R.M.K. COLLEGE OF ENGINEERING AND TECHNOLOGY

(An Autonomous Institution)

R.S.M. Nagar, Puduvoyal -601 206

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (CYBERSECURITY)

22CS938

REST APPLICATION DEVELOPMENT USING SPRING BOOT AND JPA

(Lab Integrated)

LAB MANUAL

ACADEMIC YEAR: 2025-2026

ODD SEMESTER

B.E. COMPUTER SCIENCE AND ENGINEERING (CYBERSRCURITY)

R.M.K. COLLEGE OF ENGINEERING AND TECHNOLOGY

R.S.M. Nagar, Puduvoyal -601 206

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

22CS938

-

REST APPLICATION DEVELOPMENT USING SPRING BOOT AND JPA

(Lab Integrated)

LAB MANUAL

2025-2026 ODD SEMESTER

B.E. COMPUTER SCIENCE AND ENGINEERING (CYBERSECURITY)

Prepared by	Approved by	Approved by
Ms. J. Monisha,	**Dr. S. M. Udhaya Sankar**	**Dr. N. Suresh Kumar**
Assistant Professor / CSE(CS)	Professor & Head / CSE(CS)	Principal

R.M.K. COLLEGE OF ENGINEERING AND TECHNOLOGY

R.S.M. Nagar, Puduvoyal -601 206

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (CYBERSECURITY)

Vision

To excel and take the lead in Cyber Security education, profession and research globally with a commitment to effectively address societal needs.

Mission

- To collaborate with innovators to provide real-world, standards-based Cyber Security capabilities that address business needs
- To prepare the professionals in both academic and industrial settings capable of solving real-world Cyber Security threats
- To inculcate in students the knowledge of designing and developing various projects in different areas of Cyber Security by providing a distinguished and high-quality education.

Programme Educational Objectives

PEO 1: Acquire the knowledge, skills and attitude necessary for effective Cyber Security analysis.

PEO 2: Apply the cutting-edge latest technology within a professional, legal and ethical framework to operate effectively in a multidisciplinary stream.

PEO 3: Practise continued self-learning to keep their knowledge and skills up to date and to remain abreast of the latest developments in Cyber Security.

Programme Specific Outcome

The Computer Science and Engineering (Cyber Security) Graduates should be able to

a) Understand, analyze, design, and develop computing solutions by applying algorithms, web design, database management, and networking concepts in the field of Cyber Security.

b) Develop Cyber Security skills including network defense, ethical hacking, penetration testing, application security and cryptography to provide real time solutions.

c) Apply standard tools, practices and strategies in Cyber Security for successful career and entrepreneurship.

R.M.K. COLLEGE OF ENGINEERING AND TECHNOLOGY

(An Autonomous Institution)

R.S.M. Nagar, Puduvoyal -601 206

PROGRAM OUTCOMES (POs)

Engineering Graduates will be able to:

1. **Engineering Knowledge:** Apply the knowledge of Mathematics, Science, Engineering Fundamentals, and an Engineering Specialization to the solution of complex engineering problems.
2. **Problem Analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of Mathematics, Natural Sciences, and Engineering Sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern Tool Usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.
6. **The Engineer and Society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and Sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and Team Work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project Management and Finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long Learning:** Recognize the need for and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Syllabus

22IT910	Rest Application Development Using Spring Boot and JPA	L	T	P	C
		2	0	2	3

OBJECTIVES:

The Course will enable learners to:

- Provide comprehensive knowledge of RESTful APIs and the HTTP methods used in the Spring Boot framework.
- Cover advanced querying techniques using JPA, including LIKE queries, and to manage CRUD operations using JPQL.
- Explore various relational mappings in JPA, such as one-to-one and one-to-many associations, and their practical implementations.
- Implement and manage Spring AOP applications using annotation-based configurations for method interception and post-execution operations.
- Build production-grade Spring Boot applications with integrated security using JWT, detailed API documentation with SwaggerUI and OpenUI, and effective logging practices.

UNIT I	Introduction to REST API	9 +6

RESTful APIs – overview about data exchange between client and server - separating concerns between handling HTTP requests and executing business logic - retrieving server resources via HTTP requests - injection of property values - self-contained application - serialization and deserialization – JSON properties - managing data access.

List of Exercises/Experiments:

1. Develop a RESTful API for retrieving a welcome message, emphasizing the basics of data exchange between client and server.
2. Implement a RESTful API to acknowledge the user's favorite color choice, highlighting property value injection principles.
3. Create a Spring Boot application that retrieves and displays application information, demonstrating the usage of the @Value annotation to inject property values from the application configuration file.
4. Construct a RESTful API for student details retrieval, illustrating the utilization of @JsonIgnore annotation, focusing on advanced JSON property handling and data access control.

UNIT II	Advanced Data Management With Java and Mysql	9 +6

Build production-grade applications – MYSQL - mapping Java classes to relational database - repository interface - data access operations – retrieving data from the database –mapping of request body to entity - retrieve an entity - capture data from API requests - building complex queries using keywords.

List of Exercises/Experiments:

1. Develop a web application for managing patient details using RESTful APIs, implementing POST and GET operations.
2. Create a web application for managing product details using RESTful APIs, enabling POST and GET operations.
3. Build an application for managing employee details using RESTful APIs, supporting POST, PUT, and DELETE operations

UNIT III	Advanced JPA Queries and Annotations	9 +6

Pagination & Sorting using JPA, @Transient Annotation, Queries using JPA, Starts and Ends with query using JPA, JPQL with @Query Annotation, custom JPQL queries.

List of Exercises/Experiments:

1. Develop a web application for pagination and sorting of children details using RESTful APIs, implementing POST and GET operations.
2. Create a web application for managing Person details using JPA methods via RESTful APIs, enabling POST and GET operations.
3. Retrieve person details using JPQL with conditions for names starting or ending with specific patterns.
4. Build a web application for managing Person details using custom JPQL queries via RESTful APIs, supporting POST and GET operations.

UNIT IV	JPA Associations and Mapping	9+6

JPA Mapping of One-to-One Associations - fetching entities using queries – Loading optimization technique - Two-way One-to-One Relationship Mapping with JPA - single entity instance associated with multiple instances - Adding Data with One-to-One and One-to-Many Associations using JPA.

List of Exercises/Experiments:

1. Develop a Spring Boot application with "Person" and "Address" entities, where each person has exactly one address. Utilize Spring JPA to establish a one-to-one mapping between these entities.
2. Create a Spring Boot application with "Author" and "Book" entities, where each author can have multiple books, and each book belongs to only one author. Use Spring JPA to establish a one-to-many bidirectional mapping between these entities.
3. Build a Spring Boot application with "Employee" and "Address" entities, ensuring that each employee has exactly one address, and each address belongs to only one employee. Establish a one-to-one mapping between these entities using Spring JPA and utilize the Criteria API to retrieve employee details efficiently

UNIT V	Spring Boot Essentials: API Security, Logging, AOP, and Build Management	9+6

SwaggerUI with Spring Boot, OpenUI with Spring Boot, Logging with Spring Boot, Changing Log Level, Logging Request and Response- Managing Spring Boot Logging Configuration - Aspect-Oriented Programming (AOP) Concepts – Method Parameter Handling - Post- Execution Operations - Returning Data Handling - Comprehensive Advice Handling. API security using JWT, Gradle for build management, Sonar Lint for coding standards and guidelines.

List of Exercises/Experiments:

1. Develop a web application for managing Employee and Payroll details via RESTful APIs. Utilize Spring JPA to establish a one-to-one mapping between Employee and Payroll entities. Demonstrate the usage of Swagger for API documentation and interaction.
2. Develop a Spring Boot application focused on handling person details and integrate comprehensive logging capabilities to track application activities effectively.
3. Explore the implementation of Aspect-Oriented Programming (AOP) in a Spring application to enhance the behavior of a service method and demonstrate its impact on application functionality

Total: 30+30=60 Periods

OUTCOMES:

Upon completion of the course, the students will be able to:

CO1: Create simple applications using RESTful APIs and effectively manage HTTP methods within the Spring Boot framework.

CO2: Apply database connectivity with JPA, utilizing advanced queries to interact with the database.

CO3: Build applications using Spring Boot and perform CRUD operations efficiently using JPQL

CO4: Demonstrate the implementation of various relational mappings in JPA, including one- to-one and one-to-many associations.

CO5: Develop real-time applications that integrate user interfaces and utilize Spring AOP for method interception and advice handling.

CO6: Apply security measures to REST APIs using Spring Security and JWT to protect sensitive data and ensure secure communication between clients and servers.

TEXTBOOKS:

1. Raja CSP Raman, Ludovic Dewailly, “Building RESTful Web Services with Spring 5 - Second Edition: Leverage the power of Spring 5.0, Java SE 9, and Spring Boot 2.0”, Packt Publishing, 2018
2. Leonard Richardson, Sam Ruby “RESTful Web Services” O'Reilly Media, 2008.
3. Ludovic Dewailly, “Building a RESTful Web Service with Spring: A hands-on guide to building an enterprise-grade, scalable RESTful web service using the Spring Framework”, Packt Publishing, 2015

REFERENCES:

1. Ranga Karanam, “Master Java Web Services and REST API with Spring Boot”, Packt Publishing, 2018.
2. Balaji Varanasi, Sudha Belida, “Spring REST”, Apress, 2015.
3. Greg L. Turnquist, “Learning Spring Boot 2.0”, Packt Publishing, 2021
4. Sourabh Sharma, “Modern API Development with Spring and Spring Boot”, Packt Publishing, 2021.

S. No.	List of Exercises
1.	Develop a RESTful API for retrieving a welcome message emphasizing the basics of data exchange between client and server.
2.	Implement a RESTful API to acknowledge the user's favorite color choice highlighting property value injection principles.
3.	Create a Spring Boot application that retrieves and displays application information, demonstrating the usage of the @Value annotation to inject property values from the application configuration file
4.	Construct a RESTful API for student details retrieval, illustrating the utilization of @JsonIgnore annotation, focusing on advanced JSON property handling and data access control.
5.	Develop a web application for managing patient details using RESTful APIs, implementing POST and GET operations.
6.	Create a web application for managing product details using RESTful APIs, enabling POST and GET operations.
7.	Build an application for managing employee details using RESTful APIs, supporting POST, PUT, and DELETE operations
8.	Develop a web application for pagination and sorting of children details using RESTful APIs, implementing POST and GET operations.
9.	Create a web application for managing Person details using JPA methods via RESTful APIs, enabling POST and GET operations.
10.	Retrieve person details using JPQL with conditions for names starting or ending with specific patterns.
11.	Build a web application for managing Person details using custom JPQL queries via RESTful APIs, supporting POST and GET operations.
12.	Develop a Spring Boot application with "Person" and "Address" entities, where each person has exactly one address. Utilize Spring JPA to establish a one
13.	Create a Spring Boot application with "Author" and "Book" entities, where each author can have multiple books, and each book belongs to only one author. Use Spring JPA to establish a one
14.	Build a Spring Boot application with "Employee" and "Address" entities, ensuring that each employee has exactly one address, and each address belongs to only one employee. Establish a one
15.	Develop a web application for managing Employee and Payroll details via RESTful APIs. Utilize Spring JPA to establish a one-to-one mapping between Employee and Payroll entities. Demonstrate the usage of Swagger for API documentation and interaction.
16.	Develop a Spring Boot application focused on handling person details and integrate comprehensive logging capabilities to track application activities effectively
17.	Explore the implementation of Aspect-Oriented Programming (AOP) in a Spring application to enhance the behavior of a service method and demonstrate its impact on application functionality

Date :
Exercise Number 1
Title: Welcome Message Exchange

Aim:

To Develop a RESTful API for retrieving a welcome message, emphasizing the basics of data exchange between client and server

Algorithm:

Step 1: Set up Spring Boot Project

- Initialize a Spring Boot project with the necessary dependencies, such as Spring Web, using Spring Initializer or your preferred development environment.

Step 2: Create the WelcomeController Class

- In the controller package, create a new Java class named WelcomeController.
- Use the @RestController annotation to define the class as a RESTful controller that will handle HTTP requests.

Step 3: Define a GET Mapping

- Inside the WelcomeController class, define a method welcome() that returns a simple welcome message.
- Annotate the method with @GetMapping("/welcome") to map it to the /welcome endpoint.

Step 4: Handle Client Request

- When a client sends a GET request to the /welcome endpoint, the welcome() method is executed.
- The method returns a string message, which is sent back to the client as the response.

Step 5: Build and Run the Application

Pseudo Code:

```
BEGIN

// Step 1: Initialize Spring Boot Project
INITIALIZE Spring Boot Project
INCLUDE necessary dependencies (e.g., Spring Web)

// Step 2: Create Controller Class
DEFINE class WelcomeController
ANNOTATE WelcomeController with @RestController to define it as a REST API controller

   // Step 3: Define a GET endpoint
   DEFINE METHOD welcome()
   ANNOTATE welcome() with @GetMapping("/welcome")
      RETURN "Welcome Spring Boot!" as a String response

// Step 4: Handle Client Request
WHEN client sends GET request to /welcome endpoint
   CALL welcome() method
   RETURN the message "Welcome Spring Boot!" to client as HTTP response
```

// Step 5: Build and Run the Application
BUILD the project
RUN the application on server (e.g., localhost)
ACCESS endpoint http://localhost:<port>/welcome in browser or API tool

END

Output:
Welcome Spring Boot!

Result:

Thus to Develop a RESTful API for retrieving a welcome message, emphasizing the basics of data exchange between client and server was successfully Completed.

Date :
Exercise Number 2
Title: Favourite Colour Choice

Aim:

To Implement a RESTful API to acknowledge the user's favourite colour choice, highlighting property value injection principles

Algorithm:

Step 1: Set up Spring Boot Project

- ❖ Initialize a Spring Boot project with necessary dependencies (Spring Web).
- ❖ Create the basic structure with folders such as controller, service, and model.

Step 2: Define the Controller

- ❖ Create a controller class ApiController in the controller package.
- ❖ Use @RestController annotation to designate this class as a REST controller.

Step 3: Handle Client Request

- ❖ Define a GET API method that takes a query parameter using @RequestParam.
- ❖ Use the parameter to generate a personalized response string.

Step 4: Test the API

- ❖ Build and run the Spring Boot application.
- ❖ Send a GET request to the endpoint /favouriteColor with a color query parameter to test the API.

Pseudo Code:

```
BEGIN

// Step 1: Initialize Spring Boot Project
INITIALIZE Spring Boot Project
ADD necessary dependency: Spring Web

CREATE basic folder structure:
  - controller
  - service (optional)
  - model (optional)

// Step 2: Define the Controller
DEFINE class ApiController in controller package
```

ANNOTATE ApiController with @RestController to make it a REST API controller

// Step 3: Handle Client Request
DEFINE METHOD welcome(color)
ANNOTATE welcome() with @GetMapping("/favouriteColor")
ACCEPT query parameter "color" using @RequestParam
RETURN string: "My favorite color is " + color + "!"

// Step 4: Test the API
BUILD the project
RUN the Spring Boot application on server (e.g., localhost)
SEND GET request to endpoint: /favouriteColor?color=blue
=> RESPONSE: "My favorite color is blue!"

END

Output:

My favorite color is blue!

Result:

Thus to Implement a RESTful API to acknowledge the user's favorite color choice, highlighting property value injection principles were successfully completed.

Date :
Exercise Number 3
Title: Usuage of @Value annotation

Aim:

To Create a Spring Boot application that retrieves and displays application information, demonstrating the usage of the @Value annotation to inject property values from the application configuration file

Algorithm:

Step 1: Set Up Spring Boot Project

- Create a Spring Boot project using Spring Initializer or IDE, and include the necessary dependencies (Spring Web).
- Create the necessary folder structure, such as controller, config, and resources.

Step 2: Define application.properties File

- In the resources directory, create an application.properties file.
- Add the properties for the app name and app version in the file:

Step 3: Create AppConfig.java Class

- Inside the config package, create a class AppConfig.java.
- Use the @Configuration annotation to mark the class as a Spring configuration class.
- Use @PropertySource to load the application.properties file.
- Inject the property values using the @Value annotation to map them to fields in the class.
- Define getters and setters for these properties to allow access to the injected values.

step 4: Create ApiController.java Class

- In the controller package, create a class ApiController.java.
- Use the @RestController annotation to define the class as a REST controller.
- Create a GET method using the @GetMapping annotation to map a URL endpoint (/info).
- In the method, instantiate the AppConfig class, retrieve the application name and version, and return this information as a response.

Step 5: Build and Run the Application

Pseudo Code:

```
BEGIN

// Step 1: Initialize Spring Boot Project
INITIALIZE Spring Boot Project
ADD dependency: Spring Web
CREATE folders: config, controller, resources

// Step 2: Define Properties File
CREATE file: application.properties in resources
SET properties:
   app.name = MySpringApp
   app.version = 1.0

// Step 3: Create Configuration Class
DEFINE class AppConfig in config package
ANNOTATE class with @Configuration
LOAD properties using @PropertySource("classpath:application.properties")

DECLARE private field appName
INJECT value using @Value("${app.name}")

DECLARE private field appVersion
INJECT value using @Value("${app.version}")

CREATE getter and setter methods:
   getAppName(), setAppName()
   getAppVersion(), setAppVersion()

// Step 4: Create REST Controller
DEFINE class ApiController in controller package
ANNOTATE with @RestController

DEFINE method get()
ANNOTATE with @GetMapping("/info")

INSTANTIATE AppConfig ( ⚠️ Note: This is incorrect in real Spring, see note below)
RETURN string: "App Name: " + appName + ", Version: " + appVersion

// Step 5: Build and Run the Application
BUILD the application
RUN server on localhost
SEND GET request to: /info
RECEIVE response: App Name: MySpringApp, Version: 1.0

END
```

Output:

App Name: MyApp, Version: 1.0.0

Result:

Thus, to Create a Spring Boot application that retrieves and displays application information, demonstrating the usage of the @Value annotation to inject property values from the application configuration file was successfully completed.

Date :
Exercise Number 4
Title: Usage of @JsonIgnore annotation

Aim:

To Construct a RESTful API for student details retrieval, illustrating the utilization of @JsonIgnore annotation, focusing on advanced JSON property handling and data access control.

Algorithm:

Step 1: Set Up Spring Boot Project

- Initialize a Spring Boot project with dependencies (Spring Web, Jackson for JSON handling).
- Create the basic structure with necessary folders such as controller and model.

Step 2: Create the Model Class

- Define a Student class in the model package.
- Include attributes such as id, name, and description.
- Use the @JsonIgnore annotation to exclude the description property from JSON serialization.

Step 3: Implement the Controller

- Create a StudentController class in the controller package.
- Define a GET endpoint /student that returns a Student object.
- Use the Student model to provide the response, which will include id and name, but exclude description.

Step 4: Run and Test the API

- Build and run the Spring Boot application.
- Send a GET request to the endpoint /student to verify the API response.

Pseudo Code:

```
BEGIN

// Step 1: Set Up Spring Boot Project
INITIALIZE Spring Boot Project
INCLUDE dependencies: Spring Web, Jackson (for JSON handling)
CREATE folder structure:
  - model
  - controller
```

```
// Step 2: Create the Student Model
DEFINE class Student in model package

DECLARE private fields:
   - id (Long)
   - name (String)
   - description (String)

ANNOTATE description field with @JsonIgnore to exclude it from JSON response

DEFINE constructor:
   Student(id, name, description)

DEFINE getter and setter methods for:
   - getId(), setId()
   - getName(), setName()
   - getDescription(), setDescription()

// Step 3: Create the Controller
DEFINE class StudentController in controller package
ANNOTATE with @RestController

DEFINE method get()
ANNOTATE with @GetMapping("/student")

CREATE a Student object with sample data:
   id = 1
   name = "John Doe"
   description = "This is a student description"

RETURN the Student object

// Step 4: Run and Test the API
BUILD the project
RUN the Spring Boot application
SEND GET request to endpoint: /student

EXPECTED RESPONSE:
{
   "id": 1,
   "name": "John Doe"
}
// Note: "description" is excluded due to @JsonIgnore
```

END

Output:


```
{
    "id": 1,
    "name": "John Doe"
}
```


Result:

Thus, to Construct a RESTful API for student details retrieval, illustrating the utilization of @JsonIgnore annotation, focusing on advanced JSON property handling and data access control was successfully completed.

Date :

Exercise Number 5

Title: Implementation of POST and GET Operations

Aim:

To Develop a web application for managing patient details using RESTful APIs, implementing POST and GET operations.

Algorithm:

1. Set up a Spring Boot project with necessary dependencies for RESTful APIs.
2. Create a Patient model to represent patient details.
3. Create a PatientController to handle POST and GET requests.
4. Implement service layer logic for handling patient data.
5. Implement data persistence using JPA and a relational database (e.g., MySQL).
6. Set up validation for input data using Spring Validation.
7. Create view templates (optional) for displaying patient details.
8. Test the application by making API calls.

Pseudo Code:

1. Initialize Spring Boot Project

BEGIN

INITIALIZE Spring Boot project

ADD dependencies:

- Spring Web
- Spring Data JPA
- MySQL Driver
- Spring Validation

CREATE folder structure:

- model
- repository
- service
- controller
- resources

---

2. Define Patient Model

// Create Patient class in model package

ANNOTATE with @Entity

DECLARE fields:

- id (auto-generated)
- name (must not be blank)
- email (must be valid and not blank)
- phone (must not be blank)

ADD annotations:
- @Id, @GeneratedValue
- @NotBlank, @Email for validation

DEFINE getters and setters

---

3. Define Repository Interface
// Create PatientRepository in repository package
EXTEND JpaRepository<Patient, Long>

---

4. Create Service Layer
// Create PatientService in service package
ANNOTATE with @Service

AUTOWIRE PatientRepository

DEFINE method savePatient(patient)
CALL patientRepository.save(patient)
RETURN saved patient

DEFINE method getAllPatients()
RETURN patientRepository.findAll()

---

5. Create Controller
// Create PatientController in controller package
ANNOTATE with @RestController
SET base path using @RequestMapping("/patients")

AUTOWIRE PatientService

DEFINE POST method createPatient(patient)
ANNOTATE with @PostMapping
VALIDATE input using @Valid
RETURN saved patient with HTTP 200 OK

DEFINE GET method getAllPatients()
ANNOTATE with @GetMapping
RETURN list of all patients with HTTP 200 OK

---

6. Create Main Application Class
// Create Application.java

ANNOTATE with @SpringBootApplication

DEFINE main method:
  CALL SpringApplication.run(Application.class)

---

7. Configure application.properties
SET spring.datasource.url = jdbc:mysql://localhost:3306/patient_db
SET spring.datasource.username = root
SET spring.datasource.password = yourpassword
SET spring.jpa.hibernate.ddl-auto = update
SET spring.jpa.show-sql = true

---

8. Run and Test Application
BUILD and RUN Spring Boot Application
// Test POST request
SEND POST request to: http://localhost:8080/patients
  BODY: {
    "name": "John Doe",
    "email": "john.doe@example.com",
    "phone": "1234567890"
  }
EXPECTED RESPONSE:
  Status: 200 OK
  Body: {
    "id": 1,
    "name": "John Doe",
    "email": "john.doe@example.com",
    "phone": "1234567890"
  }

// Test GET request
SEND GET request to: http://localhost:8080/patients
EXPECTED RESPONSE:
  Status: 200 OK
  Body: [
    {
      "id": 1,
      "name": "John Doe",
      "email": "john.doe@example.com",
      "phone": "1234567890"
    }
  ]

Sample Input/Output:

Input 1: (POST Request)

URL: http://localhost:8080/patients Request Body:

```
{
   "name": "John Doe",
   "email": "john.doe@example.com", "phone":
   "1234567890"
}
```

Output 1: (Successful Registration)

Response Status: 200 OK Response Body:

```
{
   "id": 1,
   "name": "John Doe",
   "email": "john.doe@example.com",
   "phone": "1234567890"
}
```

Input 2: (GET Request)

URL: http://localhost:8080/patients **Output 2: (List of Patients)** Response Status: 200 OK

Response Body: [

```
   {
      "id": 1,
      "name": "John Doe",
      "email": "john.doe@example.com", "phone":
      "1234567890"
   }
]
```

Result:

Thus to Develop a web application for managing patient details using RESTful APIs, implementing POST and GET operations were successfully completed.

Date :

Exercise Number 6

Title: Enabling POST and GET Operations

Aim:

To Create a web application for managing product details using RESTful APIs, enabling POST and GET operations.

Algorithm:

1. Set up a Spring Boot project.
2. Create a Product model to represent product details.
3. Create a ProductController to handle POST and GET requests.
4. Implement a service layer to manage business logic.
5. Implement data persistence using JPA and a database (e.g., MySQL).
6. Set up validation for input data using Spring Validation.
7. Test the application by making API requests using Postman or a similar tool.

Pseudo Code:

1. Initialize Spring Boot Project

BEGIN

INITIALIZE Spring Boot project

ADD dependencies:

- Spring Web
- Spring Data JPA
- MySQL Driver
- Spring Validation

CREATE folders:

- model
- repository
- service
- controller
- resources

2. Define Product Model

// Define class Product in model package

ANNOTATE with @Entity

DECLARE private fields:

- id (Long) → @Id and @GeneratedValue
- name (String) → @NotBlank
- price (Double) → @NotNull
- description (String) → @NotBlank

CREATE constructor (optional)

DEFINE getter and setter methods for all fields

3. Create Product Repository

// Define interface ProductRepository

EXTEND JpaRepository<Product, Long>

4. Implement Service Layer

// Define class ProductService in service package

ANNOTATE with @Service

AUTOWIRE ProductRepository

DEFINE method saveProduct(product)

CALL productRepository.save(product)

RETURN saved product

DEFINE method getAllProducts()

RETURN productRepository.findAll()

5. Implement Controller

// Define class ProductController in controller package

ANNOTATE with @RestController

SET base path using @RequestMapping("/products")

AUTOWIRE ProductService

DEFINE method createProduct(product)

ANNOTATE with @PostMapping

VALIDATE input with @Valid and @RequestBody

```
    CALL productService.saveProduct()
    RETURN ResponseEntity with saved product and status 200 OK

DEFINE method getAllProducts()
    ANNOTATE with @GetMapping
    CALL productService.getAllProducts()
    RETURN ResponseEntity with product list and status 200 OK
```

6. Configure application.properties

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

7. Run and Test Application

```
// BUILD and RUN the Spring Boot Application
// Test POST Request
SEND POST request to: http://localhost:8080/products
REQUEST BODY:
{
    "name": "Laptop",
    "price": 1200.99,
    "description": "High-end gaming laptop"
}
EXPECTED RESPONSE:
{
    "id": 1,
    "name": "Laptop",
    "price": 1200.99,
    "description": "High-end gaming laptop"
}

// Test GET Request
```

SEND GET request to: http://localhost:8080/products

EXPECTED RESPONSE:

```
[
  {
    "id": 1,
    "name": "Laptop",
    "price": 1200.99,
    "description": "High-end gaming laptop"
  }
]
```

END

Sample Input/Output:

Input 1: (POST Request)
URL: http://localhost:8080/products Request Body:

```
{
  "name": "Laptop",
  "price": 1200.99,
  "description": "High-end gaming laptop"
}
```

Output 1: (Successful Product Creation)
Response Status: 200 OK Response Body:

```
{
  "id": 1,
  "name": "Laptop",
  "price": 1200.99,
  "description": "High-end gaming laptop"
}
```

Input 2: (GET Request)
URL: http://localhost:8080/products **Output 2: (List of Products)** Response Status: 200 OK
Response Body:

```
[
    {
        "id": 1,
        "name": "Laptop",
        "price": 1200.99,
        "description": "High-end gaming laptop"
    }
]
```


Result:

Thus to Create a web application for managing product details using RESTful APIs, enabling POST and GET operations were successfully completed.

Date :

Exercise Number 7

Title: Supporting POST, PUT and DELATE Operations

Aim:

To Build an application for managing employee details using RESTful APIs, supporting POST, PUT, and DELETE operations

Algorithm:

1. Set up a Spring Boot project.
2. Create an Employee model to represent employee details.
3. Create a EmployeeController to handle POST, PUT, and DELETE requests.
4. Implement a service layer to manage business logic.
5. Implement data persistence using JPA and a database (e.g., MySQL).
6. Set up validation for input data using Spring Validation.
7. Test the application using API requests with tools like Postman.

Pseudo Code:

1. Initialize Spring Boot Project

BEGIN

INITIALIZE Spring Boot project

ADD dependencies:

- Spring Web
- Spring Data JPA
- MySQL Driver
- Spring Validation

CREATE folder structure:

- model
- repository
- service
- controller
- resources

2. Define Employee Model

// Define class Employee in model package

ANNOTATE with @Entity

DECLARE private fields:

- id (Long) → @Id, @GeneratedValue
- name (String) → @NotBlank
- email (String) → @Email, @NotBlank
- department (String) → @NotBlank

DEFINE getters and setters for all fields

3. Create Repository Interface

// Define interface EmployeeRepository

EXTEND JpaRepository<Employee, Long>

4. Create Service Layer

// Define class EmployeeService in service package

ANNOTATE with @Service

AUTOWIRE EmployeeRepository

// Save employee

DEFINE method saveEmployee(employee)

RETURN employeeRepository.save(employee)

// Update employee

DEFINE method updateEmployee(id, updatedEmployee)

FIND employee by id using repository

IF not found → THROW exception

UPDATE fields (name, email, department)

RETURN updated employee via save()

// Delete employee

DEFINE method deleteEmployee(id)

CALL employeeRepository.deleteById(id)

```
// Get all employees
DEFINE method getAllEmployees()
    RETURN employeeRepository.findAll()
```

5. Create Controller

```
// Define class EmployeeController in controller package
ANNOTATE with @RestController
SET base path using @RequestMapping("/employees")

AUTOWIRE EmployeeService

// POST: Create employee
DEFINE method createEmployee(@RequestBody employee)
    VALIDATE with @Valid
    CALL employeeService.saveEmployee()
    RETURN ResponseEntity with saved employee (HTTP 200 OK)

// PUT: Update employee
DEFINE method updateEmployee(@PathVariable id, @RequestBody employee)
    VALIDATE with @Valid
    CALL employeeService.updateEmployee(id, employee)
    RETURN ResponseEntity with updated employee (HTTP 200 OK)

// DELETE: Delete employee
DEFINE method deleteEmployee(@PathVariable id)
    CALL employeeService.deleteEmployee(id)
    RETURN ResponseEntity with no content (HTTP 204 No Content)

// GET: Get all employees
DEFINE method getAllEmployees()
    CALL employeeService.getAllEmployees()
```

RETURN ResponseEntity with employee list (HTTP 200 OK)

6. Configure application.properties

SET spring.datasource.url = jdbc:mysql://localhost:3306/employee_db

SET spring.datasource.username = root

SET spring.datasource.password = yourpassword

SET spring.jpa.hibernate.ddl-auto = update

SET spring.jpa.show-sql = true

7. Run and Test Application

// BUILD and RUN the application

// Test POST request

SEND POST request to http://localhost:8080/employees

REQUEST BODY:

```
{
   "name": "Alice Johnson",
   "email": "alice.johnson@example.com",
   "department": "IT"
}
```

EXPECTED RESPONSE:

Status: 200 OK

Body:

```
{
   "id": 1,
   "name": "Alice Johnson",
   "email": "alice.johnson@example.com",
   "department": "IT"
}
```

// Test PUT request (update)

SEND PUT request to http://localhost:8080/employees/1

REQUEST BODY:

```
{
   "name": "Alice Johnson",
   "email": "alice.johnson@newdomain.com",
   "department": "HR"
}
EXPECTED RESPONSE:
Status: 200 OK
Body:
{
   "id": 1,
   "name": "Alice Johnson",
   "email": "alice.johnson@newdomain.com",
   "department": "HR"
}

// Test DELETE request
SEND DELETE request to http://localhost:8080/employees/1
EXPECTED RESPONSE:
Status: 204 No Content
Body: (empty)

// Test GET request
SEND GET request to http://localhost:8080/employees
EXPECTED RESPONSE:
Status: 200 OK
Body: [ List of Employee Objects ]

END
```

Sample Input/Output:

Input 1: (POST Request)

URL: http://localhost:8080/employees Request Body:

```
{
```

```
    "name": "Alice Johnson",
    "email": "alice.johnson@example.com", "department": "IT"
}
```

Output 1: (Successful Employee Creation)

Response Status: 200 OK Response Body:

```
{
    "id": 1,
    "name": "Alice Johnson",
    "email": "alice.johnson@example.com", "department": "IT"
}
```

Input 2: (PUT Request to update employee details)

URL: http://localhost:8080/employees/1 Request Body:

json

Copy code

```
{
    "name": "Alice Johnson",
    "email": "alice.johnson@newdomain.com", "department": "HR"
}
```

Output 2: (Successful Employee Update)

Response Status: 200 OK Response Body:

```
{
    "id": 1,
    "name": "Alice Johnson",
    "email": "alice.johnson@newdomain.com", "department": "HR"
}
```

Input 3: (DELETE Request)

URL: http://localhost:8080/employees/1 **Output 3:**
(Successful Employee Deletion) Response Status: 204 No
Content

Result:

Thus, to Build an application for managing employee details using RESTful APIs, supporting POST, PUT, and DELETE operations were successfully completed.

Date :
Exercise Number 8
Title: Pagination and Sorting

Aim:

To develop a web application for pagination and sorting of children details using RESTful APIs, implementing POST and GET operations.

Algorithm:

1. Set up a Spring Boot project with MySQL and JPA dependencies.
2. Create an entity class for `Child` that maps to a database table.
3. Implement sorting and pagination using Spring Data JPA's `Pageable` feature.
4. Create a service layer to handle pagination and sorting logic.
5. Create a controller layer for handling the API requests and responses.
6. Test the application using Postman.

Project Structure:

```
src/main/java/com/example/childrenapi/
├── controller
│   └── ChildController.java
├── model
│   └── Child.java
├── repository
│   └── ChildRepository.java
├── service
│   └── ChildService.java
└── ChildrenApiApplication.java
```

Pseudo Code:

1. Initialize Spring Boot Project

```
BEGIN

INITIALIZE Spring Boot project
ADD dependencies:
   - Spring Web
   - Spring Data JPA
   - MySQL Driver

CREATE packages:
   - model
   - repository
   - service
   - controller
```

2. Create Model Class - Child.java

// Define Child class in model package
ANNOTATE with @Entity

DECLARE fields:
- id (Long) → @Id, @GeneratedValue
- name (String)
- age (int)

DEFINE default constructor and parameterized constructor

DEFINE getter and setter methods for:
- id
- name
- age

---

3. Create Repository Interface - ChildRepository.java

// Define interface ChildRepository
EXTEND JpaRepository<Child, Long>
ANNOTATE with @Repository

---

4. Create Service Layer - ChildService.java

// Define class ChildService
ANNOTATE with @Service

AUTOWIRE ChildRepository

DEFINE method saveChild(child)
  RETURN repository.save(child)

DEFINE method getAllChildren(pageable)
  RETURN repository.findAll(pageable)

---

5. Create Controller - ChildController.java

// Define class ChildController
ANNOTATE with @RestController
SET base path using @RequestMapping("/api/children")

AUTOWIRE ChildService

// POST endpoint to add new child

```
DEFINE method addChild(@RequestBody child)
   CALL service.saveChild(child)
   RETURN saved child

// GET endpoint to retrieve paginated & sorted list of children
DEFINE method getChildren(@RequestParam page, size, sortBy)
   CREATE PageRequest using PageRequest.of(page, size, Sort.by(sortBy))
   CALL service.getAllChildren(pageable)
   RETURN Page<Child> as response
```

---

6. Configure application.properties (not shown but implied)

```
SET spring.datasource.url = jdbc:mysql://localhost:3306/your_db_name
SET spring.datasource.username = your_username
SET spring.datasource.password = your_password
SET spring.jpa.hibernate.ddl-auto = update
SET spring.jpa.show-sql = true
```

---

7. Run and Test Application

```
// BUILD and RUN Spring Boot application

// POST Request to add child
URL: http://localhost:8080/api/children
BODY:
{
   "name": "Aarav",
   "age": 6
}
RESPONSE:
{
   "id": 1,
   "name": "Aarav",
   "age": 6
}

// GET Request for paginated and sorted data
URL: http://localhost:8080/api/children?page=0&size=5&sortBy=name
RESPONSE: Page of sorted children objects by name

END
```

Result:

Thus, to Develop a web application for pagination and sorting of children details using RESTful APIs, implementing POST and GET operations were successfully completed.

Date :
Exercise Number 9
Title:

Aim:

To Create a web application for managing Person details using JPA methods via RESTful APIs, enabling POST and GET operations.

Algorithm:

1. Set up a Spring Boot project with JPA and MySQL dependencies.
2. Create a `Person` entity class that maps to a database table.
3. Create repository interfaces extending `JpaRepository` for CRUD operations.
4. Develop service and controller layers to handle POST and GET operations.
5. Test the application using Postman.

Project Structure:

```
src/main/java/com/example/personapi/
├── controller
│   └── PersonController.java
├── model
│   └── Person.java
├── repository
│   └── PersonRepository.java
├── service
│   └── PersonService.java
└── PersonApiApplication.java
```

Pseudo Code:

```
1. Initialize Spring Boot Project
BEGIN

INITIALIZE Spring Boot project
ADD dependencies:
  - Spring Web
  - Spring Data JPA
  - MySQL Driver

CREATE base package: com.example.personapi
CREATE sub-packages:
  - model
  - repository
  - service
```

- controller

2. Define Entity Class: Person.java

// Define class Person in model package

ANNOTATE with @Entity

DECLARE fields:

- id (Long) → @Id, @GeneratedValue
- firstName (String)
- lastName (String)
- email (String)
- age (int)

DEFINE:

- Default constructor
- Parameterized constructor
- Getters and setters for all fields

3. Create Repository Interface: PersonRepository.java

// Define interface PersonRepository in repository package

EXTEND JpaRepository<Person, Long>

ANNOTATE with @Repository

4. Implement Service Layer: PersonService.java

// Define class PersonService in service package

ANNOTATE with @Service

AUTOWIRE PersonRepository

DEFINE method savePerson(person)

CALL repository.save(person)

RETURN saved person

DEFINE method getAllPersons()

CALL repository.findAll()

RETURN list of persons

5. Create Controller Class: PersonController.java

// Define class PersonController in controller package

ANNOTATE with @RestController

MAP base URL using @RequestMapping("/api/persons")

AUTOWIRE PersonService

```
// POST: Add a new person
DEFINE method addPerson(@RequestBody person)
   CALL service.savePerson(person)
   RETURN saved person

// GET: Retrieve all persons
DEFINE method getAllPersons()
   CALL service.getAllPersons()
   RETURN list of persons
```

6. Configure application.properties (not shown in program but required)

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

7. Run and Test the Application

```
// RUN the Spring Boot Application

// POST request to add a person
URL: http://localhost:8080/api/persons
BODY:
{
   "firstName": "John",
   "lastName": "Doe",
   "email": "john.doe@example.com",
   "age": 30
}
RESPONSE:
{
   "id": 1,
   "firstName": "John",
   "lastName": "Doe",
   "email": "john.doe@example.com",
   "age": 30
}
// GET request to retrieve all persons
URL: http://localhost:8080/api/persons
RESPONSE:
[
   {
      "id": 1,
```

```
    "firstName": "John",
    "lastName": "Doe",
    "email": "john.doe@example.com",
    "age": 30
  },
  ...
]

END
```

Result:

Thus, to Create a web application for managing Person details using JPA methods via RESTful APIs, enabling POST and GET operations were successfully completed.

Date :
Exercise Number 10
Title:

Aim:

To Retrieve person details using JPQL with conditions for names starting or ending with specific patterns.

Algorithm:

1. Set up a Spring Boot project with JPA and MySQL dependencies.
2. Create a `Person` entity class and repository interface.
3. Define custom JPQL queries using the `@Query` annotation to search by name patterns.
4. Develop service and controller layers to handle JPQL queries.
5. Test the application using Postman.

Project Structure:

```
src/main/java/com/example/personjpql/
├── controller
│   └── PersonController.java
├── model
│   └── Person.java
├── repository
│   └── PersonRepository.java
├── service
│   └── PersonService.java
└── PersonJPQLApplication.java
```

Pseudo Code:

```
1. Initialize Spring Boot Project
BEGIN

INITIALIZE Spring Boot project
ADD dependencies:
   - Spring Web
   - Spring Data JPA
   - MySQL Driver

CREATE base package: com.example.personjpql
CREATE sub-packages:
   - model
   - repository
   - service
   - controller

2. Define Entity Class: Person.java
// Define Person class in model package
ANNOTATE with @Entity

DECLARE fields:
```

- id (Long) → @Id, @GeneratedValue
- firstName (String)
- lastName (String)
- email (String)
- age (int)

DEFINE:
- Default constructor
- Parameterized constructor
- Getters and setters

3. Create Repository with JPQL Queries: PersonRepository.java
// Define PersonRepository interface in repository package
EXTEND JpaRepository<Person, Long>

// Define custom JPQL methods with @Query

DEFINE method findByFirstNameStartingWith(prefix)
  ANNOTATE with:
    @Query("SELECT p FROM Person p WHERE p.firstName LIKE :prefix%")
    @Param("prefix")

DEFINE method findByLastNameEndingWith(suffix)
  ANNOTATE with:
    @Query("SELECT p FROM Person p WHERE p.lastName LIKE %:suffix")
    @Param("suffix")

4. Implement Service Layer: PersonService.java
// Define PersonService class in service package
ANNOTATE with @Service

AUTOWIRE PersonRepository

DEFINE method getPersonsByFirstNamePrefix(prefix)
  RETURN repository.findByFirstNameStartingWith(prefix)

DEFINE method getPersonsByLastNameSuffix(suffix)
  RETURN repository.findByLastNameEndingWith(suffix)

---

5. Create Controller Layer: PersonController.java
// Define PersonController class in controller package
ANNOTATE with @RestController
SET base path using @RequestMapping("/api/persons")

AUTOWIRE PersonService

// GET endpoint to search by first name prefix
DEFINE method getPersonsByFirstNamePrefix(@RequestParam prefix)
  RETURN service.getPersonsByFirstNamePrefix(prefix)

// GET endpoint to search by last name suffix
DEFINE method getPersonsByLastNameSuffix(@RequestParam suffix)
  RETURN service.getPersonsByLastNameSuffix(suffix)

6. Configure application.properties (required)
SET spring.datasource.url = jdbc:mysql://localhost:3306/person_db
SET spring.datasource.username = root
SET spring.datasource.password = yourpassword
SET spring.jpa.hibernate.ddl-auto = update
SET spring.jpa.show-sql = true

7. Run and Test the Application
// RUN the Spring Boot application

// TEST GET by first name prefix
URL: http://localhost:8080/api/persons/firstname?prefix=Al
RESPONSE: List of persons whose first names start with "Al"

// TEST GET by last name suffix
URL: http://localhost:8080/api/persons/lastname?suffix=son
RESPONSE: List of persons whose last names end with "son"

END

Result:

Thus, to Retrieve person details using JPQL with conditions for names starting or ending with specific patterns were successfully completed.

Date :
Exercise Number 11
Title:

Aim:

To Build a web application for managing Person details using custom JPQL queries via RESTful APIs, supporting POST and GET operations.

Algorithm:

1. Set up a Spring Boot project with JPA and MySQL dependencies.
2. Create a `Person` entity class and repository interface.
3. Define custom JPQL queries using the `@Query` annotation to handle complex queries.
4. Develop service and controller layers to handle POST and GET requests.
5. Test the application using Postman or another REST client.

Project Structure:

src/main/java/com/example/personcustomjpql/
├── controller
│ └── PersonController.java
├── model
│ └── Person.java
├── repository
│ └── PersonRepository.java
├── service
│ └── PersonService.java
└── PersonCustomJPQLApplication.java

Pseudo Code:

1. Initialize Spring Boot Project
BEGIN

INITIALIZE Spring Boot project
ADD dependencies:
- Spring Web
- Spring Data JPA
- MySQL Driver

CREATE base package: com.example.personcustomjpql
CREATE sub-packages:
- model
- repository
- service
- controller

2. Define Entity Class: Person.java
// Define Person class in model package
ANNOTATE with @Entity

DECLARE fields:

- id (Long) → @Id, @GeneratedValue
- firstName (String)
- lastName (String)
- email (String)
- age (int)

DEFINE:
- Default constructor
- Parameterized constructor
- Getters and Setters for all fields

3. Create Repository Interface with Custom JPQL: PersonRepository.java
// Define PersonRepository interface in repository package
EXTEND JpaRepository<Person, Long>
ANNOTATE with @Repository

DEFINE custom JPQL methods using @Query:

// Search persons whose first name starts with a given prefix
@Query("SELECT p FROM Person p WHERE p.firstName LIKE :prefix%")
List<Person> findByFirstNameStartingWith(@Param("prefix"))

// Search persons whose age is greater than or equal to a given value
@Query("SELECT p FROM Person p WHERE p.age >= :age")
List<Person> findByAgeGreaterThanEqual(@Param("age"))

// Search person by exact email match
@Query("SELECT p FROM Person p WHERE p.email = :email")
Person findByEmail(@Param("email"))

4. Implement Business Logic: PersonService.java
// Define PersonService class in service package
ANNOTATE with @Service

AUTOWIRE PersonRepository

// Save a new person
DEFINE method savePerson(Person)
  RETURN repository.save(Person)

// Get persons by first name prefix
DEFINE method getPersonsByFirstNamePrefix(String prefix)
  RETURN repository.findByFirstNameStartingWith(prefix)

// Get persons by age >= specified value
DEFINE method getPersonsByAge(int age)
  RETURN repository.findByAgeGreaterThanEqual(age)

// Get person by exact email
DEFINE method getPersonByEmail(String email)
  RETURN repository.findByEmail(email)

5. Define REST Endpoints: PersonController.java
less

```
CopyEdit
// Define PersonController class in controller package
ANNOTATE with @RestController
SET base path: @RequestMapping("/api/persons")

AUTOWIRE PersonService

// POST: Add a new person
DEFINE endpoint: @PostMapping
METHOD: addPerson(@RequestBody person)
   CALL service.savePerson(person)
   RETURN saved person

// GET: Get persons by first name prefix
DEFINE endpoint: @GetMapping("/firstname")
METHOD: getPersonsByFirstNamePrefix(@RequestParam prefix)
   RETURN service.getPersonsByFirstNamePrefix(prefix)

// GET: Get persons by age >= given value
DEFINE endpoint: @GetMapping("/age")
METHOD: getPersonsByAge(@RequestParam age)
   RETURN service.getPersonsByAge(age)

// GET: Get person by email
DEFINE endpoint: @GetMapping("/email")
METHOD: getPersonByEmail(@RequestParam email)
   RETURN service.getPersonByEmail(email)

6. Configure application.properties
pgsql
CopyEdit
SET spring.datasource.url = jdbc:mysql://localhost:3306/person_db
SET spring.datasource.username = root
SET spring.datasource.password = yourpassword
SET spring.jpa.hibernate.ddl-auto = update
SET spring.jpa.show-sql = true

7. Run and Test the Application
cpp
CopyEdit
// RUN Spring Boot Application

// Test 1: POST Request
URL: http://localhost:8080/api/persons
BODY:
{
  "firstName": "Alice",
  "lastName": "Smith",
  "email": "alice@example.com",
  "age": 28
}

// Test 2: GET by first name prefix
URL: http://localhost:8080/api/persons/firstname?prefix=Al
```

// Test 3: GET by age
URL: http://localhost:8080/api/persons/age?age=25

// Test 4: GET by email
URL: http://localhost:8080/api/persons/email?email=alice@example.com

Result:

Thus to Build a web application for managing Person details using custom JPQL queries via RESTful APIs, supporting POST and GET operations were successfully completed.

Date :
Exercise Number 12
Title:

Aim:

To Develop a Spring Boot application with "Person" and "Address" entities, where each person has exactly one address. Utilize Spring JPA to establish a one-to-one mapping between these entities.

Algorithm:

1. Create Person and Address entities with a one-to-one mapping relationship using JPA annotations.
2. Develop PersonRepository and AddressRepository interfaces to interact with the database for each entity.
3. Build a PersonService to provide business logic for CRUD operations on Person and Address.
4. Create PersonController with endpoints to manage Person and Address objects through HTTP requests.
5. Run the Spring Boot application and use tools like Postman to test the API endpoints for creating, updating, fetching, and deleting Person and Address entities.
6. Ensure that each Person entity is correctly linked with one Address entity, verifying the relationship integrity in the database.

Pseudo Code:

1. Initialize Spring Boot Project

BEGIN

CREATE Spring Boot project
ADD dependencies:
- Spring Web
- Spring Data JPA
- MySQL Driver (or H2 for in-memory testing)

DEFINE package structure:
- model
- repository
- service
- controller

2. Define Entity: Address.java

// Define Address as @Entity
CLASS Address:
FIELDS:
- id (Long) → @Id, @GeneratedValue
- street (String)

- city (String)

METHODS:

- Getters and Setters

3. Define Entity: Person.java

// Define Person as @Entity

CLASS Person:

FIELDS:

- id (Long) → @Id, @GeneratedValue
- name (String)
- address (Address) → @OneToOne(cascade = ALL), @JoinColumn(name="address_id", referencedColumnName="id")

METHODS:

- Getters and Setters

4. Define Repository Interfaces

// PersonRepository: for CRUD on Person

INTERFACE PersonRepository EXTENDS JpaRepository<Person, Long>

// AddressRepository: for CRUD on Address (if needed separately)

INTERFACE AddressRepository EXTENDS JpaRepository<Address, Long>

5. Create Service Layer: PersonService.java

// CLASS PersonService

ANNOTATE with @Service

INJECT PersonRepository

DEFINE METHODS:

- getAllPersons(): return personRepository.findAll()
- getPersonById(id): return personRepository.findById(id)
- createOrUpdatePerson(person): return personRepository.save(person)
- deletePerson(id): personRepository.deleteById(id)

6. Define Controller: PersonController.java

// CLASS PersonController

ANNOTATE with @RestController

SET base path: @RequestMapping("/api/persons")

INJECT PersonService

DEFINE ENDPOINTS:

// GET: Fetch all persons
@GetMapping("/")
→ CALL getAllPersons() → RETURN list

// GET: Fetch person by ID
@GetMapping("/{id}")
→ CALL getPersonById(id)
→ IF person found RETURN person
→ ELSE RETURN 404 Not Found

// POST: Create a new person with address
@PostMapping("/")
→ CALL createOrUpdatePerson(person)
→ RETURN saved person

// PUT: Update existing person details
@PutMapping("/{id}")
→ CHECK if person exists
- IF exists: update name and address
- SAVE updated person
- RETURN updated person
- ELSE RETURN 404 Not Found

// DELETE: Remove person by ID
@DeleteMapping("/{id}")
→ CALL deletePerson(id)
→ RETURN 204 No Content

7. Configure application.properties

pgsql
CopyEdit
SET spring.datasource.url = jdbc:mysql://localhost:3306/your_db
SET spring.datasource.username = root
SET spring.datasource.password = your_password
SET spring.jpa.hibernate.ddl-auto = update
SET spring.jpa.show-sql = true

8. Run and Test the Application

bash
CopyEdit
START Spring Boot application

TEST using Postman:

1. POST /api/persons
   BODY:
   ```
   {
    "name": "John Doe",
    "address": {
     "street": "123 Main St",
     "city": "New York"
    }
   }
   ```

2. GET /api/persons
3. GET /api/persons/{id}
4. PUT /api/persons/{id}
5. DELETE /api/persons/{id}

Result:

Thus, to Develop a Spring Boot application with "Person" and "Address" entities, where each person has exactly one address. Utilize Spring JPA to establish a one-to-one mapping between these entities were successfully completed.

Date :

Exercise Number 13

Title: Use Spring JPA to establish a one-to-many bidirectional mapping between these entities.

Aim:

To Create a Spring Boot application with "Author" and "Book" entities, where each author can have multiple books, and each book belongs to only one author. Use Spring JPA to establish a one-to-many bidirectional mapping between these entities.

Algorithm:

1. Create Author and Book entities. Establish a one-to-many bidirectional relationship between Author and Book using JPA annotations.
2. Develop AuthorRepository and BookRepository interfaces to handle CRUD operations for each entity.
3. Build an AuthorService to provide business logic for CRUD operations involving Author and the associated Book entities.
4. Create AuthorController with endpoints to manage Author and Book objects through HTTP requests.
5. Run the Spring Boot application and use tools like Postman to test the REST API endpoints for creating, updating, fetching, and deleting Author and Book entities.
6. Ensure that each Author entity is correctly linked to multiple Book entities, verifying the relationship integrity in the database.

Pseudo Code:

1. Initialize Spring Boot Project
BEGIN

CREATE Spring Boot project
ADD dependencies:
- Spring Web
- Spring Data JPA
- MySQL Driver (or H2 for testing)

SETUP package structure:
- model
- repository
- service
- controller

CONFIGURE application.properties for database

---

2. Define Entity: Author.java
// Author entity with one-to-many relationship
CLASS Author:
FIELDS:
- id: Long → @Id, @GeneratedValue
- name: String

- books: List<Book> → @OneToMany(mappedBy = "author", cascade = ALL, orphanRemoval = true)

METHODS:
- Getters and Setters

---

3. Define Entity: Book.java
// Book entity with many-to-one reference to Author
CLASS Book:
FIELDS:
- id: Long → @Id, @GeneratedValue
- title: String
- author: Author → @ManyToOne, @JoinColumn(name = "author_id", nullable = false)

METHODS:
- Getters and Setters

---

4. Define Repository Interfaces
// AuthorRepository for CRUD on Author
INTERFACE AuthorRepository EXTENDS JpaRepository<Author, Long>

// BookRepository for CRUD on Book (optional here)
INTERFACE BookRepository EXTENDS JpaRepository<Book, Long>

---

5. Create Service Layer: AuthorService.java
// CLASS AuthorService
ANNOTATE with @Service

INJECT AuthorRepository

DEFINE METHODS:
- getAllAuthors(): return list of all authors
- getAuthorById(id): return author by id if present
- createOrUpdateAuthor(author): save new or updated author (and associated books)
- deleteAuthor(id): remove author and cascade delete associated books

---

6. Define Controller: AuthorController.java
// CLASS AuthorController
ANNOTATE with @RestController
SET base path: @RequestMapping("/api/authors")

INJECT AuthorService

DEFINE ENDPOINTS:

// GET: fetch all authors
@GetMapping("/")
→ CALL getAllAuthors()
→ RETURN list

// GET: fetch author by ID
@GetMapping("/{id}")
→ CALL getAuthorById(id)
→ IF found RETURN author

→ ELSE RETURN 404 Not Found

// POST: create new author with books
@PostMapping("/")
→ CALL createOrUpdateAuthor(author)
→ RETURN created author

// PUT: update existing author and books
@PutMapping("/{id}")
→ CHECK if author exists
- IF exists:
→ update name
→ update books list
→ save updated author
→ RETURN updated author
- ELSE RETURN 404 Not Found

// DELETE: delete author by ID
@DeleteMapping("/{id}")
→ CALL deleteAuthor(id)
→ RETURN 204 No Content

---

7. Configure application.properties

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

---

8. Run and Test the Application
START the Spring Boot application

TEST with Postman:

// POST /api/authors
BODY:

```
{
  "name": "J.K. Rowling",
  "books": [
    {"title": "Harry Potter 1"},
    {"title": "Harry Potter 2"}
  ]
}
```

// GET /api/authors
// GET /api/authors/{id}
// PUT /api/authors/{id}
// DELETE /api/authors/{id}

Result:

Thus, to Create a Spring Boot application with "Author" and "Book" entities, where each author can have multiple books, and each book belongs to only one author. Use Spring JPA to establish a one-to-many bidirectional mapping between these entities were successfully completed.

Date :
Exercise Number 14
Title: Establish a one-to-one mapping between these entities using Spring JPA

Aim:

To Build a Spring Boot application with "Employee" and "Address" entities, ensuring that each employee has exactly one address, and each address belongs to only one employee. Establish a one-to-one mapping between these entities using Spring JPA and utilize the Criteria API to retrieve employee details efficiently

Algorithm:

1. Create Employee and Address entities, and establish a one-to-one relationship between them using JPA annotations (@OneToOne).
2. Develop the EmployeeRepository interface to interact with the Employee entity in the database.
3. Create the EmployeeService class to handle business logic. Use the Criteria API within the service to retrieve employee details based on certain criteria (e.g., employees by city).
4. Implement the EmployeeController class to provide REST endpoints for managing employees and retrieving employee details using HTTP requests.
5. Run the Spring Boot application, and use tools like Postman to test the endpoints for creating, updating, retrieving, and deleting employees. Also, verify the Criteria API-based retrieval.
6. Ensure that each Employee is correctly linked to exactly one Address and vice versa, verifying the integrity of the one-to-one relationship in the database.

Pseudo Code:

1. Initialize Spring Boot Project
BEGIN

CREATE Spring Boot project
ADD dependencies:
- Spring Web
- Spring Data JPA
- MySQL Driver (or H2 for local test)

CONFIGURE application.properties with database credentials

2. Create Entity: Employee.java
// Employee entity with @OneToOne mapping to Address
CLASS Employee:
FIELDS:
- id: Long → @Id, @GeneratedValue
- name: String
- address: Address → @OneToOne(cascade = ALL), @JoinColumn(name = "address_id", referencedColumnName = "id")

METHODS:
- Getters and Setters

```
3. Create Entity: Address.java
// Address entity with reverse @OneToOne mapping to Employee
CLASS Address:
   FIELDS:
      - id: Long → @Id, @GeneratedValue
      - street: String
      - city: String
      - employee: Employee → @OneToOne(mappedBy = "address")

   METHODS:
      - Getters and Setters

4. Define Repository Interface
// Repository for Employee (CRUD)
INTERFACE EmployeeRepository EXTENDS JpaRepository<Employee, Long>

5. Implement Service Layer: EmployeeService.java
CLASS EmployeeService:

   INJECT EmployeeRepository
   INJECT EntityManager (for Criteria API)

   METHOD getAllEmployees():
      RETURN employeeRepository.findAll()

   METHOD getEmployeeById(id):
      RETURN employeeRepository.findById(id)

   METHOD createOrUpdateEmployee(employee):
      RETURN employeeRepository.save(employee)

   METHOD deleteEmployee(id):
      CALL employeeRepository.deleteById(id)

   METHOD getEmployeesByCity(city):
      USE EntityManager to create CriteriaBuilder
      DEFINE CriteriaQuery for Employee
      DEFINE Root<Employee> from query
      BUILD WHERE condition: employee.address.city == city
      RETURN resultList of matching employees

6. Define Controller Layer: EmployeeController.java
CLASS EmployeeController:
   BASE URL: "/api/employees"

   INJECT EmployeeService

   ENDPOINT GET /:
      RETURN employeeService.getAllEmployees()

   ENDPOINT GET /{id}:
      IF employee exists:
         RETURN employee
      ELSE:
```

```
        RETURN 404 Not Found

    ENDPOINT POST /:
      CALL employeeService.createOrUpdateEmployee(employee)
      RETURN created employee

    ENDPOINT PUT /{id}:
      IF employee exists:
        UPDATE name and address
        RETURN updated employee
      ELSE:
        RETURN 404 Not Found

    ENDPOINT DELETE /{id}:
      CALL employeeService.deleteEmployee(id)
      RETURN 204 No Content

    ENDPOINT GET /by-city?city={city}:
      CALL employeeService.getEmployeesByCity(city)
      RETURN list of employees in that city
```

7. Configure application.properties

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

8. Test Using Postman

TEST ENDPOINTS:

```
// Create Employee with Address
POST /api/employees
BODY:
{
  "name": "John Doe",
  "address": {
    "street": "123 Main St",
    "city": "Chennai"
  }
}

// Get Employee by ID
GET /api/employees/1

// Get Employees by City
GET /api/employees/by-city?city=Chennai

// Update Employee
PUT /api/employees/1
BODY: {...}

// Delete Employee
DELETE /api/employees/1
```

Result:

Thus, to Build a Spring Boot application with "Employee" and "Address" entities, ensuring that each employee has exactly one address, and each address belongs to only one employee. Establish a one-to-one mapping between these entities using Spring JPA and utilize the Criteria API to retrieve employee details efficiently was successfully completed.

Date :

Exercise Number 15

Title: Establish a one-to-one mapping between Employee and Payroll entities.

Aim:

To Develop a web application for managing Employee and Payroll details via RESTful APIs. Utilize Spring JPA to establish a one-to-one mapping between Employee and Payroll entities. Demonstrate the usage of Swagger for API documentation and interaction.

Algorithm:

Step 1: Initialize Application

- Input: Application configuration (database, server settings).
- Output: Running Spring Boot application.

Step 2: Define Data Models

- Create Employee Model
  - Attributes: id, name, department.
- Create Payroll Model
  - Attributes: id, salary, employeeId (foreign key).

Step 3: Set Up Database Connection

- Configure MySQL database connection in application.properties.
- Use Spring JPA to manage data persistence.

Step 4: Create Repositories

- EmployeeRepository
  - Interface for CRUD operations on Employee data.
- PayrollRepository
  - Interface for CRUD operations on Payroll data.

Step 5: Develop Service Layer

1. **EmployeeService**

- Methods:
  - addEmployee(Employee employee): Save employee details.
  - getEmployeeById(Long id): Retrieve employee details by ID.
  - updateEmployee(Employee employee): Update existing employee details.

- deleteEmployee(Long id): Remove employee from the database.

2. **PayrollService**

❖ Methods:

- addPayroll(Payroll payroll): Save payroll details linked to an employee.
- getPayrollByEmployeeId(Long employeeId): Retrieve payroll for a specific employee.
- updatePayroll(Payroll payroll): Update existing payroll details.
- deletePayroll(Long id): Remove payroll entry.

Step 6: Implement REST Controllers

1. **EmployeeController**

❖ Endpoint: /api/employees

- POST /: Add new employee.
- GET /{id}: Get employee details by ID.
- PUT /: Update employee details.
- DELETE /{id}: Delete employee.

2. **PayrollController**

❖ Endpoint: /api/payrolls

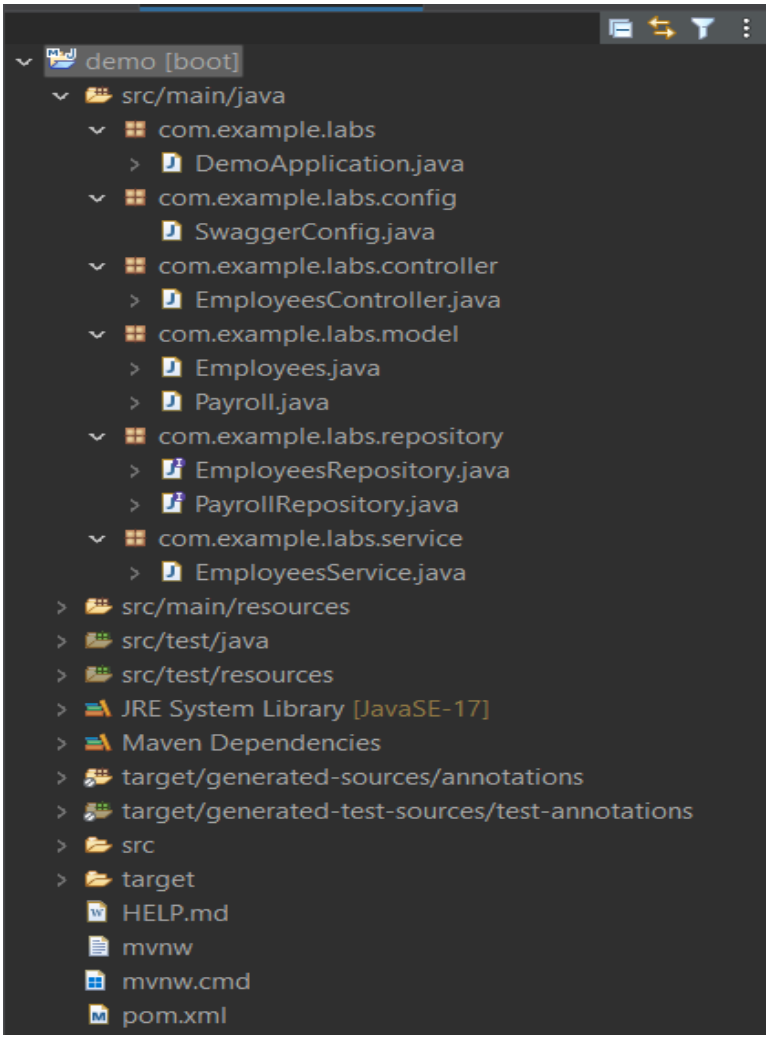
- POST /: Add new payroll.
- GET /employee/{employeeId}: Get payroll details by employee ID.
- PUT /: Update payroll details.
- DELETE /{id}: Delete payroll entry.

Step 7: API Documentation

❖ Integrate Swagger (using SpringDoc OpenAPI).

❖ Document API endpoints for easy access and testing.

Project Structure:



Pseudo Code:

Step 1: Initialize Application
START Spring Boot Application
LOAD application.properties
CONFIGURE MySQL database
LOAD dependencies:
- Spring Web
- Spring Data JPA
- MySQL Connector
- Swagger (SpringDoc OpenAPI)

OUTPUT: Running Application

Step 2: Define Entities
Employees.java
DEFINE Entity: Employees
FIELDS:
- id: Long (Primary Key, Auto Generated)
- name: String
- department: String
- payroll: Payroll (One-to-One, mappedBy = "employee", cascade = ALL)

ANNOTATIONS:
- @Entity
- @OneToOne(mappedBy = "employee", cascade = ALL)

Payroll.java
DEFINE Entity: Payroll
FIELDS:
- id: Long (Primary Key, Auto Generated)
- salary: double
- employee: Employees (One-to-One with JoinColumn to employee_id)

ANNOTATIONS:
- @Entity
- @OneToOne
- @JoinColumn(name = "employee_id")

Step 3: Configure Database (application.properties)
SET spring.datasource.url = jdbc:mysql://localhost:3307/your_db_name
SET spring.datasource.username = your_username
SET spring.datasource.password = your_password
SET spring.jpa.hibernate.ddl-auto = update
SET spring.jpa.show-sql = true
SET spring.jpa.properties.hibernate.dialect = MySQL8Dialect

Step 4: Repositories
EmployeesRepository.java
INTERFACE EmployeesRepository EXTENDS JpaRepository<Employees, Long>
PayrollRepository.java
INTERFACE PayrollRepository EXTENDS JpaRepository<Payroll, Long>

Step 5: Service Layer – EmployeesService.java
Employee Methods
METHOD createEmployee(employee):
  CALL employeeRepository.save(employee)

METHOD getAllEmployees():
  RETURN employeeRepository.findAll()

METHOD getEmployeeById(id):
  FIND employee BY id
  IF NOT FOUND:
    THROW Exception "Employee not found"
  RETURN employee
Payroll Methods
METHOD createPayrollForEmployee(employeeId, payroll):
  FIND employee BY employeeId
  IF NOT FOUND:

```
    THROW Exception "Employee not found"
  SET payroll.employee = employee
  RETURN payrollRepository.save(payroll)

METHOD getPayrollByEmployeeId(employeeId):
  FIND payroll BY employeeId
  IF NOT FOUND:
    THROW Exception "Payroll not found"
  RETURN payroll
```

Step 6: REST Controllers – EmployeesController.java

Employee Endpoints

```
@POST /api/employees
  CALL createEmployee()
  RETURN created employee

@GET /api/employees
  RETURN list of all employees

@GET /api/employees/{id}
  RETURN employee BY ID
```

Payroll Endpoints (Nested)

```
@POST /api/employees/{employeeId}/payroll
  CALL createPayrollForEmployee(employeeId, payroll)
  RETURN created payroll

@GET /api/employees/{employeeId}/payroll
  CALL getPayrollByEmployeeId(employeeId)
  RETURN payroll for the employee
```

Step 7: Swagger Integration

ADD dependency:

- springdoc-openapi-starter-webmvc-ui (version 2.6.0)

ACCESS Swagger UI:

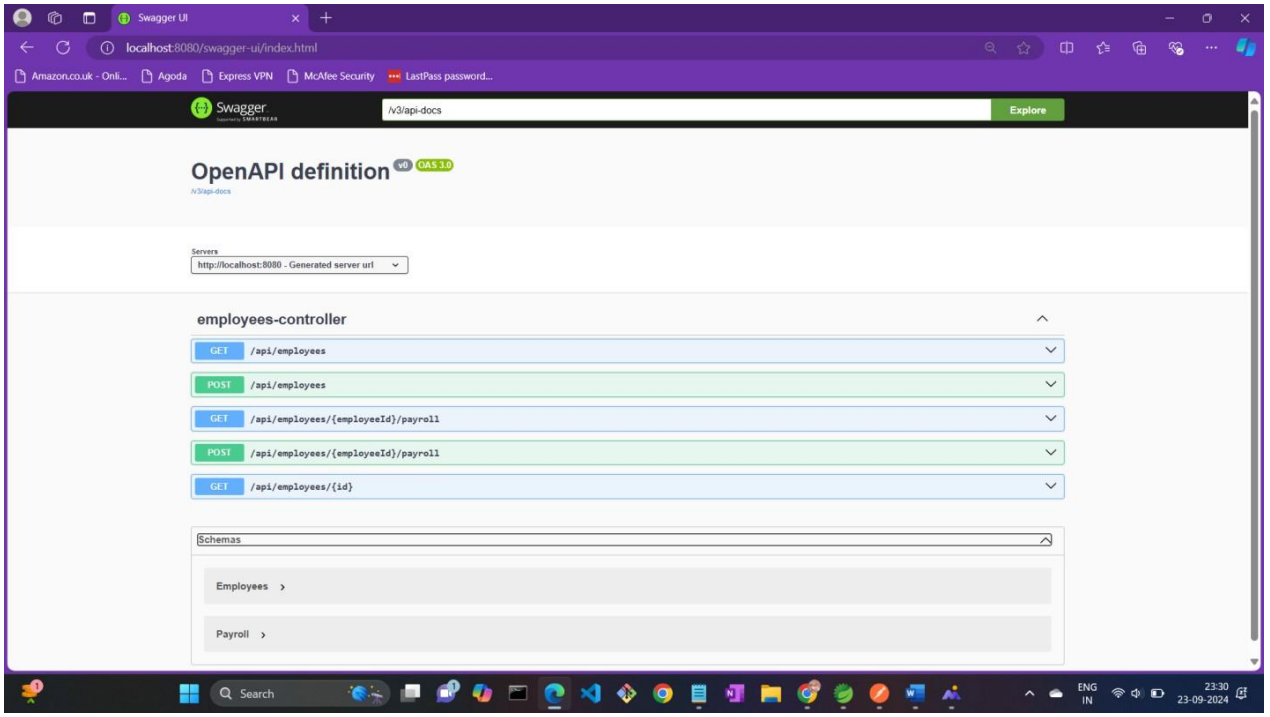
RUN Application

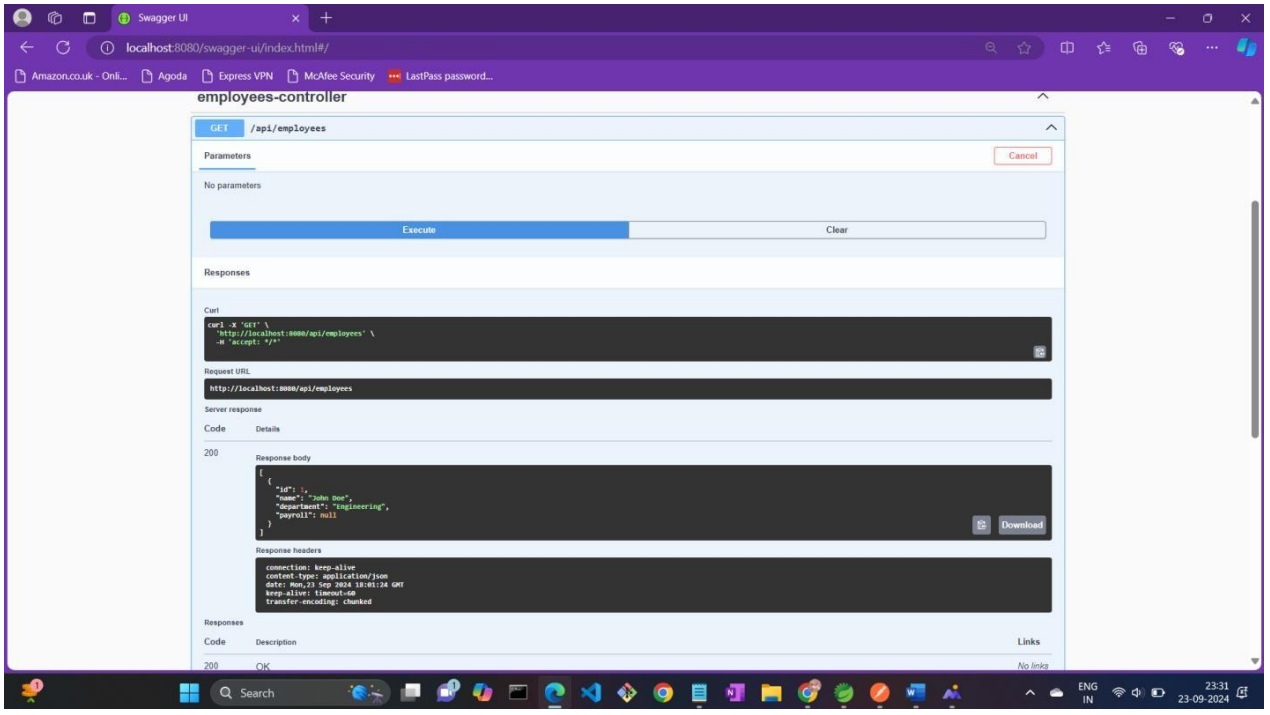
VISIT http://localhost:8080/swagger-ui.html OR /swagger-ui/index.html

INTERACT with API:

- Test POST, GET endpoints for Employee and Payroll
- Auto-generated documentation available

Output:





Result:

Thus, to Develop a web application for managing Employee and Payroll details via RESTful APIs. Utilize Spring JPA to establish a one-to-one mapping between Employee and Payroll entities. Demonstrate the usage of Swagger for API documentation and interaction was successfully completed.

Date :

Exercise Number 16

Title: Integrate comprehensive logging capabilities

Aim:

To Develop a Spring Boot application focused on handling person details and integrate comprehensive logging capabilities to track application activities effectively.

Algorithm:

Step 1: Initialize Application

- **Input**: Application configuration (database, server settings).
- **Output**: Running Spring Boot application.

Step 2: Define Data Models

- **Create Person Model**
  - Attributes: id, firstName, lastName, email, and age.

Step 3: Set Up Database Connection

- Configure MySQL database connection in application.properties.
- Use Spring JPA to manage data persistence.

Step 4: Create Repositories

- **PersonRepository**
  - Interface for CRUD operations on Employee data.

Step 5: Develop Service Layer

- **PersonService**
  - Methods:
    - addPerson(Person person): Save person details.
    - getPersonById(Long id): Retrieve person details by ID.
    - updatePerson(Person person): Update existing person details.
    - deletePerson(Long id): Remove person from the database.

Step 6: Implement REST Controllers

- **PersonController**
  - Endpoint: /api/employees
    - POST /: Add new person.
    - GET /{id}: Get person details by ID.
    - PUT /: Update person details.

- DELETE /{id}: Delete person.

Step 7: Integrate Logging with AOP

Define an aspect in LoggingAspectConfig to log method entry, exit, and exceptions:

- Use **SLF4J** logger to print messages.
- Log method names and parameters using @Before.
- Log exceptions using @AfterThrowing.

Step 8: Configure Logging Levels

In application.properties, set appropriate logging levels.

Example)

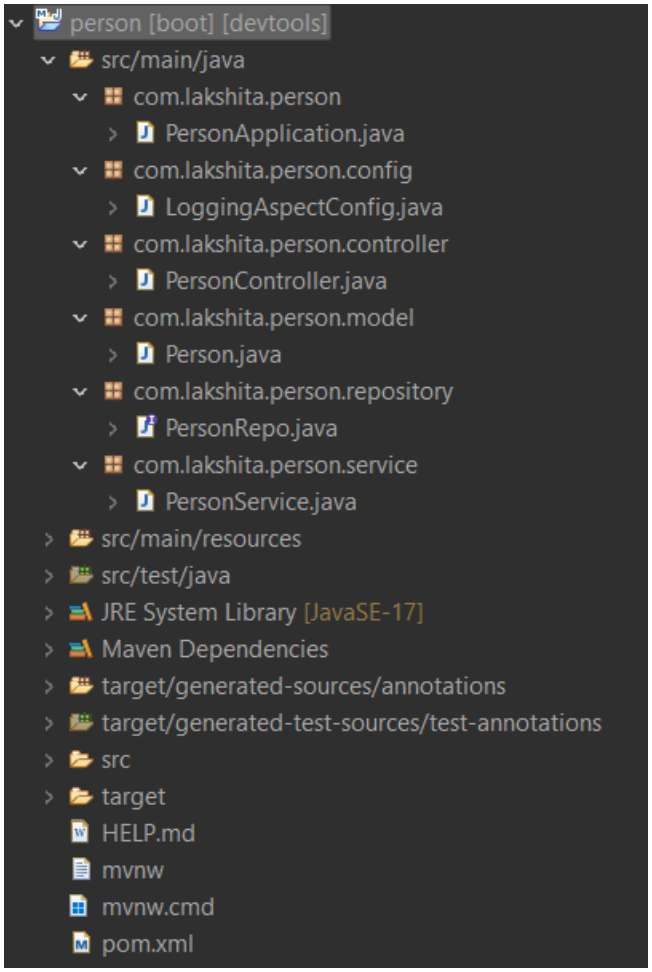
logging.level.root=INFO

logging.level.com.lakshita.person=DEBUG

Step 9: Run the Application and Verify Logs

- Test the application by making API calls (e.g., using Postman).
- Observe log outputs in the console or log files .

Project Structure:



Pseudo Code:

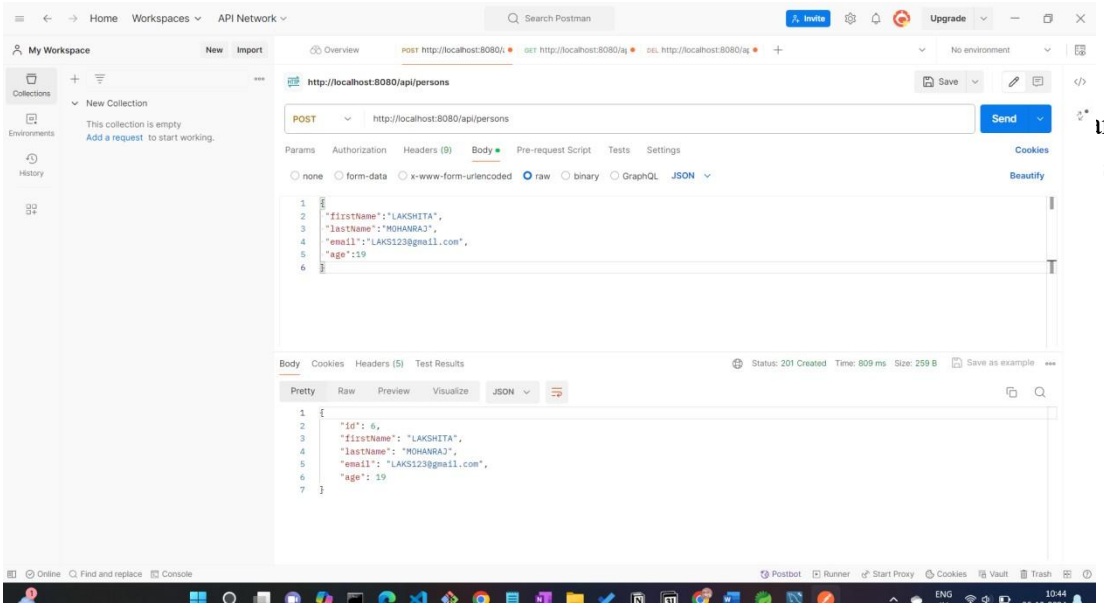
```
Step 1: Initialize Application
START Spring Boot application
  └── Load configurations from application.properties
  └── Connect to MySQL database using JPA
  └── Initialize application context and beans

Step 2: Define Person Entity
DEFINE Entity: Person
  ATTRIBUTES:
    - id (Primary Key, auto-generated)
    - firstName
    - lastName
    - email
    - age
  ANNOTATE with @Entity
  GENERATE:
    - Constructors
    - Getters and Setters

Step 3: Configure Database
SET spring.datasource.url = jdbc:mysql://localhost:3306/your_db
SET spring.datasource.username = your_username
SET spring.datasource.password = your_password
SET spring.jpa.hibernate.ddl-auto = update
SET spring.jpa.show-sql = true

Step 4: Create Repository
INTERFACE PersonRepo EXTENDS JpaRepository<Person, Long>
  └── Provides default CRUD operations for Person entity

Step 5: Implement Service Layer
pseudo
CopyEdit
CLASS PersonService
  DEPENDENCIES: PersonRepo

  METHOD addPerson(person)
    └── SAVE person to database using personRepo

  METHOD getAllPersons()
    └── RETURN all person records from personRepo

  METHOD getPersonById(id)
    └── FETCH person by ID from personRepo
    └── RETURN person or null

  METHOD updatePerson(id, personDetails)
    └── FETCH existing person by ID
    └── UPDATE person attributes
    └── SAVE updated person to DB
```

```
  METHOD deletePerson(id)
    └── DELETE person from DB using personRepo

Step 6: Implement REST Controller
CLASS PersonController
  MAPPING BASE: /api/persons

  METHOD POST /
    └── CALL service.addPerson(person)
    └── RETURN ResponseEntity with status 201

  METHOD GET /
    └── CALL service.getAllPersons()
    └── RETURN list of persons

  METHOD GET /{id}
    └── CALL service.getPersonById(id)
    └── RETURN ResponseEntity with person

  METHOD PUT /{id}
    └── CALL service.updatePerson(id, personDetails)
    └── RETURN updated person

  METHOD DELETE /{id}
    └── CALL service.deletePerson(id)
    └── RETURN status 204 (No Content)

Step 7: Logging with AOP
CLASS LoggingAspectConfig
  ANNOTATE with @Aspect and @Component

  LOGGER logger = LoggerFactory.getLogger(LoggingAspectConfig)

  METHOD logBefore(joinPoint)
    └── LOG "Entering method: methodName"
    └── LOG method arguments using joinPoint.getArgs()

  METHOD logAfterThrowing(joinPoint, exception)
    └── LOG "Exception in method: methodName"
    └── LOG stack trace

Step 8: Configure Logging Levels
SET logging.level.root=INFO
SET logging.level.com.lakshita.person=DEBUG

Step 9: Run and Test Application
START Spring Boot Application
USE Postman to:
  - Add new person (POST)
  - View all persons (GET)
  - View person by ID (GET)
  - Update person (PUT)
  - Delete person (DELETE)
```

OBSERVE console logs:
- Method entry logs
- Exception logs (if any)

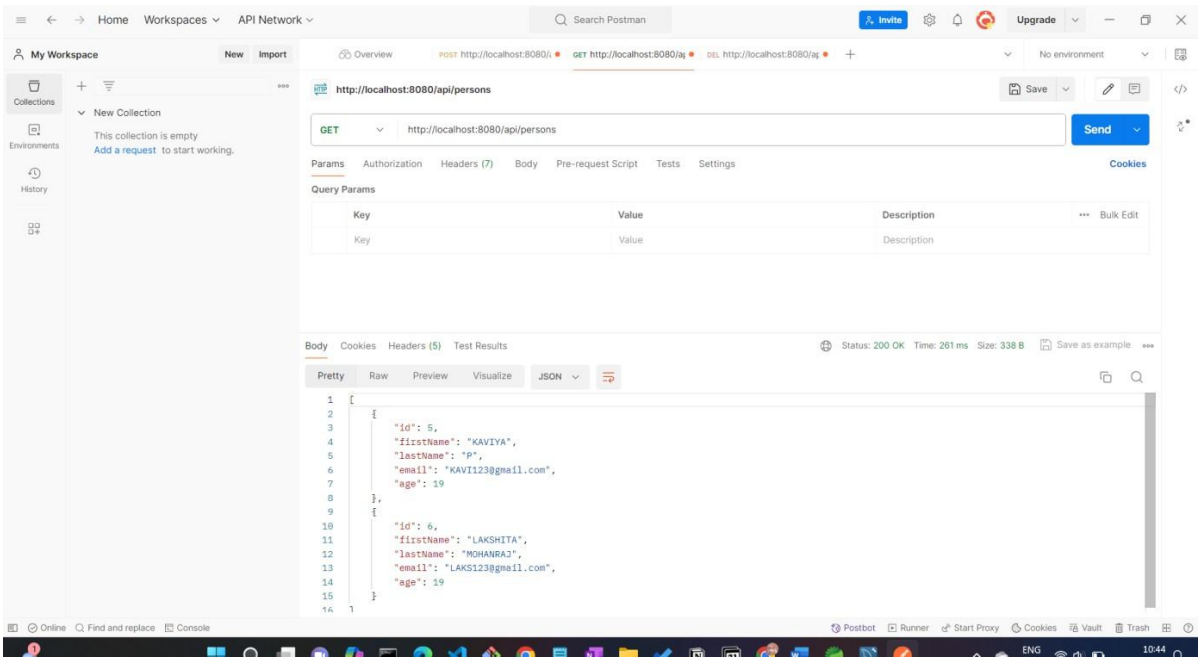
Output:

Postman for API Calls Test - Post

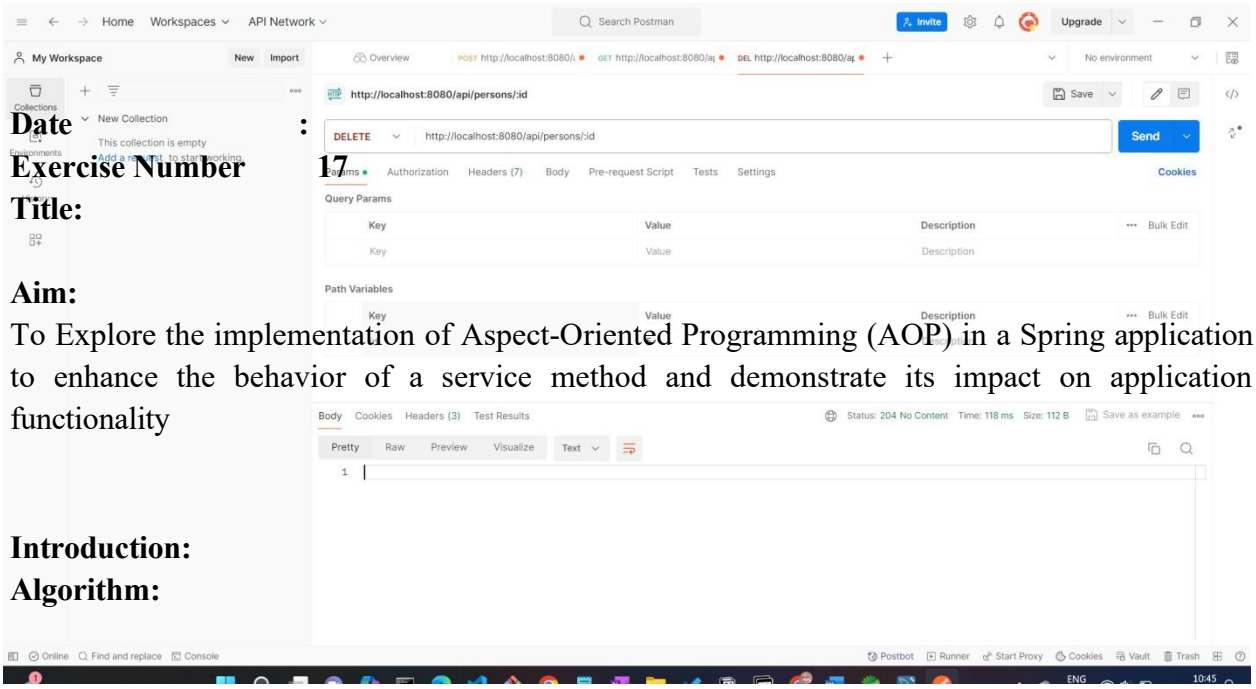


and integrate successfully

Postman for API Calls Test - Get



Postman for API Calls Test - Delete



Date :

Exercise Number 17

Title:

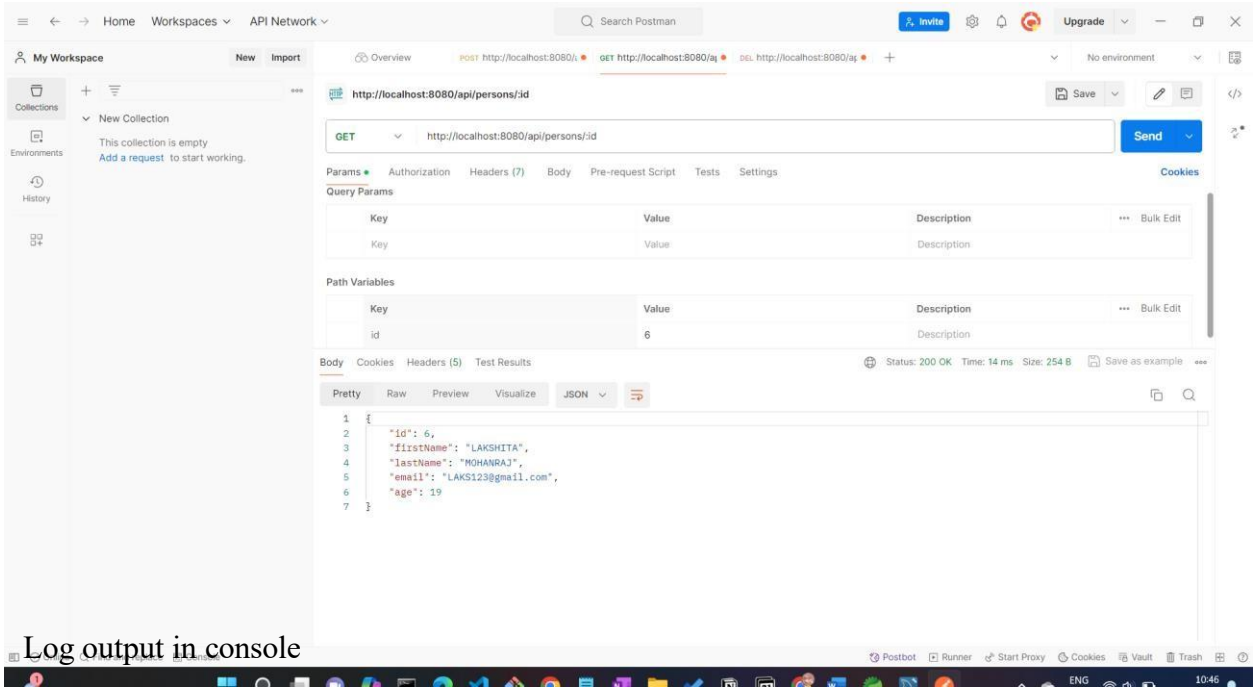
Aim:

To Explore the implementation of Aspect-Oriented Programming (AOP) in a Spring application to enhance the behavior of a service method and demonstrate its impact on application functionality

Introduction:

Algorithm:

Postman for API Calls Test - Get – By Path Variable



Log output in console

2024-10-06 10:43:56 - Entering method: addPerson()
Hibernate: insert into person (age,email,first_name,last_name) values (?,?,?,?)
2024-10-06 10:44:29 - Entering method: getAllPersons() Hibernate: select p1_0.id,p1_0.age,p1_0.email,p1_0.first_name,p1_0.last_name from person p1_0
2024-10-06 10:45:10 - Entering method: deletePerson()

Hibernate: select p1_0.id,p1_0.age,p1_0.email,p1_0.first_name,p1_0.last_name from person p1_0 where p1_0.id=?
Hibernate: delete from person where id=?
2024-10-06 10:46:19 - Entering method: getPersonById() Hibernate: select p1_0.id,p1_0.age,p1_0.email,p1_0.first_name,p1_0.last_name from person p1_0 where p1_0.id=?

Result:

Thus, to Explore the implementation of Aspect-Oriented Programming (AOP) in a Spring application to enhance the behavior of a service method and demonstrate its impact on application functionality was successfully completed.

Date :

Exercise Number 17

Title: Aspect-Oriented Programming (AOP) in a Spring application

Aim:

To explore the implementation of Aspect-Oriented Programming (AOP) in a Spring application to enhance the behavior of a service method and demonstrate its impact on application functionality

Algorithm:

Step 1: Initialize Application

Use Spring Initializer or Maven to create a Spring Boot project with the following dependencies:

- Spring Web
- Spring AOP
- Spring Boot DevTools
- Lombok (for simplifying the code)
- SLF4J/Logback (for logging)

Step 2 : Create a Service Class (ExampleService)

Implement a service method that you will enhance using AOP. The method could perform some business logic, like returning a message.

- Configure MySQL database connection in application.properties.
- Use Spring JPA to manage data persistence.

Step 3: Define an Aspect Class (LoggingAspect)

- In this class, you will define the cross-cutting logic using advice. You can implement aspects like logging, method execution time calculation, or exception handling.

Explanation of Advice:

- @Before – This advice runs before the target method is invoked.
- @After – This advice runs after the target method completes, regardless of its outcome.
- @AfterReturning – This advice runs after the method successfully returns a value, and you can access the return value.

Step 5: Create a REST Controller to Test the Service (ExampleController)

Step 6: Run the Application

Pseudo Code:

Step 1: Initialize Spring Boot Application
START Spring Boot Application with dependencies:
- Spring Web
- Spring AOP
- Lombok
- SLF4J / Logback
- Spring Boot DevTools

Step 2: Define Business Logic in Service
CLASS ExampleService
ANNOTATE with @Service

METHOD sayHello(name: String)
PRINT "Executing sayHello method"
RETURN "Hello, " + name

Step 3: Create Aspect Class for Logging
CLASS LoggingAspect
ANNOTATE with @Aspect and @Component
LOGGER = LoggerFactory.getLogger(LoggingAspect)

BEFORE execution of sayHello(..)
LOG "Before Method: methodSignature"

AFTER execution of sayHello(..)
LOG "After Method: methodSignature"

AFTER RETURNING from sayHello(..)
LOG "Method: methodSignature, Return: result"

Step 4: Enable Aspect Support
CLASS SpringAppApplication
ANNOTATE with @SpringBootApplication
ANNOTATE with @EnableAspectJAutoProxy

MAIN METHOD:
SpringApplication.run(SpringAppApplication.class, args)

Step 5: Create REST Controller

CLASS ExampleController
ANNOTATE with @RestController

DEPENDENCY: ExampleService

CONSTRUCTOR INJECTION

METHOD hello(@RequestParam name)
CALL exampleService.sayHello(name)
RETURN message

Step 6: Run & Test
RUN application

INVOKE URL:
http://localhost:8080/hello?name=John

OBSERVE CONSOLE OUTPUT:
INFO - Before Method: sayHello(String)
PRINT - Executing sayHello method
INFO - Method: sayHello(String), Return: Hello, John
INFO - After Method: sayHello(String)

Outcome

- You invoked /hello?name=John
- The method executed normally
- LoggingAspect successfully intercepted:
  - Before method execution
  - After method return
  - After method execution

Console output:

INFO - Before Method: String com.example.springapp.service.ExampleService.sayHello(String) Executing sayHello method
INFO - Method: String com.example.springapp.service.ExampleService.sayHello(String), Return: Hello, John
INFO - After Method: String com.example.springapp.service.ExampleService.sayHello(String)

Result:

Thus, to Explore the implementation of Aspect-Oriented Programming (AOP) in a Spring application to enhance the behavior of a service method and demonstrate its impact on application functionality was successfully completed.