup tasks.

4. **Request Handling**: For each client request, the servlet's `service()` method is invoked. The `service()` method handles the request, processes any Java code embedded in the JSP, and generates the dynamic content to be sent back to the client.

5. **Destruction**: When the JSP container decides to remove the JSP servlet (for example, during web application shutdown), the `destroy()` method is called. This method can be overridden to perform cleanup tasks such as closing database connections.

Understanding the advantages of JSP and its lifecycle is essential for building efficient and maintainable Java web applications.

Components of JSP Page:

1. **Directives**: Directives provide global information about an entire JSP page and are defined using `<%@ directive attribute="value" %>`. There are three types of directives:

   - **Page Directive**: Provides instructions for the container, such as error handling and buffering.

   - **Include Directive**: Includes a file during the translation phase.

   - **Taglib Directive**: Declares a custom tag library for use in the JSP page.

2. **Comments**: Comments in JSP are similar to HTML and Java comments. They are not visible in the output HTML and are used for documentation within the JSP code. JSP supports both HTML-style (`<!-- -->`) and Java-style (`<%-- --%>`) comments.

3. **Expressions**: Expressions are used to output data onto the client's web browser. They are defined using `${expression}` and are evaluated, converted to a string, and included in the response. For example, `<%= variable %>` is an expression.

4. **Scriptlets**: Scriptlets are blocks of Java code enclosed within `<% %>` tags. They are used to perform tasks like calculations, database operations, and conditional logic. However, extensive use of scriptlets is discouraged due to the mixing of presentation and business logic.

5. **Declarations**: Declarations are used to declare variables or methods in a JSP page. They are enclosed within `<%! %>` tags and allow you to define variables and methods that can be used throughout the JSP page.

6. **Implicit Objects**: JSP provides several implicit objects that can be used directly in the JSP code without any initialization. Examples include `request`, `response`, `session`, `out` (for output), `config` (for servlet configuration), and `application` (for application-wide information).

7. **Standard Actions**: Standard actions are XML-like tags used for controlling the flow of the JSP page and for performing various tasks like including files (`<jsp:include>`), forwarding requests (`<jsp:forward>`), handling exceptions (`<jsp:exception>`), etc. They provide a way to modularize the code and separate concerns.

8. **Tag Extensions**: Tag extensions allow you to create custom tags to encapsulate specific functionality. There are two types of tag extensions:

   - **Custom Tags**: Created using Java classes implementing the `Tag` interface or extending existing tag handler classes.

   - **Tag Files**: Tag files are simple XML files with a `.tag` extension that define custom tags using a tag library descriptor (`*.tld`) file.

Understanding these components is crucial for effective JSP development, allowing developers to create dynamic and maintainable web applications.

Elements Created with the INPUT Tag:

The `<input>` tag in HTML is used to create various types of form elements that allow users to input data. Some common `input` types include:

- **Text Input**: `<input type="text">` creates a single-line text input field where users can enter alphanumeric text.

- **Password Input**: `<input type="password">` creates a password input field where the entered text is masked (usually as dots or asterisks) for security.

- **Checkbox**: `<input type="checkbox">` creates a checkbox allowing users to select multiple options from a list.

- **Radio Button**: `<input type="radio">` creates a radio button allowing users to select a single option from a list.

- **File Upload**: `<input type="file">` creates a file upload button, enabling users to select and upload files from their device.

- **Hidden Input**: `<input type="hidden">` creates an invisible input field that stores data on the form but is not displayed to the user.

- **Submit Button**: `<input type="submit">` creates a button that submits the form data to the server.

- **Reset Button**: `<input type="reset">` creates a button that resets the form's input fields to their default values.

- **Date Input**: `<input type="date">` creates a date picker allowing users to select a date from a calendar.

- **Email Input**: `<input type="email">` validates that the input is a valid email address.

- **Number Input**: `<input type="number">` validates that the input is a numeric value.

- **URL Input**: `<input type="url">` validates that the input is a valid URL.

Elements Created with `<select>` and `<option>` Tags:

The `<select>` tag is used to create a dropdown list, and `<option>` tags define the options within the dropdown. For example:

```html
<select>
```

    <option value="option1">Option 1</option>

    <option value="option2">Option 2</option>

    <option value="option3">Option 3</option>

</select>

- **Dropdown List (`<select>`)**: `<select>` creates a dropdown list allowing users to select a single option from a list of options.

- **Option (`<option>`)**: `<option>` defines an option within the dropdown list. It can have attributes like `value` (the value sent to the server when the form is submitted) and `selected` (to preselect the option).

`<textarea>` Element:

The `<textarea>` tag is used to create a multiline text input field. It is used when users are required to enter a large amount of text, such as comments or descriptions. For example:

```html

<textarea rows="4" cols="50">

Enter your text here...

</textarea>

```

- **Multiline Text Input (`<textarea>`)**: `<textarea>` creates a text input area where users can enter multiple lines of text. Attributes like `rows` and `cols` specify the visible height and width of the textarea.

Understanding these HTML elements is fundamental for creating interactive and user-friendly web forms. Developers can use these elements to gather various types of input from users, enhancing the interactivity of web applications.

JSP Form Validation:

Form validation in JSP (JavaServer Pages) involves ensuring that the data submitted by users through HTML forms meets the required criteria before it is processed by the server-side application. Validating user input is crucial for data integrity, security, and a better user experience. Here's how you can perform form validation in JSP:

1. **Client-Side Validation:**

   - Use JavaScript to perform basic validation on the client side before the form is submitted to the server.

   - JavaScript can validate fields such as email addresses, phone numbers, and check if required fields are filled.

   - Client-side validation provides instant feedback to users without making a round trip to the server.

2. **Server-Side Validation:**

   - Perform thorough validation on the server side even if client-side validation is in place. Client-side validation can be bypassed, so server-side validation is essential for security.

   - In the JSP, after form submission, use Java code to validate the form fields. This can be done in the servlet associated with the JSP page.

3. **Using JavaBeans for Validation:**

   - Create a JavaBean class representing the form data. Include validation logic within this class, defining methods to validate individual fields.

   - For example, you can have methods like `isValidEmail()`, `isValidPhoneNumber()`, etc., within the JavaBean class to validate specific fields.

4. **Displaying Validation Errors:**

   - If validation fails, redirect the user back to the form page and display error messages near the respective form fields.

   - Use JSP tags or expressions to display error messages dynamically. For example:

     ```jsp
     <span style="color: red;"><%= errorMessage %></span>
     ```

5. **Handling Server-Side Validation in Servlet:**

   - In the associated servlet, retrieve form parameters using `request.getParameter("parameterName")`.

   - Validate the input and store error messages in request attributes if validation fails.

   - Redirect the request back to the form JSP page, where error messages can be displayed using JSP expressions.

Example Servlet Code for Server-Side Validation:

```java
String username = request.getParameter("username");
String password = request.getParameter("password");
String errorMessage = "";

if (username == null || username.isEmpty() || password == null || password.isEmpty()) {
    errorMessage = "Username and password are required fields.";
```

```
} else {
    // Perform further validation if needed
}

if (!errorMessage.isEmpty()) {
    request.setAttribute("errorMessage", errorMessage);
    RequestDispatcher dispatcher = request.getRequestDispatcher("form.jsp");
    dispatcher.forward(request, response);
} else {
    // Process the form data as valid
}
```

By combining client-side and server-side validation techniques, you can ensure that the data submitted through JSP forms is accurate, secure, and meets the desired criteria before processing it in your server-side application.

Certainly! Here's an example of a JSP form validation code along with the recommended file structure:

File Structure:

```
- WEB-INF
  - classes
    - User.java
  - lib
```

  - web.xml

- index.jsp

- login.jsp

- error.jsp

- success.jsp

- validationServlet.java

`index.jsp` (Home Page):

```html
<!DOCTYPE html>
<html>
<head>
    <title>Form Validation Example</title>
</head>
<body>
    <h1>Welcome to Form Validation Example</h1>
    <a href="login.jsp">Login</a>
</body>
</html>
```

`login.jsp` (Login Form Page):

```html
<!DOCTYPE html>
<html>
<head>
  <title>Login</title>
</head>
<body>
  <h1>Login</h1>
  <form action="validate" method="post">
    Username: <input type="text" name="username"><br>
    Password: <input type="password" name="password"><br>
    <input type="submit" value="Login">
  </form>
</body>
</html>
```

### `error.jsp` (Error Page):

```html
<!DOCTYPE html>
<html>
<head>
  <title>Error</title>
</head>
```

```html
<body>
    <h1>Error</h1>
    <p>${errorMessage}</p>
    <a href="login.jsp">Back to Login</a>
</body>
</html>
```

`success.jsp` (Success Page):

```html
<!DOCTYPE html>
<html>
<head>
    <title>Success</title>
</head>
<body>
    <h1>Success</h1>
    <p>Login Successful!</p>
</body>
</html>
```

`User.java` (JavaBean for User Data):

```java
public class User {
    private String username;
    private String password;

    // Getters and setters
}
```

`validationServlet.java` (Servlet for Form Validation):

```java
import java.io.IOException;
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("/validate")
public class ValidationServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        String username = request.getParameter("username");
        String password = request.getParameter("password");
        String errorMessage = "";

        if (username == null || username.isEmpty() || password == null || password.isEmpty()) {
            errorMessage = "Username and password are required fields.";
            request.setAttribute("errorMessage", errorMessage);
            request.getRequestDispatcher("error.jsp").forward(request, response);
        } else {
            // Validate username and password (for example, from a database)
            // If validation succeeds, redirect to success page
            // If validation fails, set appropriate error message and forward to error.jsp
            // ...
            // For this example, assume validation always fails
            errorMessage = "Invalid username or password.";
            request.setAttribute("errorMessage", errorMessage);
            request.getRequestDispatcher("error.jsp").forward(request, response);
        }
    }
}
```

`web.xml` (Servlet Configuration):

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

```xml
<web-app 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_4_0.xsd"
         version="4.0">

    <servlet>
        <servlet-name>ValidationServlet</servlet-name>
        <servlet-class>ValidationServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>ValidationServlet</servlet-name>
        <url-pattern>/validate</url-pattern>
    </servlet-mapping>

    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

In this example, the `ValidationServlet` handles form validation. If the form submission fails validation, the user is redirected to the `error.jsp` page with an error message. If the validation succeeds, the user is redirected to the `success.jsp` page. The `User` class is a simple JavaBean for holding user data. The `web.xml` file configures the servlet and specifies the welcome file.

Certainly! Below is an explanation along with full code examples for reading and deleting data from cookies, maintaining sessions and tracking session IDs, and using JSP Core tags, SQL tags, XML tags, and JSTL (JavaServer Pages Standard Tag Library) functions. Let's start with the file structure:

File Structure:

```
- WEB-INF
  - classes
    - User.java
    - DatabaseUtil.java
  - lib
    - standard.jar (JSTL library)
  - web.xml
- index.jsp
- login.jsp
- display.jsp
- WEB-INF
  - userData.xml
```

Reading and Deleting Data from Cookies:

`login.jsp` (Set Cookie):

```jsp
<%@ page import="javax.servlet.http.Cookie" %>
<%
    // Set a cookie with user information
    Cookie userCookie = new Cookie("username", "exampleUser");
    userCookie.setMaxAge(3600); // Cookie will expire in 1 hour
    response.addCookie(userCookie);
%>
<!DOCTYPE html>
<html>
<head>
    <title>Login</title>
</head>
<body>
    <h1>Login Successful</h1>
    <a href="display.jsp">Display User Data</a>
</body>
</html>
```

`display.jsp` (Read and Delete Cookie):

```jsp
<%@ page import="javax.servlet.http.Cookie" %>
<%
    // Read username from the cookie
    String username = "";
    Cookie[] cookies = request.getCookies();
    if (cookies != null) {
        for (Cookie cookie : cookies) {
            if (cookie.getName().equals("username")) {
                username = cookie.getValue();
                // Delete the cookie
                cookie.setMaxAge(0);
                response.addCookie(cookie);
                break;
            }
        }
    }
%>
<!DOCTYPE html>
<html>
<head>
    <title>User Data</title>
</head>
<body>
```

```
    <h1>Welcome, <%= username %></h1>

    <p>Cookie deleted after use.</p>

</body>

</html>
```

Maintaining Session and Tracking Session ID:

`index.jsp` (Set and Display Session ID):

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

<%

    // Create a new session ID or retrieve existing one

    String sessionID = (String) session.getAttribute("sessionID");

    if (sessionID == null) {

        sessionID = UUID.randomUUID().toString();

        session.setAttribute("sessionID", sessionID);

    }

%>

<!DOCTYPE html>

<html>

<head>

    <title>Session ID Example</title>

</head>
```

<body>

    <h1>Session ID: <%= sessionID %></h1>

    <a href="display.jsp">Display User Data</a>

</body>

</html>

~~~

JSP Core Tags, SQL Tags, XML Tags, and JSTL Functions:

For using JSP Core tags, SQL tags, XML tags, and JSTL functions, you need to include the appropriate tag libraries in your JSP pages and import the required classes.

`display.jsp` (Example Usage of Core Tags, SQL Tags, XML Tags, and JSTL Functions):

```jsp

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

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

<%@ taglib uri="http://java.sun.com/jsp/jstl/xml" prefix="x" %>

<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>

<%@ page import="java.util.List" %>

<%@ page import="java.sql.*" %>

<%@ page import="org.w3c.dom.Document" %>

<%@ page import="java.io.InputStream" %>

<%@ page import="javax.xml.parsers.DocumentBuilder" %>

<%@ page import="javax.xml.parsers.DocumentBuilderFactory" %>

<%@ page import="org.xml.sax.InputSource" %>
~~~

```
<%
    // Perform database query (assuming DatabaseUtil class is implemented)
    List<User> users = DatabaseUtil.getAllUsers();

    // Parse XML data (assuming userData.xml contains user data)
    InputStream xmlStream = application.getResourceAsStream("/WEB-INF/userData.xml");
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    DocumentBuilder builder = factory.newDocumentBuilder();
    Document document = builder.parse(new InputSource(xmlStream));
%>
<!DOCTYPE html>
<html>
<head>
    <title>User Data</title>
</head>
<body>
    <h1>User Data</h1>
    <ul>
        <c:forEach var="user" items="<%= users %>">
            <li>${user.name} - ${user.email}</li>
        </c:forEach>
    </ul>
    <h2>XML Data</h2>
    <x:parse var="xmlDoc" xml="<%= document %>"/>
    <x:forEach select="$xmlDoc//user">
```

```
        <p>Name: <x:out select="name"/> | Email: <x:out select="email"/></p>

    </x:forEach>

    <h2>JSTL Functions</h2>

    <p>Current Date and Time: ${fn:substring(fn:currentDate(), 0, 19)}</p>

</body>

</html>
```

In this example, the JSP page uses various JSTL features such as core tags (`<c:forEach>`), SQL tags (`<sql:query>`), XML tags (`<x:parse>`, `<x:forEach>`), and JSTL functions (`fn:currentDate()`) to iterate through user data, parse XML data, and display the current date and time.

Please ensure you have the appropriate JSTL library (standard.jar) in the `WEB-INF/lib` folder for these tags to work. Also, implement the `DatabaseUtil` class to fetch user data from the database.

Creating a table, performing SELECT, INSERT, DELETE, and UPDATE operations using JSP involves integrating Java code with SQL queries to interact with a database. Below is a step-by-step explanation along with full code examples and the file structure:

File Structure:

```
- WEB-INF
  - classes
    - DatabaseUtil.java
  - lib
    - your-database-driver.jar (e.g., mysql-connector-java.jar for MySQL)
  - web.xml
- index.jsp
- display.jsp
- insert.jsp
- delete.jsp
- update.jsp
```

`DatabaseUtil.java` (Database Utility Class):

```java
import java.sql.*;

public class DatabaseUtil {

    static final String JDBC_DRIVER = "your_database_driver";
    static final String DB_URL = "your_database_url";
    static final String USER = "your_username";
    static final String PASS = "your_password";
    public static Connection getConnection() throws SQLException, ClassNotFoundException {
        Class.forName(JDBC_DRIVER);
        return DriverManager.getConnection(DB_URL, USER, PASS);
    }

    public static void closeConnection(Connection conn, Statement stmt, ResultSet rs) {
        try {
            if (rs != null) rs.close();
            if (stmt != null) stmt.close();
            if (conn != null) conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

`index.jsp` (Display Table Data):

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

```
    Connection conn = null;
    Statement stmt = null;
    ResultSet rs = null;
    List<Map<String, String>> tableData = new ArrayList<>();

    try {
        conn = DatabaseUtil.getConnection();
        stmt = conn.createStatement();
        rs = stmt.executeQuery("SELECT * FROM your_table_name");

        while (rs.next()) {
            Map<String, String> row = new HashMap<>();
            row.put("column1", rs.getString("column1"));
            row.put("column2", rs.getString("column2"));
            // Add more columns as needed
            tableData.add(row);
        }
    } catch (SQLException | ClassNotFoundException e) {
        e.printStackTrace();
    } finally {
        DatabaseUtil.closeConnection(conn, stmt, rs);
    }
%>
<!DOCTYPE html>
<html>
```

```
<head>
    <title>Table Data</title>
</head>
<body>
    <h1>Table Data</h1>
    <table border="1">
        <thead>
            <tr>
                <th>Column 1</th>
                <th>Column 2</th>
                <!-- Add more headers as needed -->
            </tr>
        </thead>
        <tbody>
            <c:forEach var="row" items="<%= tableData %>">
                <tr>
                    <td>${row.column1}</td>
                    <td>${row.column2}</td>
                    <!-- Add more columns as needed -->
                </tr>
            </c:forEach>
        </tbody>
    </table>
</body>
</html>
```

~~~

`insert.jsp` (Insert Data into Table):

```jsp
<%@ page import="java.sql.*" %>
<%@ page import="DatabaseUtil" %>
<%
    String column1Value = request.getParameter("column1");
    String column2Value = request.getParameter("column2");
    // Get more parameters as needed

    Connection conn = null;
    PreparedStatement pstmt = null;

    try {
        conn = DatabaseUtil.getConnection();
        String insertQuery = "INSERT INTO your_table_name (column1, column2) VALUES (?, ?)";
        pstmt = conn.prepareStatement(insertQuery);
        pstmt.setString(1, column1Value);
        pstmt.setString(2, column2Value);
        // Set more parameters as needed
        pstmt.executeUpdate();
    } catch (SQLException | ClassNotFoundException e) {
        e.printStackTrace();
~~~

```
    } finally {
        DatabaseUtil.closeConnection(conn, pstmt, null);
    }
    response.sendRedirect("index.jsp"); // Redirect to display data page
%>
```

`delete.jsp` (Delete Data from Table):

```jsp
<%@ page import="java.sql.*" %>
<%@ page import="DatabaseUtil" %>
<%
    String idToDelete = request.getParameter("id");

    Connection conn = null;
    PreparedStatement pstmt = null;

    try {
        conn = DatabaseUtil.getConnection();
        String deleteQuery = "DELETE FROM your_table_name WHERE id = ?";
        pstmt = conn.prepareStatement(deleteQuery);
        pstmt.setString(1, idToDelete);
        pstmt.executeUpdate();
    } catch (SQLException | ClassNotFoundException e) {
        e.printStackTrace();
    } finally {
        DatabaseUtil.closeConnection(conn, pstmt, null);
    }
    response.sendRedirect("index.jsp"); // Redirect to display data page
%>
```

`update.jsp` (Update Data in Table):

```jsp
<%@ page import="java.sql.*" %>
<%@ page import="DatabaseUtil" %>
<%
    String idToUpdate = request.getParameter("id");
    String updatedColumn1Value = request.getParameter("updatedColumn1");
    String updatedColumn2Value = request.getParameter("updatedColumn2");
    // Get more parameters as needed

    Connection conn = null;
    PreparedStatement pstmt = null;

    try {
        conn = DatabaseUtil.getConnection();
        String updateQuery = "UPDATE your_table_name SET column1 = ?, column2 = ? WHERE id =
 ?";
```

```
        pstmt = conn.prepareStatement(updateQuery);
        pstmt.setString(1, updatedColumn1Value);
        pstmt.setString(2, updatedColumn2Value);
        pstmt.setString(3, idToUpdate);
        // Set more parameters as needed
        pstmt.executeUpdate();
    } catch (SQLException | ClassNotFoundException e) {
        e.printStackTrace();
    } finally {
        DatabaseUtil.closeConnection(conn, pstmt, null);
    }
    response.sendRedirect("index.jsp"); // Redirect to display data page
%>
```

In these examples, replace placeholders such as `your_database_driver`, `your_database_url`, `your_username`, `your_password`, `your_table_name`, `column1`, `column2`, `id`, `updatedColumn1`, `updatedColumn2` with appropriate values for your database and table structure.

Ensure that you have the necessary database driver JAR file (`your-database-driver.jar`) in the `WEB-INF/lib` folder for the database connectivity to work.

MVC (Model-View-Controller) Architecture:

MVC is a design pattern used in software engineering for organizing code in a way that separates different concerns within an application. It divides the application logic into three interconnected components:

1. **Model (M)**: Represents the data and business logic of the application. It retrieves and stores data, processes it, and responds to requests for information about its state.

2. **View (V)**: Represents the user interface and displays the data to the user. It presents the model's data to the user and forwards user input to the controller.

3. **Controller (C)**: Acts as an intermediary between the model and the view. It receives user input from the view, processes it (possibly involving the model), and updates the view accordingly.

Importance of MVC Architecture:

1. **Separation of Concerns**: MVC separates application logic into distinct components, making the codebase more organized and easier to maintain. Changes in one component do not directly affect the others, promoting modularity and reusability.

2. **Scalability**: MVC allows for easier scalability of the application. Changes or additions to one component do not require extensive modifications in other components, making it simpler to expand or modify the application.

3. **Parallel Development**: Different developers or teams can work on different components simultaneously without interfering with each other. The clear separation of concerns in MVC facilitates parallel development and collaboration.

4. **Code Reusability**: Due to the modular structure, individual components can be reused across different parts of the application or in entirely different applications, saving development time and effort.

5. **Improved Testing**: Each component (model, view, and controller) can be tested independently, allowing for more effective unit testing. This separation simplifies the testing process and ensures that each component works as intended.

Advantages of MVC Architecture:

1. **Flexibility**: MVC allows changes in one component without affecting the others. This flexibility is crucial in applications where user interface and business logic are subject to frequent changes.

2. **Maintenance**: Due to the separation of concerns, maintenance becomes easier. Developers can modify or enhance one component without disturbing the others, reducing the risk of unintended side effects.

3. **Enhanced User Experience**: Since the view is responsible for the user interface, it can be tailored to provide an optimal user experience. Changes in the view do not affect the underlying business logic, allowing for easy UI enhancements.

4. **Modular Development**: MVC supports modular development, enabling the development of each component in isolation. This modularity simplifies collaboration among developers and teams, leading to efficient project development.

5. **Better Collaboration**: In team environments, developers can work concurrently on different components of the application, making collaboration smoother and more streamlined.

Example File Structure for MVC:

```
- WEB-INF
```

```
  - classes
    - Model.java
    - View.java
    - Controller.java
  - lib
  - web.xml
- index.jsp (Entry Point)
```

In this structure, `Model.java` represents the application data and business logic, `View.java` handles the user interface, and `Controller.java` manages user input and communication between the model and the view. `web.xml` configures the servlets, and `index.jsp` serves as the entry point of the application. The separation of concerns and clear division of responsibilities among these components exemplify the MVC architecture.

Model Layer, View Layer, and Controller Layer:

In the context of the MVC (Model-View-Controller) architecture, the application is divided into three interconnected components: Model, View, and Controller. Each component has distinct responsibilities, ensuring a clear separation of concerns and enhancing the maintainability and scalability of the application.

Model Layer:

Responsibilities:

- Manages data and business logic.

- Represents the application state and behavior.

- Interacts with the database, APIs, or any external data source.

Example File Structure for Model Layer:

```
- WEB-INF
  - classes
    - Model.java
    - DatabaseHandler.java
```

`Model.java` (Example Model Class):

```java
public class Model {
    private String data;

    // Getters and setters
}
```

`DatabaseHandler.java` (Example Database Handler Class within Model Layer):

```java
import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;


public class DatabaseHandler {

   public Connection establishConnection() throws SQLException, ClassNotFoundException {

       Class.forName("your_database_driver");

       return DriverManager.getConnection("your_database_url", "your_username", "your_password");

   }


   // Additional database handling methods

}
```

View Layer:

Responsibilities:

- Presents data to the user.

- Renders the user interface based on the data received from the model.

- Collects and forwards user input to the controller.

Example File Structure for View Layer:

```
- WEB-INF
  - jsp
    - index.jsp
    - display.jsp
```

`index.jsp` (Example View Page for User Input):

```jsp
<!DOCTYPE html>
<html>
<head>
    <title>Input Form</title>
</head>
<body>
    <form action="controller" method="post">
        <input type="text" name="data">
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

`display.jsp` (Example View Page to Display Data):

```jsp
<%@ page import="Model" %>
<%@ page import="java.util.List" %>
<%@ page import="java.util.Iterator" %>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<%
    Model model = (Model) request.getAttribute("model");
    List<String> dataList = model.getDataList();
%>
<!DOCTYPE html>
<html>
<head>
    <title>Data Display</title>
</head>
<body>
    <h1>Displaying Data:</h1>
    <ul>
        <c:forEach var="data" items="<%= dataList %>">
            <li>${data}</li>
        </c:forEach>
    </ul>
```

</body>

</html>

```

### Controller Layer:

**Responsibilities:**

- Receives user input from the view.

- Processes user input, interacts with the model to perform necessary operations.

- Determines the appropriate view to present the processed data.

#### Example File Structure for Controller Layer:

```

- WEB-INF

  - classes

    - ControllerServlet.java

~~~

`ControllerServlet.java` (Example Controller Servlet):

```java

import java.io.IOException;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;
~~~

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ControllerServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        String inputData = request.getParameter("data");

        // Process input data (interact with model, perform operations)
        Model model = new Model();
        model.processData(inputData);

        // Set model in request attribute for the view
        request.setAttribute("model", model);

        // Forward the request to the appropriate view
        request.getRequestDispatcher("display.jsp").forward(request, response);
    }
}
```

In this example, the `Model` class represents the data and contains the logic to process the input data. The `index.jsp` page collects user input and sends it to the `ControllerServlet` for processing. The controller processes the input, interacts with the `Model`, and forwards the processed data to the `display.jsp` page for presentation.

This clear separation of concerns enhances the maintainability and scalability of the application,

}

allowing for easier modifications and updates in the future.

Aspect-Oriented Programming (AOP), Dependency Injection (DI), Plain Old Java Object (POJO), and Spring Framework Architecture:

Aspect-Oriented Programming (AOP):

Explanation:

AOP is a programming paradigm that allows modularization of cross-cutting concerns in software applications. Cross-cutting concerns are aspects of a program that affect multiple modules and are hard to separate out. AOP provides a way to define these concerns (such as logging, security, and transaction management) as reusable modules.

Example Code (AOP in Spring):

```java
public aspect LoggingAspect {
    pointcut loggableMethods(): execution(* com.example.service.*.*(..));

    before(): loggableMethods() {
        System.out.println("Logging before method execution.");
    }
}
```

Dependency Injection (DI):

Explanation:

DI is a design pattern used to enable the creation of objects with their dependencies injected from external sources, rather than creating them within the class. It promotes loose coupling and facilitates easier testing and flexibility in changing implementations without modifying client code.

Example Code (DI in Spring):

```java
public class CustomerService {
    private CustomerDao customerDao;

    // Constructor Injection
    public CustomerService(CustomerDao customerDao) {
        this.customerDao = customerDao;
    }

    // Business logic methods using customerDao
}
```

Plain Old Java Object (POJO):

Explanation:

POJO refers to a Java object that does not depend on any special framework, container, or library. It adheres to the JavaBeans conventions, making it simple, reusable, and easily integrable with various technologies.

Example Code (POJO):

```java
public class User {
    private String username;
    private String password;

    // Getters and setters
}
```

Spring Framework Architecture:

Explanation:

Spring is a popular Java framework that provides comprehensive infrastructure support while promoting loose coupling and easy integration with other frameworks. It includes modules for various functionalities like data access, security, transaction management, and more. The core of the Spring framework is the Inversion of Control (IoC) container, which manages object lifecycles and dependencies.

Example Code (Spring Architecture):

```xml
<!-- applicationContext.xml -->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- Define beans and their dependencies -->
    <bean id="customerDao" class="com.example.dao.CustomerDaoImpl">
        <!-- Inject dependencies if any -->
    </bean>

    <bean id="customerService" class="com.example.service.CustomerService">
        <!-- Constructor Injection -->
        <constructor-arg ref="customerDao" />
    </bean>
</beans>
```

File Structure for Spring Application:

```
- src
  - com
    - example
      - aspect
        - LoggingAspect.java
      - dao
        - CustomerDao.java
        - CustomerDaoImpl.java

      - service

        - CustomerService.java

      - MainApp.java

- WebContent

  - WEB-INF

    - applicationContext.xml
```

In this structure, `LoggingAspect` represents an AOP aspect. `CustomerDao` and `CustomerDaoImpl` demonstrate dependency injection and POJO usage. `applicationContext.xml` configures Spring beans and their dependencies. `MainApp.java` is the entry point of the application, where Spring's IoC container is initialized and used to manage beans.

By utilizing AOP, DI, and POJO principles within the Spring framework, applications can achieve modularity, maintainability, and flexibility while promoting best practices in software design and development.

Features in Spring Boot:

Spring Boot is a framework that simplifies the setup and development of new Spring applications. It comes with several powerful features that enhance productivity and reduce development time.

Key Features of Spring Boot:

1. **Auto Configuration:** Spring Boot provides default configurations for various components based on the classpath. It reduces the need for manual configuration, allowing developers to focus on application logic.

2. **Standalone:** Spring Boot applications are standalone, meaning they do not require an external application server. They embed an embedded servlet container (like Tomcat or Jetty) within the application.

3. **Spring Boot Starter:** Spring Boot Starter packages various dependencies related to specific tasks into a single JAR. For example, `spring-boot-starter-web` includes dependencies for building web applications.

4. **Production-Ready Defaults:** Spring Boot sets up sensible default configurations for production use, ensuring security, logging, and monitoring features are in place without manual intervention.

5. **Spring Boot Actuator:** Actuator provides production-ready features like health checks, metrics, and environment information. It helps monitor and manage applications.

6. **Externalized Configuration:** Spring Boot allows externalizing configuration properties, reducing the need to hardcode values within the application code.

Example File Structure for Spring Boot Application:

```
- src
  - main
```

```
    - java
      - com
        - example
          - controller
            - MyController.java
          - model
            - User.java
          - service
            - UserService.java
          - Application.java (Main Spring Boot Application)
    - resources
      - application.properties (Configuration Properties)
- pom.xml (Maven Build Configuration)
```

`MyController.java` (Example Controller Class):

```java
@RestController
public class MyController {
    @Autowired
    private UserService userService;

    @GetMapping("/users/{id}")
    public User getUser(@PathVariable Long id) {
```

        return userService.getUserById(id);
    }
}

`UserService.java` (Example Service Class):

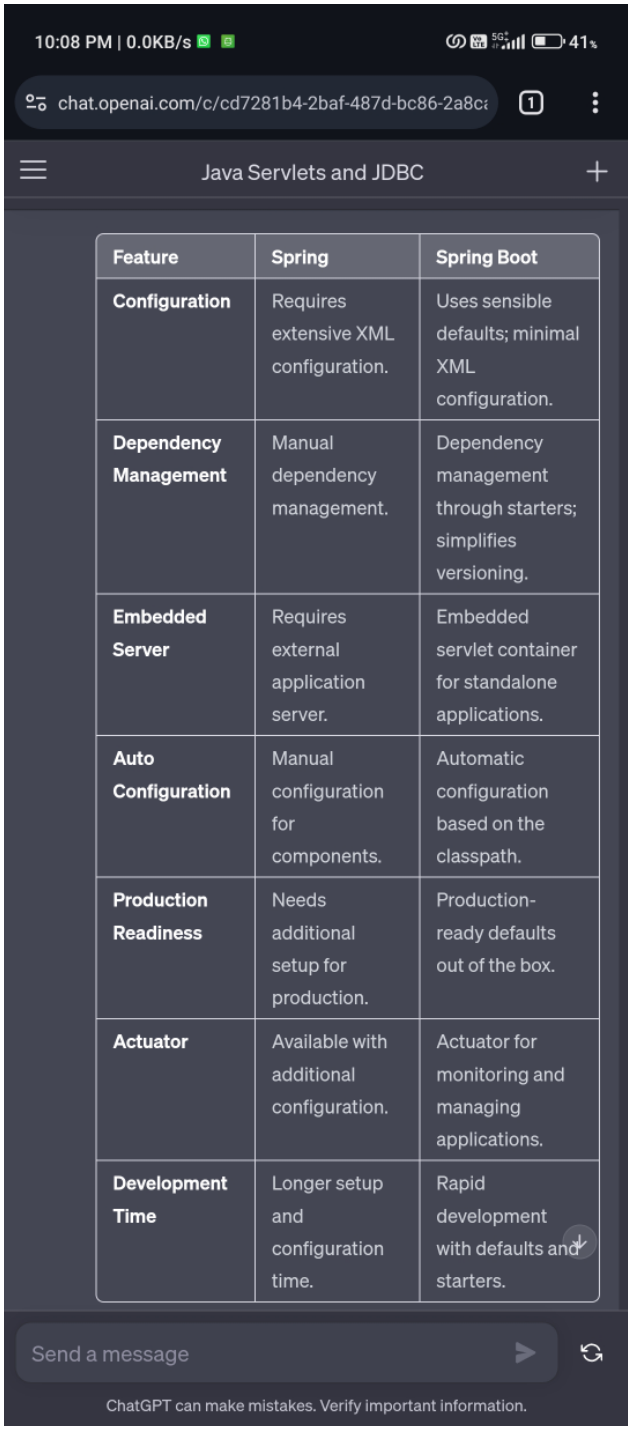
```java
@Service
public class UserService {
    public User getUserById(Long id) {
        // Logic to fetch user from database
        // Return User object
    }
}
```

`Application.java` (Main Spring Boot Application Class):

```java
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
```

}

```

---

**Comparison Between Spring and Spring Boot:**
```



10:08 PM | 0.0KB/s 41%

chat.openai.com/c/cd7281b4-2baf-487d-bc86-2a8c

Java Servlets and JDBC

Feature	Spring	Spring Boot
Configuration	Requires extensive XML configuration.	Uses sensible defaults; minimal XML configuration.
Dependency Management	Manual dependency management.	Dependency management through starters; simplifies versioning.
Embedded Server	Requires external application server.	Embedded servlet container for standalone applications.
Auto Configuration	Manual configuration for components.	Automatic configuration based on the classpath.
Production Readiness	Needs additional setup for production.	Production-ready defaults out of the box.
Actuator	Available with additional configuration.	Actuator for monitoring and managing applications.
Development Time	Longer setup and configuration time.	Rapid development with defaults and starters.

ChatGPT can make mistakes. Verify important information.