1. What is need of design pattern? Explain the use of Repository Design Pattern with the help of an example.

### Ans Importance of Design Patterns

Design patterns are reusable solutions to common software design problems. They provide best practices for building flexible, scalable, and maintainable code. The key benefits of design patterns include:

1. **Reusability**: Patterns allow developers to reuse successful techniques, leading to faster development and fewer bugs.
2. **Maintainability**: Code that follows well-known design patterns is easier to understand and modify by others, enhancing maintainability.
3. **Scalability**: Patterns help in organizing code in a way that can be easily scaled up as the application grows.
4. **Separation of Concerns**: They promote the separation of concerns, leading to better modularity and clearer structure.

**Repository Design Pattern**

The **Repository Design Pattern** is a data access pattern that abstracts the data layer of an application. It helps in separating the logic that retrieves data from the business logic, allowing for a more modular, testable, and maintainable codebase.

**Key Uses of Repository Design Pattern:**

1. **Separation of Concerns**: It decouples the business logic from data access logic.
2. **Testability**: It makes unit testing easier because the data access layer can be mocked.
3. **Flexibility**: It allows you to switch between different data sources (like databases or external services) without affecting the business logic.
4. **Encapsulation**: It encapsulates the logic needed to query the data source, making the system more robust to changes in the underlying data access technology.

#### Example: Repository Design Pattern in Java

Let's consider a simple example where we are managing hospital patients using the Repository Design Pattern.

// Step 1: Create an Entity Class (Patient)

public class Patient {

private int id;

private String name;

private String disease;

public Patient(int id, String name, String disease) {

this.id = id;

this.name = name;

this.disease = disease;

}

// Getters and Setters

public int getId() { return id; }

public void setId(int id) { this.id = id; }

public String getName() { return name; }

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

public String getDisease() { return disease; }

public void setDisease(String disease) { this.disease = disease; }

}

// Step 2: Define a Repository Interface

public interface PatientRepository {

void addPatient(Patient patient);

Patient getPatientById(int id);

List<Patient> getAllPatients();

void updatePatient(Patient patient);

void deletePatient(int id);

}

// Step 3: Implement the Repository (In-memory)

import java.util.ArrayList;

import java.util.List;

public class InMemoryPatientRepository implements PatientRepository {

private List<Patient> patients = new ArrayList<>();

@Override

public void addPatient(Patient patient) {

patients.add(patient);

}

@Override

public Patient getPatientById(int id) {

return patients.stream()

.filter(p -> p.getId() == id)

.findFirst()

.orElse(null);

}

@Override

public List<Patient> getAllPatients() {

return new ArrayList<>(patients);

}

@Override

public void updatePatient(Patient patient) {

for (int i = 0; i < patients.size(); i++) {

if (patients.get(i).getId() == patient.getId()) {

patients.set(i, patient);

break;

}

}

}

@Override

public void deletePatient(int id) {

patients.removeIf(p -> p.getId() == id);

}

}

// Step 4: Use the Repository in the Business Logic

public class HospitalManagementSystem {

private PatientRepository patientRepository;

public HospitalManagementSystem(PatientRepository patientRepository) {

this.patientRepository = patientRepository;

}

public void managePatients() {

// Add a new patient

Patient patient1 = new Patient(1, "John Doe", "Flu");

patientRepository.addPatient(patient1);

// Get and print patient details

Patient fetchedPatient = patientRepository.getPatientById(1);

System.out.println("Patient Name: " + fetchedPatient.getName());

// Update patient details

fetchedPatient.setDisease("Recovered");

patientRepository.updatePatient(fetchedPatient);

// Get all patients

List<Patient> allPatients = patientRepository.getAllPatients();

for (Patient patient : allPatients) {

System.out.println(patient.getName() + " - " + patient.getDisease());

}

// Delete patient

patientRepository.deletePatient(1);

}

public static void main(String[] args) {

PatientRepository repo = new InMemoryPatientRepository();

HospitalManagementSystem hospitalSystem = new HospitalManagementSystem(repo);

hospitalSystem.managePatients();

}

}

(b) What is the difference between JAR and WAR files? Describe the process of creation, deployment and extraction of WAR files.

**Difference between JAR and WAR Files**

**JAR (Java ARchive)**

* **Purpose**: JAR files are used to package **Java classes, libraries, and resources** into a single file. They are mainly used for deploying standalone Java applications or libraries.
* **Contents**: A JAR file contains compiled .class files, resource files (such as images or properties), and metadata (usually in a META-INF folder that contains the MANIFEST.MF file).
* **Usage**: Used to distribute and execute **Java desktop applications** or reusable Java libraries.
* **Command to create**: jar -cvf app.jar -C path/to/classes .

**WAR (Web Application Archive)**

* **Purpose**: WAR files are used to package **web applications** (Java web projects) that can be deployed on a servlet container (such as Apache Tomcat, Jetty, etc.).
* **Contents**: A WAR file contains web-related components like:
  + **Servlets**, **JSPs**, and classes (in the /WEB-INF/classes folder)
  + **Libraries** (in the /WEB-INF/lib folder)
  + **Web resources** (HTML, CSS, JavaScript, images)
  + web.xml (deployment descriptor in /WEB-INF)

Q2: (a) What is Servlet interface? Differentiate between GenericServlet and HTTPServlet?

### Ans : Servlet Interface

The **Servlet Interface** is a core part of the Java Servlet API, which defines the basic structure and behavior of a servlet. A servlet is a Java class that processes requests and generates responses, typically for web applications. The Servlet interface is the foundation for building servlets that handle web-based requests.

**Key Methods of the Servlet Interface:**

1. **init(ServletConfig config)**: This method is called once when the servlet is instantiated and initialized. It is used to perform any one-time setup.
2. **service(ServletRequest req, ServletResponse res)**: This method is called for each request made to the servlet. It processes the request and generates a response.
3. **destroy()**: This method is called once when the servlet is taken out of service, typically used to release any resources or perform cleanup.
4. **getServletConfig()**: Returns the ServletConfig object that contains the servlet's configuration and initialization parameters.
5. **getServletInfo()**: Provides information about the servlet, such as its version and author details.

**Difference between GenericServlet and HttpServlet**

Both GenericServlet and HttpServlet are abstract classes that implement the Servlet interface, but they differ in the type of protocols they handle and their purpose.

| **Feature** | **GenericServlet** | **HttpServlet** |
| --- | --- | --- |
| **Protocol Handling** | Protocol-independent (generic). | Specifically designed to handle **HTTP** requests (used for web applications). |
| **Purpose** | Can be used for any type of protocol (SMTP, FTP, etc.) by overriding the service() method. | Handles **HTTP requests** such as GET, POST, PUT, DELETE, etc. |
| **Methods** | Must override service() method to handle requests. | Provides **doGet()**, **doPost()**, **doPut()**, **doDelete()** for handling HTTP methods. |
| **Use Case** | For non-web-specific protocols (e.g., FTP, mail systems) or custom protocol handling. | Primarily used for building **web applications**. |
| **Predefined Methods** | Does not have predefined methods for specific HTTP tasks. | Has predefined methods to simplify handling of common HTTP methods like doGet() for GET requests and doPost() for POST requests. |
| **Common Usage** | Less commonly used because it's more generic. | The most commonly used class for **servlets** in web development. |
| **State** | Stateless (like all servlets). | Stateless (but HTTP is inherently stateless). |

(b) Briefly explain servlet life cycle. Also, explain the request and response in the context of HTTP?

### Ans : Servlet Life Cycle

The **Servlet life cycle** is defined by the **Java Servlet API**, and it describes the steps from when a servlet is loaded until it is destroyed. The life cycle has three main phases: **Initialization**, **Request Handling**, and **Termination**.

**1. Initialization (init()):**

* When the servlet is first requested or loaded by the server (such as Apache Tomcat), the init() method is called.
* This method is executed once, and it is responsible for initializing resources (e.g., database connections, configurations).
* The init() method receives a ServletConfig object, which holds initialization parameters defined in the web.xml configuration file.

**2. Request Handling (service()):**

* After initialization, the servlet is ready to handle client requests.
* For every incoming request, the **service(ServletRequest req, ServletResponse res)** method is invoked.
* The service() method determines the type of request (GET, POST, PUT, DELETE, etc.) and forwards it to the corresponding method (doGet(), doPost(), etc.).
  + For example, a GET request is handled by the doGet() method, while a POST request is handled by doPost().
* This method processes the request, performs necessary actions (e.g., interacting with a database, processing data), and generates a response.

**3. Termination (destroy()):**

* When the servlet is no longer needed, or when the server shuts down, the destroy() method is called.
* This method is used for cleanup, such as closing database connections or releasing other resources.
* Once destroy() is called, the servlet is removed from memory and can no longer handle requests.

**Servlet Life Cycle Methods**

| **Method** | **Description** |
| --- | --- |
| **init()** | Called once when the servlet is initialized. |
| **service()** | Called for each client request. |
| **destroy()** | Called once when the servlet is taken out of service. |

**HTTP Request and Response in the Context of Servlets**

In the context of servlets, the **HTTP request** and **response** are central to the communication between a web client (typically a browser) and the server.

**1. HTTP Request:**

An **HTTP request** is sent by the client to the server when it wants to access some resource (e.g., a webpage or an API endpoint).

* **Request Types** (Methods):
  + **GET**: Requests data from the server. Typically used to fetch resources (e.g., HTML pages, images).
  + **POST**: Sends data to the server. Often used for form submissions or sending sensitive information.
  + **PUT**, **DELETE**, **HEAD**, **OPTIONS**, **PATCH**: Other methods used for modifying resources or querying server capabilities.
* **Request Components**:
  + **Request Line**: Contains the HTTP method (GET, POST, etc.), the URI of the requested resource, and the HTTP version.
    - Example: GET /index.html HTTP/1.1
  + **Headers**: Additional information sent to the server, such as content type, user-agent, or cookies.
    - Example: Content-Type: application/json
  + **Body**: Data sent to the server, typically with a POST or PUT request (e.g., form data, JSON).
* **Servlet Request Object** (HttpServletRequest):
  + This object allows the servlet to access data from the client's request, such as parameters, headers, cookies, or session information.
  + **Methods**:
    - getParameter(String name): Retrieves a parameter value from the request.
    - getHeader(String name): Retrieves the value of a header.
    - getCookies(): Retrieves cookies sent by the client.

**2. HTTP Response:**

The **HTTP response** is the data sent back by the server in response to an HTTP request.

* **Response Components**:
  + **Status Line**: Indicates the status of the request (e.g., success or failure) with a status code and a reason phrase.
    - Example: HTTP/1.1 200 OK
  + **Headers**: Additional information sent to the client, such as the content type, length, or cache control.
    - Example: Content-Type: text/html
  + **Body**: The actual content being returned (HTML, JSON, etc.). In the case of a servlet, the body might contain a dynamically generated web page or data.
* **Servlet Response Object** (HttpServletResponse):
  + The HttpServletResponse object allows the servlet to send data back to the client.
  + **Methods**:
    - setContentType(String type): Sets the MIME type of the response (e.g., text/html for an HTML page).
    - getWriter(): Returns a PrintWriter object that can be used to write the response (e.g., HTML content or JSON data).
    - setStatus(int sc): Sets the HTTP status code (e.g., 200 for success, 404 for not found).

1. Explain the various components of JSP with suitable code.

Ans : Java Server Pages (JSP) has various components that help developers build dynamic web pages efficiently. These components allow mixing HTML with Java code for business logic, making JSP a powerful tool for web development.

**Key Components of JSP:**

1. **Directives**
2. **Scriptlets**
3. **Expressions**
4. **Declarations**
5. **Comments**
6. **Implicit Objects**
7. **Actions**
8. **Expression Language (EL)**

Let’s explore each component with examples:

**1. Directives**

Directives provide global information about the JSP page to the JSP engine. There are three main types of directives:

* **page**: Defines page-level attributes such as import statements, session control, etc.
* **include**: Includes content from another file during translation (compile) time.
* **taglib**: Declares a custom tag library for use in the JSP.

**Syntax**: <%@ directive attribute="value" %>

**Example:**

jsp

Copy code

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

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

<%@ include file="header.jsp" %>

Here:

* The page directive imports the Java utility class List and sets the encoding.
* The include directive includes another JSP file (header.jsp).

**2. Scriptlets**

Scriptlets contain Java code that gets executed when the JSP is processed. Scriptlets are written inside <% ... %> tags.

**Syntax**: <% Java code %>

**Example:**

jsp

Copy code

<%

int num1 = 10;

int num2 = 20;

int sum = num1 + num2;

%>

<p>The sum is: <%= sum %></p>

Here, Java code is executed inside the scriptlet, and the result (sum) is displayed in the HTML output.

**3. Expressions**

Expressions are used to insert the value of a Java expression directly into the HTML output. The expression is evaluated, and the result is converted into a string.

**Syntax**: <%= expression %>

**Example:**

jsp

Copy code

<%

String name = "Alice";

%>

<p>Hello, <%= name %>!</p>

This will display: Hello, Alice!

**4. Declarations**

Declarations are used to declare variables and methods in a JSP page that can be reused in multiple scriptlets or expressions.

**Syntax**: <%! declaration %>

**Example:**

jsp

Copy code

<%!

int counter = 0;

public int incrementCounter() {

return ++counter;

}

%>

<p>Counter: <%= incrementCounter() %></p>

Here, the counter variable and the incrementCounter() method are declared, and the method is used in the page.

**5. Comments**

JSP supports two types of comments: HTML comments and JSP comments.

* **HTML Comments**: These comments will be visible to the client when viewing the page source.

html

Copy code

<!-- This is an HTML comment visible in the browser -->

* **JSP Comments**: These comments are not sent to the client and are only visible in the JSP source.

jsp

Copy code

<%-- This is a JSP comment and will not appear in the browser --%>

**Example:**

jsp

Copy code

<!-- This is an HTML comment -->

<%-- This is a JSP comment --%>

**6. Implicit Objects**

JSP provides a set of implicit objects that developers can use without explicitly declaring them. These objects are created by the JSP engine and provide easy access to various elements such as request, response, session, etc.

* **request**: An object of HttpServletRequest, used to access request data.
* **response**: An object of HttpServletResponse, used to send a response to the client.
* **session**: An object of HttpSession, used to track user sessions.
* **out**: An object of JspWriter, used to write content to the response.
* Other implicit objects include application, config, pageContext, exception, etc.

**Example:**

jsp

Copy code

<%

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

%>

<p>Welcome, <%= username %>!</p>

Here, the request object is used to retrieve the username parameter from the client request.

**7. Actions**

JSP actions are XML-like tags that invoke built-in functionalities. They enable you to use JavaBeans, forward requests, include other resources, etc.

**Common JSP Actions:**

* **<jsp:include>**: Includes content from another resource at runtime.
* **<jsp:forward>**: Forwards the request to another resource.
* **<jsp:useBean>**: Creates or accesses JavaBean components.

**Example:**

jsp

Copy code

<jsp:include page="header.jsp" />

<jsp:useBean id="user" class="com.example.User" />

<jsp:getProperty name="user" property="name" />

In this example:

* jsp:include dynamically includes another JSP.
* jsp:useBean creates a JavaBean named user, and jsp:getProperty retrieves its name property.

**8. Expression Language (EL)**

Expression Language (EL) simplifies accessing data in Java objects. EL can directly access objects like request parameters, session attributes, and beans without needing explicit Java code.

**Syntax**: ${expression}

**Example:**

jsp

Copy code

<p>Username: ${param.username}</p>

<p>Session ID: ${session.id}</p>

In this example:

* ${param.username} retrieves the username parameter from the request.
* ${session.id} retrieves the session ID.

Q4: What do you mean by JDBC? Explain how we retrieve data from database using suitable JSP program.

### Ans: What is JDBC?

**Java Database Connectivity (JDBC)** is an API (Application Programming Interface) that enables Java applications to interact with various databases. It provides a standard interface for connecting to relational databases, executing SQL queries, and retrieving results. JDBC allows developers to write Java code that can perform operations like creating, reading, updating, and deleting data in a database.

**Key Components of JDBC:**

1. **JDBC Driver**: A software component that enables Java applications to interact with a specific database. Different databases require different JDBC drivers.
2. **Connection**: An interface that represents a session with a specific database. It is used to establish a connection and communicate with the database.
3. **Statement**: An interface that allows you to execute SQL queries against the database.
4. **ResultSet**: An interface that holds the data retrieved from the database after executing a query.
5. **PreparedStatement**: An interface used for executing precompiled SQL statements with or without parameters.

**Retrieving Data from a Database Using JSP**

To demonstrate how to retrieve data from a database using JSP, we will create a simple example that connects to a database, executes a SQL SELECT query, and displays the results in an HTML table.

**Prerequisites:**

1. **Database Setup**: Ensure you have a database (e.g., MySQL) running with a sample table. Below is an example table schema:
   * Database: testdb
   * Table: users
     + Columns: id (INT), name (VARCHAR), email (VARCHAR)

Sample SQL to create the table:

sql

Copy code

CREATE DATABASE testdb;

USE testdb;

CREATE TABLE users (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(100),

email VARCHAR(100)

);

INSERT INTO users (name, email) VALUES

('Alice', 'alice@example.com'),

('Bob', 'bob@example.com'),

('Charlie', 'charlie@example.com');

1. **JDBC Driver**: Include the JDBC driver (e.g., mysql-connector-java-x.x.x.jar) in your project.

**JSP Program to Retrieve Data**

Below is a sample JSP program that connects to the database, retrieves data from the users table, and displays it.

**JSP Page (retrieveUsers.jsp)**:

jsp

Copy code

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

<html>

<head>

<title>User List</title>

</head>

<body>

<h1>User List</h1>

<table border="1">

<tr>

<th>ID</th>

<th>Name</th>

<th>Email</th>

</tr>

<%

// Database connection parameters

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

String user = "root"; // replace with your database username

String password = "password"; // replace with your database password

Connection conn = null;

Statement stmt = null;

ResultSet rs = null;

try {

// Load JDBC driver

Class.forName("com.mysql.cj.jdbc.Driver");

// Establish connection

conn = DriverManager.getConnection(url, user, password);

// Create a statement

stmt = conn.createStatement();

// Execute query

String sql = "SELECT \* FROM users";

rs = stmt.executeQuery(sql);

// Process the result set

while (rs.next()) {

int id = rs.getInt("id");

String name = rs.getString("name");

String email = rs.getString("email");

%>

<tr>

<td><%= id %></td>

<td><%= name %></td>

<td><%= email %></td>

</tr>

<%

}

} catch (SQLException e) {

e.printStackTrace();

} catch (ClassNotFoundException e) {

e.printStackTrace();

} finally {

// Close resources

try {

if (rs != null) rs.close();

if (stmt != null) stmt.close();

if (conn != null) conn.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

%>

</table>

</body>

</html>

**Explanation of the Code:**

1. **Imports**: The required classes (java.sql.\*) are imported to enable database connectivity.
2. **Database Connection Parameters**: The JDBC URL, username, and password are defined for connecting to the MySQL database.
3. **Loading JDBC Driver**: The Class.forName() method loads the JDBC driver class. For MySQL, it's com.mysql.cj.jdbc.Driver.
4. **Establishing Connection**: The DriverManager.getConnection() method is used to establish a connection to the database.
5. **Creating a Statement**: A Statement object is created to execute SQL queries.
6. **Executing the Query**: The executeQuery() method runs the SQL query to retrieve all records from the users table.
7. **Processing the Result Set**: The result set (ResultSet) is processed in a loop using rs.next(). Each user's ID, name, and email are retrieved using getInt() and getString() methods.
8. **Displaying Results**: The data is displayed in an HTML table format.
9. **Closing Resources**: The resources (ResultSet, Statement, and Connection) are closed in the finally block to avoid memory leaks.

5) What are the Strut2 core components? Explain the working and flow of Struts 2 with the help of suitable diagram.

Struts 2 Core Components

Struts 2 is a popular framework for developing Java web applications. It is built on top of the popular Model-View-Controller (MVC) design pattern and provides a range of core components that facilitate web application development. The main core components of Struts 2 include:

Action: Represents the business logic of the application. An action class is responsible for processing requests and returning results. It typically contains methods that correspond to user actions.

ActionForm: A simple Java class that holds the data from the request and is used to transfer data between the view and action.

Interceptor: A component that processes requests and responses before and after the action is executed. Interceptors can perform tasks such as input validation, logging, and authentication.

Result: Defines what happens after an action is executed, such as forwarding to a JSP page or redirecting to another action.

Configuration File: The struts.xml file is used to configure actions, interceptors, results, and other components of the Struts 2 application.

View: Typically, JSP pages or other view technologies that render the user interface and display data returned from the action.

Servlet: The Struts 2 framework uses a servlet (typically ActionServlet) to handle incoming requests and route them to the appropriate action.

Working and Flow of Struts 2

The flow of a Struts 2 application can be described in the following steps:

Request Handling: A user sends a request (HTTP request) to the web application, which is handled by the Struts 2 front controller servlet (usually FilterDispatcher or StrutsPrepareAndExecuteFilter).

Configuration Lookup: The servlet looks up the request in the struts.xml configuration file to determine which action class to invoke based on the request URL.

Action Invocation: The corresponding action class is instantiated, and its execution begins. Before the action method is called, any configured interceptors are executed. Interceptors can perform various tasks such as logging, input validation, and preparing data.

Business Logic: The action class executes its business logic. This may involve interacting with the model (e.g., retrieving data from a database) and setting values in the action class.

Result Handling: Once the action method completes, it returns a string that represents the result type (e.g., "success," "error"). The framework then looks up the appropriate result configuration in the struts.xml file.

View Rendering: Based on the result, the framework determines which view to render (e.g., a JSP page). The data from the action class is made available to the view for rendering.

Response Generation: The JSP page generates the final HTML response, which is sent back to the client.

Client Receives Response: The client receives the response and renders it in the browser.

Diagram of Struts 2 Workflow

Below is a simplified diagram illustrating the workflow of a Struts 2 application:

sql

Copy code

+------------------+ +---------------------+

| | | |

| Client | -----> | Struts 2 Filter |

| (Browser) | | (Action Dispatcher) |

| | | |

+------------------+ +---------------------+

|

|

V

+---------------------+

| struts.xml |

| Configuration |

+---------------------+

|

|

V

+-----------------------+

| Action Class |

| (Business Logic) |

+-----------------------+

|

|

V

+---------------------+

| Result Type |

+---------------------+

|

|

V

+---------------------+

| View (JSP) |

+---------------------+

|

|

V

+---------------------+

| HTTP Response |

+---------------------+

Explanation of the Diagram

Client (Browser): The user interacts with the web application through a browser, sending HTTP requests to the server.

Struts 2 Filter: Acts as the front controller, routing requests to the appropriate action class based on the configuration in struts.xml.

struts.xml Configuration: This file contains mappings between request URLs, action classes, and results.

Action Class: Represents the application's business logic. It processes input and interacts with the model, retrieving or updating data as necessary.

Result Type: Based on the action's outcome, the appropriate view (JSP) is selected for rendering.

View (JSP): Generates the final HTML output that is sent back to the client.

HTTP Response: The server sends the rendered HTML back to the client's browser, completing the request-response cycle.

Q6: (a) Explain process of creating records using Spring Boot and Hibernate.

Creating records in a database using **Spring Boot** and **Hibernate** involves several steps, including setting up the Spring Boot application, configuring Hibernate, defining an entity, creating a repository, and implementing a service layer. Here’s a step-by-step guide on how to do this.

### Step 1: Set Up Spring Boot Application

1. **Create a New Spring Boot Project**: You can use Spring Initializr (<https://start.spring.io/>) to generate a new Spring Boot project. Choose the following dependencies:
   * Spring Web
   * Spring Data JPA
   * Your chosen database driver (e.g., H2, MySQL, PostgreSQL)
2. **Project Structure**: Your project should have the following structure:

├── src

│ └── main

│ ├── java

│ │ └── com

│ │ └── example

│ │ └── demo

│ │ ├── DemoApplication.java

│ │ ├── controller

│ │ ├── entity

│ │ ├── repository

│ │ └── service

│ └── resources

│ ├── application.properties

│ └── ...

### Step 2: Configure Database Connection

In the application.properties file (or application.yml), configure the database connection. For example, if you are using MySQL:

spring.datasource.url=jdbc:mysql://localhost:3306/your\_database\_name

spring.datasource.username=your\_username

spring.datasource.password=your\_password

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

### Step 3: Define the Entity

Create an entity class that represents the records you want to create. For example, let’s create a User entity.

// src/main/java/com/example/demo/entity/User.java

package com.example.demo.entity;

import javax.persistence.\*;

@Entity

@Table(name = "users")

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

// Getters and Setters

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

}

### Step 4: Create the Repository

Create a repository interface that extends JpaRepository. This interface provides methods to perform CRUD operations.

// src/main/java/com/example/demo/repository/UserRepository.java

package com.example.demo.repository;

import com.example.demo.entity.User;

import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long> {

// Custom query methods can be defined here if needed

}

### Step 5: Implement the Service Layer

Create a service class to encapsulate the business logic. This class will use the repository to create records.

// src/main/java/com/example/demo/service/UserService.java

package com.example.demo.service;

import com.example.demo.entity.User;

import com.example.demo.repository.UserRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

@Service

public class UserService {

@Autowired

private UserRepository userRepository;

public User createUser(User user) {

return userRepository.save(user);

}

}

### Step 6: Create a Controller

Create a REST controller to handle HTTP requests for creating records.

// src/main/java/com/example/demo/controller/UserController.java

package com.example.demo.controller;

import com.example.demo.entity.User;

import com.example.demo.service.UserService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping("/api/users")

public class UserController {

@Autowired

private UserService userService;

@PostMapping

public ResponseEntity<User> createUser(@RequestBody User user) {

User createdUser = userService.createUser(user);

return new ResponseEntity<>(createdUser, HttpStatus.CREATED);

}

}

### Step 7: Run the Application

1. **Run the Spring Boot Application**: You can run your application by executing the DemoApplication class.

// src/main/java/com/example/demo/DemoApplication.java

package com.example.demo;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class DemoApplication {

public static void main(String[] args) {

SpringApplication.run(DemoApplication.class, args);

}

}

**Test the Endpoint**: You can test the API using tools like Postman or curl. Send a POST request to http://localhost:8080/api/users with the following JSON body:

{

"name": "John Doe",

"email": "john.doe@example.com"

}

(b) Explain how testing of custom login form can be performed with the help of an example.

esting a custom login form involves verifying that the login functionality works as expected under various scenarios. You can perform testing using automated tools like **Selenium** or manual testing approaches. Below, I will demonstrate both manual and automated testing using **Selenium WebDriver** in a Java context.

### Manual Testing of a Custom Login Form

1. **Test Scenarios**: Identify the different scenarios you want to test:
   * Valid login with correct credentials.
   * Invalid login with incorrect credentials.
   * Empty username/password fields.
   * SQL injection or other security vulnerabilities.
   * Password visibility toggle (if applicable).
2. **Test Steps**:
   * Open the login page.
   * Enter credentials and submit the form.
   * Verify the expected behavior:
     + Redirect to the homepage/dashboard for valid login.
     + Show error messages for invalid login.
     + Check that both username and password fields are required.
     + Validate that the application is secure against SQL injection.
3. **Expected Outcomes**:
   * For valid login: User should be redirected to the dashboard.
   * For invalid login: Error message should be displayed.
   * For empty fields: Error messages indicating fields are required should be shown.
   * For security tests: Ensure the application handles malicious input without compromising security.

### Automated Testing with Selenium WebDriver

Here’s how you can automate the testing of a custom login form using **Selenium WebDriver** in Java.

#### Step 1: Set Up Selenium

Add Selenium dependencies to your project. If you are using Maven, include the following in your pom.xml:

xml

Copy code

<dependencies>

<dependency>

<groupId>org.seleniumhq.selenium</groupId>

<artifactId>selenium-java</artifactId>

<version>4.21.0</version>

</dependency>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-api</artifactId>

<version>5.9.0</version>

</dependency>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-engine</artifactId>

<version>5.9.0</version>

<scope>test</scope>

</dependency>

</dependencies>

#### Step 2: Create the Login Test Class

Here is an example of a login test class that tests the login form.

java

Copy code

import org.junit.jupiter.api.AfterEach;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

import static org.junit.jupiter.api.Assertions.\*;

public class LoginTest {

private WebDriver driver;

@BeforeEach

public void setUp() {

// Set the path for the WebDriver executable (Chrome in this case)

System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");

driver = new ChromeDriver();

driver.get("http://localhost:8080/login"); // URL of the login page

}

@Test

public void testValidLogin() {

WebElement usernameField = driver.findElement(By.id("username"));

WebElement passwordField = driver.findElement(By.id("password"));

WebElement loginButton = driver.findElement(By.id("loginButton"));

usernameField.sendKeys("validUser"); // Replace with a valid username

passwordField.sendKeys("validPassword"); // Replace with a valid password

loginButton.click();

String expectedUrl = "http://localhost:8080/dashboard"; // URL after successful login

assertEquals(expectedUrl, driver.getCurrentUrl());

}

@Test

public void testInvalidLogin() {

WebElement usernameField = driver.findElement(By.id("username"));

WebElement passwordField = driver.findElement(By.id("password"));

WebElement loginButton = driver.findElement(By.id("loginButton"));

usernameField.sendKeys("invalidUser"); // Invalid username

passwordField.sendKeys("invalidPassword"); // Invalid password

loginButton.click();

WebElement errorMessage = driver.findElement(By.id("errorMessage"));

assertTrue(errorMessage.isDisplayed());

assertEquals("Invalid username or password.", errorMessage.getText());

}

@Test

public void testEmptyFields() {

WebElement loginButton = driver.findElement(By.id("loginButton"));

loginButton.click();

WebElement usernameError = driver.findElement(By.id("usernameError"));

WebElement passwordError = driver.findElement(By.id("passwordError"));

assertTrue(usernameError.isDisplayed());

assertEquals("Username is required.", usernameError.getText());

assertTrue(passwordError.isDisplayed());

assertEquals("Password is required.", passwordError.getText());

}

@AfterEach

public void tearDown() {

driver.quit(); // Close the browser after each test

}

}

### Explanation of the Code

* **Setup and Teardown**:
  + setUp(): Initializes the WebDriver and opens the login page before each test.
  + tearDown(): Closes the browser after each test.
* **Test Cases**:
  + testValidLogin(): Tests logging in with valid credentials and verifies the redirection to the dashboard.
  + testInvalidLogin(): Tests logging in with invalid credentials and checks for the display of an error message.
  + testEmptyFields(): Tests the login button click without entering credentials and checks for error messages indicating that fields are required.

Q7: Explain how CRUD operations are mapped to SQL statements, with suitable example.

CRUD operations—Create, Read, Update, and Delete—are the fundamental operations that can be performed on data in a database. Each of these operations corresponds to specific SQL statements. Here’s a breakdown of how each CRUD operation maps to SQL statements, along with examples.

**1. Create (Insert)**

**SQL Statement**: INSERT INTO

The **Create** operation is used to add new records to a table.

**Example**: Inserting a new user into a users table.

sql

Copy code

INSERT INTO users (username, email, password)

VALUES ('john\_doe', 'john@example.com', 'password123');

**2. Read (Select)**

**SQL Statement**: SELECT

The **Read** operation retrieves data from the database. It can be used to get specific records or all records from a table.

**Example**: Selecting all users from the users table.

sql

Copy code

SELECT \* FROM users;

**Example**: Selecting a specific user by username.

sql

Copy code

SELECT \* FROM users WHERE username = 'john\_doe';

**3. Update**

**SQL Statement**: UPDATE

The **Update** operation modifies existing records in a table.

**Example**: Updating the email of a user with the username 'john\_doe'.

sql

Copy code

UPDATE users

SET email = 'john\_new@example.com'

WHERE username = 'john\_doe';

**4. Delete**

**SQL Statement**: DELETE FROM

The **Delete** operation removes records from a table.

**Example**: Deleting a user with the username 'john\_doe'.

sql

Copy code

DELETE FROM users

WHERE username = 'john\_doe';

**Summary of CRUD Operations and Their SQL Mappings**

| **CRUD Operation** | **SQL Statement** | **Example** |
| --- | --- | --- |
| Create | INSERT INTO | INSERT INTO users (username, email, password) VALUES ('john\_doe', 'john@example.com', 'password123'); |
| Read | SELECT | SELECT \* FROM users; or SELECT \* FROM users WHERE username = 'john\_doe'; |
| Update | UPDATE | UPDATE users SET email = 'john\_new@example.com' WHERE username = 'john\_doe'; |
| Delete | DELETE FROM | DELETE FROM users WHERE username = 'john\_doe'; |

Q8: (a) Write the unit test case to execute it as a user with RequestPostProcessor for the URL pattern “/” which returns model attribute with key as “message” and value as “Hello World”.

To write a unit test case that executes a request with a RequestPostProcessor for the URL pattern /, returning a model attribute with the key as message and value as Hello World, you can use **Spring Test** along with **MockMvc**. Here's a step-by-step guide, including the necessary code.

### Step 1: Setting Up Your Spring Controller

First, ensure you have a Spring controller that maps the root URL ("/") and returns the desired model attribute.

#### Example Controller

java

Copy code

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.GetMapping;

@Controller

public class HelloController {

@GetMapping("/")

public String hello(Model model) {

model.addAttribute("message", "Hello World");

return "hello"; // Assuming this returns a view named "hello"

}

}

### Step 2: Writing the Unit Test Case

Now, let’s write the unit test case using **MockMvc** with a RequestPostProcessor to test this functionality.

#### Example Unit Test Case

java

Copy code

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.http.MediaType;

import org.springframework.test.web.servlet.MockMvc;

import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;

import org.springframework.test.web.servlet.result.MockMvcResultMatchers;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.model;

@SpringBootTest

@AutoConfigureMockMvc

public class HelloControllerTest {

@Autowired

private MockMvc mockMvc;

@Test

public void testHelloWorldMessage() throws Exception {

mockMvc.perform(MockMvcRequestBuilders.get("/")

.contentType(MediaType.TEXT\_HTML))

.andExpect(MockMvcResultMatchers.status().isOk())

.andExpect(model().attribute("message", "Hello World"));

}

}

### Explanation of the Code

1. **Annotations**:
   * @SpringBootTest: This annotation is used to create a Spring application context for the test.
   * @AutoConfigureMockMvc: This annotation auto-configures the MockMvc instance, allowing us to perform requests against the controller without starting the entire server.
2. **MockMvc**:
   * This is a central class in Spring MVC Test that allows for the execution of requests against the controller and the verification of results.
3. **Test Method**:
   * mockMvc.perform(...): This performs a GET request to the root URL ("/").
   * contentType(MediaType.TEXT\_HTML): Sets the content type of the request to text/html.
   * .andExpect(...): These methods assert the expected outcomes:
     + status().isOk(): Asserts that the response status is HTTP 200 (OK).
     + model().attribute(...): Asserts that the model contains an attribute with the key message and the value Hello World.
4. What is Role-based Login? Explain how user’s access can be restricted using Role-based Login.

Role-based login is an authentication and authorization mechanism that restricts access to system resources based on the roles assigned to a user. This approach is widely used in web applications and enterprise software to ensure that users can only perform actions and access data that are appropriate for their roles within an organization. Here’s a detailed explanation of how role-based login works and how user access can be restricted using this mechanism.

### What is Role-based Login?

1. **Roles**: In a role-based access control (RBAC) system, roles are defined based on job functions or responsibilities. For example, common roles in a system may include:
   * **Admin**: Full access to all features and data.
   * **Manager**: Limited access to user management and reporting features.
   * **User**: Basic access to view and edit their own data.
2. **Users**: Each user is assigned one or more roles, which dictate what they can see and do within the application.
3. **Access Control**: When a user attempts to access a resource (like a webpage or an API endpoint), the application checks the user's roles and determines whether they have the necessary permissions to access that resource.

### How User Access Can Be Restricted Using Role-based Login

1. **Role Assignment**:
   * When a user is created in the system, they are assigned one or more roles. This can be done through an admin panel or during user registration.
   * For example, a user named "Alice" might be assigned the role of "Manager."
2. **Authorization Checks**:
   * When a user tries to access a specific resource, the system checks their assigned roles against the required roles for that resource.
   * For example, if a user tries to access an admin dashboard, the system checks if the user has the "Admin" role.
3. **Access Control Lists (ACL)**:
   * The application maintains a list of which roles have access to which resources. This can be implemented using configuration files, databases, or annotations in code.
   * For example, the admin role might have access to the following resources:
     + View user profiles
     + Edit user roles
     + Access system settings
4. **Implementation Example**: Here’s a simplified example of how you might implement role-based login in a web application:

#### Pseudocode Example

java

Copy code

public class User {

private String username;

private List<Role> roles;

// Constructor, getters, and setters

}

public enum Role {

ADMIN, MANAGER, USER;

}

// Service for authorization

public class AuthorizationService {

public boolean hasAccess(User user, String resource) {

if (resource.equals("adminDashboard") && user.getRoles().contains(Role.ADMIN)) {

return true;

} else if (resource.equals("managerDashboard") && user.getRoles().contains(Role.MANAGER)) {

return true;

} else if (resource.equals("userProfile") && user.getRoles().contains(Role.USER)) {

return true;

}

return false; // No access

}

}

// Usage

User alice = new User("Alice", Arrays.asList(Role.MANAGER));

AuthorizationService authService = new AuthorizationService();

boolean canAccessAdminDashboard = authService.hasAccess(alice, "adminDashboard"); // Returns false

boolean canAccessManagerDashboard = authService.hasAccess(alice, "managerDashboard"); // Returns true

### Benefits of Role-based Login

1. **Enhanced Security**: By restricting access based on roles, you minimize the risk of unauthorized access to sensitive data and functionalities.
2. **Simplified User Management**: Managing permissions through roles rather than individual users simplifies user management, especially in large organizations.
3. **Scalability**: As new roles and permissions are needed, they can be added to the system without affecting existing configurations.
4. **Auditing and Compliance**: Role-based access control makes it easier to audit user activities and ensure compliance with regulatory requirements by tracking what actions were taken by users based on their roles.

Write short notes on the following: (15 Marks)

(a) JSP Standard Tag Library (JSTL)

(b) Spring Framework

(c) Cross Site Request Forgery (CSRF)

### (a) JSP Standard Tag Library (JSTL)

**JSTL** is a collection of tags that encapsulates the core functionality common to many JSP applications, making it easier to work with JSP pages. It allows developers to write cleaner and more maintainable code by providing a set of reusable components for tasks such as iteration, conditionals, and formatting.

#### Key Features:

* **Core Tags**: For basic operations like looping and conditional statements (e.g., <c:forEach>, <c:if>).
* **Formatting Tags**: For formatting numbers, dates, and other data types (e.g., <fmt:formatNumber>).
* **SQL Tags**: For database operations (e.g., <sql:query>).
* **XML Tags**: For XML manipulation (e.g., <x:out>).

#### Benefits:

* Reduces Java code in JSPs, enhancing readability.
* Simplifies common tasks, making JSP development faster and more efficient.
* Provides a standardized way of handling operations in JSP.

### (b) Spring Framework

**Spring Framework** is a comprehensive framework for building enterprise applications in Java. It provides a wide range of features for developing robust applications, including dependency injection, aspect-oriented programming, transaction management, and support for various data access technologies.

#### Key Components:

* **Core Container**: Provides the fundamental building blocks of the framework (e.g., IoC container).
* **Spring MVC**: A web framework for building web applications with a Model-View-Controller architecture.
* **Data Access/Integration**: Supports various data access technologies, including JDBC, JPA, and Hibernate.
* **Aspect-Oriented Programming (AOP)**: Enables the separation of cross-cutting concerns, such as logging and transaction management.
* **Security**: Provides a comprehensive security framework for authentication and authorization.

#### Benefits:

* Promotes loose coupling and easier testability through dependency injection.
* Simplifies complex configurations with annotations and Java-based configuration.
* Enhances productivity with various features and integrations.

### (c) Cross-Site Request Forgery (CSRF)

**CSRF** is a type of attack that tricks a user into executing unwanted actions on a web application in which they are authenticated. This can lead to unauthorized actions being performed on behalf of the user without their consent.

#### How CSRF Works:

1. A user is logged into a website (e.g., a banking site).
2. The attacker sends a malicious link or script to the user.
3. When the user clicks the link, the browser sends a request to the banking site using the user's credentials (stored in cookies).
4. The banking site processes the request as if it was initiated by the legitimate user.

#### Prevention Measures:

* **Anti-CSRF Tokens**: Generate unique tokens for each session or request that must be included in forms or requests.
* **SameSite Cookies**: Use cookies with the SameSite attribute to restrict how cookies are sent with cross-site requests.
* **Referer Header Check**: Validate the referer header to ensure requests originate from the same site.
* **User Confirmation**: Require users to confirm sensitive actions, such as changing passwords or making transactions.