THANTHAI PERIYAR GOVERNMENT INSTITUTE OF TECHNOLOGY, VELLORE – 632 002

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ODD SEMESTER

ACADEMIC YEAR (2025 – 2026)

REGULATION – 2021

NM1117 – FULL STACK WITH JAVA

THANTHAI PERIYAR
GOVERNMENT INSTITUTE OF TECHNOLOGY
VELLORE - 02

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NM1117 – FULL STACK WITH JAVA

(R2021)

NAME: ……..

REGISTER NUMBER: ……..

YEAR AND SEMESTER: ……..

ACADEMIC YEAR: ……...

THANTHAI PERIYAR GOVERNMENT INSTITUTE OF TECHNOLOGY, VELLORE – 632002

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NM1117 – FULL STACK WITH JAVA

Certified that this is a bonafide record of work done by…….

with Register Number ……. of IV year and VII Semester in

B.E Computer Science and Engineering submitted for the Practical Examination

NM1117 – FULL STACK WITH JAVA during the academic year 2025 - 2026.

Staff In-charge **Head of the Department**

Submitted for Practical Examination held on.....at TPGIT, Bagayam, Vellore - 02.

Internal Examiner **External Examiner**

NM1117 – Full Stack with Java

INDEX

Spring IOC

EX 1
Date:

Aim:
To understand and implement Spring Inversion of Control (IoC) container and demonstrate how objects are created, managed, and injected by the Spring framework instead of being created manually in the code.
Procedure:

Introduction to IoC:
The Inversion of Control (IoC) principle in Spring allows the framework to manage object creation and dependencies automatically.

It promotes loose coupling through Dependency Injection (DI), where the container injects required dependencies into objects at runtime.
Setting up the Environment:
Install Java JDK (version 8 or above).

Download and configure Spring Framework libraries or use Spring Tool Suite (STS) or Eclipse IDE.

Create a new Java project and add the Spring JAR files to the build path.
Creating the Bean Class:

```
public class Student {
   private String name;

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

   public void displayInfo() {
      System.out.println("Student name: " + name);
   }
}
```

Configuring the Bean in XML:

Create a file named applicationContext.xml in the src folder.

```
<?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 id="studentBean" class="com.example.Student">
        <property name="name" value="John"/>
     </bean>
</beans>
```

Creating the Main Class:

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

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
      Student student = (Student) context.getBean("studentBean");
      student.displayInfo();
   }
}
```

Running the Application:

- Compile and run MainApp.java.
- The Spring container reads the XML configuration file, creates the Student object, and injects the value “John” into it.
- Output appears on the console.

Output:

Student name: John

Result:

The Spring IoC container successfully created and managed the Student object using the configuration defined in applicationContext.xml. Thus, the concept of Inversion of Control (IoC) was successfully demonstrated using the Spring Framework.

Spring Boot

EX 2
Date:

Aim:
To develop and run a simple Spring Boot application that demonstrates auto-configuration, embedded server support, and simplified project setup without requiring manual XML configuration.

Procedure:

Introduction to Spring Boot:

- Spring Boot is a framework built on top of the Spring Framework that simplifies Java application development.
- It provides auto-configuration, standalone applications, and an embedded web server (Tomcat/Jetty).
- The main purpose is to minimize configuration and accelerate application setup.

Setting up the Environment:

- Install Java JDK (version 8 or above).
- Install an IDE like Spring Tool Suite (STS) or Eclipse.
- Go to https://start.spring.io and create a new Spring Boot project with the following details:
- Project: Maven Project
- Language: Java
- Spring Boot Version: Latest Stable Version
- Dependencies: Spring Web
- Download and extract the generated project and open it in your IDE.

Creating the Main Application Class:
In the src/main/java directory, create the main class file.

```
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);
   }
}
```

Creating a REST Controller:

Add a simple controller to display a welcome message.

```
package com.example.demo.controller;

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

@RestController
public class HelloController {
   @GetMapping("/hello")
   public String sayHello() {
      return "Welcome to Spring Boot Application!";
   }
}
```

Running the Application:

- Right-click on the project → Run As → Spring Boot App.
- The application starts on the embedded Tomcat server (default port: 8080).
- Open a web browser and visit:
- http://localhost:8080/hello
- The browser displays the output message.

Understanding Auto-Configuration:

- Spring Boot automatically configures beans and web server settings based on dependencies.
- No need for explicit XML configuration as in traditional Spring applications.
- The @SpringBootApplication annotation combines @Configuration, @EnableAutoConfiguration, @ComponentScan.

Output:
Welcome to Spring Boot Application!

Result:

A Spring Boot web application was successfully created and executed. The embedded Tomcat server hosted the application, and accessing the URL http://localhost:8080/hello displayed the message:

EX 3
Date: **Constructor Injection**

Aim:
To understand and implement Constructor Injection in the Spring Framework, where dependencies are provided to a class through its constructor at the time of object creation using the IoC container..

Procedure:

Introduction to Constructor Injection:

- Constructor Injection is one of the types of Dependency Injection (DI) in Spring.
- In this method, dependencies are injected into a class through its constructor parameters.
- This promotes loose coupling and ensures that required dependencies are supplied when the object is created by the Spring container.

Setting up the Environment:

- Install Java JDK (version 8 or above).
- Download and configure the Spring Framework JAR files or use Spring Tool Suite (STS) / Eclipse IDE.
- Create a new Java project and add the Spring libraries to the classpath.

Creating the Dependent Class (Address.java):

```
public class Address {
   private String city;
   private String state;

   public Address(String city, String state) {
      this.city = city;
      this.state = state;
   }

   public void displayAddress() {
      System.out.println("City: " + city + ", State: " + state);
   }
}
```

Creating the Main Bean Class (Employee.java):

```
public class Employee {
   private int id;
   private String name;
   private Address address;
   public Employee(int id, String name, Address address) {
      this.id = id;
```

```
 this.name = name;
        this.address = address;
    }

    public void displayInfo() {
        System.out.println("Employee ID: " + id);
        System.out.println("Employee Name: " + name);
        address.displayAddress();
    }
}
```

Creating the Spring Configuration File (applicationContext.xml):

```
    <?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 id="addressBean" class="com.example.Address">
            <constructor-arg value="Chennai"/>
            <constructor-arg value="Tamil Nadu"/>
        </bean>

        <bean id="employeeBean" class="com.example.Employee">
            <constructor-arg value="101"/>
            <constructor-arg value="John"/>
            <constructor-arg ref="addressBean"/>
        </bean>
    </beans>
```

Creating the Main Class (MainApp.java):

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

public class MainApp {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
        Employee emp = (Employee) context.getBean("employeeBean");
        emp.displayInfo();
    }
}
```

Running the Application:

- Compile and run the program.
- The Spring container automatically creates the Address and Employee beans and injects dependencies via the constructor

Output:

Employee ID: 101
Employee Name: John
City: Chennai, State: Tamil Nadu

Result:

The Spring IoC container successfully created and managed the Employee and Address objects. The dependencies were injected through the constructor, proving the working of Constructor Injection in the Spring Framework.

EX4

Date: **Pagination and Sorting**

Aim:

To implement Pagination and Sorting in a Spring Boot application using Spring Data JPA, allowing data to be retrieved and displayed in a structured, page-wise, and ordered manner.Procedure:

Procedure:

Introduction:

- In real-world applications, displaying large sets of data at once can reduce performance and readability.
- Pagination divides data into multiple pages, while Sorting arranges the data in a specific order (ascending or descending).
- Spring Data JPA provides built-in support for both using PagingAndSortingRepository or Pageable interface.

Setting up the Environment:

- Install Java JDK (version 8 or above).
- Use Spring Tool Suite (STS) or Eclipse IDE.
- Create a new Spring Boot project using https://start.spring.io with dependencies:
- Spring Web
- Spring Data JPA
- MySQL Driver

Database Configuration (application.properties):

```
spring.datasource.url=jdbc:mysql://localhost:3306/studentdb
spring.datasource.username=root
spring.datasource.password=****
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Creating the Entity Class (Student.java):

```
package com.example.demo.entity;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
public class Student {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private int age;

    // Getters and Setters
}
```

Creating the Repository Interface (StudentRepository.java):

```
package com.example.demo.repository;

import org.springframework.data.repository.PagingAndSortingRepository;
import com.example.demo.entity.Student;

public interface StudentRepository extends PagingAndSortingRepository<Student, Long> {
}
```

Creating the Controller Class (StudentController.java):

```
package com.example.demo.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Sort;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import com.example.demo.entity.Student;
import com.example.demo.repository.StudentRepository;

@RestController
public class StudentController {

    @Autowired
    private StudentRepository repo;

    @GetMapping("/students")
    public Page<Student> getStudents(@RequestParam int page, @RequestParam int size) {
        return repo.findAll(PageRequest.of(page, size, Sort.by("name").ascending()));
    }
}
```

Running the Application:

- Run the Spring Boot project.
- Insert some sample records in the studentdb database table.

:

- http://localhost:8080/students?page=0&size=3

- The response shows 3 student records (first page), sorted by name.

Output:


```
[
  {"id":1,"name":"Anitha","age":21},
  {"id":2,"name":"Bala","age":22},
  {"id":3,"name":"Kiran","age":23}
]
```


Result:

The Spring Boot application successfully implemented Pagination and Sorting using Spring Data JPA. The data was displayed in pages with sorted order, demonstrating efficient retrieval of large datasets in a manageable format.

EX 5

Date: **Query Creation Based on Method Names**

Aim:

To understand and implement Query Creation using Method Names in Spring Data JPA, allowing automatic query generation based on repository method naming conventions without writing explicit SQL or JPQL queries.

Procedure:

Introduction:

- Spring Data JPA automatically generates SQL queries by interpreting repository method names.
- Developers can define query methods in repository interfaces by following a specific naming pattern (e.g., findBy, countBy, deleteBy, etc.).
- This feature improves productivity and reduces boilerplate code.

Setting up the Environment:

- Install Java JDK (version 8 or above).
- Use Spring Tool Suite (STS) or Eclipse IDE.
- Create a new Spring Boot project using https://start.spring.io with dependencies:
- Spring Web
- Spring Data JPA
- MySQL Driver

Database Configuration (application.properties):

```
spring.datasource.url=jdbc:mysql://localhost:3306/studentdb
spring.datasource.username=root
spring.datasource.password=****
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Creating the Entity Class (Student.java):

```
package com.example.demo.entity;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private int age;
    private String city;

    // Getters and Setters
}
```

Creating the Repository Interface (StudentRepository.java):

```
package com.example.demo.repository;

import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;
import com.example.demo.entity.Student;

public interface StudentRepository extends JpaRepository<Student, Long> {
    List<Student> findByCity(String city);
    List<Student> findByAgeGreaterThan(int age);
    List<Student> findByNameStartingWith(String prefix);
}
```

Creating the Controller (StudentController.java):

```
package com.example.demo.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import java.util.List;
import com.example.demo.entity.Student;
import com.example.demo.repository.StudentRepository;

@RestController
public class StudentController {

    @Autowired
    private StudentRepository repo;
```

```
 @GetMapping("/byCity")
   public List<Student> getByCity(@RequestParam String city) {
      return repo.findByCity(city);
   }

   @GetMapping("/byAge")
   public List<Student> getByAge(@RequestParam int age) {
      return repo.findByAgeGreaterThan(age);
   }

   @GetMapping("/byName")
   public List<Student> getByName(@RequestParam String prefix) {
      return repo.findByNameStartingWith(prefix);
   }
}
```

Running the Application:

Run the Spring Boot project.

Insert sample records into the studentdb table.

Access the endpoints using URLs such as:

http://localhost:8080/byCity?city=Chennai
http://localhost:8080/byAge?age=21
http://localhost:8080/byName?prefix=A

Output:


```
[
  {"id":1,"name":"Anitha","age":20,"city":"Chennai"},
  {"id":4,"name":"Arjun","age":23,"city":"Chennai"}
]
```


Result:

The Spring Boot application successfully generated and executed queries based on method names defined in the repository interface.This experiment demonstrates how Spring Data JPA simplifies query creation, eliminating the need for manually written SQL or JPQL queries

EX 6

Date: **Update Using Spring Data JPA**

Aim:

To implement the update operation on database records using Spring Data JPA, demonstrating how entity objects can be modified and persisted automatically by the Spring framework.

Procedure:

Introduction:

- Spring Data JPA allows easy CRUD operations on database entities.
- The update operation is performed by retrieving an existing entity, modifying its fields, and saving it back using the repository's save() method.
- This ensures the changes are reflected in the database without manually writing SQL update statements.

Setting up the Environment:

- Install Java JDK (version 8 or above).
- Use Spring Tool Suite (STS) or Eclipse IDE.

Create a new Spring Boot project with dependencies:

- Spring Web
- Spring Data JPA
- MySQL Driver

Database Configuration (application.properties):

```
spring.datasource.url=jdbc:mysql://localhost:3306/studentdb
spring.datasource.username=root
spring.datasource.password=****
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Creating the Entity Class (Student.java):

```
package com.example.demo.entity;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
@Entity
public class Student {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String name;
   private int age;
   private String city;

   // Getters and Setters
}
```

Creating the Repository Interface (StudentRepository.java):

```
package com.example.demo.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.example.demo.entity.Student;

public interface StudentRepository extends JpaRepository<Student, Long> {
}
```

Creating the Service/Controller for Update (StudentController.java):

```
package com.example.demo.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import com.example.demo.entity.Student;
import com.example.demo.repository.StudentRepository;
```

```
@RestController
public class StudentController {

    @Autowired
    private StudentRepository repo;

    @PutMapping("/updateStudent/{id}")
    public Student updateStudent(@PathVariable Long id, @RequestBody Student updatedStudent) {
        Student student = repo.findById(id).orElseThrow(() -> new RuntimeException("Student not found"));
        student.setName(updatedStudent.getName());
        student.setAge(updatedStudent.getAge());
        student.setCity(updatedStudent.getCity());
        return repo.save(student);
    }
}
```

Running the Application:

- Run the Spring Boot project.
- Insert sample records in the studentdb table.

Use a REST client (like Postman) to send a PUT request:

URL: http://localhost:8080/updateStudent/1

Body (JSON):

```
{
  "name": "Anitha",
  "age": 22,
  "city": "Bangalore"
}
```

The repository updates the entity and persists the changes in the database.

Output:

```
{
  "id": 1,
  "name": "Anitha",
  "age": 22,
  "city": "Bangalore"
}
```

Result:

The update operation using Spring Data JPA was successfully performed. The Student entity was retrieved, modified, and saved back to the database automatically. This demonstrates the simplicity and efficiency of performing updates with Spring Data JPA without manually writing SQL queries.

EX 7
Date: **Custom Repository Implementation Using Spring Data JPA**

Aim:

To implement custom repository methods in Spring Data JPA, enabling execution of complex queries that are not achievable using standard repository methods.

Procedure:

Introduction:

- Spring Data JPA allows creation of custom repository implementations when default JpaRepository or CrudRepository methods are insufficient.
- Custom methods can include complex queries using JPQL, native SQL, or programmatic logic.
- This improves flexibility while maintaining Spring Data JPA's convenience.

Setting up the Environment:

- Install Java JDK (version 8 or above).
- Use Spring Tool Suite (STS) or Eclipse IDE.

Create a Spring Boot project with dependencies:

- Spring Web
- Spring Data JPA
- MySQL Driver

Database **Configuration** (application.properties):

```
spring.datasource.url=jdbc:mysql://localhost:3306/studentdb
spring.datasource.username=root
spring.datasource.password=****
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Creating the Entity Class (Student.java):

```
package com.example.demo.entity;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
@Entity
public class Student {
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)


    private Long id;
    private String name;
    private int age;


  private String city;

    // Getters and Setters
}
```

Creating the Repository Interface (StudentRepository.java):

```
package com.example.demo.repository;

import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import com.example.demo.entity.Student;

public interface StudentRepository extends JpaRepository<Student, Long> {

    // Custom query using JPQL
    @Query("SELECT s FROM Student s WHERE s.age > ?1 AND s.city = ?2")
    List<Student> findStudentsByAgeAndCity(int age, String city);
}
```

Creating the Controller (StudentController.java):

```
package com.example.demo.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import java.util.List;
import com.example.demo.entity.Student;
import com.example.demo.repository.StudentRepository;

@RestController
public class StudentController {

    @Autowired
    private StudentRepository repo;

    @GetMapping("/customStudents")
    public List<Student> getCustomStudents(@RequestParam int age, @RequestParam String city) {
        return repo.findStudentsByAgeAndCity(age, city);
    }
}
```

Running the Application:

- Run the Spring Boot project.
- Insert sample records into the studentdb table.

Access the endpoint using:

http://localhost:8080/customStudents?age=20&city=Chennai

The query returns students older than 20 from Chennai.

Output:


```
[
  {"id":2,"name":"Bala","age":22,"city":"Chennai"},
  {"id":4,"name":"Arjun","age":23,"city":"Chennai"}
]
```


Result:

The custom repository implementation using JPQL successfully retrieved records based on complex criteria.
This experiment demonstrates how Spring Data JPA allows creating custom query methods for advanced data retrieval without manually writing boilerplate SQL code.

EX 8
Date: **CRUD Operations Using Spring Data JPA**

Aim:

To implement CRUD (Create, Read, Update, Delete) operations on a database entity using Spring Data JPA, demonstrating how Spring simplifies database interaction.

Procedure:

Introduction:

CRUD operations represent the basic operations in a database:

- Create – Insert new records
- Read – Retrieve existing records
- Update – Modify existing records
- Delete – Remove records

Spring Data JPA provides ready-made methods in JpaRepository to perform all CRUD operations without writing SQL queries manually.

Setting up the Environment:

- Install Java JDK (version 8 or above).
- Use Spring Tool Suite (STS) or Eclipse IDE.

Create a Spring Boot project with dependencies:

- Spring Web
- Spring Data JPA
- MySQL Driver

Database Configuration (application.properties):

```
spring.datasource.url=jdbc:mysql://localhost:3306/studentdb
spring.datasource.username=root
spring.datasource.password=****
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Creating the Entity Class (Student.java):

```
package com.example.demo.entity;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
```

```
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private int age;
    private String city;

    // Getters and Setters
}
```

Creating the Repository Interface (StudentRepository.java):

```
package com.example.demo.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.example.demo.entity.Student;

public interface StudentRepository extends JpaRepository<Student, Long> {
}
```

Creating the Controller for CRUD Operations (StudentController.java):

```
package com.example.demo.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import java.util.List;
import com.example.demo.entity.Student;
import com.example.demo.repository.StudentRepository;

@RestController
public class StudentController {

    @Autowired
    private StudentRepository repo;

    // Create
    @PostMapping("/students")
    public Student addStudent(@RequestBody Student student) {
        return repo.save(student);
    }

    // Read All
    @GetMapping("/students")
    public List<Student> getAllStudents() {
        return repo.findAll();
    }
    // Read by ID
    @GetMapping("/students/{id}")

public Student getStudentById(@PathVariable Long id) {
        return repo.findById(id).orElseThrow(() -> new RuntimeException("Student not found"));
    }
```

```
    // Update
    @PutMapping("/students/{id}")
    public Student updateStudent(@PathVariable Long id, @RequestBody Student updatedStudent) {
        Student student = repo.findById(id).orElseThrow(() -> new RuntimeException("Student not found"));
        student.setName(updatedStudent.getName());
        student.setAge(updatedStudent.getAge());
        student.setCity(updatedStudent.getCity());
        return repo.save(student);
    }

    // Delete
    @DeleteMapping("/students/{id}")
    public String deleteStudent(@PathVariable Long id) {
        repo.deleteById(id);
        return "Student deleted with ID: " + id;
    }
}
```

Running the Application:

Run the Spring Boot project.

Use a REST client (Postman) to perform POST, GET, PUT, and DELETE requests to the endpoints:

POST /students → Add new student
GET /students → View all students
GET /students/{id} → View a student by ID
PUT /students/{id} → Update student
DELETE /students/{id} → Delete student

Output:

Example of GET /students:

```
[
  {"id":1,"name":"Anitha","age":22,"city":"Bangalore"},
  {"id":2,"name":"Bala","age":23,"city":"Chennai"}
]
```

Example of DELETE /students/1:

Student deleted with ID: 1

Result:

All CRUD operations were successfully implemented using Spring Data JPA. The JpaRepository methods simplified database interaction, allowing creation, retrieval, update, and deletion of Student records without manual SQL queries.

EX 9
Date: **Spring 5 Basics Transport Domain Capstone – InfyGo**

Aim:

To develop a Spring 5-based transport domain application (InfyGo) that demonstrates the basics of Spring 5 framework including IoC, Dependency Injection, and RESTful services, simulating a travel booking system.

Procedure:

Introduction:

InfyGo is a capstone project that represents a transport booking system for buses, trains, or flights.

The project demonstrates Spring 5 fundamentals such as:

- Inversion of Control (IoC)
- Dependency Injection (DI)
- RESTful API development
- Data persistence using Spring Data JPA

Setting up the Environment:

- Install Java JDK (version 8 or above).
- Use Spring Tool Suite (STS) or Eclipse IDE.

Create a new Spring Boot project with dependencies:

- Spring Web
- Spring Data JPA
- MySQL Driver

Database Configuration (application.properties):

```
spring.datasource.url=jdbc:mysql://localhost:3306/infygo
spring.datasource.username=root
spring.datasource.password=****
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Creating the Entity Class (TravelBooking.java):

```
package com.example.infygo.entity;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
```

```
@Entity


public class TravelBooking {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String passengerName;
   private String source;
   private String destination;
   private String travelDate;

   // Getters and Setters
}
```

Creating the Repository Interface (BookingRepository.java):

```
package com.example.infygo.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.example.infygo.entity.TravelBooking;

public interface BookingRepository extends JpaRepository<TravelBooking, Long> {
}
```

Creating the Controller Class (BookingController.java):

```
package com.example.infygo.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import java.util.List;
import com.example.infygo.entity.TravelBooking;
import com.example.infygo.repository.BookingRepository;

@RestController
@RequestMapping("/bookings")
public class BookingController {

   @Autowired
   private BookingRepository repo;

   // Create a booking
   @PostMapping
   public TravelBooking addBooking(@RequestBody TravelBooking booking) {
      return repo.save(booking);
   }

   // Get all bookings
   @GetMapping
   public List<TravelBooking> getAllBookings() {
      return repo.findAll();
   }
}
```

Running the Application:

Run the Spring Boot project.

Insert sample booking data using POST requests in a REST client (Postman).

Access the endpoint using:

POST /bookings → Add new booking
GET /bookings → View all bookings

Output:

Example GET /bookings response:


```
[
  {"id":1,"passengerName":"Anitha","source":"Chennai","destination":"Bangalore","travelDate":"2025-10-20"},
  {"id":2,"passengerName":"Bala","source":"Chennai","destination":"Hyderabad","travelDate":"2025-10-21"}
]
```


Result:

The InfyGo transport domain application was successfully implemented using Spring 5.
The project demonstrated IoC, Dependency Injection, RESTful API development, and database persistence using Spring Data JPA.This experiment shows a practical application of Spring 5 concepts in a real-world transport booking system.

EX 10
Date: **Aspect-Oriented Programming (AOP)**

Aim:

To understand and implement Aspect-Oriented Programming (AOP) in Spring, demonstrating how cross-cutting concerns (like logging, security, or transactions) can be modularized using aspects, advice, and pointcuts.

Procedure:

Introduction:

- AOP separates cross-cutting concerns from the main business logic.
- Common use cases include:
- Logging
- Security checks
- Transaction management

Key concepts:

- Aspect – a module that contains cross-cutting concerns
- Advice – action taken by an aspect (before, after, or around method execution)
- Pointcut – expression that selects join points (methods) where advice is applied

Setting up the Environment:

- Install Java JDK (version 8 or above).
- Use Spring Tool Suite (STS) or Eclipse IDE.

Create a Spring Boot project with dependencies:

- Spring AOP
- Spring Web

Creating the Business Class (CustomerService.java):

```
package com.example.demo.service;

import org.springframework.stereotype.Service;

@Service
public class CustomerService {
   public void addCustomer(String name) {
      System.out.println("Adding customer: " + name);
   }
```

```
    public void deleteCustomer(String name) {
        System.out.println("Deleting customer: " + name);
    }
}
```

Creating the Aspect Class (LoggingAspect.java):

```
package com.example.demo.aspect;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class LoggingAspect {

    @Before("execution(* com.example.demo.service.CustomerService.*(..))")
    public void logBeforeMethods() {
        System.out.println("Logging before method execution");
    }
}
```

Creating the Main Application (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);
    }
}
```

Running the Application:

Run the Spring Boot project.
Autowire CustomerService in a test runner or REST controller and call methods:
customerService.addCustomer("Anitha");
customerService.deleteCustomer("Bala");
The AOP before advice executes automatically before each method in CustomerService.

Output:

```
Logging before method execution
Adding customer: Anitha
Logging before method execution
Deleting customer: Bala
```

Result:

The Aspect-Oriented Programming (AOP) concept in Spring was successfully implemented.
The LoggingAspect executed cross-cutting logging behavior before each method in the CustomerService class without modifying the business logic.This demonstrates how AOP modularizes cross-cutting concerns efficiently in Spring applications.

EX 11
Date: **Spring REST Health Care Domain Capstone – WeCARE**

Aim:

To develop a Spring REST-based healthcare application (WeCARE) demonstrating RESTful API design, CRUD operations, and Spring Data JPA integration in a real-world health care domain.

Procedure:
Introduction:
WeCARE is a health care domain capstone project designed to manage patient records, appointments, and health services.
The project demonstrates:

- Spring REST API development
- CRUD operations with Spring Data JPA
- Database interaction using MySQL
- IoC and Dependency Injection

Setting up the Environment:

- Install Java JDK (version 8 or above).
- Use Spring Tool Suite (STS) or Eclipse IDE.

Create a Spring Boot project with dependencies:

- Spring Web
- Spring Data JPA
- MySQL Driver

Database Configuration (application.properties):

```
spring.datasource.url=jdbc:mysql://localhost:3306/wecare
spring.datasource.username=root
spring.datasource.password=****
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Creating the Entity Class (Patient.java):

```
package com.example.wecare.entity;import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
@Entity
public class Patient {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private int age;
    private String disease;

    // Getters and Setters
}
```

Creating the Repository Interface (PatientRepository.java):

```
package com.example.wecare.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.example.wecare.entity.Patient;

public interface PatientRepository extends JpaRepository<Patient, Long> {
}
```

Creating the REST Controller (PatientController.java):

```
package com.example.wecare.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import java.util.List;
import com.example.wecare.entity.Patient;
import com.example.wecare.repository.PatientRepository;

@RestController
@RequestMapping("/patients")
public class PatientController {

    @Autowired
    private PatientRepository repo;

    // Create patient
    @PostMapping
    public Patient addPatient(@RequestBody Patient patient) {
        return repo.save(patient);
    }

    // Get all patients
    @GetMapping
    public List<Patient> getAllPatients() {
        return repo.findAll();
    }

    // Get patient by ID
```

```
public Patient getPatientById(@PathVariable Long id) {

    return repo.findById(id).orElseThrow(() -> new RuntimeException("Patient not found"));
   }

 // Update patient
   @PutMapping("/{id}")
   public Patient updatePatient(@PathVariable Long id, @RequestBody Patient updatedPatient) {
      Patient patient = repo.findById(id).orElseThrow(() -> new RuntimeException("Patient not found"));
      patient.setName(updatedPatient.getName());
      patient.setAge(updatedPatient.getAge());
      patient.setDisease(updatedPatient.getDisease());
      return repo.save(patient);
   }

   // Delete patient
   @DeleteMapping("/{id}")
   public String deletePatient(@PathVariable Long id) {
      repo.deleteById(id);
      return "Patient deleted with ID: " + id;
   }
}
```

Running the Application:

Run the Spring Boot project.

Use Postman or a REST client to perform POST, GET, PUT, DELETE requests:

```
POST   /patients       → Add patient
GET    /patients       → View all patients
GET    /patients/{id}  → View patient by ID
PUT    /patients/{id}  → Update patient
DELETE /patients/{id}  → Delete patient
```

Output:

Example GET /patients response:

```
[
  {"id":1,"name":"Anitha","age":30,"disease":"Flu"},
  {"id":2,"name":"Bala","age":45,"disease":"Diabetes"}
]
```

Example DELETE /patients/1 response:
Patient deleted with ID: 1

Result:
The WeCARE healthcare application was successfully implemented using Spring REST and Spring Data JPA. The application supports CRUD operations on patient records and demonstrates practical use of Spring REST in a health care domain scenario.This experiment highlights real-world integration of Spring REST API, JPA, and database interaction

EX 12
Date: **Spring REST Transport & Logistic Domain Capstone – InfyRail**

Aim:
To develop a Spring REST-based transport and logistics application (InfyRail) demonstrating RESTful API development, CRUD operations, and Spring Data JPA integration for a railway transport system.

Procedure:

Introduction:

InfyRail is a transport and logistics domain capstone project designed to manage train bookings, schedules, and passenger records.

The project demonstrates:

- RESTful API development using Spring Boot
- CRUD operations with Spring Data JPA
- Database integration using MySQL
- IoC and Dependency Injection in Spring

Setting up the Environment:

- Install Java JDK (version 8 or above).
- Use Spring Tool Suite (STS) or Eclipse IDE.

Create a Spring Boot project with dependencies:

- Spring Web
- Spring Data JPA
- MySQL Driver

Database Configuration (application.properties):

```
spring.datasource.url=jdbc:mysql://localhost:3306/infyrail
spring.datasource.username=root
spring.datasource.password=****
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Creating the Entity Class (TrainBooking.java):

```
package com.example.infyrail.entity;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
```

```
@Entity
public class TrainBooking {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
private String passengerName;
private String source;
private String destination;
private String travelDate;

// Getters and Setters
}
```

Creating the Repository Interface (TrainBookingRepository.java):

```
package com.example.infyrail.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.example.infyrail.entity.TrainBooking;

public interface TrainBookingRepository extends JpaRepository<TrainBooking, Long> {
}
```

Creating the REST Controller (TrainBookingController.java):

```
package com.example.infyrail.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import java.util.List;
import com.example.infyrail.entity.TrainBooking;
import com.example.infyrail.repository.TrainBookingRepository;

@RestController
@RequestMapping("/trainBookings")
public class TrainBookingController {

@Autowired
private TrainBookingRepository repo;
// Create booking
@PostMapping
public TrainBooking addBooking(@RequestBody TrainBooking booking) {
return repo.save(booking);
}
// Get all bookings
@GetMapping
public List<TrainBooking> getAllBookings() {
return repo.findAll();
}

// Get booking by ID
@GetMapping("/{id}")
public TrainBooking getBookingById(@PathVariable Long id) {
```

```
return repo.findById(id).orElseThrow(() -> new RuntimeException("Booking not found"));
}

// Update booking
   @PutMapping("/{id}")
   public TrainBooking updateBooking(@PathVariable Long id, @RequestBody TrainBooking updatedBooking) {
      TrainBooking booking = repo.findById(id).orElseThrow(() -> new RuntimeException("Booking not found"));
      booking.setPassengerName(updatedBooking.getPassengerName());
      booking.setSource(updatedBooking.getSource());
      booking.setDestination(updatedBooking.getDestination());
      booking.setTravelDate(updatedBooking.getTravelDate());
      return repo.save(booking);
   }

   // Delete booking
   @DeleteMapping("/{id}")
   public String deleteBooking(@PathVariable Long id) {
      repo.deleteById(id);
      return "Booking deleted with ID: " + id;
   }
}
```

Running the Application:
Run the Spring Boot project.
Use Postman or REST client to perform POST, GET, PUT, DELETE requests:
POST /trainBookings → Add train booking
GET /trainBookings → View all bookings
GET /trainBookings/{id} → View booking by ID
PUT /trainBookings/{id} → Update booking
DELETE /trainBookings/{id} → Delete booking

Output: Example GET /trainBookings response:

```
[
  {"id":1,"passengerName":"Anitha","source":"Chennai","destination":"Bangalore","travelDate":"2025-10-25"},
  {"id":2,"passengerName":"Bala","source":"Chennai","destination":"Hyderabad","travelDate":"2025-10-26"}
]
```

Example DELETE /trainBookings/1 response:

```
Booking deleted with ID: 1
```

Result:

The InfyRail transport and logistics application was successfully implemented using Spring REST and Spring Data JPA. The project demonstrated CRUD operations on train bookings, RESTful API development, and effective database integration, highlighting practical application of Spring Boot in transport and logistics systems

EX 13
Date: **Spring REST Transport & Logistic Domain Capstone – InfyGo**

Aim:

To develop a Spring REST-based transport and logistics application (InfyGo) demonstrating RESTful API development, CRUD operations, and Spring Data JPA integration for a transport system.

Procedure:

Introduction:

InfyGo in this capstone represents a transport booking and logistics management system.

It demonstrates:

- REST API development using Spring Boot
- CRUD operations with Spring Data JPA
- Database integration using MySQL
- IoC and Dependency Injection in Spring

Setting up the Environment:

- Install Java JDK (version 8 or above).
- Use Spring Tool Suite (STS) or Eclipse IDE.

Create a Spring Boot project with dependencies:

- Spring Web
- Spring Data JPA
- MySQL Driver

Database Configuration (application.properties):

```
spring.datasource.url=jdbc:mysql://localhost:3306/infygo
spring.datasource.username=root
spring.datasource.password=****
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Creating the Entity Class (TransportBooking.java):

```
package com.example.infygo.entity;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
```

```
import jakarta.persistence.Id;
@Entity
public class TransportBooking {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String passengerName;
    private String source;
    private String destination;
    private String travelDate;

    // Getters and Setters
}
```

Creating the Repository Interface (TransportBookingRepository.java):

```
package com.example.infygo.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.example.infygo.entity.TransportBooking;

public interface TransportBookingRepository extends JpaRepository<TransportBooking, Long> {
}
```

Creating the REST Controller (TransportBookingController.java):

```
package com.example.infygo.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import java.util.List;
import com.example.infygo.entity.TransportBooking;
import com.example.infygo.repository.TransportBookingRepository;

@RestController
@RequestMapping("/transportBookings")
public class TransportBookingController {

    @Autowired
    private TransportBookingRepository repo;

    // Create booking
    @PostMapping
    public TransportBooking addBooking(@RequestBody TransportBooking booking) {
        return repo.save(booking);
    }

    // Get all bookings
    @GetMapping
    public List<TransportBooking> getAllBookings() {
        return repo.findAll();
    }
// Get booking by ID
    @GetMapping("/{id}")
```

```
public TransportBooking getBookingById(@PathVariable Long id) {
        return repo.findById(id).orElseThrow(() -> new RuntimeException("Booking not found"));
    }

    // Update booking
    @PutMapping("/{id}")
    public TransportBooking updateBooking(@PathVariable Long id, @RequestBody TransportBooking updatedBooking)
{
        TransportBooking booking = repo.findById(id).orElseThrow(() -> new RuntimeException("Booking not found"));
        booking.setPassengerName(updatedBooking.getPassengerName());
        booking.setSource(updatedBooking.getSource());
        booking.setDestination(updatedBooking.getDestination());
        booking.setTravelDate(updatedBooking.getTravelDate());
        return repo.save(booking);
    }

    // Delete booking
    @DeleteMapping("/{id}")
    public String deleteBooking(@PathVariable Long id) {
        repo.deleteById(id);
        return "Booking deleted with ID: " + id;
    }
}
```

Running the Application:

Run the Spring Boot project.

Use Postman or REST client to perform POST, GET, PUT, DELETE requests:

POST /transportBookings → Add booking
GET /transportBookings → View all bookings
GET /transportBookings/{id} → View booking by ID
PUT /transportBookings/{id} → Update booking
DELETE /transportBookings/{id} → Delete booking

Output:

Example GET /transportBookings response:

```
[
  {"id":1,"passengerName":"Anitha","source":"Chennai","destination":"Bangalore","travelDate":"2025-10-30"},
  {"id":2,"passengerName":"Bala","source":"Chennai","destination":"Hyderabad","travelDate":"2025-10-31"}
]
```

Example DELETE /transportBookings/1 response:

Booking deleted with ID: 1

Result:

The InfyGo transport and logistics application was successfully implemented using Spring REST and Spring Data JPA.The application supports CRUD operations on transport bookings, demonstrates RESTful API design, and integrates effectively with a MySQL database.This experiment highlights practical application of Spring Boot, REST API, and JPA in a transport and logistics scenario**.**

EX 14
Date: **Spring REST Business Domain Capstone – InfyDUGuesstimate**

Aim:

To develop a Spring REST-based business domain application (InfyDUGuesstimate) demonstrating RESTful API development, CRUD operations, and Spring Data JPA integration in a business estimation context.

Procedure:

Introduction:

InfyDUGuesstimate is a capstone project designed to manage business estimation and project data, helping users submit, update, and view business estimates.

The project demonstrates:

- RESTful API development using Spring Boot
- CRUD operations with Spring Data JPA
- Database integration using MySQL
- IoC and Dependency Injection in Spring

Setting up the Environment:

- Install Java JDK (version 8 or above).
- Use Spring Tool Suite (STS) or Eclipse IDE.

Create a Spring Boot project with dependencies:

- Spring Web
- Spring Data JPA
- MySQL Driver

Database Configuration (application.properties):

```
spring.datasource.url=jdbc:mysql://localhost:3306/infyduguesstimate
spring.datasource.username=root
spring.datasource.password=****
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Creating the Entity Class (Estimate.java):

```
package com.example.infyduguesstimate.entity;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
```

```
import jakarta.persistence.Id;

@Entity
public class Estimate {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String projectName;
   private String clientName;
   private double estimatedCost;

   // Getters and Setters
}
```

Creating the Repository Interface (EstimateRepository.java):

```
package com.example.infyduguesstimate.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.example.infyduguesstimate.entity.Estimate;

public interface EstimateRepository extends JpaRepository<Estimate, Long> {
}
```

Creating the REST Controller (EstimateController.java):

```
package com.example.infyduguesstimate.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import java.util.List;
import com.example.infyduguesstimate.entity.Estimate;
import com.example.infyduguesstimate.repository.EstimateRepository;

@RestController
@RequestMapping("/estimates")
public class EstimateController {

   @Autowired
   private EstimateRepository repo;

   // Create estimate
   @PostMapping
   public Estimate addEstimate(@RequestBody Estimate estimate) {
      return repo.save(estimate);
   }

   // Get all estimates
   @GetMapping
   public List<Estimate> getAllEstimates() {
      return repo.findAll();
   }

   // Get estimate by ID
```

```
 @GetMapping("/{id}")
   public Estimate getEstimateById(@PathVariable Long id) {
      return repo.findById(id).orElseThrow(() -> new RuntimeException("Estimate not found"));
   }

   // Update estimate
   @PutMapping("/{id}")
   public Estimate updateEstimate(@PathVariable Long id, @RequestBody Estimate updatedEstimate) {
      Estimate estimate = repo.findById(id).orElseThrow(() -> new RuntimeException("Estimate not found"));
      estimate.setProjectName(updatedEstimate.getProjectName());
      estimate.setClientName(updatedEstimate.getClientName());
      estimate.setEstimatedCost(updatedEstimate.getEstimatedCost());
      return repo.save(estimate);
   }

   // Delete estimate
   @DeleteMapping("/{id}")
   public String deleteEstimate(@PathVariable Long id) {
      repo.deleteById(id);
      return "Estimate deleted with ID: " + id;
   }
}
```

Running the Application:

Run the Spring Boot project.

Use Postman or REST client to perform POST, GET, PUT, DELETE requests:

```
POST   /estimates       → Add estimate
GET    /estimates       → View all estimates
GET    /estimates/{id}  → View estimate by ID
PUT    /estimates/{id}  → Update estimate
DELETE /estimates/{id}  → Delete estimate
```

Output:
Example GET /estimates response:

```
[
  {"id":1,"projectName":"Project A","clientName":"ABC Corp","estimatedCost":50000.0},
  {"id":2,"projectName":"Project B","clientName":"XYZ Ltd","estimatedCost":75000.0}
]
```

Example DELETE /estimates/1 response:

Estimate deleted with ID: 1

Result:

The InfyDUGuesstimate business application was successfully implemented using Spring REST and Spring Data JPA. The application supports CRUD operations on business estimates, demonstrates RESTful API design, and integrates effectively with a MySQL database.This experiment highlights practical application of Spring Boot, REST API, and JPA in a business domain scenario.

EX 15
Date: **Spring Data JPA Entertainment Domain Capstone – ExpressMovies**

Aim:

To develop a Spring Data JPA-based entertainment domain application (ExpressMovies) demonstrating CRUD operations, repository management, and RESTful API integration for a movie management system.

Procedure:

Introduction:

ExpressMovies is a capstone project designed to manage movie records, schedules, and ticketing information.

The project demonstrates:

- Spring Data JPA for database interaction
- CRUD operations
- REST API integration using Spring Boot
- Dependency Injection and IoC in Spring

Setting up the Environment:

- Install Java JDK (version 8 or above).
- Use Spring Tool Suite (STS) or Eclipse IDE.

Create a Spring Boot project with dependencies:

- Spring Web
- Spring Data JPA
- MySQL Driver

Database Configuration (application.properties):

```
spring.datasource.url=jdbc:mysql://localhost:3306/expressmovies
spring.datasource.username=root
spring.datasource.password=****
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Creating the Entity Class (Movie.java):

```
package com.example.expressmovies.entity;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
```

```
@Entity
public class Movie {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;
    private String genre;
    private String releaseDate;
    private double rating;

    // Getters and Setters
}
```

Creating the Repository Interface (MovieRepository.java):

```
package com.example.expressmovies.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.example.expressmovies.entity.Movie;

public interface MovieRepository extends JpaRepository<Movie, Long> {
}
```

Creating the REST Controller (MovieController.java):

```
package com.example.expressmovies.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import java.util.List;
import com.example.expressmovies.entity.Movie;
import com.example.expressmovies.repository.MovieRepository;

@RestController
@RequestMapping("/movies")
public class MovieController {

    @Autowired
    private MovieRepository repo;

    // Create movie
    @PostMapping
    public Movie addMovie(@RequestBody Movie movie) {
        return repo.save(movie);
    }

    // Get all movies
    @GetMapping
    public List<Movie> getAllMovies() {
        return repo.findAll();
    }
```

```
  // Get movie by ID
   @GetMapping("/{id}")
   public Movie getMovieById(@PathVariable Long id) {
      return repo.findById(id).orElseThrow(() -> new RuntimeException("Movie not found"));
   }

   // Update movie
   @PutMapping("/{id}")
   public Movie updateMovie(@PathVariable Long id, @RequestBody Movie updatedMovie) {
      Movie movie = repo.findById(id).orElseThrow(() -> new RuntimeException("Movie not found"));
      movie.setTitle(updatedMovie.getTitle());
      movie.setGenre(updatedMovie.getGenre());
      movie.setReleaseDate(updatedMovie.getReleaseDate());
      movie.setRating(updatedMovie.getRating());
      return repo.save(movie);
   }

   // Delete movie
   @DeleteMapping("/{id}")
   public String deleteMovie(@PathVariable Long id) {
      repo.deleteById(id);
      return "Movie deleted with ID: " + id;
   }
}
```

Running the Application:

Run the Spring Boot project.

Use Postman or REST client to perform POST, GET, PUT, DELETE requests:

```
POST   /movies       → Add movie
GET    /movies       → View all movies
GET    /movies/{id}  → View movie by ID
PUT    /movies/{id}  → Update movie
DELETE /movies/{id}  → Delete movie
```

Output:
Example GET /movies response:

```
[
  {"id":1,"title":"Avengers","genre":"Action","releaseDate":"2025-05-01","rating":9.0},
  {"id":2,"title":"Inception","genre":"Sci-Fi","releaseDate":"2025-07-15","rating":8.8}
]
```

Example DELETE /movies/1 response:
Movie deleted with ID: 1

Result:
The ExpressMovies entertainment application was successfully implemented using Spring Data JPA and Spring Boot REST APIs. The application supports CRUD operations on movie records, demonstrating repository management, RESTful API design, and database interaction.This experiment highlights the practical application of Spring Data JPA in an entertainment domain scenario.

EX 16
Date: **Spring REST Telecom Domain Capstone – SIM Activation Portal**

Aim:

To develop a Spring REST-based telecom application that automates the SIM activation process, demonstrating RESTful API design, validation mechanisms, and database integration for telecom customer management.

Procedure:

Introduction:

The SIM Activation Portal automates the SIM activation process for customers.

Functionalities include:

- SIM validation
- Display of special offers
- Customer validation based on email and date of birth
- Validation using first name, last name, and email
- Address update
- Customer ID proof validation

The project demonstrates Spring REST, Spring Data JPA, database integration, and validation mechanisms.

Software and Hardware Requirements:
Software:

- Spring Tool Suite (STS)
- MySQL Database
- SoapUI (for API testing)

Hardware:

- i5/i7 processor or AMD R5
- 16 GB RAM, 500 GB storage

Setting up the Environment:

Create a Spring Boot project with dependencies:

- Spring Web
- Spring Data JPA
- MySQL Driver

- Validation (Spring Boot Starter Validation)

Database Configuration (application.properties):

```
spring.datasource.url=jdbc:mysql://localhost:3306/sim_activation
spring.datasource.username=root
spring.datasource.password=****
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Creating the Entity Class (Customer.java):

```
package com.example.simactivation.entity;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String dob;
    private String address;
    private String idProof;

    // Getters and Setters
}
```

Creating the Repository Interface (CustomerRepository.java):

```
package com.example.simactivation.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.example.simactivation.entity.Customer;

public interface CustomerRepository extends JpaRepository<Customer, Long> {
    Customer findByEmailAndDob(String email, String dob);
    Customer findByFirstNameAndLastNameAndEmail(String firstName, String lastName, String email);
}
```

Creating the REST Controller (CustomerController.java):

```
package com.example.simactivation.controller;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
```

```
import java.util.List;

import com.example.simactivation.entity.Customer;
import com.example.simactivation.repository.CustomerRepository;

@RestController
@RequestMapping("/customers")
public class CustomerController {

    @Autowired
    private CustomerRepository repo;

    // Create customer
    @PostMapping
    public Customer addCustomer(@RequestBody Customer customer) {
        return repo.save(customer);
    }

    // Get all customers
    @GetMapping
    public List<Customer> getAllCustomers() {
        return repo.findAll();
    }

    // Validate customer by email and DOB
    @GetMapping("/validate/emailDob")
    public Customer validateByEmailDob(@RequestParam String email, @RequestParam String dob) {
        return repo.findByEmailAndDob(email, dob);
    }

    // Validate customer by name and email
    @GetMapping("/validate/nameEmail")
    public Customer validateByNameEmail(@RequestParam String firstName, @RequestParam String lastName,
@RequestParam String email) {
        return repo.findByFirstNameAndLastNameAndEmail(firstName, lastName, email);
    }

    // Update address
    @PutMapping("/{id}/address")
    public Customer updateAddress(@PathVariable Long id, @RequestParam String address) {
        Customer customer = repo.findById(id).orElseThrow(() -> new RuntimeException("Customer not found"));
        customer.setAddress(address);
        return repo.save(customer);
    }
}
```

Running the Application:

Run the Spring Boot project.

Use Postman or SoapUI to perform POST, GET, PUT requests for SIM activation, validation, and updates:

POST /customers → Add new customer
GET /customers → View all customers

GET /customers/validate/emailDob?email=&dob= → Validate by email & DOB
GET /customers/validate/nameEmail?firstName=&lastName=&email= → Validate by name & email
PUT /customers/{id}/address?address= → Update address

Output:

Example GET /customers response:


```
[
  {"id":1,"firstName":"Anitha","lastName":"R","email":"anitha@example.com","dob":"2000-01-
15","address":"Chennai","idProof":"Aadhar"},
  {"id":2,"firstName":"Bala","lastName":"S","email":"bala@example.com","dob":"1995-05-
20","address":"Bangalore","idProof":"PAN"}
]
```


Example GET /customers/validate/emailDob response:


```
{"id":1,"firstName":"Anitha","lastName":"R","email":"anitha@example.com","dob":"2000-01-
15","address":"Chennai","idProof":"Aadhar"}
```


Result:
The SIM Activation Portal was successfully implemented using Spring REST and Spring Data JPA.
The application automates SIM validation, customer verification, and address updates, demonstrating real-world telecom domain functionality.This experiment highlights practical application of REST APIs, data validation, and database operations in a telecom domain scenari