INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

Dundigal, Hyderabad - 500 043

COURSE CONTENT

JAVA FULL STACK DEVELOPMENT LABORATORY								
IV Semester: Common for all branches								
Course Code	**Category**	**Hours / Week**			**Credits**	**Maximum Marks**		
ACSE20	**Core**	**L**	**T**	**P**	**C**	**CIA**	**SEE**	**Total**
		1	0	2	2	40	60	100
Contact Classes: 16	**Tutorial Classes: Nil**	**Practical Classes: 32**				**Total Classes: 48**		
Prerequisite: Object Oriented Programming, Web Systems Engineering								

I. COURSE OVERVIEW:

This course covers core Java concepts including exception handling, collections, and I/O streams. It introduces Java 8 features like lambda expressions and streams for efficient coding. Students learn advanced Java with database integration using JDBC and Hibernate. The course includes Spring Framework, Spring Boot, RESTful APIs, and responsive UI design. It concludes with API testing using Postman and deploying Java applications using Docker.

II. COURSES OBJECTIVES:

The students will try to learn

I. The foundations in core Java concepts, including collections, exception handling, I/O streams, and Java 8 features.
II. The database integration techniques using JDBC and Hibernate for building data-driven web applications.
III. The Spring Framework and its components such as IoC, DI, and Spring MVC for developing enterprise-level applications.
IV. The development of micro-services using Spring Boot and RESTful APIs, along with responsive web design practices.
V. The practical tools like Postman and Docker for testing APIs and deploying containerized Java applications.

III. COURSE OUTCOMES:

At the end of the course students should be able to:

CO 1	**Apply** Java Collections and Java 8 features to develop efficient data manipulation solutions using streams and functional interfaces.	**Apply**
CO 2	**Implement** exception handling, I/O operations, and serialization techniques for building robust Java applications.	**Apply**
CO 3	**Develop** database-integrated web applications using session management and Hibernate ORM with entity relationships in project showcasing user sessions and CRUD operations.	**Create**
CO 4	**Construct** enterprise-level applications using Spring Framework components in implementing of Spring-based modules.	**Create**
CO 5	**Design** micro-services and REST APIs using Spring Boot with responsive UI and Spring Data JPA for database interaction.	**Create**
CO 6	**Demonstrate** API testing using Postman and application deployment through Docker containerization techniques for functional REST APIs and Dockerized Java applications.	**Understand**

IV. COURSE SYLLABUS:

MODULE - I COLLECTIONS AND STREAMS
Exception Handling, Collections Framework – List, Set, Map interfaces and inherited classes. I/O Streams – Byte and Character streams, Serialization. Introduction to Java 8 features – Lambda Expressions, Streams and Functional Interfaces.

MODULE - II ADVANCE JAVA WITH DATABASE INTEGRATION
Session Management – Cookies, HttpSession, URL rewriting, Hibernate ORM – Configuration.
Mapping Java classes to database tables, Relationships – One-to-One, One-to-Many, Many-to-Many.

MODULE - III SPRING FRAMEWORK
Introduction to Spring Framework – Features, Architecture, Spring Core – Inversion of Control (IoC), Dependency Injection (DI).

Spring Bean lifecycle, Spring MVC – DispatcherServlet, Controllers.

MODULE - IV: MICROSERVICES-BASED WITH SPING BOOT
Responsive Web Designing (RWD) for building adaptable UIs across devices. Introduces Spring Boot with Microservices, including Auto Configuration. REST API development, and database integration using Spring Data JPA and Hibernate.

MODULE - V: INTRODUCTION TO DOCKER
Postman for testing REST APIs, Introduction to Docker – Images, Containers, Dockerizing Java applications.

V. TEXT BOOKS:

1. Kathy Sierra, Bert Bates, Head First Java, O'Reilly Media, 2nd edition, 2005
2. Juha Hinkula, Full Stack Development with Spring Boot and React, Packt publishing, 1st edition, 2022.

VI. REFERENCE BOOKS:

1. Arun Gupta, Docker for Java Developers, O'Reilly Media, 1st edition, 2017.
2. Greg L. Turnquist, Learning Spring Boot 3.0, Packt Publishing, 3rd edition 2022.
3. Craig Walls, Spring in Action, Manning Publications, 6th edition, 2022.
4. Anghel Leonard, Pro Hibernate and MongoDB, Apresss, 1st edition 2013.

VII. ELECTRONICS RESOURCES:

1. https://www.geeksforgeeks.org/java/
2. https://www.javatpoint.com/java-full-stack
3. https://spring.io/guides
4. https://www.w3schools.com/react/
5. https://www.baeldung.com/

JAVA FULL STACK

1. Collections and Streams

1.1 Exception Handling

Java provides a robust mechanism to handle runtime errors through Exception Handling.

Types of Exceptions: Checked, Unchecked

Keywords: try, catch, finally, throw, throws

```
try
{
    int result = 10 / 0;
}
catch (ArithmeticException e)
{
    System.out.println("Cannot divide by zero");
}
finally
{
    System.out.println("Always executed");
}
```

1.2 Collections Framework

A unified architecture for representing and manipulating collections.
List: Ordered collection (e.g., ArrayList, LinkedList)

Set: No duplicate elements (e.g., HashSet, TreeSet)

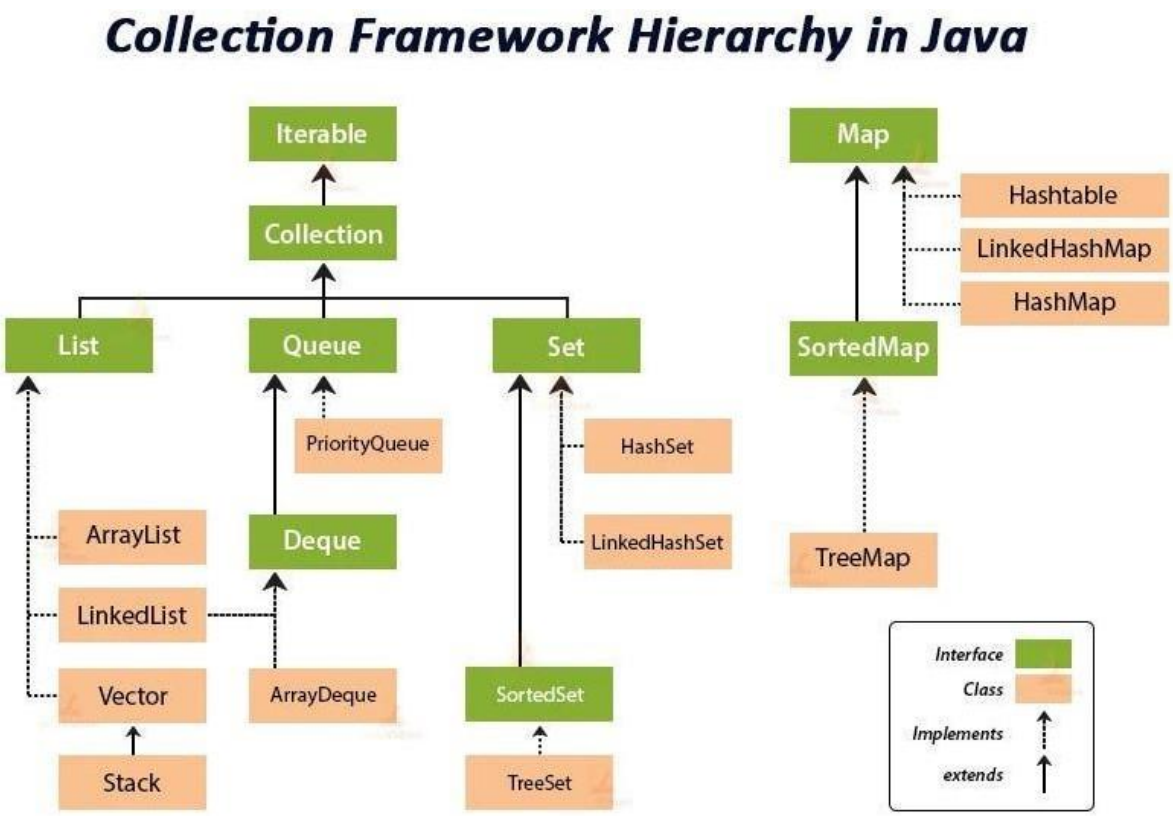
Map: Key-value pairs (e.g., HashMap, TreeMap)

List example:

```
List<String> list = new ArrayList<>();
list.add("Apple");
list.add("Banana");

Set<Integer> set = new HashSet<>();
set.add(10);
set.add(20);

Map<Integer, String> map = new HashMap<>();
map.put(1, "One");
map.put(2, "Two");
```



Each with for-each loop, for loop, iterator, and forEach() lambda:

List Example (ArrayList):

```
import java.util.*;
public class ListExample {
    public static void main(String[] args) {
        List<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Mango");
        System.out.println("For-each loop:");
        for (String fruit : fruits) {
            System.out.println(fruit);
        }
        System.out.println("\nTraditional for loop:");
        for (int i = 0; i < fruits.size(); i++) {
            System.out.println(fruits.get(i));
        }
        System.out.println("\nUsing iterator:");
        Iterator<String> it = fruits.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
```

```
        }
        System.out.println("\nUsing forEach and lambda:");
        fruits.forEach(f -> System.out.println(f));
    }
}
```

Set Example (HashSet):

```
import java.util.*;
public class SetExample {
    public static void main(String[] args) {
        Set<String> names = new HashSet<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");
        System.out.println("For-each loop:");
        for (String name : names) {
            System.out.println(name);
        }
        System.out.println("\nUsing iterator:");
        Iterator<String> it = names.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }
        System.out.println("\nUsing forEach and lambda:");
        names.forEach(name -> System.out.println(name));
    }
```

Map Example (HashMap)

```
import java.util.*;
public class MapExample {
    public static void main(String[] args) {
        Map<Integer, String> students = new HashMap<>();
        students.put(101, "John");
        students.put(102, "Emma");
        students.put(103, "David");
        System.out.println("For-each loop on entry set:");
        for (Map.Entry<Integer, String> entry : students.entrySet()) {
```

```
            System.out.println("ID: " + entry.getKey() + ", Name: " +
entry.getValue());
        }
        System.out.println("\nUsing keySet and get():");
        for (Integer id : students.keySet()) {
            System.out.println("ID: " + id + ", Name: " +
students.get(id));
        }
        System.out.println("\nUsing forEach and lambda:");
        students.forEach((id, name) -> System.out.println("ID: " + id + ",
Name: " + name));
    }
}
```

1.3 I/O Streams

Used for reading and writing data (files, memory, etc.)

Byte Streams: InputStream, OutputStream

In Java, **FileInputStream** and **FileOutputStream** are part of the java.io package and are used for **reading from and writing to files** in the form of **raw bytes** (binary data). They are most commonly used for reading and writing binary files such as images, audio, or other non-text files.

FileInputStream (Reading Data from File)

Used to read bytes from a file.

Syntax:

```
FileInputStream fis = new FileInputStream("file.txt");
int data = fis.read();
```

```
import java.io.FileInputStream;
import java.io.IOException;

public class ReadFileExample {
   public static void main(String[] args) {
     try {
       FileInputStream fis = new FileInputStream("example.txt");
       int i;
       while ((i = fis.read()) != -1) {
                System.out.print((char) i);
            }
            fis.close();
```

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

FileOutputStream (Writing Data to File)

Used to write bytes to a file.

Syntax:

```
FileOutput: Stream fos = new FileOutputStream("file.txt");
fos.write(byteData);
```

Character Streams: Reader, Writer

Serialization: Saving object state

```
ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("data.ser"));
out.writeObject(new MyClass());
out.close();
```

1.4 Java 8 Features

Lambda Expressions: (a, b) -> a + b

Lambda Expressions with Custom Functional Interfaces:

- Use lambda to implement custom logic:

```
@FunctionalInterface
interface Calculator {
   int operate(int a, int b);
}
Calculator add = (a, b) -> a + b;
```

Streams API: Process collections with map(), filter(), reduce() methods.

Stream API Advanced Operations:

```
 Intermadiate methods: map(), filter(), flatMap(), reduce(), sorted(), distinct(), limit(), skip()
Terminal methods: collect(), forEach(), count(), findAny(), allMatch(), etc.
```

Collectors and Collectors API:

- Used with streams to collect data:

```
List<String> names = Arrays.asList("rama","Krishna","shiva");
```

```
List<String> list = names.stream().collect(Collectors.toList());
Set<String> set = names.stream().collect(Collectors.toSet());
Map<Integer, String> map = list.stream().collect(Collectors.toMap(String::length, Function.identit
```

Parallel Streams:

- For multi-core parallelism in stream processing:

```
list.parallelStream().forEach(System.out::println);
```

Functional Interfaces: @FunctionalInterface, e.g., Runnable, Comparator

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4);
numbers.stream()
    .filter(n -> n % 2 == 0)
    .forEach(System.out::println);
```

Date and Time API :

- Replaces the old Date and Calendar classes
- Classes: LocalDate, LocalTime, LocalDateTime, ZonedDateTime, Period, Duration

```
LocalDate today = LocalDate.now();
LocalDate birthday = LocalDate.of(1995, Month.JANUARY, 1);
```

1.5 Mini Project – Student Management System

Objective: To build a simple console-based application that manages student records using:

• Collections Framework (List, Map)
• Exception Handling
• File I/O and Serialization
• Java 8 Features (Streams, Lambda expressions, Functional Interfaces)

Tools & Technologies

- Java SE 8+
- IDE: IntelliJ IDEA / Eclipse / VS Code
- JDK installed (8+)

Features

- Add Student:
  - Input: Name, Roll Number, Grade
  - Store in a List and Map collection.
- Delete Student
  - Remove based on Roll Number.
- View All Students
  - Display all student details in tabular form.
- Filter Students by Grade
  - Use Java 8 Streams to display only students matching a given grade.
- Save and Load Student Data

- Use Serialization to persist the student list to a file.

Deliverables

- Student.java (data class)
- StudentManagementSystem.java (main logic)
- students.ser (file for persistent storage)
- Documentation/README for compilation and usage

```
class Student {
    int id;
    String name;
    double grade;
}
List<Student> students = new ArrayList<>();
students.add(new Student(1, "John", 85));
students.stream()
        .filter(s -> s.grade > 80)
        .forEach(s -> System.out.println(s.name));
```

1.6 Questions and Answers

1. What are the main components of Java Collections Framework?
Ans. Interfaces (List, Set, Map) and classes (ArrayList, HashMap, etc.)

2. Difference between ArrayList and LinkedList?
Ans. ArrayList is backed by an array; LinkedList uses doubly linked list.

3. What is the purpose of Set in Java?
Ans.To store unique elements.

4. What is a Map?
Ans. A key-value pair collection.

5. What is the difference between InputStream and Reader?
Ans. InputStream is for byte data; Reader is for character data.

6. What is Serialization in Java?
Ans. Converting object state to byte stream.

7. What is a lambda expression?
Ans. A concise way to write anonymous functions.

8. What is the Streams API?
Ans. A Java 8 feature for processing collections declaratively.

9. What is a functional interface?
Ans. Interface with exactly one abstract method.

10. Give an example of using filter in Streams.
Ans. list.stream().filter(s -> s.startsWith("A"));

11. How can you remove duplicate elements from a list using Java Streams?

Answer:

```
List<String> uniqueNames = names.stream()
                                .distinct()
                                .collect(Collectors.toList());
System.out.println(uniqueNames);
distinct() is a stream List<String> names = Arrays.asList("A", "B", "A", "J");
```

- method that returns a stream with unique elements.

How do you sort a list of custom objects using Streams?
Answer:
Suppose you have a Person class:

```
class Person {
    String name;
    int age;

    // constructor, getters, setters
}
```

2. Advanced Java with Database Integration

2.1 Session Management

Session Management Session management is a way to persist user information across multiple pages. Common techniques include:

- **Cookies**: are small text files stored on the client's browser, used to save user-specific information like login details or preferences. They are sent with every HTTP request to the server. Cookies can be persistent (saved across sessions) or temporary (deleted after the session ends).
- **HttpSession**: is a server-side mechanism to store user data between requests. When a user accesses the server, a unique session ID is generated and maintained (usually via cookies). It allows storing objects like user info without exposing data to the client.
- **URL Rewriting**: is a technique where session information (like session ID) is appended to the URL query stringIt.is useful when cookies are disabled. Example: http://example.com/page?sessionId=1234. It keeps the session alive without relying on cookies.

Cookie Example:

CookieServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class CookieServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
```

```
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        // Create a cookie
        Cookie cookie = new Cookie("user", "Alice");
        response.addCookie(cookie);
        out.println("Cookie set for user=Alice");
    }
}
```

HttpSession

HttpSession is an interface provided by the **Servlet API** in Java. It is used to **store and manage user-specific data** between multiple requests in a web application.

Since HTTP is a **stateless protocol**, the server does not remember any user information between requests. HttpSession helps solve this problem by storing data on the server side for each user.

Key Features of HttpSession:

- Stores **user-specific data** (like login info, preferences).
- Maintains data across **multiple HTTP requests**.
- Automatically created when needed (request.getSession()).
- Can be **invalidated** when the user logs out.
- Has a **timeout** period (can be configured).

SessionServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SessionServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        // Get or create session
        HttpSession session = request.getSession();
        session.setAttribute("username", "Bob");
        out.println("Session created. Username stored: Bob");
```

```
    }
}
```

URL Rewriting Example

LoginServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;


public class LoginServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();


        // Append data in URL
        String username = "Charlie";
        out.println("<a href='WelcomeServlet?user=" + username + "'>Click
here</a>");
    }
}
```

WelcomeServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class WelcomeServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String user = request.getParameter("user");
        out.println("Welcome via URL rewriting, " + user);
    }
}
```

2.2 Hibernate ORM (Object Relational Mapping)

Hibernate is an ORM framework to map Java classes to database tables.

- **Configuration**: Done using hibernate.cfg.xml and annotated classes.
- **Entity Classes**: Java classes annotated with @Entity, @Id, etc.
- **Relationships**:
    - **One-to-One**: Each entity has exactly one related entity.
    - **One-to-Many**: One entity is related to many others.
    - **Many-to-Many**: Both entities have a collection of each other.

```
@Entity
@Table(name = "students")
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;

    @OneToMany(mappedBy = "student", cascade = CascadeType.ALL)
    private List<Course> courses;
}
```

2.3 Mini Project – Library Management System

Objective: This project aims to efficiently manage library operations including the handling of books, registered users, and their interactions. It allows staff to maintain an inventory of books and record user details. The system supports issuing books to users and tracking their return. It ensures the availability status of books and maintains transactional data for reporting and control.

Features: Tools & Technologies:

• Java SE 8+
• Hibernate ORM
• MySQL
• IDE: IntelliJ IDEA / Eclipse / VS Code

Technologies Used: Java, Hibernate ORM, MySQL

Key Features:

- Add/Remove books
- Register users
- Issue and return books

Database Tables:

- Book(id, title, author, available)

- User(id, name, email)
- Transaction(id, user_id, book_id, issue_date, return_date)

```
public class Book {
    private int id;
    private String title;
    private String author;
    private boolean available;
}
public class User {
    private int id;
    private String name;
    private String email;
}
public class Transaction {
    private int id;
    private User user;
    private Book book;
    private LocalDate issueDate;
    private LocalDate returnDate;
}
```

Book.hbm.xml

```
<hibernate-mapping>
    <class name="Book" table="Book">
        <id name="id" column="id">
            <generator class="increment"/>
        </id>
        <property name="title"/>
        <property name="author"/>
        <property name="available"/>
    </class>
</hibernate-mapping>
```

2.4 Questions and Answers

1. What is session management?
Ans. Mechanism to maintain state across multiple HTTP requests.

2. Difference between cookie and HttpSession?
Ans. Cookies are stored on the client; HttpSession is stored on the server.

3. Why use Hibernate over JDBC?

Ans. Hibernate automates SQL generation and handles relationships more easily.

4. What is @Entity in Hibernate?
Ans. Annotation to mark a class as a persistent entity.

5. What is the use of SessionFactory?
Ans. It creates Session objects to interact with the DB.

6. Explain One-to-Many relationship.
Ans. One entity is associated with a collection of another entity.

7. What is lazy loading in Hibernate?
Ans. Objects are not loaded until they are accessed.

8. What is the use of @Table annotation?
Ans. Specifies the table name in DB for the entity.

9. Can we use Hibernate with multiple databases?
Ans. Yes, with separate configuration files.

10. How does Hibernate handle transactions?
Ans. With Transaction interface and beginTransaction() method.

3. Spring Frame Work

3.1 Introduction to Spring

Spring is a lightweight framework used for building Java applications.

- **Core Features**: IoC, DI, AOP, Transaction Management
- **Modules**: Spring Core, Spring MVC, Spring Boot, Spring Security

3.2 Inversion of Control (IoC)and Dependency Injection(DI)

- **IoC**: Objects are created and managed by Spring container.
- **DI**: Dependencies are injected by Spring rather than created by the class itself.

```
package com.example;


import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;


public class App {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");
        NotificationManager manager = (NotificationManager)
context.getBean("notificationManager");
        manager.notifyUser("Hello from Spring (no annotations)!");
    }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<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">
    <!-- Bean for EmailService -->
    <bean id="emailService" class="com.example.EmailService" />
    <!-- Bean for NotificationManager with DI -->
    <bean id="notificationManager" class="com.example.NotificationManager">
        <constructor-arg ref="emailService" />
    </bean>
</beans>
```

3.4 Spring Bean Lifecycle

- **Lifecycle methods:** init-method, destroy-method
- Managed by Spring Container

Lifecycle methods:
init-method

- Called **immediately after the bean is initialized**.
- Use this to write any **startup logic**: opening a file, initializing a connection, etc.

destroy-method

- Called **before the bean is destroyed** (when Spring context sh
- uts down).
- Use this for **cleanup tasks**: closing resources, releasing memory, saving state, etc.

Example: XML Configuration

```
<bean id="myBean" class="com.example.MyBean"
      init-method="myInit"
      destroy-method="myDestroy" />
```

Java Bean Class:

```
public class MyBean {
    public void myInit() {
        System.out.println("Initializing bean...");
    }
    public void myDestroy() {
        System.out.println("Destroying bean...");
    }
}
```

Java-based Configuration Example:

```
@Configuration
public class AppConfig {
    @Bean(initMethod = "myInit", destroyMethod = "myDestroy")
    public MyBean myBean() {
        return new MyBean();
    }
}
```

Using Annotations (Alternative Approach):

If you don't want to use init-method/destroy-method in XML or Java config, you can use annotations:

```
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
public class MyBean {
    @PostConstruct
    public void init() {
        System.out.println("PostConstruct: Bean initialized");
    }
```

```
    @PreDestroy
    public void destroy() {
        System.out.println("PreDestroy: Bean will be destroyed");
    }
}
```

3.4 Spring MVC Architecture

- **DispatcherServlet**: Central controller
- **Controller**: Handles user input
- **Model**: Data layer
- **View**: Presentation layer

DispatcherServlet:

- It is the **front controller** in Spring MVC.
- Receives all **incoming HTTP requests**.
- Delegates requests to the **appropriate controller** based on configuration.
- Coordinates between **Controller, ViewResolver, and Model**.
- Configured in web.xml or via Spring Boot auto-configuration.

web.xml – To load DispatcherServlet

```
<web-app>
  <servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```

dispatcher-servlet.xml – Spring configuration

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:context="http://www.springframework.org/schema/context"
```

```
       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
         http://www.springframework.org/schema/mvc
         http://www.springframework.org/schema/mvc/spring-mvc.xsd
         http://www.springframework.org/schema/context
         http://www.springframework.org/schema/context/spring-
context.xsd">
    <context:component-scan base-package="com.example" />
    <mvc:annotation-driven/>
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/views/"/>
        <property name="suffix" value=".jsp"/>
    </bean>
</beans>
```

Model:

- Represents the **application data**, usually fetched from the **database**.
- Often composed of **POJOs (Plain Old Java Objects)**.
- In Spring MVC, the controller adds data to the model using Model, ModelMap, or ModelAndView.

Model – Employee.java

```
package com.example.model;
public class Employee {
    private String name;
    private String department;
    // Getters and Setters
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public String getDepartment() { return department; }
    public void setDepartment(String department) { this.department =
department; }
}
```

Controller – EmployeeController.java

```
package com.example.controller;
import com.example.model.Employee;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class EmployeeController {
    @RequestMapping("/show")
    public String showEmployee(Model model) {
        Employee emp = new Employee();
        emp.setName("Ravi");
        emp.setDepartment("IT");
        model.addAttribute("employee", emp);
        return "employeeView";
    }
}
```

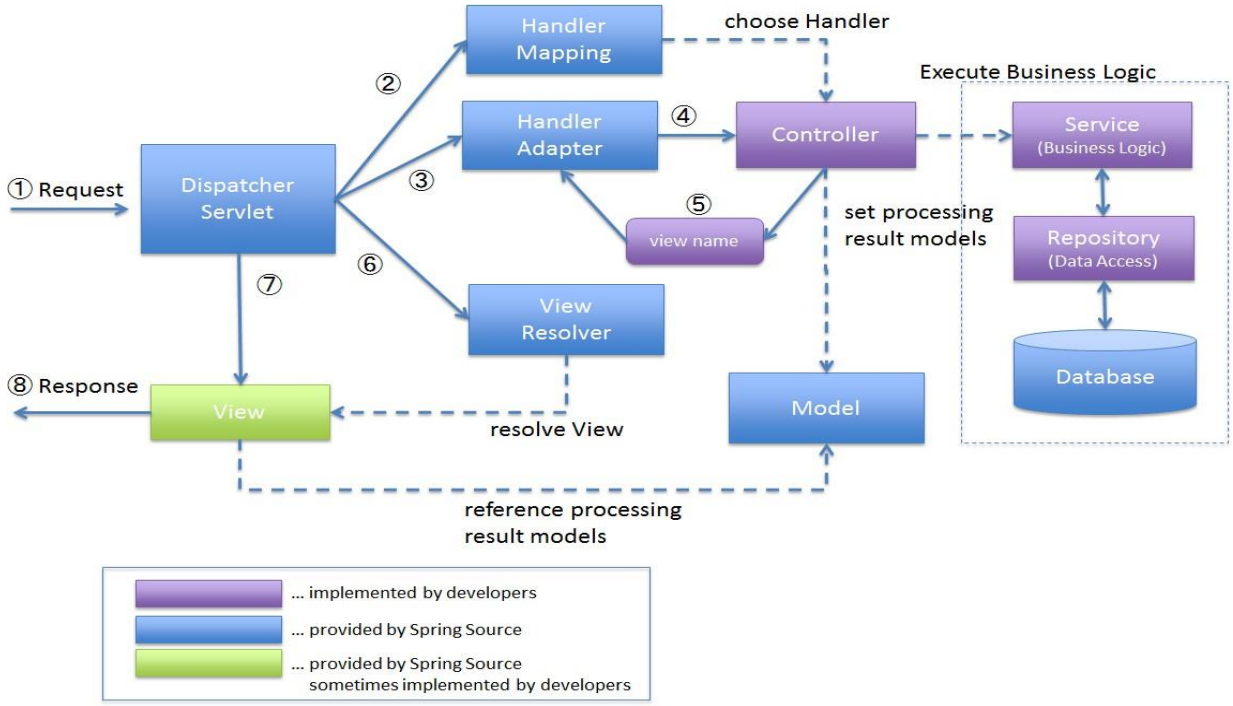
View : The Presentation Layer

- Responsible for **displaying the response** to the user.
- The controller returns the **view name**, and the view resolver maps it to an actual file (like JSP, Thymeleaf, etc.).
- Receives data from the model and renders it in HTML.

View – employeeView.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head><title>Employee Info</title></head>
<body>
    <h2>Employee Details</h2>
    <p>Name: ${employee.name}</p>
    <p>Department: ${employee.department}</p>
</body>
</html>
```

Spring MVC Architecture:




```
@Controller
public class HelloController {
    @RequestMapping("/hello")
    public String sayHello(Model model) {
        model.addAttribute("message", "Hello, Spring MVC!");
        return "hello.jsp";
    }
}
```

3.5 Mini Project – Employee Management System

Objective:
Implement CRUD operations (Create, Read, Update, Delete) for Employee data using Spring MVC framework. Use Spring Controller to handle HTTP requests and JSP/Thymeleaf for views. Connect to a database (like MySQL) using Spring JDBC or Hibernate/JPA. Enable routing via Spring DispatcherServlet and map views through a Model-View-Controller structure.

Features:

- Add/Update/Delete/View employees

Technologies:

- Spring MVC
- Spring Core
- JSP
- MySQL
- Spring JDBC / Hibernate (choose one)

Structure of the project:

```
EmployeeCRUD/
├── src/
│   └── com/example/controller/
│       └── EmployeeController.java
│   └── com/example/dao/
│       └── EmployeeDAO.java
│   └── com/example/model/
│       └── Employee.java
│   └── com/example/service/
│       └── EmployeeService.java
│
├── WebContent/
│   ├── WEB-INF/
│   │   ├── views/
│   │   │   ├── add.jsp
│   │   │   ├── edit.jsp
│   │   │   ├── list.jsp
│   │   └── web.xml
│   │   └── dispatcher-servlet.xml
```

```
public class Employee {
   private int id;
   private String name;
   private double salary;
   // getters, setters
}
```

```
@Controller
public class EmployeeController {
   @Autowired
   private EmployeeService service;

   @RequestMapping("/list")
   public String listEmployees(Model model) {
       model.addAttribute("employees", service.getAll());
       return "list";
   }
}
```

```
<!-- list.jsp -->
<table>
   <c:forEach var="emp" items="${employees}">
      <tr><td>${emp.name}</td><td>${emp.salary}</td></tr>
   </c:forEach>
</table>
```

3.6 Questions and Answers

1. What is IoC?
Ans. Design principle where control of objects is transferred to a container.

2. What is DI?
Ans. Technique where an object receives its dependencies from an external source.

3. What is DispatcherServlet?
Ans. It acts as the front controller in Spring MVC.

4. What is a Spring Bean?
Ans. A Java object managed by the Spring container.

5. How are beans defined in Spring?
Ans. Through XML, annotations, or Java configuration.

6. What is the use of @Controller?
Ans. Marks a class as a web controller.

7. What is Model in Spring MVC?
Ans. Used to pass data from controller to view.

8. What is difference between BeanFactory and ApplicationContext?
Ans. ApplicationContext is a superset with more enterprise features.

9. What are the types of DI in Spring?
Ans. Constructor-based and setter-based.

10. How is ViewResolver used in Spring MVC?
Ans. It resolves the logical view name to an actual view.

4. Microservices with Spring Boot

4.1 Responsive Web Design (RWD)

- Uses CSS3 and media queries to design adaptive interfaces
- Frameworks: Bootstrap

Responsive Web Design (RWD) – HTML + CSS + Bootstrap:

Create an HTML file (index.html) that works well on mobile and desktop:

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Catalog Page</title>
  <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css">
</head>
<body>

  <div class="container text-center mt-5">
    <h2 class="mb-4">Product Catalog</h2>
    <div class="row">
       <div class="col-md-4 mb-3">
        <div class="card">
          <div class="card-body">
            <h5 class="card-title">Product 1</h5>
            <p class="card-text">Description</p>
            <button class="btn btn-primary">View</button>
          </div>
        </div>
      </div>
      <div class="col-md-4 mb-3">
        <div class="card">
          <div class="card-body">
            <h5 class="card-title">Product 2</h5>
            <p class="card-text">Description</p>
            <button class="btn btn-primary">View</button>
          </div>
        </div>
      </div>
    </div>
  </div>

</body>
</html>
```

4.2 Spring Boot

- Convention over configuration
- Auto-configures application
- Embedded Tomcat

Spring Boot Example – Simple REST API

Product.java (model):

```
public class Product {
    private Long id;
    private String name;
    private String description;
    private double price;
}
```

ProductController.java:

```
@RestController
@RequestMapping("/api/products")
public class ProductController {
    private List<Product> products = new ArrayList<>();
    @GetMapping
    public List<Product> getAll() {
        return products;
    }
    @PostMapping
    public Product add(@RequestBody Product product) {
        product.setId((long) (products.size() + 1));
        products.add(product);
        return product;
    }
}
```

CatalogApplication.java:

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

```
}
```

Run this Spring Boot app and test with Postman at http://localhost:8080/api/products

4.3 Microservices Architecture

- Decomposed services
- RESTful APIs
- Independent deployment

Each service:

- Has its own database
- Communicates using **REST APIs**
- Can be deployed independently

4.4 Spring Data JPA

- Simplifies data access
- Auto-implements repositories

```
@RestController
@RequestMapping("/products")
public class ProductController {
    @Autowired
    ProductRepository repo;
    @GetMapping
    public List<Product> list() {
        return repo.findAll();
    }
}
```

4.5 Mini Project – Product Catalog System

Objective:
Build a **Catalog Microservice** that exposes REST APIs to manage product data (create, read, update, delete). Use **Spring Boot** and **Spring MVC** to handle HTTP requests and JSON responses. Integrate with a **database (e.g., MySQL or H2)** for persistent product storage. Ensure the service is modular, scalable, and ready for integration into a larger microservices architecture.

Features:

- CRUD operations
- API testing with Postman
- H2 database integration

Technologies Used:

- Spring Boot
- Spring MVC (REST Controllers)
- H2 In-Memory Database
- Maven (for dependency management)
- Postman (for API testing)
- Java

Structure of the project:

```
catalog-service/
├── src/
│   └── main/
│       ├── java/
│       │   └── com/example/catalog/
│       │       ├── CatalogApplication.java
│       │       ├── controller/ProductController.java
│       │       ├── model/Product.java
│       │       └── repository/ProductRepository.java
│       └── resources/
│           ├── application.properties
│           └── data.sql (optional for demo data)
└── pom.xml
```

4.6 Questions and Answers

1. **What is Spring Boot?**

Ans. A framework that simplifies Spring-based development.

2. **What are microservices?**

Ans. Small, loosely coupled, independently deployable services.

3. **What is a REST API?**

Ans. Web service that uses HTTP methods.

4. **What is @RestController?**

Ans. Combines @Controller and @ResponseBody.

5. **What is @SpringBootApplication?**

Ans. Main annotation that includes @Configuration, @EnableAutoConfiguration, and @ComponentScan.

6. What is H2 database?

Ans. In-memory database useful for development and testing.

7. How do you create a Spring Boot project?

Ans. Using Spring Initializr or IDE plugin.

8. What is application.properties used for?

Ans. Configuration file in Spring Boot.

9. What is dependency injection in Spring Boot?

Ans. Automatically provides required beans using annotations like @Autowired.

10. How is Postman used in Spring Boot development?

Ans. For testing and debugging RESTful web services.

5. Introduction to Docker

5.1 Postman

What is Postman?

Postman is a popular **Graphical User Interface (GUI) tool** used to:

- Send HTTP requests to REST APIs
- View responses
- Test APIs without writing code
- GUI tool to test REST APIs
- Supports GET, POST, PUT, DELETE requests

```
src/main/java/com/example/demo/
│
├── DemoApplication.java
└── StudentController.java
```

How to use Postman:

- Open Postman.
- Select the request type (GET, POST, etc.).
- Enter URL (http://localhost:8080/api/students).
- For POST/PUT, go to **Body > raw > JSON**, and type:

```
{ "name": "YourName" }
```

Hit **Send** to see response.

5.2 Docker Overview

- **Images**: Blueprint of container
- **Containers**: Running instance of image
- **Dockerfile**: Script to create custom images

```
Code Example: Dockerfile
FROM openjdk:17
COPY target/app.jar app.jar
ENTRYPOINT ["java", "-jar", "app.jar"]
```

5.3 Mini Project - Dockerized Spring Boot Application

Objective:

- Define a Docker image for the Spring Boot application.
- Package the application into a deployable container.
- Run the container with necessary port mappings.
- Access the application from the mapped host port.

Steps:

1. Create Spring Boot app
2. Generate jar using mvn package
3. Create Dockerfile
4. Build image: docker build -t app .
5. Run container: docker run -p 8080:8080 app

1. Create Spring Boot App:

CatalogApplication.java

```
package com.example.catalog;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class CatalogApplication {
    public static void main(String[] args) {
```

```
        SpringApplication.run(CatalogApplication.class, args);
    }
}
```

ProductController.java

```
package com.example.catalog;
import org.springframework.web.bind.annotation.*;
import java.util.*;
@RestController
@RequestMapping("/products")
public class ProductController {
    private List<String> products = new ArrayList<>(List.of("Laptop", "Phone",
"Tablet"));
    @GetMapping
    public List<String> getProducts() {
        return products;
    }
}
```

```
pom.xml (minimal required for Spring Boot)
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>catalog</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>Catalog</name>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.1.0</version>
  </parent>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
```

```
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

2. Generate JAR using Maven

Open terminal and run:

```
mvn clean package
```

This creates a JAR file in target/catalog-0.0.1-SNAPSHOT.jar

3. Create Dockerfile

In the root directory:

```
# Use official Java base image
FROM openjdk:17-jdk-alpine
# Set JAR file name
ARG JAR_FILE=target/catalog-0.0.1-SNAPSHOT.jar
# Copy JAR into container
COPY ${JAR_FILE} app.jar
# Run the JAR
ENTRYPOINT ["java", "-jar", "app.jar"]
```

4. Build Docker Image

```
docker build -t catalog-app .
```

5. Run Docker Container

```
docker run -p 8080:8080 catalog-app
```

Test the App

```
 http://localhost:8080/products
["Laptop", "Phone", "Tablet"]
```

Structure of the project:

```
springboot-docker/
```

```
├── src/
│   └── main/
│       └── java/
│           └── com/example/demo/
│               └── DemoApplication.java
│               └── HelloWorldController.java
├── pom.xml
```

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);
    }
}
```

HelloWorldController.java

```
package com.example.demo;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class HelloWorldController {

    @GetMapping("/hello")
    public String sayHello() {
        return "Hello, Docker + Spring Boot!";
    }
}
```

pom.xml

Add the necessary Spring Boot dependencies:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>springboot-docker</artifactId>
    <version>1.0.0</version>
    <packaging>jar</packaging>
    <name>Spring Boot Docker Example</name>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.2.0</version> <!-- use the latest version -->
    </parent>
    <dependencies>
        <!-- Spring Web -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <!-- For Testing (optional) -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <!-- Package the app as an executable JAR -->
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

Dockerfile

This file creates a Docker image from your Spring Boot JAR.

```
# Use OpenJDK base image
FROM openjdk:17-jdk-alpine
# Set working directory inside the container
WORKDIR /app
# Copy the built JAR file
COPY target/springboot-docker-1.0.0.jar app.jar
# Expose port
EXPOSE 8080
# Run the JAR file
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Build & Run

Step 1: Package the JAR file

In terminal (from root folder):

```
mvn clean package
```

Step 2: Build Docker image

```
docker build -t springboot-docker-app .
```

Step 3: Run the container

```
docker run -p 8080:8080 springboot-docker-app
```

Test theAPI

Open a browser or use Postman:

```
GET http://localhost:8080/hello
```

Response:

```
Hello, Docker + Spring Boot!
```

5.4 Questions and Answers

1. **What is Docker?**

Ans. A platform to develop, ship, and run applications in containers.

2. **What is a Docker container?**

Ans. A lightweight, standalone, executable package.

3. **What is the difference between image and container?**

Ans. Image is a blueprint; container is a running instance.

4. What is a Dockerfile?

Ans. A script with instructions to build a Docker image.

5. What command is used to build Docker image?

Ans. docker build -t name.

6. How do you run a Docker container?

Ans. docker run -p 8080:8080 image-name

7. What is the benefit of using Docker?

Ans. Environment consistency and scalability.

8. What is port mapping in Docker?

Ans. Mapping container port to host port.

9. How do you list Docker containers?

Ans. docker ps

10. How do you stop a running container?

Ans. docker stop container-id