## Module 1 – Core Java

1. Introduction to Java

* History of Java:
  + Developed by **James Gosling** in 1991 as “**OAK**”.
  + In 1995 Renamed as “**JAVA**” by **James Gosling**.
* Features of Java (Platform Independent, Object-Oriented, etc.)
  + Simple
  + secure
  + Object Oriented
  + interpreter
  + Robust
  + platform independent
  + high performance
  + portable
  + dynamic
  + multithreading
* Understanding JVM, JRE, and JDK
  + **JDK:** Java Development kit (It includes development tools like (compiler) + JRE)
  + **JRE:** Java Runtime Environment (it provides an environment to run + JVM)
  + **JVM:** Java Virtual Machine (Convert bytecode into native machine code)
  + **JIT:** Just in Time
* Java Program Structure (Packages, Classes, Methods)
  + - **Class** is a collection of data member and member function.
    - **Methods** is a set of instructions that perform a task.
    - **Packages** is a namespace that is a collection of related classes and interfaces.

1. Data Types, Variables, and Operators

* Primitive Data Types in Java (int, float, char, etc.)
  + fixed size of data types // inbuild datatypes
  + Integral points: byte, short, int, long, char.
  + floating points: float, double
* Variable Declaration and Initialization
  + Variable Declaration:
    - int a;
    - String s;
    - char c;
    - byte b;
    - boolean bl;
    - long l;
  + Variable Initialization:
    - a = 12;
    - s = “Hello”;
    - c = ‘a’;
    - b = 2;
    - bl = true; // by default its false
    - l = 1234;
* Operators: Arithmetic, Relational, Logical, Assignment, Unary, and Bitwise
  + **Arithmetic: (\*, /, +, -, %)**
  + **Unary: (-, +, ++, --, !)**
  + **Assignment: (+=, -=, \*=, /=, %=)**
  + **Relational: (==, !=, <, >, <=, >=)**
  + **Bitwise: (&&, ||, !)**
* Type Conversion and Type Casting
  + Convert from one data type to another data type.
  + There are mainly 2 types:
    - **implicit:** automatically convert from smaller data type to size convert into Bigger Data type.
    - **explicit:** convert from bigger data type into smaller data type.

1. Control Flow Statements
   * If-Else Statements
     + **If** the condition is true, then if block will be executed otherwise nothing.
     + **If else:** if the condition is true, then your if block will be executed otherwise else block will be executed.
     + **nested if:** if inside if
     + **else if ladder:** we have multiple options/conditions
   * Switch Case Statements
     + **switch statement** is used to evaluate one expression and compare it with different values/cases.
   * Loops (For, While, Do-While)
     + **loops** are used to execute a block of code repeatedly as long as a specified condition is true.
   * Break and Continue Keywords
     + The **break** keyword is used to exit from a loop or switch case.
     + The **break** statement is used to terminate the current loop and jump to the statement after the loop.
     + **Continue** keyword is used to skip the current iteration of a loop and proceed with the next iteration.
     + **Continue** will skip the rest of the code in the current iteration and move to the next iteration of the loop.
2. Classes and Objects

* Defining a Class and Object in Java
  + **Class** is a collection of data member and member function.
  + The **object** is an instance of a class.
  + It uses new keyword and class constructor to create **object.**
  + We cannot access private properties of a **class**.
* Constructors and Overloading
  + **Constructor** is a special member function because it uses same name as a class name.
  + Does not have return type.
  + When we class object, you create at that at time to called **constructor**.
* this Keyword
  + **this** is a reference type to specify the current class member.

1. Methods in Java
   * Defining Methods
     + **Methods** are defined inside classes, is a block of code that performs a specific task.
   * Method Parameters and Return Types
     + **Parameters** are used to pass information into a method.
     + **Return Type** indicates what type of data the method will return.
   * Method Overloading
     + **Method overloading** is a concept in object-oriented programming where you can have multiple methods with the same name but different parameters (either in number or type).
   * Static Methods and Variables
     + A **static variable** is common to all instances of a class.
     + A **static method** belongs to the class rather than to any specific object, and it can be invoked directly using the class name.
2. Object-Oriented Programming (OOPs) Concepts
   * Basics of OOP: Encapsulation, Inheritance, Polymorphism, Abstraction
     + **Encapsulation** is a binding of data into a single unit.
     + **Inheritance** allows access to the properties of the other class and interface.
     + **Polymorphism** means many forms
       - Method Overloading (Compiler time)
       - Method Overriding (Run time)
   * Inheritance: Single, Multilevel, Hierarchical
     + **Inheritance** allows access to the properties of the other class and interface.
     + **Single:** In Single inheritance a single class extends other single class.
     + **Multilevel:** In multilevel every single class extends every single class. (A => B => C)
       - class C extends class B
       - class B extends class A
     + **Hierarchical:** In hierarchical two different classes extend the same parent class.
   * Method Overriding:
     + The whole signature of the method should be the same in super class as well as in subclass, but its behaviors (body part of the method) are different.
3. Constructors and Destructors

##### Theory:

* + - Constructor Types (Default, Parameterized)
    - Copy Constructor (Emulated in Java)
    - Constructor Overloading
    - Object Life Cycle and Garbage Collection

##### Lab Exercise:

* + - Write a program to create and initialize an object using a parameterized constructor.
    - Demonstrate constructor overloading by passing different types of parameters.

1. Arrays and Strings

##### Theory:

* + - One-Dimensional and Multidimensional Arrays
    - String Handling in Java: String Class, StringBuffer, StringBuilder
    - Array of Objects
    - String Methods (length, charAt, substring, etc.)

##### Lab Exercise:

* + - Write a program to perform matrix addition and subtraction using 2D arrays.
    - Create a program to reverse a string and check for palindromes.
    - Implement string comparison using equals() and compareTo() methods.

1. Inheritance and Polymorphism

##### Theory:

* + - Inheritance Types and Benefits
    - Method Overriding
    - Dynamic Binding (Run-Time Polymorphism)
    - Super Keyword and Method Hiding

##### Lab Exercise:

* + - Write a program that demonstrates inheritance using extends keyword.
    - Implement runtime polymorphism by overriding methods in the child class.
    - Use the super keyword to call the parent class constructor and methods.

1. Interfaces and Abstract Classes

##### Theory:

* + - Abstract Classes and Methods
    - Interfaces: Multiple Inheritance in Java
    - Implementing Multiple Interfaces

##### Lab Exercise:

* + - Create an abstract class and implement its methods in a subclass.
    - Write a program that implements multiple interfaces in a single class.
    - Implement an interface for a real-world example, such as a payment gateway.

1. Packages and Access Modifiers

##### Theory:

* + - Java Packages: Built-in and User-Defined Packages
    - Access Modifiers: Private, Default, Protected, Public
    - Importing Packages and Classpath

##### Lab Exercise:

* + - Create a user-defined package and import it into another program.
    - Demonstrate the use of different access modifiers within the same package and across different packages.

1. Exception Handling

##### Theory:

* + - Types of Exceptions: Checked and Unchecked
    - try, catch, finally, throw, throws
    - Custom Exception Classes

##### Lab Exercise:

* + - Write a program to demonstrate exception handling using try-catch-finally.
    - Implement multiple catch blocks for different types of exceptions.
    - Create a custom exception class and use it in your program.

1. Multithreading

##### Theory:

* + - Introduction to Threads
    - Creating Threads by Extending Thread Class or Implementing Runnable Interface
    - Thread Life Cycle
    - Synchronization and Inter-thread Communication

##### Lab Exercise:

* + - Write a program to create and run multiple threads using the Thread class.
    - Implement thread synchronization using synchronized blocks or methods.
    - Use inter-thread communication methods like wait(), notify(), and

notifyAll().

1. File Handling

##### Theory:

* + - Introduction to File I/O in Java (java.io package)
    - FileReader and FileWriter Classes
    - BufferedReader and BufferedWriter
    - Serialization and Deserialization

##### Lab Exercise:

* + - Write a program to read and write content to a file using FileReader and

FileWriter.

* + - Implement a program that reads a file line by line using BufferedReader.
    - Create a program that demonstrates object serialization and deserialization.

1. Collections Framework

##### Theory:

* + - Introduction to Collections Framework
    - List, Set, Map, and Queue Interfaces
    - ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap
    - Iterators and ListIterators

##### Lab Exercise:

* + - Write a program that demonstrates the use of an ArrayList and LinkedList.
    - Implement a program using HashSet to remove duplicate elements from a list.
    - Create a HashMap to store and retrieve key-value pairs.

1. Java Input/Output (I/O)

##### Theory:

* + - Streams in Java (InputStream, OutputStream)
    - Reading and Writing Data Using Streams
    - Handling File I/O Operations

##### Lab Exercise:

* + - Write a program to read input from the console using Scanner.
    - Implement a file copy program using FileInputStream and FileOutputStream.
    - Create a program that reads from one file and writes the content to another file.

## Module 2 – Java – RDBMS & Database Programming with JDBC

Introduction to JDBC

##### Theory:

* + - What is JDBC (Java Database Connectivity)?
    - Importance of JDBC in Java Programming
    - JDBC Architecture: Driver Manager, Driver, Connection, Statement, and ResultSet

##### Lab Exercise:

* + - Write a simple Java program to connect to a MySQL database using JDBC.
    - Demonstrate the process of loading a JDBC driver and establishing a connection.

1. JDBC Driver Types

##### Theory:

* + - Overview of JDBC Driver Types:
      * Type 1: JDBC-ODBC Bridge Driver
      * Type 2: Native-API Driver
      * Type 3: Network Protocol Driver
      * Type 4: Thin Driver
    - Comparison and Usage of Each Driver Type

##### Lab Exercise:

* + - Identify which driver your Java program uses to connect to MySQL.
    - Research and explain the best JDBC driver for your database and Java environment.

1. Steps for Creating JDBC Connections

##### Theory:

* + - Step-by-Step Process to Establish a JDBC Connection:

1. Import the JDBC packages
2. Register the JDBC driver
3. Open a connection to the database
4. Create a statement
5. Execute SQL queries
6. Process the result set
7. Close the connection

##### Lab Exercise:

* + - Write a Java program to establish a connection to a database and print a confirmation message upon successful connection.

1. Types of JDBC Statements

##### Theory:

* + - Overview of JDBC Statements:
      * Statement: Executes simple SQL queries without parameters.
      * PreparedStatement: Precompiled SQL statements for queries with parameters.
      * CallableStatement: Used to call stored procedures.
    - Differences between Statement, PreparedStatement, and

CallableStatement

##### Lab Exercise:

* + - Create a program that inserts, updates, selects, and deletes data using Statement.
    - Modify the program to use PreparedStatement for parameterized queries.

1. JDBC CRUD Operations (Insert, Update, Select, Delete)

##### Theory:

* + - Insert: Adding a new record to the database.
    - Update: Modifying existing records.
    - Select: Retrieving records from the database.
    - Delete: Removing records from the database.

##### Lab Exercise:

* + - Write a Java program that performs the following CRUD operations:
      * Insert a new record.
      * Update an existing record.
      * Select and display records.
      * Delete a record from the database.

1. ResultSet Interface

##### Theory:

* + - What is ResultSet in JDBC?
    - Navigating through ResultSet (first, last, next, previous)
    - Working with ResultSet to retrieve data from SQL queries

##### Lab Exercise:

* + - Write a program that executes a SELECT query and processes the ResultSet to display records from the database.
    - Demonstrate how to navigate through the ResultSet using methods like next(), previous(), etc.

1. Database Metadata

##### Theory:

* + - What is DatabaseMetaData?
    - Importance of Database Metadata in JDBC
    - Methods provided by DatabaseMetaData (getDatabaseProductName, getTables, etc.)

##### Lab Exercise:

* + - Write a program that retrieves and displays metadata information about your database using DatabaseMetaData.
    - Display database name, version, list of tables, and supported SQL features.

1. ResultSet Metadata

##### Theory:

* + - What is ResultSetMetaData?
    - Importance of ResultSet Metadata in analyzing the structure of query results
    - Methods in ResultSetMetaData (getColumnCount, getColumnName, getColumnType)

##### Lab Exercise:

* + - Write a program that retrieves and displays column names, types, and count of a

ResultSet using ResultSetMetaData.

* + - Use a SELECT query to display this metadata for a specific table.

1. Practical SQL Query Examples

##### Lab Exercise:

* + - Write SQL queries for:
      * Inserting a record into a table.
      * Updating specific fields of a record.
      * Selecting records based on certain conditions.
      * Deleting specific records.
    - Implement these queries in Java using JDBC.

1. Practical Example 1: Swing GUI for CRUD Operations

##### Theory:

* + - Introduction to Java Swing for GUI development
    - How to integrate Swing components with JDBC for CRUD operations

##### Lab Exercise:

* + - Create a simple Swing GUI with input fields for id, fname, lname, and email.
    - Implement CRUD operations (Insert, Update, Select, Delete) using JDBC and MySQL.
    - On button clicks, the program should interact with the database and perform the appropriate operation (insert, update, display records, or delete records).

1. Practical Example 2: Callable Statement with IN and OUT Parameters

##### Theory:

* + - What is a CallableStatement?
    - How to call stored procedures using CallableStatement in JDBC
    - Working with IN and OUT parameters in stored procedures

##### Lab Exercise:

* + - Create a stored procedure in MySQL with IN and OUT parameters (e.g., a procedure that takes an employee ID as input and returns the employee's full name as output).
    - Write a Java program that uses CallableStatement to call this stored procedure.
    - Demonstrate how to pass IN parameters and retrieve OUT parameters.

Sample Lab Assignments Summary:

# Lab Assignment 1: Simple JDBC Program

1. Write a Java program that connects to a MySQL database and executes a simple query to retrieve all records from a table.

# Lab Assignment 2: CRUD Operations using JDBC

1. Write a Java program that performs the following operations on a MySQL database:
   * Insert a new record.
   * Update an existing record.
   * Select and display records.
   * Delete a record.

# Lab Assignment 3: Swing GUI with JDBC

1. Create a Swing-based GUI with fields for id, fname, lname, and email.
2. Implement buttons for Insert, Update, Select, and Delete.
3. Perform the corresponding JDBC operations for each button click.

# Lab Assignment 4: Using CallableStatement

1. Create a stored procedure in MySQL with IN and OUT parameters.
2. Write a Java program that calls the stored procedure using CallableStatement and demonstrates how to pass parameters and retrieve results.

## Module 3) Web Technologies in Java

HTML Tags: Anchor, Form, Table, Image, List Tags, Paragraph, Break, Label

# Theory:

* + Introduction to HTML and its structure.
  + Explanation of key tags:
    - <a>: Anchor tag for hyperlinks.
    - <form>: Form tag for user input.
    - <table>: Table tag for data representation.
    - <img>: Image tag for embedding images.
    - List tags: <ul>, <ol>, and <li>.
    - <p>: Paragraph tag.
    - <br>: Line break.
    - <label>: Label for form inputs.

# Lab Exercise:

1. Create a webpage that includes:
   * A navigation menu with anchor tags.
   * A form with input fields, labels, and a submit button.
   * A table that displays user data.
   * Images with appropriate alt text.
   * Both ordered and unordered lists.

CSS: Inline CSS, Internal CSS, External CSS

# Theory:

* Overview of CSS and its importance in web design.
* Types of CSS:
  + **Inline CSS**: Directly in HTML elements.
  + **Internal CSS**: Inside a <style> tag in the head section.
  + **External CSS**: Linked to an external file.

# Lab Exercise:

* + You apply inline CSS to an element.
  + Use internal CSS for another element.
  + Link an external CSS file to style other elements.

CSS: Margin and Padding

# Theory:

* Definition and difference between margin and padding.
* How margins create space outside the element and padding creates space inside.

# Lab Exercise:

1. Create a webpage and use CSS to demonstrate:
   * Margin applied to an element.
   * Padding applied to a div.
   * The effect of different margin and padding values on the layout.

CSS: Pseudo-Class

# Theory:

* Introduction to CSS pseudo-classes like :hover, :focus, :active, etc.
* Use of pseudo-classes to style elements based on their state.

# Lab Exercise:

1. Create a navigation menu and use pseudo-classes to:
   * Change the color of links on hover.
   * Style form inputs when they are focused.

CSS: ID and Class Selectors

# Theory:

* Difference between id and class in CSS.
* Usage scenarios for id (unique) and class (reusable).

# Lab Exercise:

* + You apply an id to an element and style it uniquely.
  + Use class to apply the same style to multiple elements.

Introduction to Client-Server Architecture

# Theory:

* Overview of client-server architecture.
* Difference between client-side and server-side processing.
* Roles of a client, server, and communication protocols.

# Lab Exercise:

1. Create a diagram explaining client-server communication flow and explain how a request is processed by the server and sent back to the client.

HTTP Protocol Overview with Request and Response Headers

# Theory:

* + Introduction to the HTTP protocol and its role in web communication.
  + Explanation of HTTP request and response headers.

# Lab Exercise:

1. Create a Java servlet that:
   * Displays the HTTP request headers.
   * Sends an HTTP response with custom headers.

J2EE Architecture Overview

# Theory:

* Introduction to J2EE and its multi-tier architecture.
* Role of web containers, application servers, and database servers.

# Lab Exercise:

1. Draw and explain the J2EE architecture, labeling the layers like the presentation layer, business logic layer, and data layer.

Web Component Development in Java (CGI Programming)

# Theory:

* + Introduction to CGI (Common Gateway Interface).
  + Process, advantages, and disadvantages of CGI programming.

# Lab Exercise:

1. Write a simple CGI script using Java to accept user input from a form and display it on a webpage.

Servlet Programming: Introduction, Advantages, and Disadvantages

# Theory:

* + Introduction to servlets and how they work.
  + Advantages and disadvantages compared to other web technologies.

# Lab Exercise:

1. Write a simple Java servlet that accepts parameters from a user and displays a response.
2. Discuss the advantages of using servlets over CGI.

Servlet Versions, Types of Servlets

# Theory:

* + History of servlet versions.
  + Types of servlets: Generic and HTTP servlets.

# Lab Exercise:

1. Create a Java servlet program using both GenericServlet and HttpServlet and compare their implementation.

Difference between HTTP Servlet and Generic Servlet

# Theory:

* + Detailed comparison between HttpServlet and GenericServlet.

# Lab Exercise:

1. Write a program using HttpServlet to handle HTTP-specific requests like GET and POST.

Servlet Life Cycle

# Theory:

* + Explanation of the servlet life cycle: init(), service(), and destroy() methods.

# Lab Exercise:

1. Write a servlet program and override all life cycle methods to log messages when each method is called.

Creating Servlets and Servlet Entry in web.xml

# Theory:

* + How to create servlets and configure them using web.xml.

# Lab Exercise:

1. Create a servlet and configure it in web.xml for deployment.

Logical URL and ServletConfig Interface

# Theory:

* + Explanation of logical URLs and their use in servlets.
  + Overview of ServletConfig and its methods.

# Lab Exercise:

1. Write a servlet that uses ServletConfig to fetch initialization parameters.

RequestDispatcher Interface: Forward and Include Methods

# Theory:

* + Explanation of RequestDispatcher and the forward() and include() methods.

# Lab Exercise:

1. Create a login form in JSP, send the data to a servlet, and use RequestDispatcher to forward or include a response based on input validity.

ServletContext Interface and Web Application Listener

# Theory:

* + Introduction to ServletContext and its scope.
  + How to use web application listeners for lifecycle events.

# Lab Exercise:

1. Use ServletContext to share data across multiple servlets.
2. Create a web application listener that logs application start and stop events.

Practical Example 1: Fetch Data Using ServletConfig

# Lab Exercise:

1. Write a servlet to fetch and display initialization parameters from web.xml using

ServletConfig.

Practical Example 2: Fetch Data Using ServletContext

# Lab Exercise:

1. Create multiple servlets that fetch shared data from web.xml using ServletContext.

Practical Example 3: JSP-Servlet Registration Form with RequestDispatcher

# Lab Exercise:

1. Create a registration form in JSP.
2. Send form data to a servlet, process it, and forward the response back to a JSP using

RequestDispatcher.

Java Filters: Introduction and Filter Life Cycle

# Theory:

* + What are filters in Java and when are they needed?
  + Filter lifecycle and how to configure them in web.xml.

# Lab Exercise:

1. Implement a filter to perform server-side validation of user input.

Practical Example: Server-Side Validation Using Filters

# Lab Exercise:

1. Write a filter that checks whether form input fields are empty. If they are, forward back to the input form; otherwise, proceed with the request.

JSP Basics: JSTL, Custom Tags, Scriplets, and Implicit Objects

# Theory:

* + Introduction to JSP and its key components: JSTL, custom tags, scriplets, and implicit objects.

# Lab Exercise:

1. Create a JSP page that uses JSTL to iterate through a list, display scriplets, and access implicit objects.

Session Management and Cookies

# Theory:

* + Overview of session management techniques: cookies, hidden form fields, URL rewriting, and sessions.
  + How to track user sessions in web applications.

# Lab Exercise:

1. Implement a login system in JSP and servlet that uses cookies and session tracking to manage user authentication.

*Module 4) Java – Software Design Patter and Project* Software Design Patterns and Project (MVC + DAO) **Theory:**

##### Introduction to Software Design Patterns:

* + - Definition and purpose of design patterns.
    - Classification: Creational, Structural, and Behavioral patterns.
    - Examples of popular patterns: Singleton, Factory, Observer, Decorator, etc.

##### Introduction to MVC Pattern:

* + - Model-View-Controller (MVC) architecture explained.
    - Separation of concerns and how MVC helps in structuring applications.

##### Introduction to Data Access Object (DAO):

* + - Purpose of the DAO pattern in decoupling data access logic from business logic.
    - How DAO works in combination with MVC to interact with databases.

# Lab Exercise:

##### Build a simple web application using MVC + DAO:

* **Step 1**: Create a simple CRUD web application for user management (register, login, update profile, delete user).
* **Step 2**: Implement DAO pattern to handle database interactions (e.g., for MySQL database).
* **Step 3**: Follow the MVC pattern:
  + Model: Contains business logic and DAO.
  + View: JSP files for the user interface.
  + Controller: Java servlets to handle requests and manage responses.

1. Session Management (Session, Cookie, Hidden Form Field, URL Rewriting)

# Theory:

### Session Management Overview:

* + Why session management is essential in web applications.
  + Difference between client-side and server-side session management.

### Session:

* + Definition of a session and its importance in tracking user activity.
  + How to create, retrieve, and destroy sessions using Java servlets.

### Cookies:

* + What cookies are and how they store small amounts of data on the client-side.
  + Creating, reading, updating, and deleting cookies in Java servlets.

### Hidden Form Fields:

* + Explanation of hidden form fields and their role in passing data between pages.

### URL Rewriting:

* + How URL rewriting can be used to track sessions when cookies are disabled.

# Lab Exercise:

### Session Management in Web Application:

* + **Step 1**: Create a login page in JSP.
  + **Step 2**: Use a session to track the logged-in user and display a welcome page with their details.
  + **Step 3**: Implement logout functionality that invalidates the session.

### Cookie Implementation:

* + **Step 1**: Store the user’s preferences (e.g., theme) in a cookie.
  + **Step 2**: On subsequent visits, read the cookie and apply the stored preferences to the web page.

### Hidden Form Fields:

* + **Step 1**: Create a multi-step form for user registration.
  + **Step 2**: Pass data between forms using hidden fields without using sessions.

### URL Rewriting:

* + **Step 1**: Implement URL rewriting to maintain the session for a user in case cookies are disabled.

1. Project Covering Topics:

# Template Integration Theory:

* + - What is template integration in web applications.
    - Importance of using pre-built templates for faster UI development.

# Lab Exercise:

##### Integrate a Template in Your Web Application:

* + Download a free HTML/CSS template from a website (e.g., Bootstrap template).
  + Integrate the template into your MVC project to enhance the front-end design.

# Image Upload/Download Theory:

* + - Steps to upload and download files in Java web applications.
    - Explanation of the multipart request and handling file uploads using MultipartConfig.

# Lab Exercise:

##### Image Upload/Download Functionality:

* + **Step 1**: Create a JSP form to upload an image file.
  + **Step 2**: Write a servlet to handle the file upload and store the image in a designated folder on the server.
  + **Step 3**: Implement a servlet to list and download stored images by retrieving the files from the server.

# Mail Integration Theory:

* + - How to send emails from a Java web application using JavaMail API.
    - Explanation of SMTP and how it’s used for sending emails.

# Lab Exercise:

##### Integrate Email Functionality in the Project:

* + **Step 1**: Create a registration form.
  + **Step 2**: After successful registration, send a confirmation email to the user using the JavaMail API.

# OTP via Mail Integration Theory:

* + - Introduction to OTP (One-Time Password) and its importance in enhancing security.
    - How to generate and send OTP via email for verification purposes.

# Lab Exercise:

##### OTP Verification:

* + **Step 1**: Create a registration form with an email field.
  + **Step 2**: Generate an OTP upon form submission and send it to the provided email address.
  + **Step 3**: Create a form to enter the OTP and verify the user’s email before allowing account creation.

# Online Payment Integration Theory:

* + - Introduction to online payment gateways (e.g., PayPal, Stripe).
    - How to integrate payment gateways into web applications.

# Lab Exercise:

##### Payment Gateway Integration:

* + **Step 1**: Register for a sandbox account with a payment provider (e.g., PayPal Sandbox).
  + **Step 2**: Implement a checkout page for product purchases and integrate it with the payment gateway.

# AJAX

**Theory:**

* + - Introduction to AJAX and its role in improving the user experience by enabling asynchronous requests.
    - Explanation of how AJAX works in combination with JavaScript and the server.

# Lab Exercise:

##### Implement AJAX in Web Application:

* + **Step 1**: Create a form for live username validation using AJAX.
  + **Step 2**: When a user enters their username, send an asynchronous request to the server to check if the username is available.
  + **Step 3**: Display the result on the page without refreshing the form.

## Module 5) Java – Hibernate Framework

1. Introduction to Hibernate Architecture

# Theory:

### What is Hibernate?:

* + - Definition and purpose of Hibernate as an ORM (Object Relational Mapping) tool.
    - Comparison between Hibernate and JDBC.
    - Why use Hibernate? (Advantages: Database independence, automatic table creation, HQL, etc.)

### Hibernate Architecture:

* + - Explanation of the Hibernate architecture components:
      * **SessionFactory**: Configuration of Hibernate and creation of sessions.
      * **Session**: The main interface between the Java application and the database.
      * **Transaction**: Handling database transactions in Hibernate.
      * **Query**: Writing HQL (Hibernate Query Language) queries to interact with the database.
      * **Criteria**: Criteria API for building dynamic queries.
    - How Hibernate works internally from loading configuration files to executing queries.

# Lab Exercise:

##### Setting Up Hibernate in a Project:

* + **Step 1**: Download the required Hibernate dependencies (e.g., Hibernate Core, Hibernate EntityManager, Hibernate Validator, and MySQL Connector).
  + **Step 2**: Create a Hibernate configuration file (hibernate.cfg.xml) to set up the connection to a MySQL database.
  + **Step 3**: Write a simple Java application to establish a session with Hibernate and perform a basic operation (e.g., inserting data into a table).

1. Hibernate Relationships (One-to-One, One-to-Many, Many-to-One, Many-to-Many)

# Theory:

### Object Relationships in Hibernate:

* + - How Hibernate manages relationships between Java objects and database tables.
    - Overview of the different types of relationships:

##### One-to-One Relationship:

* + - * + A single instance of an entity is related to a single instance of another entity.

##### One-to-Many Relationship:

* + - * + One entity can have multiple related entities.

##### Many-to-One Relationship:

* + - * + Many entities are associated with a single entity.

##### Many-to-Many Relationship:

* + - * + Multiple instances of an entity are associated with multiple instances of another entity.

### Mapping Relationships in Hibernate:

* + - How to map relationships in Hibernate using annotations like @OneToOne, @OneToMany, @ManyToOne, and @ManyToMany.
    - The concept of owning and inverse sides in relationships.
    - Cascade types and how they affect related entities.

# Lab Exercise:

### One-to-One Relationship:

* + **Step 1**: Create two entity classes, e.g., User and Profile, where each user has one profile.
  + **Step 2**: Map the relationship using @OneToOne annotation in Hibernate.
  + **Step 3**: Write a program to save and retrieve a user and its profile using Hibernate.

### One-to-Many Relationship:

* + **Step 1**: Create two entity classes, e.g., Author and Book, where one author can have multiple books.
  + **Step 2**: Map the relationship using @OneToMany and @ManyToOne annotations.
  + **Step 3**: Write a program to add multiple books for an author and retrieve the author's details along with their books.

### Many-to-Many Relationship:

* + **Step 1**: Create two entity classes, e.g., Student and Course, where a student can enroll in multiple courses, and a course can have multiple students.
  + **Step 2**: Use the @ManyToMany annotation to map the relationship and create a join table.
  + **Step 3**: Write a program to assign multiple courses to students and retrieve student- course details.

1. Hibernate CRUD Example

# Theory:

### Understanding CRUD Operations in Hibernate:

* + - **Create (Insert)**: How to use Hibernate to insert records into a database.
    - **Read (Select)**: Fetching data from the database using Hibernate.
    - **Update**: Modifying existing records in the database.
    - **Delete**: Removing records from the database.

### Writing HQL (Hibernate Query Language):

* + - Basics of HQL and how it differs from SQL.
    - How to perform CRUD operations using HQL.
    - Introduction to the Criteria API for dynamic queries.

# Lab Exercise:

### Create (Insert) Operation:

* + **Step 1**: Define a simple entity class, e.g., Employee, with fields like id, name, department, and salary.
  + **Step 2**: Write a Hibernate program to insert employee records into a database table using Session.save() method.
  + **Step 3**: Verify the inserted data by querying the database directly.

### Read (Select) Operation:

* + **Step 1**: Write a Hibernate query to retrieve all employees from the database using

Session.get() or HQL.

* + **Step 2**: Display the retrieved employee data in the console.

### Update Operation:

* + **Step 1**: Write a Hibernate program to update the salary of an employee.
  + **Step 2**: Use Session.update() method to modify an existing record.
  + **Step 3**: Fetch and verify that the employee’s salary has been updated in the database.

### Delete Operation:

* + **Step 1**: Write a Hibernate program to delete an employee from the database.
  + **Step 2**: Use Session.delete() method to remove a record.
  + **Step 3**: Verify the deletion by querying the database.

Practical Project Example:

#### Create a simple Employee Management System using Hibernate to perform CRUD operations and manage employee details. The system should support:

* **Inserting** a new employee record.
* **Viewing** all employee records.
* **Updating** employee details (e.g., changing department, salary).
* **Deleting** an employee.

#### Incorporate Hibernate relationships such as:

* **One-to-One**: Each employee has one profile (e.g., employee details and profile picture).
* **One-to-Many**: One department can have many employees.
* **Many-to-Many**: Employees can work on multiple projects, and projects can have multiple employees assigned.

## Module 6) Java – Spring

1. Introduction to Spring Framework

# Theory:

### What is Spring Framework?

* + - Overview of the Spring Framework and its purpose in Java development.
    - Key features of Spring:
      * Inversion of Control (IoC)
      * Dependency Injection (DI)
      * Aspect-Oriented Programming (AOP)
      * Transaction Management
      * Spring's flexibility for creating both web and non-web applications.

### Spring Architecture:

* + - Overview of the core components of the Spring Framework:
      * **Core Container**: IoC and DI
      * **Spring AOP**: Aspect-Oriented Programming
      * **Spring ORM**: Integrating Spring with ORM frameworks (e.g., Hibernate, JPA)
      * **Spring Web**: Web framework for creating Java web applications.
      * **Spring MVC**: Model-View-Controller framework for building web applications.

# Lab Exercise:

##### Setting up a Spring Project:

* + **Step 1**: Install and configure Spring dependencies using Maven or Gradle.
  + **Step 2**: Create a basic Spring application.
  + **Step 3**: Configure a simple XML or annotation-based Spring application with one bean and test it by loading the Spring application context.

1. BeanFactory and ApplicationContext

# Theory:

### BeanFactory vs. ApplicationContext:

* + - What is **BeanFactory**?:
      * A simple container for managing Spring beans.
      * Pros and cons of using BeanFactory.
    - What is **ApplicationContext**?:
      * A more advanced container that includes features like event propagation, declarative mechanisms, and AOP support.
    - Differences between **BeanFactory** and **ApplicationContext** (e.g., lazy initialization in BeanFactory vs. eager initialization in ApplicationContext).

### Spring Beans:

* + - Definition of a bean in Spring.
    - Scope of beans: Singleton, Prototype, Request, Session.
    - Bean lifecycle: Initialization and destruction of beans.

# Lab Exercise:

### Using BeanFactory and ApplicationContext:

* + **Step 1**: Create a Spring configuration file (beans.xml) to define a few simple beans.
  + **Step 2**: Write Java code to load the beans using **BeanFactory** and display the bean properties.
  + **Step 3**: Modify the code to load the same beans using **ApplicationContext** and discuss the difference.

### Bean Scopes:

* + **Step 1**: Configure beans with different scopes (e.g., Singleton and Prototype) in the

beans.xml file.

* + **Step 2**: Write Java code to demonstrate the effect of different bean scopes by retrieving beans multiple times and checking if the same instance is returned.

1. Container Concepts in Spring

# Theory:

### Spring IoC (Inversion of Control):

* + - Understanding IoC and how Spring uses it to manage object creation and dependencies.
    - Benefits of IoC in application design (loose coupling, modularity, and testability).

### Dependency Injection (DI):

* + - Types of Dependency Injection:
      * Constructor-based Dependency Injection.
      * Setter-based Dependency Injection.
    - Advantages of DI in Spring.

# Lab Exercise:

### Constructor and Setter Dependency Injection:

* + **Step 1**: Create a Spring configuration file and define two beans with dependencies.
  + **Step 2**: Demonstrate constructor-based DI by wiring dependencies via the constructor.
  + **Step 3**: Demonstrate setter-based DI by wiring dependencies via setter methods.
  + **Step 4**: Test the configuration by retrieving the beans and checking the injection.

### Configuring IoC in XML and Annotations:

* + **Step 1**: Define beans in XML to implement DI.
  + **Step 2**: Modify the same beans to use annotations (@Autowired, @Qualifier) for DI.

1. Spring Data JPA Template

# Theory:

### What is Spring Data JPA?:

* + - Introduction to Spring Data JPA and how it simplifies interaction with databases.
    - Explanation of JPA (Java Persistence API) and its role in ORM (Object Relational Mapping).
    - Benefits of using Spring Data JPA over manual SQL queries.

### Spring Data JPA Components:

* + - **Repositories**: How Spring Data JPA auto-generates repository implementations.
    - **Entities**: Mapping Java objects to database tables using JPA annotations.
    - **Query Methods**: Creating custom queries using method naming conventions (e.g.,

findById, findByName).

# Lab Exercise:

### Basic CRUD Operations with Spring Data JPA:

* + **Step 1**: Set up a Spring Boot project with Spring Data JPA and a MySQL database.
  + **Step 2**: Create an entity class (Employee) with fields like id, name, department.
  + **Step 3**: Create a repository interface extending JpaRepository.
  + **Step 4**: Write a service class to perform basic CRUD operations (Insert, Update, Delete, Select) on the Employee entity.
  + **Step 5**: Test the CRUD operations using a REST controller or unit tests.

### Custom Queries Using Spring Data JPA:

* + **Step 1**: Create a repository interface with custom query methods (e.g.,

findByDepartment(String department)).

* + **Step 2**: Implement the repository and perform database queries based on method names.

1. Spring MVC

# Theory:

### What is Spring MVC?:

* + - Overview of the MVC (Model-View-Controller) design pattern.
    - Explanation of the Spring MVC framework and how it simplifies web development.

### Spring MVC Components:

* + - **Controller**: Handles HTTP requests and returns a response.
    - **Model**: Holds the data to be displayed on the view.
    - **View**: Renders the data from the model in a user-friendly format (e.g., JSP, Thymeleaf).
    - **DispatcherServlet**: Central servlet in Spring MVC that manages the request flow.

### Request Mapping in Spring MVC:

* + - Using @RequestMapping, @GetMapping, and @PostMapping annotations to map HTTP requests to controller methods.
    - Path variables, request parameters, and form handling.

# Lab Exercise:

### Building a Simple Spring MVC Application:

* + **Step 1**: Set up a Spring MVC project and configure the DispatcherServlet in

web.xml.

* + **Step 2**: Create a simple controller class with @RequestMapping to handle a basic request (e.g., /welcome).
  + **Step 3**: Create a view (e.g., JSP or Thymeleaf) to display a welcome message to the user.
  + **Step 4**: Test the application by accessing the controller endpoint and displaying the view.

### Handling Forms in Spring MVC:

* + **Step 1**: Create a Spring MVC form for user registration.
  + **Step 2**: Create a controller method to handle form submission and capture user data.
  + **Step 3**: Validate the form inputs using Spring’s form validation (@Valid,

BindingResult).

* + **Step 4**: Display validation errors on the view if inputs are invalid.

Project Example for Spring MVC + Spring Data JPA:

### Employee Management System:

* Build a basic web application using **Spring MVC** for handling requests and **Spring Data JPA**

for CRUD operations.

* Key Features:
  1. **User Registration**: Create a form for registering a new employee.
  2. **View Employees**: Display all employees from the database on a webpage.
  3. **Update Employee**: Provide an option to update employee details.
  4. **Delete Employee**: Allow the deletion of employee records.
  5. **Search Employees**: Add functionality to search for employees by name or department.

## Module 7) Java – Spring Boot

1. Introduction to STS (Spring Tool Suite)

# Theory:

##### What is Spring Tool Suite (STS)?

* + - Overview of STS: An Eclipse-based IDE for developing Spring applications.
    - Key features and benefits of using STS, including built-in support for Spring Boot, easy dependency management, and a robust debugging environment.

##### Installation and Setup:

* + - Step-by-step guide on how to download, install, and configure STS for Java/Spring development.
    - Overview of the interface, how to create a Spring Boot project, and the workspace organization.

# Lab Exercise:

##### Setting up STS and Creating a Simple Spring Boot Application:

* + **Step 1**: Install and configure STS.
  + **Step 2**: Create a new Spring Boot project in STS.
  + **Step 3**: Configure dependencies (Spring Web, Spring Data JPA, etc.) via Maven or Gradle.
  + **Step 4**: Write a simple controller and run the application to display "Hello, Spring!" on the browser.

1. Spring MVC (Model-View-Controller)

# Theory:

### Spring MVC Overview:

* + - Introduction to the MVC design pattern and how it is implemented in Spring.
    - Explanation of core components: Controller, Model, and View.

### Template Integration:

* + - Using templating engines like Thymeleaf or JSP in Spring MVC applications.
    - How template engines help in creating dynamic web pages and separating concerns.

### CRUD Operations:

* + - Implementing basic Create, Read, Update, and Delete functionality in a Spring MVC application.
    - Flow of data between the view, controller, and model.

### Form Validation:

* + - Introduction to form validation in Spring MVC using annotations like @Valid and

@NotNull.

* + - Validating user input and handling validation errors.

### Pagination:

* + - Implementing pagination in Spring MVC to handle large datasets.
    - Using Pageable and Page interfaces in Spring Data JPA.

# Lab Exercise:

### Template Integration:

* + **Step 1**: Create a Spring MVC project and integrate Thymeleaf (or JSP) as the view layer.
  + **Step 2**: Create a simple template to display dynamic content (e.g., a list of users).
  + **Step 3**: Configure the template to accept data from the Spring controller and display it on the view.

### CRUD Operations with Spring MVC:

* + **Step 1**: Set up a Spring Boot project with Spring MVC and Spring Data JPA.
  + **Step 2**: Create an entity class Product with fields id, name, price, and

description.

* + **Step 3**: Implement the CRUD operations (Create, Read, Update, Delete) in the controller, using a service layer and repository.
  + **Step 4**: Create views for adding, listing, editing, and deleting products.

### Form Validation:

* + **Step 1**: Create a form for user registration.
  + **Step 2**: Add validation to the form fields (e.g., name, email) using @NotEmpty, @Email, and other validation annotations.
  + **Step 3**: Implement validation handling in the controller and display error messages on the view when validation fails.

### Pagination:

* + **Step 1**: Create a service to fetch data in a paginated format using Pageable.
  + **Step 2**: Implement pagination in the controller and view to display large datasets (e.g., a list of products or users) across multiple pages.
  + **Step 3**: Create navigation controls to move between pages.

1. Aspect-Oriented Programming (AOP)

# Theory:

##### What is AOP (Aspect-Oriented Programming)?

* + - Definition of AOP and its importance in separating cross-cutting concerns (logging, security, transaction management).
    - Key components in AOP:
      * **Aspect**: A module that encapsulates cross-cutting concerns.
      * **Joinpoint**: A point in the program where the aspect is applied.
      * **Advice**: The action taken by an aspect at a particular joinpoint (Before, After, Around).
      * **Pointcut**: An expression to define where advice should be applied.

# Lab Exercise:

##### Logging Aspect in Spring AOP:

* + **Step 1**: Set up a Spring Boot project with AOP support.
  + **Step 2**: Create an Aspect for logging method execution times.
  + **Step 3**: Implement @Before, @After, and @Around advices to log details before and after method execution in a service class.
  + **Step 4**: Test the aspect by calling a method from the service class and checking the logs for method execution details.

1. Spring Security

# Theory:

### Introduction to Spring Security:

* + - Overview of Spring Security, its purpose, and how it secures web applications.
    - Key features: Authentication and Authorization, Security Filters, and Form-based login.

### Role-Based Authentication:

* + - How to define roles (e.g., USER, ADMIN) and restrict access to specific URLs or methods based on user roles.
    - Securing endpoints using @Secured or @PreAuthorize.

### OAuth2 Authentication:

* + - Introduction to OAuth2 and how it is used for third-party authentication (Google, Facebook).
    - Explanation of OAuth2 flows: Authorization Code Grant, Implicit Grant, etc.

### Token-Based Authentication (JWT):

* + - Introduction to token-based authentication using JSON Web Tokens (JWT).
    - Explanation of the authentication process: token generation, validation, and secure access to protected resources.

# Lab Exercise:

### Role-Based Authentication:

* + **Step 1**: Set up a Spring Boot project with Spring Security.
  + **Step 2**: Define roles (USER, ADMIN) and create a simple login form.
  + **Step 3**: Secure specific URLs (e.g., /admin, /user) and restrict access based on roles.
  + **Step 4**: Test the application by logging in with different users and checking if the correct restrictions are applied.

### OAuth2 Integration:

* + **Step 1**: Set up OAuth2 login with Google or Facebook in a Spring Boot application.
  + **Step 2**: Configure the application to redirect to Google/Facebook for authentication.
  + **Step 3**: Once authenticated, display the user’s information (name, email) on the dashboard.

### Token-Based Authentication (JWT):

* + **Step 1**: Implement JWT-based authentication in a Spring Boot REST API.
  + **Step 2**: Create an endpoint for user login and generate a JWT token upon successful authentication.
  + **Step 3**: Implement a filter to validate the JWT token for each request to protected resources.
  + **Step 4**: Test the application by logging in, obtaining a token, and accessing secured endpoints using the token.

Project Example: E-Commerce Web Application Using Spring MVC, AOP, and Security

### Key Features:

* **User Registration and Login**: Implement user registration with form validation and Spring Security.
* **Role-based Authorization**: Admin can manage products, and users can view and purchase products.
* **CRUD Operations**: Admin can create, update, delete, and view products.
* **Aspect-Oriented Programming**: Implement logging for product management operations (create, update, delete).
* **Pagination**: Display a paginated list of products for users.
* **OAuth2 Authentication**: Allow users to sign in via Google or Facebook.
* **JWT Authentication**: Implement JWT for securing REST API endpoints for managing products.

## Module 8) Java – Spring Webservices

1. Introduction to Web Services

# Theory:

### What are Web Services?

* + - Definition of web services and their importance in enabling communication between different applications over the internet.
    - Types of Web Services:
      * **SOAP (Simple Object Access Protocol)**
      * **REST (Representational State Transfer)**

### Advantages of Web Services:

* + - Platform and language independence.
    - Integration across diverse systems.
    - Enables microservices architecture.

# Lab Exercise:

##### Create a Simple Web Service:

* + **Step 1**: Set up a simple RESTful web service using Spring Boot.
  + **Step 2**: Create a REST endpoint /greeting that returns a simple greeting message (e.g., “Hello, World!”).
  + **Step 3**: Test the endpoint using Postman or Curl to verify it returns the expected response.

1. Basics of REST APIs

# Theory:

### What is REST (Representational State Transfer)?

* + - Overview of REST principles: statelessness, resource-based URLs, use of HTTP methods (GET, POST, PUT, DELETE), and status codes.
    - Key REST concepts:
      * **Resources**: Everything is treated as a resource.
      * **URI**: Uniform Resource Identifiers for identifying resources.
      * **Stateless Communication**: Each request from a client to the server must contain all the information needed to understand and process the request.

### HTTP Methods:

* + - **GET**: Retrieve data.
    - **POST**: Submit data.
    - **PUT**: Update data.
    - **DELETE**: Remove data.

# Lab Exercise:

##### Create a RESTful API for a Student Resource:

* + **Step 1**: Set up a Spring Boot project with Spring Web dependency.
  + **Step 2**: Create a Student entity with fields id, name, email, and course.
  + **Step 3**: Implement REST endpoints for CRUD operations:
    - **GET** /students: Retrieve a list of students.
    - **POST** /students: Add a new student.
    - **PUT** /students/{id}: Update an existing student’s details.
    - **DELETE** /students/{id}: Delete a student.
  + **Step 4**: Test the endpoints using Postman or any REST client.

1. Spring MVC (Model-View-Controller)

# Theory:

### Spring MVC Overview:

* + - Explanation of the MVC design pattern: Model, View, and Controller.
    - How Spring MVC handles incoming web requests and maps them to the correct controller.

### Controller and View:

* + - Creating a controller to handle user requests.
    - Using a view template engine (e.g., Thymeleaf) to render dynamic data.

# Lab Exercise:

##### Create a Spring MVC Web Application:

* + **Step 1**: Set up a Spring Boot project with Spring Web and Thymeleaf.
  + **Step 2**: Create a simple controller that handles a GET request and returns a view.
  + **Step 3**: Create a view template using Thymeleaf to display a list of students passed from the controller.

1. Aspect-Oriented Programming (AOP)

# Theory:

##### What is AOP (Aspect-Oriented Programming)?

* + - Overview of AOP and how it helps in separating cross-cutting concerns (e.g., logging, security, transaction management).
    - Key AOP terms:
      * **Aspect**: Module encapsulating cross-cutting concerns.
      * **Advice**: The action taken by an aspect (Before, After, or Around).
      * **Joinpoint**: Point in the execution of the program where the aspect is applied.
      * **Pointcut**: Expression that defines where the advice should be applied.

# Lab Exercise:

##### Implement Logging Aspect Using AOP:

* + **Step 1**: Set up a Spring Boot project with AOP dependency.
  + **Step 2**: Create an Aspect class that logs the method execution time.
  + **Step 3**: Use @Before and @After annotations to log the execution of specific methods in a service class.
  + **Step 4**: Test the logging aspect by calling methods in the service class and checking the logs.

1. Spring REST (CRUD API, Pagination, Fetching from Multiple Tables, Image Upload/Download)

# Theory:

### Spring REST Overview:

* + - Introduction to creating RESTful services in Spring Boot.
    - Use of @RestController to create REST APIs.
    - Handling HTTP requests and returning JSON or XML responses.

### Pagination:

* + - Introduction to pagination in REST APIs to handle large datasets.
    - Use of Pageable and Page interfaces from Spring Data JPA for pagination support.

### CRUD Operations:

* + - Create, Read, Update, Delete (CRUD) operations using Spring Data JPA.

### Fetching Data from Multiple Tables:

* + - Use of JPA relationships (@OneToOne, @OneToMany, @ManyToOne, and

@ManyToMany) to retrieve related data from multiple tables.

### Image Upload/Download:

* + - Handling file upload and download in a Spring REST API.

# Lab Exercise:

### CRUD API with Pagination:

* + **Step 1**: Set up a Spring Boot project with Spring Data JPA and Spring Web.
  + **Step 2**: Create two entities, Student and Course, with a **many-to-one** relationship between them.
  + **Step 3**: Implement CRUD operations for the Student entity with endpoints for adding, updating, retrieving, and deleting students.
  + **Step 4**: Implement pagination on the GET endpoint to retrieve a paginated list of students using the Pageable interface.
  + **Step 5**: Test the API using Postman or any REST client.

### Fetching Data from Multiple Tables:

* + **Step 1**: Extend the above lab by fetching a list of students enrolled in a particular course.
  + **Step 2**: Implement a GET endpoint to fetch students based on the course ID.
  + **Step 3**: Return a list of students enrolled in the course, showing the relationship between the two tables.

### Image Upload/Download in REST API:

* + **Step 1**: Implement an API endpoint that allows users to upload an image file.
  + **Step 2**: Store the uploaded image in the file system or database (e.g., as a BLOB).
  + **Step 3**: Create another API endpoint to download and display the image file.
  + **Step 4**: Test the image upload and download functionality using Postman or any REST client.

Project Example: Bookstore Application Using Spring REST, AOP, and Pagination

### Features:

* **Book Management**: Implement CRUD operations for books.
* **Author Management**: CRUD operations for authors, with a relationship between books and authors (One-to-Many).
* **Pagination**: Display paginated lists of books on the frontend.
* **AOP Logging**: Implement logging for the CRUD operations on books and authors.
* **Image Upload/Download**: Allow users to upload book cover images and download them.
* **Search Functionality**: Implement a search API to find books by title or author.

## Module 9) Java – Micro services with Spring Boot, Spring Cloud

1. Microservices with Spring Boot and Spring Cloud

# Theory:

### What are Microservices?

* + - Definition and characteristics of Microservices architecture.
    - Key principles: Decoupled services, scalability, independent deployment.

### Advantages of Microservices Over Monolithic Architecture:

* + - Scalability: Independent scaling of services.
    - Fault Isolation: Issues in one service do not affect others.
    - Flexibility: Different technologies can be used in different services.
    - Faster Deployment: Continuous delivery and deployment pipelines are easier.

### Components of Microservices Architecture:

* + - **API Gateway**: Routes and load balances requests to microservices.
    - **Service Registry (Eureka)**: Keeps track of services and their locations.
    - **Circuit Breaker**: Manages service failures.
    - **Load Balancer**: Distributes requests across services.

# Lab Exercise:

##### Create a Simple Microservice with Spring Boot:

* + **Step 1**: Set up a Spring Boot application for a simple microservice (e.g.,

UserService).

* + **Step 2**: Implement basic CRUD operations for the UserService using RESTful APIs.
  + **Step 3**: Test the APIs locally using Postman or Curl.

1. Introduction to Microservice Architecture

# Theory:

##### Microservice vs. Monolithic Architecture:

* + - **Monolithic Architecture**: All functionalities reside in one large application.
    - **Microservices**: Applications are split into independent services.

##### Key Characteristics:

* + - **Decentralization**: Each microservice has its own database.
    - **Inter-Service Communication**: Services communicate using lightweight protocols like HTTP or messaging systems like RabbitMQ.

# Lab Exercise:

##### Convert a Monolithic Application into Microservices:

* + **Step 1**: Take a sample monolithic application (e.g., a shopping app with user management and product management).
  + **Step 2**: Split the monolithic app into two microservices: UserService and

ProductService.

* + **Step 3**: Set up communication between the services using REST.

1. Developing and Deploying a Microservice Application Locally

# Theory:

##### Steps to Build a Microservice:

* + - Develop each service independently.
    - Use Spring Boot for microservice development.
    - Package and deploy each service using Docker or directly on localhost.

# Lab Exercise:

##### Deploy Two Microservices Locally:

* + **Step 1**: Create two microservices (UserService and OrderService) using Spring Boot.
  + **Step 2**: Set up the services to run on different ports (e.g., UserService on port 8081 and OrderService on port 8082).
  + **Step 3**: Test communication between the services using REST APIs locally.

##### Optional:

* + **Step 4**: Package the services as Docker containers and run them using Docker Compose.

1. Introduction to Service Discovery: Eureka Server

# Theory:

### Service Discovery:

* + - In microservices, each service may start and stop dynamically, so a **Service Registry**

is essential to keep track of service instances.

### What is Eureka?

* + - **Eureka** is a Service Registry from Netflix that allows services to register themselves and discover other services.

### Eureka Server and Eureka Client:

* + - **Eureka Server**: Acts as the registry for services.
    - **Eureka Client**: Registers itself with the Eureka Server and discovers other services.

# Lab Exercise:

### Set up a Eureka Server:

* + **Step 1**: Create a Spring Boot application and add the Eureka Server dependency.
  + **Step 2**: Enable Eureka Server in the application using @EnableEurekaServer.
  + **Step 3**: Run the Eureka Server and check the Eureka dashboard (default on

[http://localhost:8761).](http://localhost:8761/)

### Register a Service with Eureka:

* + **Step 1**: Create a simple Spring Boot microservice (OrderService) and add the Eureka Client dependency.
  + **Step 2**: Enable Eureka Client in the service using @EnableEurekaClient.
  + **Step 3**: Register the service with the Eureka Server and check if it is listed in the Eureka dashboard.

1. Client-Side and Server-Side Discovery Patterns

# Theory:

### Client-Side Discovery:

* + - The client is responsible for service discovery by interacting with the Eureka Server and finding the instances of a particular service.

### Server-Side Discovery:

* + - The client makes a request to an API Gateway or Load Balancer, which then forwards the request to the appropriate service.

# Lab Exercise:

### Client-Side Service Discovery:

* + **Step 1**: Create a microservice (UserService) and register it with the Eureka Server.
  + **Step 2**: Create another microservice (OrderService) that uses a RestTemplate

to discover UserService from Eureka and make a request to its API.

### Server-Side Discovery:

* + **Step 1**: Set up an API Gateway (e.g., Spring Cloud Gateway) that forwards requests to UserService and OrderService.
  + **Step 2**: Use Eureka for server-side service discovery, where the gateway fetches the available instances from the Eureka Server.

1. Load Balancing Configuration

# Theory:

### What is Load Balancing?

* + - Load balancing helps distribute incoming requests across multiple instances of a service to ensure better performance and fault tolerance.

### Types of Load Balancers:

* + - **Client-Side Load Balancer**: Managed at the client-side (e.g., Ribbon).
    - **Server-Side Load Balancer**: Managed centrally (e.g., API Gateway, Nginx).

# Lab Exercise:

### Client-Side Load Balancing with Ribbon:

* + **Step 1**: Create multiple instances of a microservice (e.g., two instances of

UserService running on different ports).

* + **Step 2**: Enable Ribbon client-side load balancing in another service (OrderService).
  + **Step 3**: Use RestTemplate to make a call to UserService and test if the requests are balanced across both instances.

### Server-Side Load Balancing Using Spring Cloud Gateway:

* + **Step 1**: Set up Spring Cloud Gateway to route requests to multiple instances of

UserService.

* + **Step 2**: Configure the gateway to load balance the requests between instances.
  + **Step 3**: Test load balancing by sending multiple requests to the gateway and checking the distribution.

Project Example: E-commerce Microservices with Eureka and Load Balancing

### Features:

* **Service Registry**: Use Eureka Server to register services (UserService, OrderService, ProductService).
* **API Gateway**: Set up an API Gateway to route traffic to the different services.
* **Load Balancing**: Configure load balancing for services with multiple instances.
* **Database Integration**: Use Spring Data JPA for database interactions in each service (e.g., MySQL or PostgreSQL).
* **Communication**: Use REST APIs for inter-service communication.

## Module 9) Debugging Exercises for Problem Solving

1. Simple Arithmetic Calculation (Off-by-One Error)

#### **Description**: This program is meant to calculate the sum of the first 10 natural numbers. However, there’s an off-by-one error.

java

Copy code

(String[] args) {

int sum = 0; for (int i =

public class SumOfNumbers { public static void main

0; i <= 10; i++) { // Off-by-one error here

sum += i;

}

System.out.println("Sum of first 10 natural numbers is: " + sum);

}

}

### Objective:

* Identify the off-by-one error.
* Debug the loop so that it correctly sums the first 10 natural numbers.

### Expected Output:

Sum of first 10 natural numbers is: 55

1. Array Index Out of Bound

**Description**: The program is designed to calculate the average of the numbers in an array, but it throws an ArrayIndexOutOfBoundsException.

java

Copy code

(String[] args) {

int sum = 0; for (int i =

public class ArrayAverage { public static void main int[] numbers = {10

, 20, 30, 40, 50};

0; i <= numbers.length; i++) { // Off-by-one error

sum += numbers[i];

}

double average = sum / numbers.length;

System.out.println("Average is: " + average);

}

}

### Objective:

* Identify the mistake causing the ArrayIndexOutOfBoundsException.
* Fix the error and ensure the program calculates the average correctly.

**Expected Output**: Average is: 30.0

1. Infinite Loop

#### **Description**: The following program should print numbers from 1 to 5, but it runs infinitely due to a logical error in the loop.

java

Copy code

public class PrintNumbers {

public static void main(String[] args) {

int i = 1;

while (i <= 5) {

System.out.println(i);

// Increment is missing here

}

}

}

### Objective:

* Find the cause of the infinite loop.
* Correct the code so it prints numbers from 1 to 5 without running indefinitely.

### Expected Output:

Copy code 1

2

3

4

5

1. Null Pointer Exception

#### **Description**: The following program should print the length of a string, but it throws a

NullPointerException.

java

Copy code

public class StringLength {

public static void main(String[] args) {

String str = null;

System.out.println("Length of the string is: " + str.length());

}

}

### Objective:

* Identify why the program throws a NullPointerException.
* Modify the code to avoid the exception and handle the null string properly.

### Expected Output:

Length of the string is: 0 (or handle it with an appropriate message)

1. Incorrect Output Due to Floating-Point Division

#### **Description**: The following program tries to calculate the percentage of marks, but the result is incorrect due to integer division.

java

Copy code

public class PercentageCalculator {

public static void main(String[] args) {

int totalMarks = 450;

int marksObtained = 375;

int percentage = (marksObtained / totalMarks) \* 100; // Incorrect

System.out.println("Percentage: " + percentage + "%");

division

}

}

### Objective:

* Identify the cause of the incorrect output.
* Correct the code to ensure that floating-point division is used for calculating the percentage.

**Expected Output**: Percentage: 83.33%

1. Logical Error in Prime Number Check

#### **Description**: The following program is supposed to check if a number is prime or not, but it incorrectly identifies some composite numbers as prime.

java

Copy code

public class PrimeCheck {

public static void main(String[] args) {

= true;

int number = 15; boolean isPrime

for (int i = 2; i <= number / 2; i++) { if (number % i == 0) {

isPrime = false; break;

}

}

if (isPrime) {

System.out.println(number + " is a prime number.");

} else {

System.out.println(number + " is not a prime number.");

}

}

}

### Objective:

* Identify why the program incorrectly identifies some composite numbers as prime.
* Correct the prime number logic to work for any input.

### Expected Output:

15 is not a prime number.

1. Wrong Use of Equals for String Comparison

#### **Description**: The following program tries to compare two strings for equality, but it gives incorrect results.

java

Copy code

public class StringComparison {

public static void main(String[] args) {

String str1 = "hello";

String str2 = new String("hello");

if (str1 == str2) {

System.out.println("Strings are equal.");

} else {

System.out.println("Strings are not equal.");

}

}

}

### Objective:

* Identify why the comparison gives incorrect results.
* Use the correct method for comparing strings.

**Expected Output**: Strings are equal.

1. Off-by-One Error in Array Sum

#### **Description**: The following program should calculate the sum of elements in an array, but it doesn’t add all elements correctly due to an off-by-one error.

java

Copy code

public class ArraySum {

public static void main(String[] args) { int[] arr = {1, 2, 3, 4, 5};

int sum = 0;

for (int i = 0; i < arr.length - 1; i++) { // Off-by-one error

sum += arr[i];

}

System.out.println("Sum of array elements: " + sum);

}

}

### Objective:

* Identify the off-by-one error in the array summation.
* Correct the loop to add all elements of the array.

### Expected Output:

Sum of array elements: 15

1. Wrong Output for Fibonacci Series

#### **Description**: The program should print the first 5 Fibonacci numbers, but it prints incorrect values due to improper handling of the loop variables.

java

Copy code

public class Fibonacci {

public static void main(String[] args) { int n1 = 0, n2 = 1, n3; System.out.print(n1 + " " + n2);

for (int i = 2; i <= 5; i++) { n3 = n1 + n2; System.out.print(" " + n3);

n1 = n2; // Incorrect update of n1 and n2 n2 = n3;

}

}

}

### Objective:

* Identify why the Fibonacci sequence is incorrect.
* Fix the logic to correctly generate the first 5 Fibonacci numbers.

**Expected Output**: 0 1 1 2 3 5

1. Logical Error in Palindrome Check

#### **Description**: The following program is supposed to check if a string is a palindrome, but it incorrectly identifies some non-palindromes as palindromes.

java

Copy code

public class PalindromeCheck {

public static void main(String[] args) {

String str = "madam";

String reverse = "";

for (int i = 0; i <= str.length(); i++) { // Off-by-one error

reverse += str.charAt(i);

}

if (str.equals(reverse)) {

System.out.println(str + " is a palindrome.");

} else {

System.out.println(str + " is not a palindrome.");

}

}

}

### Objective:

* Identify the off-by-one error in the loop and correct it.
* Ensure the program correctly checks if a string is a palindrome.

**Expected Output**: madam is a palindrome.