R.M.K. COLLEGE OF ENGINEERING AND TECHNOLOGY

(An Autonomous Institution)

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

Approved by AICTE, New Delhi /Affiliated to Anna University, Chennai

Accredited by NBA, New Delhi (All Eligible Courses)/ NAAC with 'A' Grade

An ISO 21001:2018 Certified Institution

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LAB MANUAL

22CS938 Rest Application Development Using Spring Boot and JPA (Lab Integrated)

Academic Year	**:**	**2025-2026**
Regulations	**:**	**2022**
Batch	**:**	**2023-2027**
Year / Semester	**:**	**III / V**

Prepared By,

Ms. JEYA RATHINAM J, AP / CSE

Ms. GOKILA E, AP/ CSE

R.M.K. COLLEGE OF ENGINEERING AND TECHNOLOGY

(An Autonomous Institution)

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

Approved by AICTE, New Delhi /Affiliated to Anna University, Chennai
Accredited by NBA, New Delhi (All Eligible Courses)/ NAAC with 'A' Grade
An ISO 21001:2018 Certified Institution

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LAB MANUAL

22CS938 Rest Application Development Using Spring Boot and JPA (Lab Integrated)

Academic Year	**:**	**2025-2026**
Regulations	**:**	**2022**
Batch	**:**	**2023-2027**
Year / Semester	**:**	**III / V**

Prepared By,

Ms. JEYA RATHINAM J, AP / CSE

Ms. GOKILA E, AP/ CSE

Institute Vision and Mission

Vision

To be knowledge hub of providing quality technical education and promoting research for building up of our nation and its contribution for the betterment of humanity

Mission

- To make the best use of state-of-the-art infrastructure to ensure quality technical education
- To develop industrial collaborations to promote innovation and research capabilities
- To inculcate values and ethics to serve humanity

Vision

To be a source of knowledge in the field of Computer Science and Engineering to cater to the growing need of industry and society

Mission

- To avail state-of-the art infrastructure for adopting cutting edge technologies and encouraging research activities
- To promote industrial collaborations for professional competency
- To nurture social responsibility and ethics to become worthy citizens

Program Educational Objectives (PEOs)

Graduates of Computer Science and Engineering Program will

1. Become a globally competent professional in all spheres and pursue higher education world over.
2. Successfully carry forward domain knowledge in computing and allied areas to solve complex real world engineering problems.
3. Continuously upgrade their technical knowledge and expertise to keep pace with the technological revolution.
4. Serve the humanity with social responsibility combined with ethics.

Program Specific Outcomes (PSOs)

Graduates of Computer Science and Engineering Program will be able to:

- Analyze, design and develop computing solutions by applying foundational concepts of Computer Science and Engineering.
- Apply software engineering principles and practices for developing quality software for scientific and business applications.
- Implement cost-effective solutions for the betterment of both industry and society with the technological skills acquired through the Centres of Excellence.

22CS938	REST APPLICATION DEVELOPMENT USING SPRING BOOT AND JPA	L	T	C
		2	0	3

COURSE OBJECTIVES:

The Course will enable the learners:

- 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	6+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	6+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	6+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	6+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	6+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”, 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
4. Raja CSP Raman, Ludovic Dewailly, “Building RESTful Web Services with Spring 5 – Second Edition
5. .Leverage the power of Spring 5.0, Java SE 9, and Spring Boot 2.0”, Packt Publishing, 2018

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

SOFTWARE REQUIREMENTS:

Java Persistence API, Spring Boot

COURSE OUTCOMES	
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.

Course Articulation Matrix

Course Code/ Course Name: 22CS938/ Rest Application Development Using Spring Boot and JPA

Course Outcome – Program Outcome & Program Specific Outcome Mapping:

Course Outcomes (COs)	Program Outcomes (POs), Program Specific Outcomes (PSOs)														
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CO1	3	2	3	-	2	-	-	-	-	1	-	2	2	3	2
CO2	3	3	3	-	2	-	-	-	-	-	-	2	3	3	2
CO3	3	3	3	-	2	-	-	-	-	1	-	2	3	3	2
CO4	3	2	3	-	2	-	-	-	-	-	-	2	2	3	2
CO5	3	2	3	-	3	-	-	2	1	2	-	3	2	3	3
CO6	3	2	3	-	2	-	-	3	2	2	-	3	2	3	2
CO	3.00	2.33	3.00	-	2.17	-	-	2.50	1.50	1.50	-	2.33	2.33	3.00	2.17

EXPERIMENTS WITH MAPPED CO

S.NO	NAME OF THE EXERCISE	Mapped CO
1	Develop a RESTful API for retrieving a welcome message emphasizing the basics of data exchange between client and server.	CO1
2	Implement a RESTful API to acknowledge the user's favorite color choice highlighting property value injection principles.	CO1
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	CO1
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.	CO1
5	Develop a web application for managing patient details using RESTful APIs, implementing POST and GET operations.	CO2
6	Create a web application for managing product details using RESTful APIs, enabling POST and GET operations.	CO2
7	Build an application for managing employee details using RESTful APIs, supporting POST, PUT, and DELETE operations	CO2
8	Develop a web application for pagination and sorting of children details using RESTful APIs, implementing POST and GET operations.	CO3
9	Create a web application for managing Person details using JPA methods via RESTful APIs, enabling POST and GET operations.	CO3
10	Retrieve person details using JPQL with conditions for names starting or ending with specific patterns.	CO3
11	Build a web application for managing Person details using custom JPQL queries via RESTful APIs, supporting POST and GET operations.	CO4
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	CO4
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	CO4

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	CO5
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.	CO5
16	Develop a Spring Boot application focused on handling person details and integrate comprehensive logging capabilities to track application activities effectively	CO5
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	CO5

VIVA QUESTIONS

1. INTRODUCTION TO REST API

1. What is a RESTful API?

Answer:

A RESTful API is an application programming interface that follows the principles of Representational State Transfer (REST) and uses HTTP methods (GET, POST, PUT, DELETE) to access and manipulate resources.

2. How does data exchange occur between a client and server in REST APIs?

Answer:

Data is exchanged using HTTP requests and responses. Clients send requests (e.g., GET, POST) to a server, which processes the request and sends back a response, usually in JSON or XML format.

3. What does 'separating concerns' mean in REST API architecture?

Answer:

It means dividing the application into different layers: one for handling HTTP requests/responses and another for executing the business logic. This improves maintainability and scalability.

4. What HTTP methods are commonly used in REST APIs?

Answer:

GET: Retrieve data
POST: Create new data
PUT: Update existing data
DELETE: Remove data

5. What are server resources in the context of REST?

Answer:

Resources are data entities (like users, products, orders) accessible via unique URIs. They can be manipulated using standard HTTP methods.

6. How are property values injected in RESTful applications?

Answer:

Property values are typically injected using dependency injection frameworks (like Spring in Java) where configuration or object references are automatically provided to classes at runtime.

7. What is a self-contained REST application?

Answer:

It is an application that contains everything needed to run independently, including web server, API logic, and configurations, often built using frameworks like Spring Boot.

8. What is serialization in the context of REST APIs?

Answer:

Serialization is the process of converting objects (like Java or Python objects) into a format like JSON or XML to be sent over the network.

9. What is deserialization?

Answer:

Deserialization is the reverse of serialization – converting JSON/XML data received from the client into programming language-specific objects.

10. Why is JSON commonly used in REST APIs?

Answer:

JSON is lightweight, easy to read and parse, and is supported by most programming languages and platforms, making it ideal for data exchange in REST APIs.

11. What are JSON properties?

Answer:

JSON properties are key-value pairs used to represent data. For example:

{ "name": "Alice", "age": 25 }

12. What is the purpose of a controller in RESTful architecture?

Answer:

A controller handles HTTP requests, maps them to appropriate methods, and returns HTTP responses. It acts as a bridge between the client and business logic layer.

13. How is data access typically managed in RESTful applications?

Answer:

Data access is managed using Data Access Objects (DAO) or Repositories which interact with databases using queries or Object-Relational Mapping (ORM) frameworks.

14. What is the role of HTTP status codes in REST APIs?

Answer:

HTTP status codes indicate the result of the request:

200: OK
201: Created
400: Bad Request
404: Not Found
500: Internal Server Error

15. What are the benefits of using REST APIs?

Answer:

Stateless communication
Scalability
Easy integration
Platform and language independence
Simple and lightweight architecture

2. ADVANCED DATA MANAGEMENT WITH JAVA AND MYSQL

1. Q: How do you map a Java class to a MySQL table using JPA?
Answer:

We use JPA annotations like @Entity, @Table, @Id, @Column to map Java classes to database tables. For example:

```
@Entity
@Table(name = "users")
public class User {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @Column(name = "username")
  private String username;
}
```

2. Q: What is a repository interface and why is it important?
Answer:

A repository interface provides an abstraction over data access operations. Using Spring Data JPA, we extend interfaces like JpaRepository or CrudRepository to auto-generate methods for CRUD operations.

3. Q: Explain how to retrieve data from a MySQL database using Spring Data JPA.
Answer:

We define a repository interface and use method naming conventions. For example:

```
List<User> findByUsername(String username);
```
Spring will automatically generate the necessary SQL based on the method name.

4. Q: What annotation is used to map the request body to an entity in a REST controller?
Answer:

The @RequestBody annotation is used to map incoming JSON payloads to Java objects.

```
@PostMapping("/users")
public ResponseEntity<User> createUser(@RequestBody User user) {
   return ResponseEntity.ok(userService.save(user));
}
```

5. Q: How do you retrieve an entity by its ID?
Answer:

We use the repository's findById(id) method which returns an Optional<T>.

```
Optional<User> user = userRepository.findById(1L);
```

6. Q: What are the benefits of using Optional in data retrieval?
Answer:

Optional helps in avoiding NullPointerException by providing a safer way to handle absent values. We can use methods like isPresent() or orElse().

7. Q: What is the role of the @Transactional annotation in data operations?
Answer:

@Transactional ensures that a method executes within a database transaction, which guarantees atomicity, consistency, isolation, and durability (ACID).

8. Q: How do you handle one-to-many and many-to-one relationships in JPA?
Answer:

We use annotations like @OneToMany, @ManyToOne, and @JoinColumn.

Example:

```
@OneToMany(mappedBy = "user")
private List<Order> orders;

@ManyToOne
@JoinColumn(name = "user_id")
private User user;
```

9. Q: What is a DTO and why is it used in API design?
Answer:

DTO (Data Transfer Object) is used to encapsulate and transfer data between layers, typically to prevent exposing the full entity or to include computed fields.

10. Q: How do you create a custom query in Spring Data JPA?
Answer:

Using the @Query annotation with JPQL or native SQL.

```
@Query("SELECT u FROM User u WHERE u.email = ?1")
User findByEmail(String email);
```

11. Q: Explain the difference between eager and lazy loading in JPA.

Answer:

Eager Loading: Fetches related entities immediately.

Lazy Loading: Fetches related entities only when accessed.

```
@OneToMany(fetch = FetchType.LAZY)
private List<Order> orders;
```

12. Q: How can you optimize performance when dealing with large datasets in MySQL using JPA?
Answer:

Use pagination (Pageable), projections, selective fetching (JPQL), and limit eager relationships.

13. Q: How do you capture form or API data in a Spring Boot controller?
Answer:

Use @RequestBody for JSON payloads, @RequestParam for URL parameters, and @ModelAttribute for form submissions.

14. Q: What are some best practices for building production-grade data access layers?

Answer:

Use DTOs to control exposure of entities.
Implement exception handling with @ControllerAdvice.
Validate input using @Valid and @NotNull.
Use service layers between controllers and repositories.

15. Q: How do you build complex queries like joins or filters in Spring Data JPA?
Answer:

Use method naming conventions: findByStatusAndDateBetween(...)
Use @Query with JPQL/native SQL for joins and filters.
Alternatively, use the Criteria API for dynamic queries.

3. ADVANCED JPA QUERIES AND ANNOTATIONS

1. What is Pagination in JPA and why is it used?
Answer:
Pagination allows retrieving a subset of records (a "page") from a large result set to improve performance and scalability. It's implemented using Pageable or PageRequest in Spring Data JPA.

2. How do you implement pagination using Spring Data JPA?
Answer:
Use the Pageable interface:
Page<User> findAll(Pageable pageable);
And call it like:
PageRequest.of(pageNumber, pageSize);

3. What is the difference between Page, Slice, and List in Spring Data JPA?
Answer:
Page: Provides total pages, total elements, etc.
Slice: Only gives data and info about next page.
List: Just returns data without pagination metadata.

4. How is Sorting achieved in JPA?
Answer:
Using Sort object with method signatures like:
findAll(Sort.by("name").ascending());

5. What is the @Transient annotation in JPA?
Answer:
@Transient is used to indicate that a field should not be persisted in the database. It is ignored by the persistence provider.

6. When would you use the @Transient annotation?
Answer:
When you have a field used for business logic or computation, but it should not be stored in the DB.

7. What is JPQL (Java Persistence Query Language)?
Answer:
JPQL is an object-oriented query language used to perform queries on entities, not tables. Syntax is similar to SQL but operates on entity classes and fields.

8. How do you define a JPQL query using the @Query annotation?
Answer:
@Query("SELECT u FROM User u WHERE u.name = ?1")
List<User> findByName(String name);

9. How can you perform a "starts with" query using JPQL?
Answer:
Using the LIKE operator:
@Query("SELECT u FROM User u WHERE u.name LIKE ?1%")
Or:
@Query("SELECT u FROM User u WHERE u.name LIKE CONCAT(:prefix, '%')")

10. How can you perform an "ends with" query in JPQL?
Answer:
@Query("SELECT u FROM User u WHERE u.name LIKE CONCAT('%', :suffix)")

11. How do you write a custom JPQL query with multiple conditions?
Answer:
@Query("SELECT u FROM User u WHERE u.name = :name AND u.age > :age")
List<User> findByNameAndAgeGreaterThan(@Param("name") String name, @Param("age") int age);

12. What is the difference between JPQL and Native Query?
Answer:
JPQL: Works on entities and their fields.
Native Query: Uses raw SQL, works on database tables and columns.

13. How do you execute a native query in Spring Data JPA?
Answer:
@Query(value = "SELECT * FROM users WHERE status = ?1", nativeQuery = true)
List<User> findByStatus(String status);

14. What are the advantages of using JPQL over native queries?
Answer:
Database-independent
Easier to maintain
Safer and more readable
Uses entity model

15. Can you sort results in a JPQL query?
Answer:
Yes. Use ORDER BY clause:
@Query("SELECT u FROM User u ORDER BY u.name ASC")
List<User> findAllOrderedByName();

4. JPA ASSOCIATIONS AND MAPPING

1. What is a JPA association?
Answer:
A JPA association represents a relationship between two entities. These can be One-to-One, One-to-Many, Many-to-One, or Many-to-Many.

2. How is a One-to-One relationship defined in JPA?
Answer:
Using the @OneToOne annotation. For example:

```
@OneToOne
@JoinColumn(name = "profile_id")
private Profile profile;
```

3. What is the difference between unidirectional and bidirectional One-to-One mapping?
Answer:

Unidirectional: Only one entity has a reference to the other.
Bidirectional: Both entities refer to each other using mappedBy in one of them.

4. How do you define a Two-way One-to-One relationship in JPA?
Answer:
One side uses @OneToOne with mappedBy to define the inverse side:

```
// Owning side
@OneToOne
@JoinColumn(name = "address_id")
private Address address;

// Inverse side
@OneToOne(mappedBy = "address")
private Person person;
```

5. What is lazy and eager loading in JPA?
Answer:

EAGER: Loads the associated entity immediately.
LAZY: Loads it only when accessed. Improves performance by loading only needed data.

6. What is the default fetch type for One-to-One association in JPA?
Answer:
The default fetch type for @OneToOne is EAGER.

7. How can you optimize loading performance in JPA associations?
Answer:
By using:

LAZY fetching,
DTO projections,
JOIN FETCH in queries,
Batch fetching and Entity graphs.

8. Write a JPQL query to fetch an entity with its One-to-One association.
Answer:
SELECT e FROM Employee e JOIN FETCH e.profile

9. How do you handle cascading in JPA relationships?
Answer:
Use the cascade attribute:
@OneToOne(cascade = CascadeType.ALL)
This ensures related entities are persisted, merged, or removed together.

10. What annotation is used to map One-to-Many relationships in JPA?
Answer:
@OneToMany is used. Example:
@OneToMany(mappedBy = "department")
private List<Employee> employees;

11. Can a single entity instance be associated with multiple instances of another entity?
Answer:
Yes, this is typically a One-to-Many or Many-to-Many relationship, depending on the use case.

12. How do you insert data into One-to-One associated entities using JPA?
Answer:
Create both objects, associate them, and persist the owning entity:

Profile p = new Profile(...);
Employee e = new Employee(...);
e.setProfile(p);
entityManager.persist(e);

13. What is the owning side in a bidirectional relationship?
Answer:
The owfning side is the one that defines the @JoinColumn. It controls the relationship and is responsible for database changes.

14. What happens if you omit mappedBy in a bidirectional relationship?

Answer:
JPA treats both sides as owning, which can result in redundant foreign keys or unexpected behavior.

15. How do you delete an entity involved in a One-to-One relationship safely?
Answer:

Ensure orphanRemoval = true or use cascade = CascadeType.REMOVE.
Break the relationship before deletion to avoid foreign key constraint violations.

5. SPRING BOOT ESSENTIALS: API SECURITY, LOGGING, AOP, AND BUILD MANAGEMENT

1. What is Swagger UI and how is it used in Spring Boot?
Answer:
Swagger UI is a web interface that visualizes REST APIs documented using OpenAPI. In Spring Boot, it can be integrated using the springdoc-openapi-ui dependency, which automatically generates documentation for all controllers and endpoints.

2. What is the difference between Swagger and OpenAPI?
Answer:
OpenAPI is the specification for defining RESTful APIs, whereas Swagger is a set of tools (including Swagger UI and Swagger Editor) that implement this specification. Swagger tools are often used to generate and visualize OpenAPI documentation.

3. How do you secure a Spring Boot REST API using JWT?
Answer:
JWT (JSON Web Token) is used to authenticate and authorize API users. In Spring Boot:
A user logs in and receives a JWT token.
The token is included in the Authorization header (Bearer <token>) in subsequent requests.
A filter validates the token before allowing access to secured endpoints.

4. What are the advantages of using JWT over traditional session-based authentication?
Answer:

- Stateless authentication (no server-side session storage)
- Scalable across multiple servers
- Easy to integrate with mobile and third-party clients
- Secure and self-contained

5. What is Aspect-Oriented Programming (AOP) in Spring Boot?
Answer:
AOP allows separation of cross-cutting concerns like logging, security, and transaction management. In Spring Boot, AOP is implemented using aspects (@Aspect) and advice types like @Before, @After, @Around, etc.

6. How do you log method entry and exit using Spring AOP?

Answer:
You can use an aspect with @Around advice to intercept methods, log entry/exit, and even measure execution time. Example:

```
@Around("execution(* com.example.service.*.*(..))")
public Object logMethod(ProceedingJoinPoint joinPoint) throws Throwable {
   log.info("Entering: " + joinPoint.getSignature());
   Object result = joinPoint.proceed();
   log.info("Exiting: " + joinPoint.getSignature());
   return result;
}
```

7. What is the purpose of SonarLint in a Spring Boot project?
Answer:
SonarLint is a static code analysis tool that helps developers detect and fix coding issues in real time, following industry standards and best practices, such as clean code, security vulnerabilities, and potential bugs.

8. How can you change the logging level in a Spring Boot application?
Answer:
Logging levels can be changed in application.properties or application.yml:

```
logging.level.org.springframework=DEBUG
logging.level.com.example=TRACE
```

9. How can you log incoming HTTP requests and responses in Spring Boot?
Answer:
You can use:

- Spring Boot's built-in logging
- Filters or Interceptors to capture request/response data
- AOP for logging at the controller or service layer

10. What are Gradle's main advantages over Maven?
Answer:

- Faster builds due to incremental compilation and caching
- More concise and readable build scripts (Groovy/Kotlin DSL)
- Better performance for large projects
- Easier integration with Kotlin and Android projects

11. What is the difference between build.gradle and settings.gradle?
Answer:
build.gradle: Contains project-specific configuration like dependencies, tasks, and plugins.
settings.gradle: Defines multi-project configurations and project structure.

12. What is Advice in Spring AOP? Name the types.

Answer:
Advice is the action taken at a join point. Types:
@Before: Runs before the method
@After: Runs after the method
@AfterReturning: Runs after method returns successfully
@AfterThrowing: Runs if method throws exception
@Around: Runs before and after method execution

13. How do you handle method parameters in AOP?
Answer:
Method parameters can be accessed via JoinPoint.getArgs() or by declaring them in the pointcut expression using args().

14. What is a Join Point and a Pointcut in AOP?
Answer:

Join Point: A point in execution (e.g., method call) where an aspect can be applied.
Pointcut: A predicate that matches join points. It defines where advice should be applied.

15. How do you ensure secure REST API design in Spring Boot?
Answer:

- Use HTTPS for secure communication
- Secure endpoints using Spring Security with JWT
- Validate input to prevent injection attacks
- Avoid exposing sensitive data in logs or responses
- Implement CORS and rate limiting if needed

Rubrics for Assessment

	Excellent	**Good**	**Average**	**Poor**
Problem Understanding & Logic Design	Clear understanding with efficient logic (3 marks)	Good logic with minor issues (2 marks)	Partial understanding (1 mark)	unclear understanding (0 mark)
Code Implementation	Error-free, well-structured, meets all requirements (3 marks)	Minor errors, mostly meets objectives (2 marks)	Major errors (1 mark)	incomplete implementation (0 mark)
Output & Viva Explanation	All test cases passed, confident explanation (4 marks)	Some test cases passed, fair explanation (3 marks)	Few test cases passed, weak explanation (2 marks)	Attempted, no test cases passed, no explanation (1 mark)

LIST OF EXPERIMENTS

Sl.NO	NAME OF THE EXPERIMENT	SIGN
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	
3	Create a Spring Boot application that retrieves and displays application information, demonstrating the usage of the @Value annotation to inject	
4	Construct a RESTful API for student details retrieval, illustrating the utilization of @JsonIgnore annotation, focusing on advanced JSON property	
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	
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	

14	Build a Spring Boot application with "Employee" and "Address" entities, ensuring that each employee has exactly one address, and each address	
15	Develop a web application for managing Employee and Payroll details via RESTful APIs. Utilize Spring JPA to establish a one-to-one mapping	
16	Develop a Spring Boot application focused on handling person details and integrate comprehensive logging capabilities to track application activities	
17	Explore the implementation of Aspect-Oriented Programming (AOP) in a Spring application to enhance the behavior of a service method and	

Ex-1: To Retrieve Welcome message

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

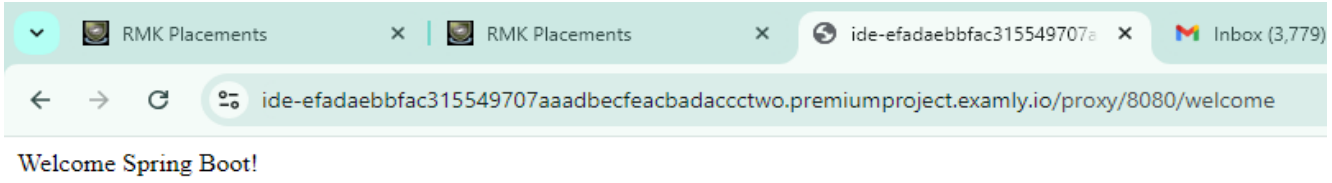
// Step 5: Build and Run the Application
BUILD the project
RUN the application on server (e.g., localhost)
ACCESS endpoint http://localhost:<port>/welcome in browser or API tool

END

PROGRAM :

```
package com.example.springapp.controller;

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

@RestController
public class ApiController{
@GetMapping("/welcome")
public String welcome()
{
  return "Welcome Spring Boot!";
}
}
```

OUTPUT:



Problem understanding and Design (3 Marks)	
Code Implementation (3 Marks)	
Output & Viva Explanation (4 Marks)	
Total (10 Marks)	

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.

Ex.2 To retrieve a color choice using @Value

Title : To retrieve a color choice using @Value

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.

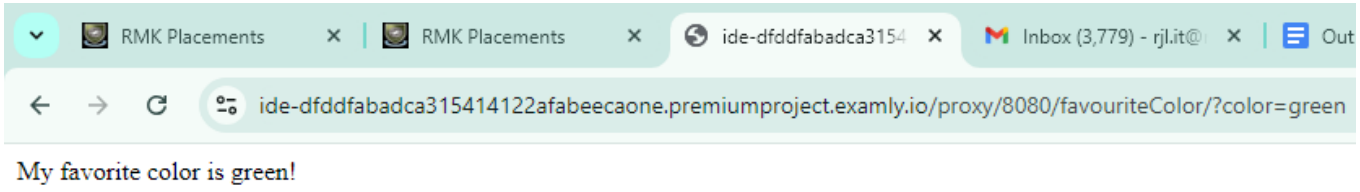
Pseudo Code:

```
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
```

PROGRAM

```
package com.example.springapp.controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class ApiController {
 @GetMapping("/favouriteColor")
  public String Color(@RequestParam String color)
 {
  return "My favorite color is "+color+"!";
 }
}
```

OUTPUT



Problem understanding and Design (3 Marks)	
Code Implementation (3 Marks)	
Output & Viva Explanation (4 Marks)	
Total (10 Marks)	

RESULT:

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

Ex.3 To display application information using @Value

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
```

PROGRAM:

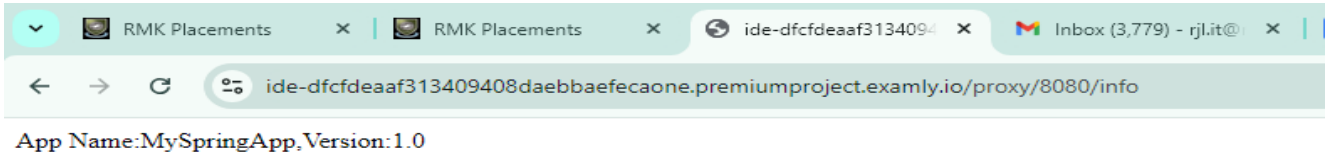
ApiController.java

```
package com.examly.springapp.controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import com.examly.springapp.config.AppConfig;
@RestController
public class ApiController {
  private AppConfig appconfig;

     public ApiController(AppConfig appconfig) {
     this.appconfig = appconfig;
  }

  @GetMapping("/info")
  public String getApp()
{
 return "App Name:" + appconfig.getAppName() + ",Version:" + appconfig.getAppVersion();
}
}
```

AppConfig.java

```
package com.examly.springapp.config;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class AppConfig {
 @Value("${app.name}")
  private String appName;
  @Value("${app.version}")
 private String appVersion;
 public AppConfig()
 {

 }
```

```
public AppConfig(String appName, String appVersion) {
  this.appName = appName;
  this.appVersion = appVersion;
}
public String getAppName() {
  return appName;
}
public String getAppVersion() {
  return appVersion;
}
public void setAppName(String appName) {
  this.appName = appName;
}
public void setAppVersion(String appVersion) {
  this.appVersion = appVersion;
}

}
```



Problem understanding and Design (3 Marks)	
Code Implementation (3 Marks)	
Output & Viva Explanation (4 Marks)	
Total (10 Marks)	

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.

EX -4 To Retrieve Student details using @JsonIgnore

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

PROGRAM:

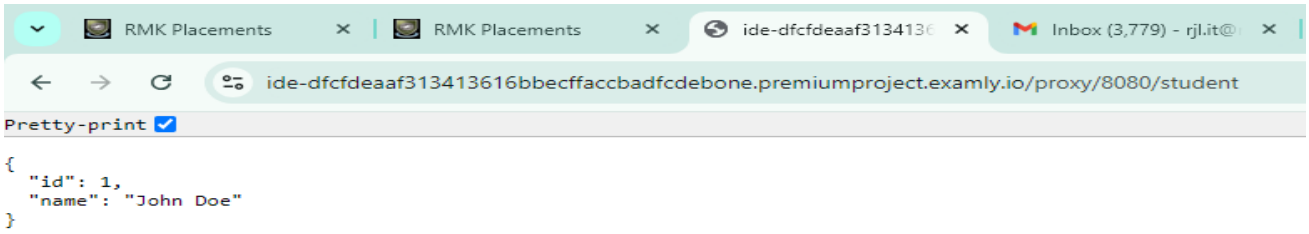
StudentControler.java

```
package com.examly.springapp.controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import com.examly.springapp.model.Student;
@RestController
public class StudentController {
@GetMapping("/student")
public Student getStu()
{
 Student stuobj=new Student(1L,"John Doe","This is a student description");
 return stuobj;
}
}
```

Student.java

```
package com.examly.springapp.model;
import com.fasterxml.jackson.annotation.JsonIgnore;
public class Student {
 private long id;
 private String name;
 @JsonIgnore
 private String description;

public Student()
{

}
public Student(long id, String name, String description) {
   this.id = id;
   this.name = name;
   this.description = description;}
public long getId() {
   return id;
}
public String getName() {
   return name;
}
public String getDescription() {
   return description;
}
public void setId(long id) {
   this.id = id;
}
public void setName(String name) {
   this.name = name;
}
public void setDescription(String description) {
   this.description = description;
}
 }
```

OUTPUT



Problem understanding and Design (3 Marks)	
Code Implementation (3 Marks)	
Output & Viva Explanation (4 Marks)	
Total (10 Marks)	

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.

Ex-5 Managing Patient details using POST and GET operations

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"
    }
  ]
```

PROGRAM:

Patient.java

```
package com.example.springapp.model;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Patient {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private int patientId;
  private String patientName;
```

```
    private String doctorName;
    private String disease;
    private String medication;
public Patient()
{
}
```

```
public Patient(int patientId, String patientName, String doctorName, String disease, String medication) {
    this.patientId = patientId;
    this.patientName = patientName;
    this.doctorName = doctorName;
    this.disease = disease;
    this.medication = medication;
}
public int getPatientId() {
    return patientId;
}
public String getPatientName() {
    return patientName;
}
public String getDoctorName() {
    return doctorName;
}
public String getDisease() {
    return disease;
}
public String getMedication() {
    return medication;
}
public void setPatientId(int patientId) {
    this.patientId = patientId;
}
public void setPatientName(String patientName) {
    this.patientName = patientName;
}
```

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




public void setMedication(String medication) {
   this.medication = medication;
}
}
```

PatientController.java

```
package com.example.springapp.controller;

import java.util.List;
import java.util.Optional;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.example.springapp.model.Patient;
import com.example.springapp.service.PatientService;

@RestController
@RequestMapping("/api")
public class PatientController {
   private PatientService patientService;
@Autowired
   public PatientController(PatientService patientService) {
      this.patientService = patientService;
```

```
    }

 @PostMapping("/patient")
  public ResponseEntity<Patient> postPatient(@RequestBody Patient obj)
  {
    Patient pobj=patientService.postPatient(obj);
    return  ResponseEntity.status(HttpStatus.CREATED).body(pobj);
 }


 @GetMapping("/patient")
 public ResponseEntity<List<Patient>> getAllpatients()
 {
   List <Patient> patients=patientService.getAllpatients();
  return new ResponseEntity<>(patients,HttpStatus.OK);
 }
 @GetMapping("/patient/{patientId}")
 public ResponseEntity<Patient> getPatientById(@PathVariable int patientId)
 {
    Optional<Patient> patient=patientService.getPatientById(patientId);
    if(patient.isPresent())
    {
        return new ResponseEntity<Patient>(patient.get(),HttpStatus.OK);
    }
    return new ResponseEntity<>(HttpStatus.NOT_FOUND);
 }
}
```

PatientService.java

```
package com.example.springapp.service;

import java.util.List;
import java.util.Optional;
import org.springframework.stereotype.Service;
import com.example.springapp.model.Patient;
import com.example.springapp.repository.PatientRepo;

@Service
public class PatientService {
  PatientRepo patientRepo;
```

```
public PatientService(PatientRepo patientRepo) {
   this.patientRepo = patientRepo;
}
public Patient postPatient(Patient obj)
{
return patientRepo.save(obj);
}



 public List<Patient> getAllpatients()
 {
  return patientRepo.findAll();
 }
public Optional<Patient> getPatientById(Integer patientId)
{
  return patientRepo.findById(patientId);
}
}
```

PatientRepo.java

```
package com.example.springapp.repository;

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

import com.example.springapp.model.Patient;
@Repository
public interface PatientRepo extends JpaRepository<Patient, Integer> {

}
```

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"
   }
]
```

Problem understanding and Design (3 Marks)	
Code Implementation (3 Marks)	
Output & Viva Explanation (4 Marks)	
Total (10 Marks)	

RESULT:

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

Ex-6 Managing Product details with POST and GET operations

Title: Enabling POST and GET Operations to Managing Product details

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

PROGRAM :

Product.java

```
package com.example.springapp.model;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Product {
 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
   private int productId;
   private String productName;
   private double price;
   private String description;
   private int quantity;

   public Product(){

   }

   public Product(int productId, String productName, double price, String description, int quantity) {
      this.productId = productId;
      this.productName = productName;
      this.price = price;
      this.description = description;
      this.quantity = quantity;
   }
   public int getProductId() {
      return productId;
   }
   public void setProductId(int productId) {
      this.productId = productId;
   }
```

```
public String getProductName() {
    return productName;
 }
 public void setProductName(String productName) {
    this.productName = productName;
 }
 public double getPrice() {
    return price;
 }
 public void setPrice(double price) {
    this.price = price;
 }
 public String getDescription() {
    return description;
 }
 public void setDescription(String description) {
    this.description = description;
 }
 public int getQuantity() {
    return quantity;
 }
 public void setQuantity(int quantity) {
    this.quantity = quantity;
 }
}
```

ProductController.java

```
package com.example.springapp.controller;
import java.util.List;
import java.util.Optional;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.example.springapp.model.Product;
import com.example.springapp.service.ProductService;

@RestController
@RequestMapping("/api")
public class ProductController {
   @Autowired
   private ProductService productService;

   public ProductController(ProductService productService) {
      this.productService = productService;
   }

   @PostMapping("/product/add")
   public ResponseEntity <Product> postProduct(@RequestBody Product obj)
   {
   Product pobj=productService.postProduct(obj);
   return ResponseEntity.status(HttpStatus.CREATED).body(pobj);
   }
```

```
  @GetMapping("/product")
  public  ResponseEntity<List<Product>> getAllProducts()
  {
   List<Product> lobj =productService.getAllProducts();
   return new ResponseEntity<>(lobj,HttpStatus.OK);
  }

  @GetMapping("/product/{productId}")
  public ResponseEntity<Product> getByProductId(@PathVariable int productid)
  {
   Optional<Product> product=productService.getProductById(productid);
    if (product.isPresent())
    {
      return new ResponseEntity<Product>(product.get(),HttpStatus.OK);

    }
    return new ResponseEntity<>(HttpStatus.NOT_FOUND);
  }

}
```

ProductService.java

```
package com.example.springapp.service;

import java.util.List;
import java.util.Optional;
import org.springframework.lang.NonNull;
import org.springframework.stereotype.Service;
import com.example.springapp.model.Product;
import com.example.springapp.repository.ProductRepo;

@Service
public class ProductService {
   private ProductRepo productRepo;
   public ProductService(ProductRepo productRepo) {
      this.productRepo = productRepo;
   }

  public Product postProduct(@NonNull Product obj)
```

```
    {
     return productRepo.save(obj);
     }
    public List<Product> getAllProducts()
    {
     return productRepo.findAll();
    }
    public Optional<Product> getProductById(@NonNull Integer productid)
    {
     return productRepo.findById(productid);
    }
}
```

ProductRepo.java

```
package com.example.springapp.repository;

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

import com.example.springapp.model.Product;

@Repository
public interface ProductRepo extends JpaRepository<Product,Integer>{

}
```

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"
    }
]
```

Problem understanding and Design (3 Marks)	
Code Implementation (3 Marks)	
Output & Viva Explanation (4 Marks)	
Total (10 Marks)	

RESULT:

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

Ex-7 Managing Employee Details with GET,POST and DELETE Operations

Title: Supporting POST, PUT and DELATE Operations to Managing Employee Details

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

PROGRAM:

Employee.java

```
package com.example.springapp.model;
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
public class Employee {
   @Id
   private int employeeId;
  private String employeeName;
  private String employeeEmail;
  private double salary;
  private String department;
public Employee()
{
 }
public Employee(int employeeId, String employeeName, String employeeEmail, double salary, String
department) {
   this.employeeId = employeeId;
   this.employeeName = employeeName;
   this.employeeEmail = employeeEmail;
   this.salary = salary;
   this.department = department;
}
public int getEmployeeId() {
   return employeeId;
}
public void setEmployeeId(int employeeId) {
   this.employeeId = employeeId;
}
public String getEmployeeName() {
   return employeeName;
}
public void setEmployeeName(String employeeName) {
   this.employeeName = employeeName;
}
public String getEmployeeEmail() {
   return employeeEmail;
}
```

```
public void setEmployeeEmail(String employeeEmail) {
   this.employeeEmail = employeeEmail;
}
public double getSalary() {
   return salary;
}
public void setSalary(double salary) {
   this.salary = salary;
}
public String getDepartment() {
   return department;
}
public void setDepartment(String department) {
   this.department = department;
}
}
```

EmployeeController.java

```
package com.example.springapp.controller;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.PathVariable;



import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.example.springapp.model.Employee;
import com.example.springapp.service.EmployeeService

@RestController
@RequestMapping("/api")
public class EmployeeController {
   private EmployeeService employeeService;
   public EmployeeController(EmployeeService employeeService) {
```

```
        this.employeeService = employeeService;
    }
    @PostMapping("/employee")
    public ResponseEntity<Employee> addEmployee(Employee e)
    {
        Employee obj=employeeService.addEmployee(e);
        return new ResponseEntity<>(obj,HttpStatus.CREATED);
    }



    @PutMapping("/employee/{employeeId}")
    public ResponseEntity<Employee> updateEmployee(@PathVariable int employeeId, @RequestBody
Employee e)
    {
        if (employeeService.updateEmployee(employeeId,e)==true) {
          return new ResponseEntity<>(e,HttpStatus.OK);
        }
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    @DeleteMapping("/employee/{employeeId}")
    public ResponseEntity<Boolean> deleteEmployeeById(@PathVariable int employeeId)
    {
        if (employeeService.deleteEmployeeById(employeeId)==true) {
            return new ResponseEntity<>(true,HttpStatus.OK);
        }
        return new ResponseEntity<>(false,HttpStatus.NOT_FOUND);
    }
}
```

EmployeeService.java

```
package com.example.springapp.service;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.example.springapp.model.Employee;
import com.example.springapp.repository.EmployeeRepo;
@Service
public class EmployeeService {
   private EmployeeRepo employeeRepo;
   @Autowired
   public EmployeeService(EmployeeRepo employeeRepo) {
      this.employeeRepo = employeeRepo;
   }
 public Employee addEmployee(Employee e)
 {
   return employeeRepo.save(e);
 }
 public boolean updateEmployee(int employeeId, Employee e)
 {
   if (employeeRepo.findById(employeeId)!=null) {
      employeeRepo.save(e);
      return true;
   }
   return false;
 }
 public boolean deleteEmployeeById(int employeeId)
 {
   if (employeeRepo.findById(employeeId)==null) {
     return false;
   }
   else{
      employeeRepo.deleteById(employeeId);
      return true;
   }
 }
}
```

EmployeeRepo.java

```
package com.example.springapp.repository;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.example.springapp.model.Employee;
@Repository
public interface EmployeeRepo extends JpaRepository<Employee, Integer> {
  }
```

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

Problem understanding and Design (3 Marks)	
Code Implementation (3 Marks)	
Output & Viva Explanation (4 Marks)	
Total (10 Marks)	

RESULT:

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

Ex-8: Managing Children details using POST and GET operations with Pagination and Sorting

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

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

PROGRAM:

Children.java

```
package com.example.springapp.model;
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
public class Children {
   @Id
   int babyId;
   String babyFirstName,babyLastName,fatherName,motherName,address;
   public Children() {
   }
   public Children(int babyId, String babyFirstName, String babyLastName, String fatherName, String
motherName,
         String address) {
      this.babyId = babyId;
      this.babyFirstName = babyFirstName;
      this.babyLastName = babyLastName;
      this.fatherName = fatherName;
      this.motherName = motherName;
      this.address = address;
   }
   public int getBabyId() {
      return babyId;
   }
   public void setBabyId(int babyId) {
      this.babyId = babyId;
   }
```

```
public String getBabyFirstName() {
   return babyFirstName;
}
public void setBabyFirstName(String babyFirstName) {
   this.babyFirstName = babyFirstName;
}
public String getBabyLastName() {
   return babyLastName;
}
public void setBabyLastName(String babyLastName) {
   this.babyLastName = babyLastName;
}
public String getFatherName() {
   return fatherName;
}
public void setFatherName(String fatherName) {
   this.fatherName = fatherName;
}
public String getMotherName() {
   return motherName;
}
public void setMotherName(String motherName) {
   this.motherName = motherName;
}
public String getAddress() {
   return address;
}
public void setAddress(String address) {
   this.address = address;
}
}
```

Children_Controller

```
package com.example.springapp.controller;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import com.example.springapp.model.Children;
import com.example.springapp.service.ChildrenService;

@RestController
public class ChildrenController {
   @Autowired
   ChildrenService cs;
   //post
   @PostMapping("/api/children")
   public ResponseEntity<Children> create(@RequestBody Children c)
   {
     Children obj = cs.create(c);
      return new ResponseEntity<>(obj,HttpStatus.CREATED);
   }
   //sorting
   @GetMapping("/api/children/sortBy/{field}")
   public List<Children> g(@PathVariable String field)
   {
      return cs.sort(field);
   }
```

```
    //pagination
    @GetMapping("/api/children/{offset}/{pagesize}")
    public List<Children> get(@PathVariable int offset,@PathVariable int pagesize)
    {
        return cs.page(pagesize, offset);
    }
        //sorting and pagination
    @GetMapping("/api/children/{offset}/{pagesize}/{field}")
    public List<Children> getsorting(@PathVariable int offset,@PathVariable int
pagesize,@PathVariable String field)
    {
        return cs.getsort(offset,pagesize,field);
    }
}
```

Children_Service

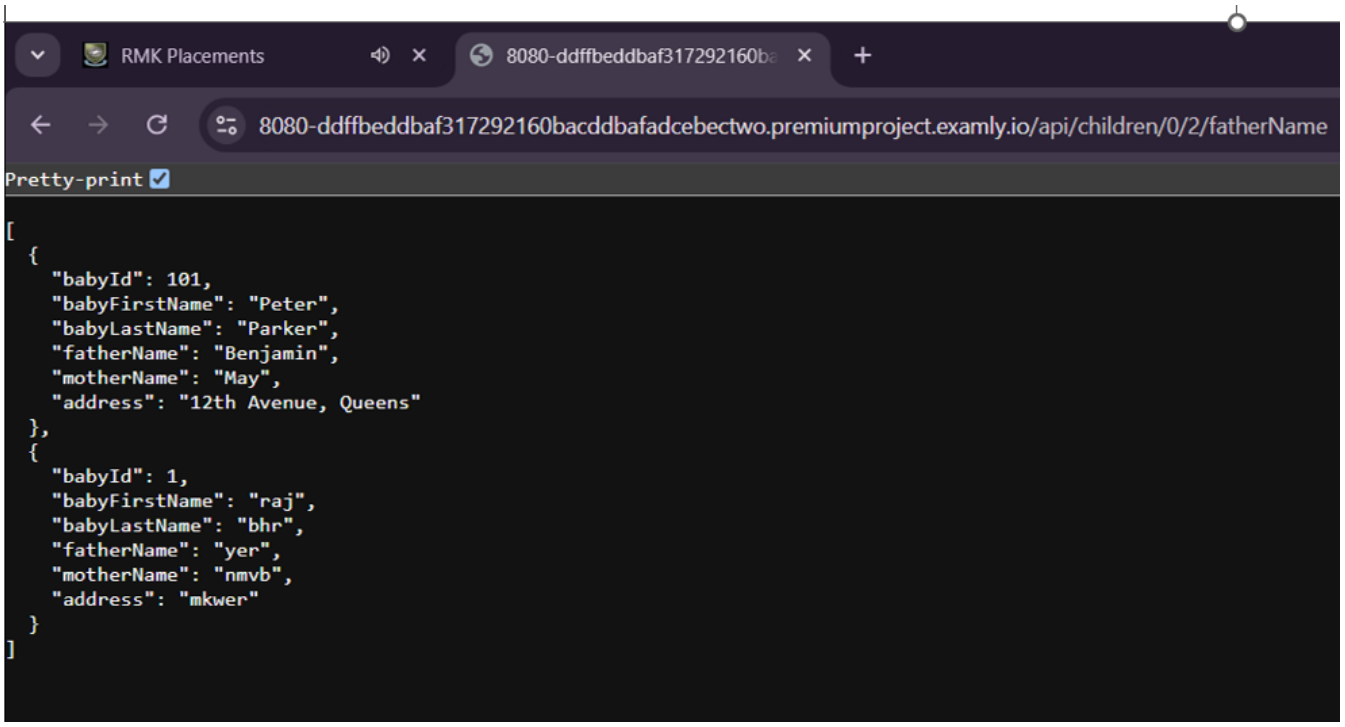
```
package com.example.springapp.service;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;
import org.springframework.stereotype.Service;
import com.example.springapp.model.Children;
import com.example.springapp.repository.ChildrenRepo;

@Service
public class ChildrenService {
    @Autowired
    ChildrenRepo cr;

    //post
    public Children create(Children c)
    {
        return cr.save(c);
    }
```

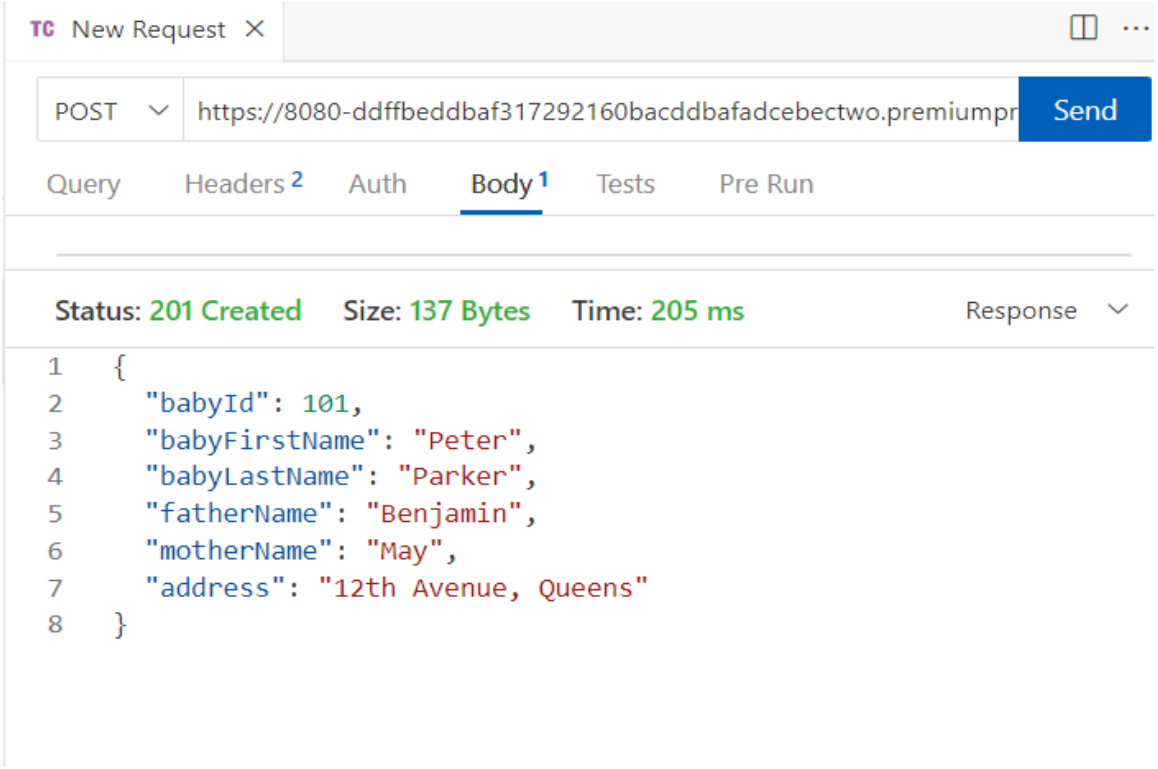
```
    //sorting by field
    public List<Children> sort(String field)
    {
        Sort sort=Sort.by(Sort.Direction.ASC,field);
        return cr.findAll(sort);
    }
    //pagination
    public List<Children> page(int pageSize,int pageNumber)
    {
        Pageable page=PageRequest.of(pageNumber, pageSize);
        return cr.findAll(page).getContent();
    }
    //sorting and pagination
    public List<Children> getsort(int pageNumber,int pageSize,String field)
    {
      return cr.findAll(PageRequest.of(pageNumber, pageSize)
          .withSort(Sort.by(Sort.Direction.ASC,field))).getContent();
    }
}
```

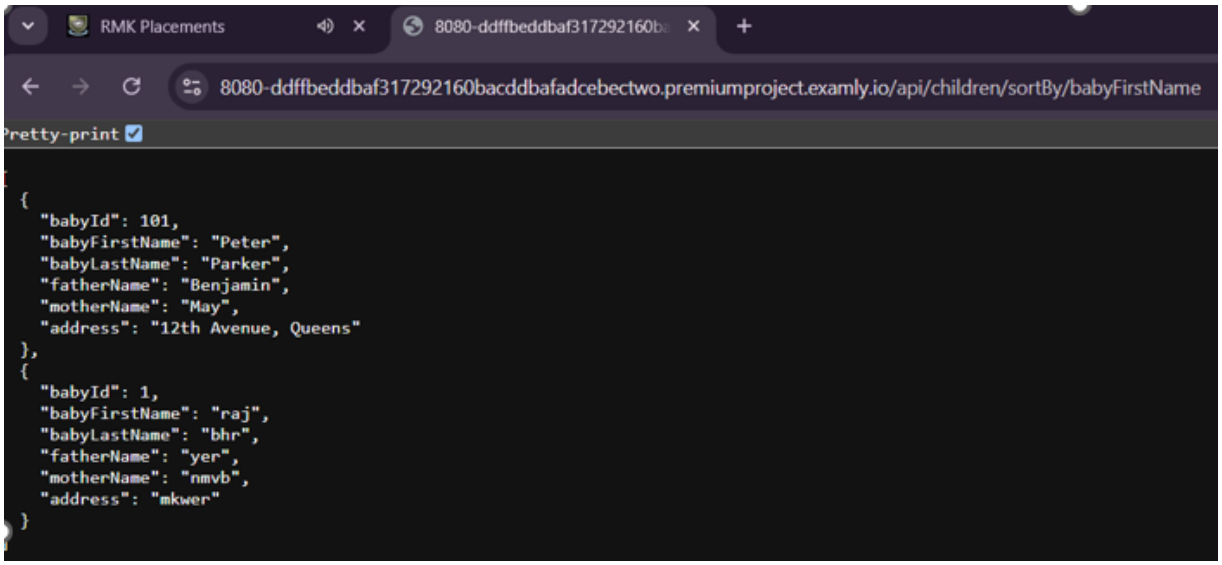
Children_Repository

```
package com.example.springapp.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.example.springapp.model.Children;

public interface ChildrenRepo extends JpaRepository<Children,Integer>{
   }
```

OUTPUT


RMK Placements
8080-ddffbeddbaf317292160ba
8080-ddffbeddbaf317292160bacddbafadcebectwo.premiumproject.examly.io/api/children/0/2/fatherName
Pretty-print
[
{
"babyId": 101,
"babyFirstName": "Peter",
"babyLastName": "Parker",
"fatherName": "Benjamin",
"motherName": "May",
"address": "12th Avenue, Queens"
},
{
"babyId": 1,
"babyFirstName": "raj",
"babyLastName": "bhr",
"fatherName": "yer",
"motherName": "nmvb",
"address": "mkwer"
}
]





Problem understanding and Design (3 Marks)	
Code Implementation (3 Marks)	
Output & Viva Explanation (4 Marks)	
Total (10 Marks)	

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.

Ex-9 Managing Person details using JPA

Title: **Managing Person details using JPA**

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

PROGRAM:

Person_Model

```
package com.example.springapp.model;

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

@Entity
public class Person {
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private int personId;
   private String firstName;
   private String lastName;
   private int age;
   private String gender;
   public Person() {
   }

   public Person(int personId, String firstName, String lastName, int age, String gender) {
      this.personId = personId;
      this.firstName = firstName;
      this.lastName = lastName;
      this.age = age;
      this.gender = gender;
   }
   public int getPersonId() {
      return personId;
   }
   public void setPersonId(int personId) {
      this.personId = personId;
   }
```

```
public String getFirstName() {
   return firstName;
}
public void setFirstName(String firstName) {
   this.firstName = firstName;
}
public String getLastName() {
   return lastName;
}
public void setLastName(String lastName) {
   this.lastName = lastName;
}
public int getAge() {
   return age;
}
public void setAge(int age) {
   this.age = age;
}
public String getGender() {
   return gender;
}
public void setGender(String gender) {
   this.gender = gender;
}
}
```

Person_Controller

```
package com.example.springapp.controller;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import com.example.springapp.model.Person;
import com.example.springapp.service.PersonService;

@RestController
public class PersonController {
   @Autowired
   private PersonService service;
   @PostMapping("/api/person")
   public ResponseEntity<?> addPerson(@RequestBody Person person){
      try {
         return new ResponseEntity<>(service.addPerson(person),HttpStatus.CREATED);
      } catch (Exception e) {
         return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
      }
   }

   @GetMapping("/api/person")
   public ResponseEntity<?>getPerson(){
      try {
         return new ResponseEntity<>(service.getPerson(),HttpStatus.OK);
      } catch (Exception e) {
         return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
      }
   }
```

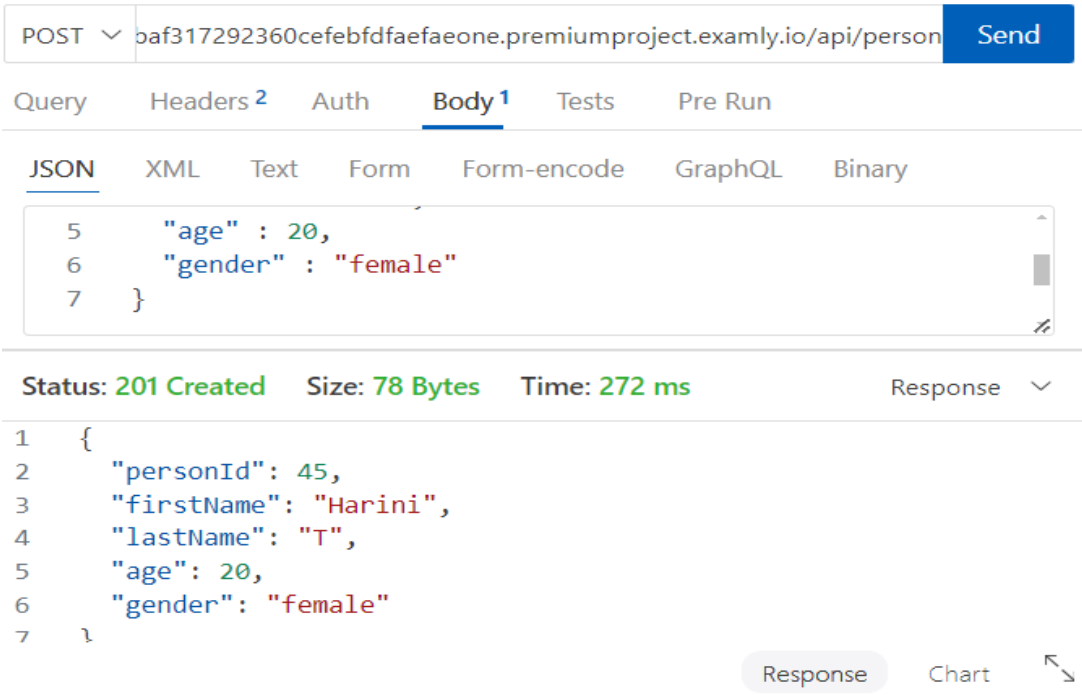
```
    @GetMapping("/api/person/byAge")
    public ResponseEntity<?>sort(@RequestParam String age){
        try {
            return new ResponseEntity<>(service.sort(age),HttpStatus.OK);
        } catch (Exception e) {
            return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }
}
```

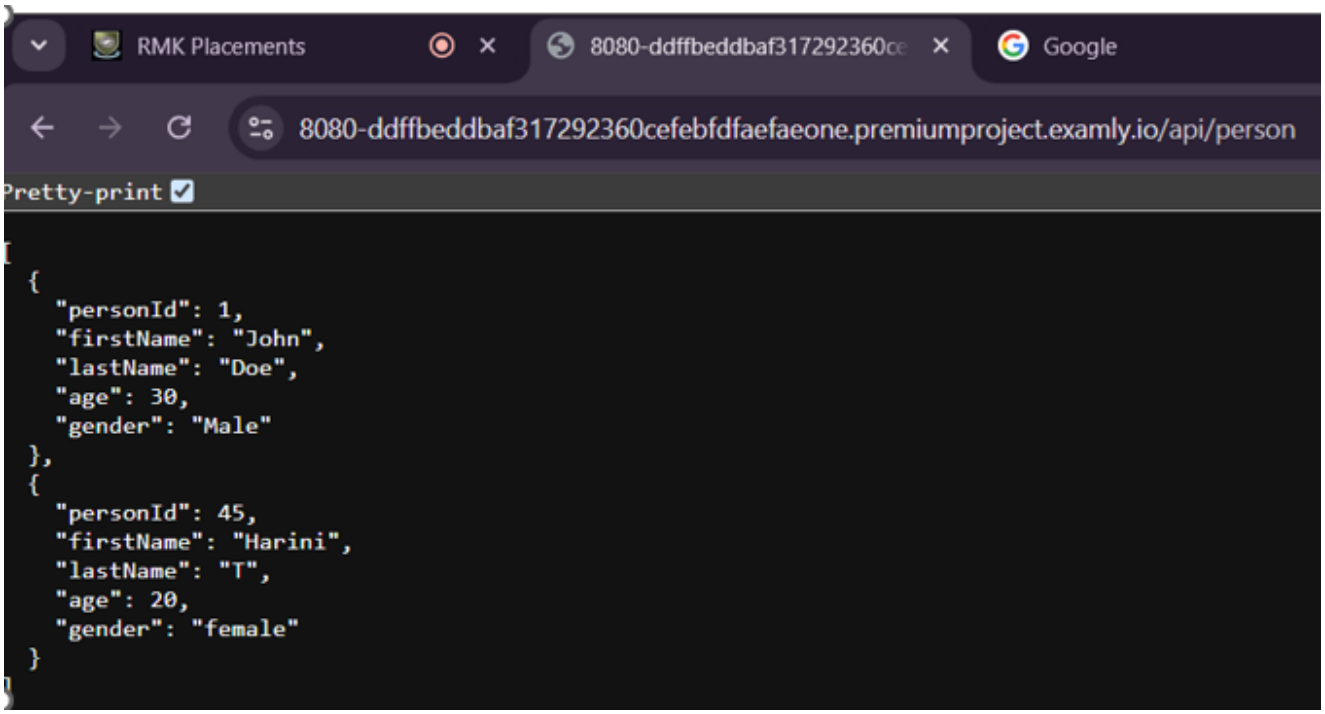
Person_Service

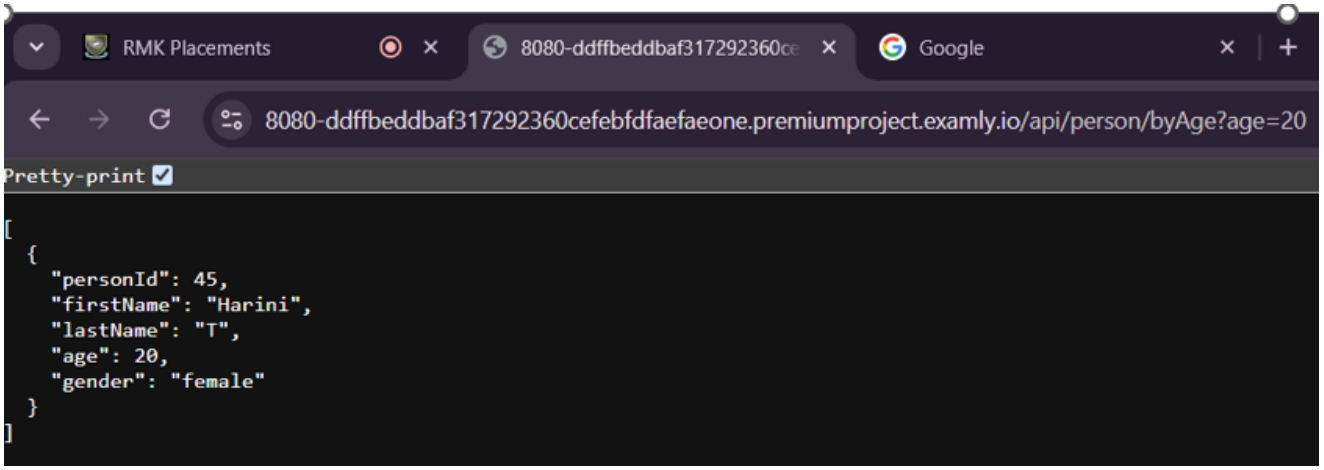
```
package com.example.springapp.service;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Sort;
import org.springframework.stereotype.Service;
import com.example.springapp.model.Person;
import com.example.springapp.repository.PersonRepo;

@Service
public class PersonService {
    @Autowired
    private PersonRepo repo;
    public Person addPerson(Person person) {
        return repo.save(person);
    }
    public List<Person> getPerson() {
        return repo.findAll();
    }
   public List <Person> sort(String age) {
        Sort sort = Sort.by(Sort.Direction.ASC,"age");
        return repo.findAll(sort);
    }
}
```

Person_Repository

```
package com.example.springapp.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.example.springapp.model.Person;

public interface PersonRepo extends JpaRepository<Person,Integer>{
}
```







Problem understanding and Design (3 Marks)	
Code Implementation (3 Marks)	
Output & Viva Explanation (4 Marks)	
Total (10 Marks)	

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.

Ex-10 Managing Person details using JPQL-specific search patterns

Title: Retrieve person details using JPQL

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
```

PROGRAM:

Person.java

```
package com.examly.springapp.model;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
@Entity
public class Person {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private long id;
   private String name;
   private int age;
   public Person()
   {
   }
```

```
public Person(long id, String name, int age) {
        this.id = id;
        this.name = name;
        this.age = age;
    }
    public long getId() {
        return id;
    }
    public void setId(long id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
   }
```

PersonController.java

```
package com.examly.springapp.controller;

import java.util.List;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import com.examly.springapp.model.Person;
import com.examly.springapp.service.PersonService;

@RestController
public class PersonController {
  private PersonService personService;
public PersonController(PersonService personService) {
   this.personService = personService;
}

@PostMapping("/person")
 public ResponseEntity<Person> addPerson(@RequestBody Person p)
{
   return new ResponseEntity<>(personService.addPerson(p),HttpStatus.CREATED);
 }

 @GetMapping("/person/startsWithName/{value}")
 public ResponseEntity<List<Person>> getPersonStartingName(@PathVariable String value)
 {
   return new ResponseEntity<>(personService.getPersonStartingName(value),HttpStatus.OK);
 }
 @GetMapping("/person/endsWithName/{value}")
 public ResponseEntity<List<Person>> getPersonEndingName(@PathVariable String value)
 {
   return new ResponseEntity<>(personService.getPersonEndingName(value),HttpStatus.OK);
```

```
 }
}
```

PersonRepo.java

```
package com.examly.springapp.repository;

import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.examly.springapp.model.Person;

@Repository

public interface PersonRepo extends JpaRepository<Person,Long>{
   List<Person> findByNameStartsWith(String value);
   List<Person> findByNameEndsWith(String value);
}
```

PersonService.java

```
package com.examly.springapp.service;

import java.util.List;
import org.springframework.stereotype.Service;
import com.examly.springapp.model.Person;
import com.examly.springapp.repository.PersonRepo;

@Service
public class PersonService {
  private PersonRepo personRepo;
public PersonService(PersonRepo personRepo) {
   this.personRepo = personRepo;
}
 public Person addPerson(Person p)
 {
   return personRepo.save(p);
 }
```

```
public List<Person> getPersonStartingName(String value)
{
  return personRepo.findByNameStartsWith(value);
}
public List<Person> getPersonEndingName(String value)
{
  return personRepo.findByNameEndsWith(value);
}
}
```

OUTPUT

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"

Problem understanding and Design (3 Marks)	
Code Implementation (3 Marks)	
Output & Viva Explanation (4 Marks)	
Total (10 Marks)	

RESULT:

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

Ex-11 Managing Person details using custom JPQL queries

Title : Managing Person details using custom JPQL queries

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
```

PROGRAM :

package com.example.springapp.model;

```
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
public class Person {
   @Id
   private int personId;
   private String firstName,lastName,gender,email;
   private int age;
   public Person() {
   }
 public Person(int personId, String firstName, String lastName, String gender, String email, int age) {
       this.personId = personId;
       this.firstName = firstName;
       this.lastName = lastName;
       this.gender = gender;
       this.email = email;
       this.age = age;
   }
   public int getPersonId() {
       return personId;
   }

   public void setPersonId(int personId) {
       this.personId = personId;
   }
   public String getFirstName() {
       return firstName;
   }
```

```
public void setFirstName(String firstName) {
   this.firstName = firstName;
}
public String getLastName() {
   return lastName;
}
public void setLastName(String lastName) {
   this.lastName = lastName;
}
public String getGender() {
   return gender;
}
public void setGender(String gender) {
   this.gender = gender;
}
public String getEmail() {
   return email;
}
public void setEmail(String email) {
   this.email = email;
}
public int getAge() {
   return age;
}
public void setAge(int age) {
   this.age = age;
}
}
```

PersonController.java

```
package com.example.springapp.controller;

import java.util.List;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import com.example.springapp.model.Person;
import com.example.springapp.service.PersonService;

@RestController
public class PersonController {
  private PersonService personService;
public PersonController(PersonService personService) {
   this.personService = personService;
}
   @PostMapping("/api/person")
   public ResponseEntity<Person> addPerson(@RequestBody Person p)
   {
      return new ResponseEntity<>(personService.addPerson(p),HttpStatus.CREATED);
   }
   @GetMapping("/api/person/byage/{age}")
   public ResponseEntity<List<Person>> getPersonByAge(@PathVariable int age)
   {
      return new ResponseEntity<>(personService.getPersonByAge(age),HttpStatus.OK);
   }
}
```

PersonService.java

```
package com.example.springapp.service;

import java.util.List;
import org.springframework.stereotype.Service;
import com.example.springapp.model.Person;
import com.example.springapp.repository.PersonRepo;
@Service
public class PersonService {
   private PersonRepo personRepo;
   public PersonService(PersonRepo personRepo) {
      this.personRepo = personRepo;
   }
   public Person addPerson(Person p)
   {
      return personRepo.save(p);
   }
   public List<Person> getPersonByAge(int age)
   {
      return personRepo.findPersonByAge(age);
   }
}
```

PerosnRepo.java

```
package com.example.springapp.repository;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.stereotype.Repository;
import com.example.springapp.model.Person;

@Repository
public interface PersonRepo extends JpaRepository<Person,Integer>{
 @Query("select p from Person p where p.age=?1")
   public List<Person> findPersonByAge(int age);
}
```

OUTPUT:

```
// 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
```

Problem understanding and Design (3 Marks)	
Code Implementation (3 Marks)	
Output & Viva Explanation (4 Marks)	
Total (10 Marks)	

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.

Ex-12 Managing Person details using Spring JPA with one-to-one mapping with Address Entity

TITLE : Spring JPA with one-to-one mapping

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

PROGRAM:

Address.java

```
package com.examly.springapp.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import com.fasterxml.jackson.annotation.JsonBackReference;

@Entity
public class Address {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;
  private String street;
  private String city;
  private String zipCode;
  public Person getPerson() {
     return person;
  }
```

```
    public void setPerson(Person person) {
        this.person = person;
    }
    @OneToOne
    @JsonBackReference
    private Person person;
public Address(){
}
public Address(Long id, String street, String city, String zipCode) {
    this.id = id;
    this.street = street;
    this.city = city;
    this.zipCode = zipCode;
}
public Long getId() {
    return id;
}
public void setId(Long id) {
    this.id = id;
}

public String getStreet() {
    return street;
}
public void setStreet(String street) {
    this.street = street;
}
public String getCity() {
    return city;
}
public void setCity(String city) {
    this.city = city;
}
public String getZipCode() {
    return zipCode;
}
public void setZipCode(String zipCode) {
    this.zipCode = zipCode;
```

```
}
}
```

Person.java

```
package com.examly.springapp.model;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import com.fasterxml.jackson.annotation.JsonManagedReference;

@Entity
public class Person {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
  private String name;
  private String email;
  private String phoneNumber;
  private String nationality;
  @OneToOne(mappedBy = "person",cascade = CascadeType.ALL)
  @JsonManagedReference
  private Address address;

public Person()
  {
  }

 public Person(Long id, String name, String email, String phoneNumber, String nationality, Address
address) {
   this.id = id;
   this.name = name;
   this.email = email;
   this.phoneNumber = phoneNumber;
   this.nationality = nationality;
   this.address = address;
}
```

```
public Long getId() {
  return id;
}
public void setId(Long id) {
  this.id = id;
}
public String getName() {
  return name;
}
public void setName(String name) {
  this.name = name;
}
public String getEmail() {
  return email;
}
public void setEmail(String email) {
  this.email = email;
}
public String getPhoneNumber() {
  return phoneNumber;
}
public void setPhoneNumber(String phoneNumber) {
  this.phoneNumber = phoneNumber;
}
public String getNationality() {
  return nationality;
}
public void setNationality(String nationality) {
  this.nationality = nationality;
}
public Address getAddress() {
  return address;
}
public void setAddress(Address address) {
  this.address = address;
}
 }
```

AddressController.java

```
package com.examly.springapp.controller;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import com.examly.springapp.model.Address;
import com.examly.springapp.model.Person;
import com.examly.springapp.service.AddressService;
import com.examly.springapp.service.PersonService;

@RestController
public class AddressController {
  private AddressService addressService;
  private PersonService personService;
    public AddressController(AddressService addressService, PersonService personService) {
      this.addressService = addressService;
      this.personService = personService;
  }

  @PostMapping("address/person/{personId}")
  public ResponseEntity<Address> postAddress(@RequestBody Address aobj,

 @PathVariable("personId") Long id)
  {
   Person pobj=personService.getPersonById(id).orElse(null);
      aobj.setPerson(pobj);
   addressService.postAddress(aobj);
   return new ResponseEntity<Address>(aobj,HttpStatus.CREATED);
  }
}
```

PersonController

```
package com.examly.springapp.controller;

import java.util.List;
import java.util.Optional;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import com.examly.springapp.model.Person;
import com.examly.springapp.service.PersonService;

@RestController
public class PersonController {
  private PersonService personService;
  public PersonController(PersonService personService) {
     this.personService = personService;
  }
 @PostMapping("/person")
 public ResponseEntity<Person> postPerson(@RequestBody Person obj)
 {
  Person pobj=personService.postPerson(obj);
  return new ResponseEntity<Person>(pobj,HttpStatus.CREATED);
 }

@GetMapping("/person")
public ResponseEntity<List<Person>> getPersons()
{
  List<Person> pobj=personService.getPersons();
  return new ResponseEntity<List<Person>>(pobj,HttpStatus.OK);
}
```

```
@GetMapping("/person/{personId}")
public ResponseEntity<Person> getPersonById(@PathVariable("personId") Long id)
{
  Optional<Person>  pobj=personService.getPersonById(id);
  return new ResponseEntity<>(pobj.get(),HttpStatus.OK);
}
}
```

PersonRepository

```
package com.examly.springapp.repository;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.examly.springapp.model.Person;
@Repository
public interface PersonRepository extends JpaRepository<Person,Long> {
 }
```

AddressRepository

```
package com.examly.springapp.repository;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.examly.springapp.model.Address;

@Repository
public interface AddressRepository extends JpaRepository<Address,Long> {
 }
```

PersonService

```
package com.examly.springapp.service;
import java.util.List;
import java.util.Optional;

import org.springframework.stereotype.Service;
import com.examly.springapp.model.Person;
import com.examly.springapp.repository.PersonRepository;
@Service
public class PersonService {
  private PersonRepository personRepo;
  public PersonService(PersonRepository personRepo) {
     this.personRepo = personRepo;
```

```
  }
  public Person postPerson(Person obj)
  {
     return personRepo.save(obj);
  }
 public List<Person> getPersons()
 {
  return personRepo.findAll();
 }
 public Optional<Person> getPersonById(Long id)
 {
  return personRepo.findById(id);
 }
}
```

AddressService.java

```
package com.examly.springapp.service;
import org.springframework.stereotype.Service;
import com.examly.springapp.model.Address;
import com.examly.springapp.repository.AddressRepository;

@Service
public class AddressService {
  private AddressRepository addressRepo;
  public AddressService(AddressRepository addressRepo) {
     this.addressRepo = addressRepo;
  }
  public Address postAddress(Address aobj)
  {
     return addressRepo.save(aobj);
  }
    }
```

OUTPUT:

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}

Problem understanding and Design (3 Marks)	
Code Implementation (3 Marks)	
Output & Viva Explanation (4 Marks)	
Total (10 Marks)	

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.

Ex-13 Managing Author and Book details using Spring JPA with one-to-many mapping

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

PROGRAM :

Author.java

```
package com.examly.springapp.model;

import java.util.List;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import com.fasterxml.jackson.annotation.JsonManagedReference;

@Entity
public class Author {
    @Id
    private int id;
    private String name;
    private String email;
    private String phoneNumber;
    private String address;
    @OneToMany(mappedBy = "author", cascade = CascadeType.ALL)
    @JsonManagedReference
    private List<Book> books;

    public Author()
    {

    }
    public Author(int id, String name, String email, String phoneNumber, String address, List<Book>
books) {
        this.id = id;
        this.name = name;
        this.email = email;
        this.phoneNumber = phoneNumber;
        this.address = address;
        this.books = books;
    }

    public int getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }
```

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

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }

    public void setPhoneNumber(String phoneNumber) {
        this.phoneNumber = phoneNumber;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public List<Book> getBooks() {
        return books;
    }

    public void setBooks(List<Book> books) {
        this.books = books;
    }
}
```

Book.java

```
package com.examly.springapp.model;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import com.fasterxml.jackson.annotation.JsonBackReference;

@Entity
public class Book {
    @Id
    //@GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String title;
    private String genre;
    private int publicationYear;
    private String isbn;
    private double price;
    @ManyToOne
    @JoinColumn(name = "authorId")
    @JsonBackReference
    private Author author;

    public Book()
    {

    }

    public Book(int id, String title, String genre, int publicationYear, String isbn, double price, Author
author) {
        this.id = id;
        this.title = title;
        this.genre = genre;
        this.publicationYear = publicationYear;
        this.isbn = isbn;
        this.price = price;
        this.author = author;
    }

    public int getId() {
        return id;
    }

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

```
    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getGenre() {
        return genre;
    }

    public void setGenre(String genre) {
        this.genre = genre;
    }

    public int getPublicationYear() {
        return publicationYear;
    }

    public void setPublicationYear(int publicationYear) {
        this.publicationYear = publicationYear;
    }

    public String getIsbn() {
        return isbn;
    }

    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public Author getAuthor() {
        return author;
    }

    public void setAuthor(Author author) {
        this.author = author;
    }

}
```

AuthorController.java

```
package com.examly.springapp.controller;

import java.util.List;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import com.examly.springapp.model.Author;
import com.examly.springapp.service.AuthorService;

@RestController
public class AuthorController {
    private AuthorService authorService;

    public AuthorController(AuthorService authorService) {
        this.authorService = authorService;
    }

 @PostMapping("/author")
 public ResponseEntity<Author> postAuthor(@RequestBody Author obj)
 {
     Author aobj=authorService.postAuthor(obj);
  if(aobj!=null)
   return new ResponseEntity<>(aobj,HttpStatus.CREATED);
  else
  return new ResponseEntity<>(aobj,HttpStatus.INTERNAL_SERVER_ERROR);
 }

@GetMapping("/author/{authorId}")
public ResponseEntity<Author> getAuthorById(@PathVariable("authorId") int authorId)
{
  Author aobj=authorService.getAuthorById(authorId);
  if (aobj!=null) {
    return new ResponseEntity<> (aobj,HttpStatus.OK);
  }
  return new ResponseEntity<>(aobj,HttpStatus.NOT_FOUND);
}
```

```
@GetMapping("/author")
public ResponseEntity<List<Author>> getAuthors()
{
//List<Author> lobj=authorService.getAuthors();
return new ResponseEntity<>(authorService.getAuthors(),HttpStatus.OK);
}

@PutMapping("/author/{authorId}")
public ResponseEntity<Author> putAuthor(@PathVariable("authorId") int authorId,@RequestBody
Author aobj)
{
Author obj=authorService.putAuthor(authorId,aobj);
if(obj!=null)
return new ResponseEntity<>(obj,HttpStatus.OK);
else
return new ResponseEntity<>(null,HttpStatus.NOT_FOUND);
}
}
```

BookController.java

```
package com.examly.springapp.controller;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import com.examly.springapp.model.Author;
import com.examly.springapp.model.Book;
import com.examly.springapp.service.AuthorService;
import com.examly.springapp.service.BookService;

@RestController
public class BookController {
    private BookService bookService;
    private AuthorService authorService;


  public BookController(BookService bookService, AuthorService authorService) {
      this.bookService = bookService;
      this.authorService = authorService;
    }
```

```
  @PostMapping("book/author/{authorId}")
  public ResponseEntity<Book>postBook( @PathVariable("authorId") int authorId,@RequestBody
Book b )
  {
    Author aobj=authorService.getAuthorById(authorId);
    b.setAuthor(aobj);
    return new ResponseEntity<Book>(bookService.postBook(b),HttpStatus.CREATED);
  }

  @DeleteMapping("book/{bookId}")
  public ResponseEntity<?> deleteBookById(@PathVariable int bookId)
  {
    boolean deleted=bookService.deleteBookById(bookId);
    if(deleted)
    return ResponseEntity.ok("Book deleted successfully");
    else
    return ResponseEntity.status(HttpStatus.NOT_FOUND).body("Book not found with ID:" + bookId);

  }
}
```

AuthorRepo.java

```
package com.examly.springapp.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.examly.springapp.model.Author;

@Repository
public interface AuthorRepository extends JpaRepository<Author,Integer>{

}
```

BookRepo.java

```
package com.examly.springapp.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.examly.springapp.model.Book;

@Repository
public interface BookRepository extends JpaRepository<Book,Integer> {

}
```

AuthorService.java

```
package com.examly.springapp.service;

import java.util.List;
import org.springframework.stereotype.Service;
import com.examly.springapp.model.Author;
import com.examly.springapp.repository.AuthorRepository;

@Service
public class AuthorService {

    private AuthorRepository authorRepo;

    public AuthorService(AuthorRepository authorRepo) {
        this.authorRepo = authorRepo;
    }

public Author postAuthor(Author obj)
   {

    return authorRepo.save(obj);
   }

   public Author getAuthorById(int id)
   {
    return authorRepo.findById(id).orElse(null);
   }

   public List<Author> getAuthors()
   {
    return authorRepo.findAll();
   }

   public Author putAuthor(int authorId,Author aobj)
   {
      Author obj=authorRepo.findById(authorId).orElse(null);
    if (obj!=null) {
       obj.setAddress(aobj.getAddress());
       obj.setEmail(aobj.getEmail());
       obj.setPhoneNumber(aobj.getPhoneNumber());
        return authorRepo.save(aobj);
    }
    return authorRepo.save(aobj);
   }
}
```

BOOKSERVICE.JAVA

```
package com.examly.springapp.service;

import org.springframework.stereotype.Service;
import com.examly.springapp.model.Book;
import com.examly.springapp.repository.BookRepository;

@Service
public class BookService {
    private BookRepository bookRepo;

    public BookService(BookRepository bookRepo) {
        this.bookRepo = bookRepo;
    }

    public Book postBook(Book book)
 {
   return bookRepo.save(book);
 }
public Book getBookById(int id)
{
   return bookRepo.findById(id).orElse(null);
}
 public Boolean deleteBookById(int bookId)
 {
   if(this.getBookById(bookId)==null)
   return false;
   bookRepo.deleteById(bookId);
   return true;
 } }
```

OUTPUT :

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}
```

Problem understanding and Design (3 Marks)	
Code Implementation (3 Marks)	
Output & Viva Explanation (4 Marks)	
Total (10 Marks)	

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.

Ex-14: Managing Employee and Address details using Spring JPA with one-one mapping using Criteria API

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
```

PROGRAM:

Address.java

```
package com.examly.springapp.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import com.fasterxml.jackson.annotation.JsonBackReference;

@Entity
public class Address {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private int id;
   private String city,street;
   @OneToOne
   @JsonBackReference
   private Employee employee;
   public Address()
   {

   }

   public Address(int id, String city, String street, Employee employee) {
      this.id = id;
      this.city = city;
      this.street = street;
      this.employee = employee;
   }

   public int getId() {
      return id;
   }

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

   public String getCity() {
      return city;
   }

   public void setCity(String city) {
      this.city = city;
   }
```

```
    public String getStreet() {
        return street;
    }

    public void setStreet(String street) {
        this.street = street;
    }

    public Employee getEmployee() {
        return employee;
    }

    public void setEmployee(Employee employee) {
        this.employee = employee;
    }
}
```

Employee.java

```
package com.examly.springapp.model;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import com.fasterxml.jackson.annotation.JsonManagedReference;

@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;
    @OneToOne(mappedBy = "employee",cascade = CascadeType.ALL)
    @JsonManagedReference
    private Address address;

   public Employee()
    {

    }
```

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

public int getId() {
   return id;
}

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

public String getName() {
   return name;
}

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

public Address getAddress() {
   return address;
}

public void setAddress(Address address) {
   this.address = address;
}


}
```

AddressController.java

```
package com.examly.springapp.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

import com.examly.springapp.model.Address;
import com.examly.springapp.model.Employee;
import com.examly.springapp.service.AddressService;
import com.examly.springapp.service.EmployeeService;
```

```
@RestController
public class AddressController {
   private AddressService addService;
   @Autowired
   private EmployeeService empService;
   public AddressController(AddressService addService) {
      this.addService = addService;
   }
   @PostMapping("/address/employee/{id}")
   public ResponseEntity<Address> addAddress(@RequestBody Address a,@PathVariable int id)
   {
      Employee eobj=empService.getEmployeeById(id);
      a.setEmployee(eobj);
      return new ResponseEntity<>(addService.addAddress(a),HttpStatus.CREATED);
   }
}
```

EmployeeController.java

```
package com.examly.springapp.controller;

import java.util.List;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

import com.examly.springapp.model.Employee;
import com.examly.springapp.service.EmployeeService;

@RestController
public class EmployeeController {
   private EmployeeService empService;

   public EmployeeController(EmployeeService empService) {
      this.empService = empService;
   }
   @PostMapping("/employee")
   public ResponseEntity<Employee> addEmployee(@RequestBody Employee e)
   {
      return new ResponseEntity<>(empService.addEmployee(e),HttpStatus.CREATED);
   }
   @GetMapping("/employees-inner-join")
   public ResponseEntity<List<Employee>> getAllEmployeesByInnerJoin()
   {
      return new ResponseEntity<>(empService.getAllEmployeesByInnerJoin(),HttpStatus.OK);
   }
```

```
    @GetMapping("/employees-left-outer-join")
    public ResponseEntity<List<Employee>> getAllEmployeesWithLeftOuterjoin()
    {
        return new ResponseEntity<>(empService.getAllEmployeesWithLeftOuterjoin(),HttpStatus.OK);
    }
}
```

AddressRepo.java

```
package com.examly.springapp.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.examly.springapp.model.Address;

@Repository
public interface AddressRepository extends JpaRepository<Address,Integer> {

}
```

EmployeeRepo.java

```
package com.examly.springapp.repository;

import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.stereotype.Repository;
import com.examly.springapp.model.Employee;

@Repository
public interface EmployeeRepository extends JpaRepository<Employee,Integer>{
    @Query("select e from Employee e inner join e.address a")
    List<Employee> findAllEmployeesWithInnerJoin();
    @Query("select e from Employee e left outer join e.address a")
    List<Employee> findAllEmployeesWithLeftOuterJoin();
}
```

AddressService.java

```
package com.examly.springapp.service;

import org.springframework.stereotype.Service;
import com.examly.springapp.model.Address;
import com.examly.springapp.repository.AddressRepository;

@Service
public class AddressService {
   private AddressRepository addRepo;

   public AddressService(AddressRepository addRepo) {
      this.addRepo = addRepo;
   }
   public Address addAddress(Address a)
   {
      return addRepo.save(a);
   }

}
```

EmployeeService.java

```
package com.examly.springapp.service;

import java.util.List;
import org.springframework.stereotype.Service;
import com.examly.springapp.model.Employee;
import com.examly.springapp.repository.EmployeeRepository;

@Service
public class EmployeeService {
   private EmployeeRepository empRepo;

   public EmployeeService(EmployeeRepository empRepo) {
      this.empRepo = empRepo;
   }
   public Employee addEmployee(Employee e)
   {
      return empRepo.save(e);
   }
   public Employee getEmployeeById(int id)
   {
      return empRepo.findById(id).orElse(null);
   }
```

```
    public List<Employee> getAllEmployeesByInnerJoin()
    {
        return empRepo.findAllEmployeesWithInnerJoin();
    }
    public List<Employee> getAllEmployeesWithLeftOuterjoin()
    {
        return empRepo.findAllEmployeesWithLeftOuterJoin();
    }
}
```

OUTPUT:

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
```

Problem understanding and Design (3 Marks)	
Code Implementation (3 Marks)	
Output & Viva Explanation (4 Marks)	
Total (10 Marks)	

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.

Ex-15 Managing Employee and Payroll details using JPA with one-to-one mapping with Swagger API

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.

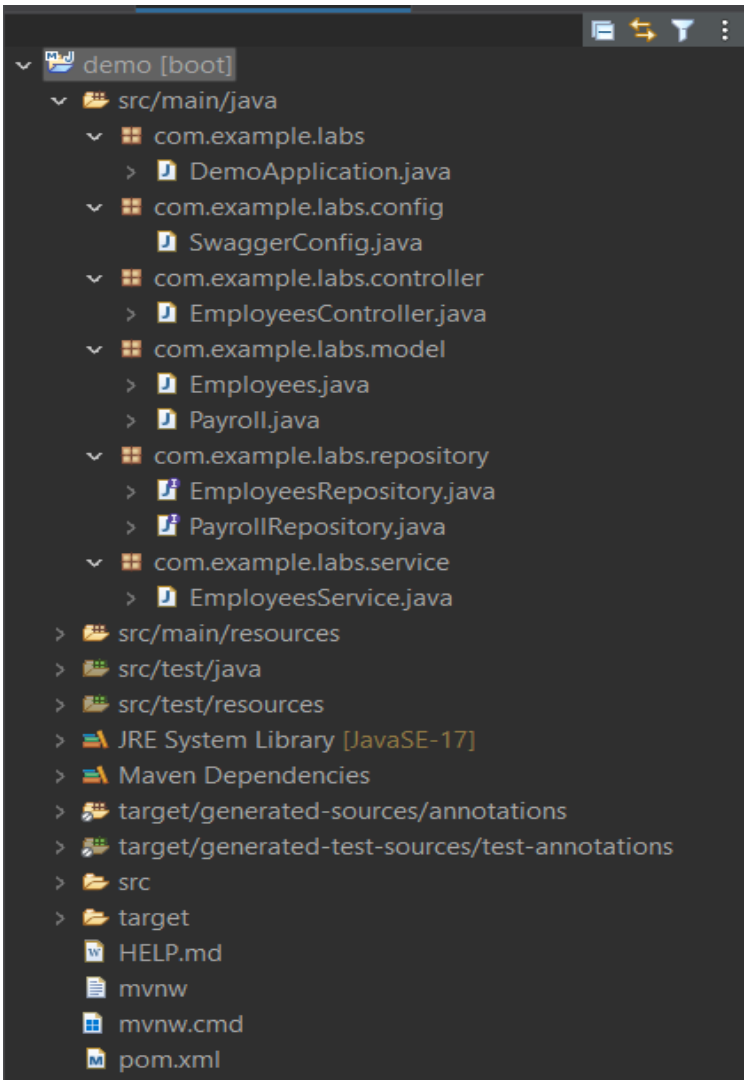
2. **PayrollController**

   - ❖ 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
```

PROGRAM:

```
package com.examly.springapp.configuration;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import springfox.documentation.builders.PathSelectors;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@Configuration
@EnableSwagger2
public class SwaggerConfig {
  @Bean
  public Docket api()
  {
   return new Docket(DocumentationType.SWAGGER_2)
   .select()
   .apis(RequestHandlerSelectors.basePackage("com.examly.springapp.controller"))
   .paths(PathSelectors.any())
   .build();
}
}

package com.examly.springapp.controller;

import java.util.List;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import com.examly.springapp.model.Employee;
import com.examly.springapp.model.Payroll;
import com.examly.springapp.service.EmployeeService;
```

```
@RestController
public class EmployeeController {
   private EmployeeService employeeService;

   public EmployeeController(EmployeeService employeeService) {
      this.employeeService = employeeService;
   }

 @PostMapping("/employee")
 public ResponseEntity<Employee> postEmployee(@RequestBody Employee obj)
 {
 Employee eobj=employeeService.postEmployee(obj);
 return new ResponseEntity<Employee>(eobj,HttpStatus.CREATED);
 }

@GetMapping("/employee")
public ResponseEntity<List<Employee>> getEmployees()

   {
    List<Employee> lobj=employeeService.getEmployees();
    return new ResponseEntity<List<Employee>>(lobj, HttpStatus.OK);
   }

 @GetMapping("/employee/{employeeId}")
 public ResponseEntity<Employee> getEmployeeById(@PathVariable Long employeeId)
 {
   Employee obj=employeeService.getEmployeeById(employeeId);
   return new ResponseEntity<Employee>(obj,HttpStatus.OK);
 }

}
```

```
package com.examly.springapp.controller;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import com.examly.springapp.model.Payroll;
import com.examly.springapp.service.PayrollService;

@RestController
public class PayRollController {
  private PayrollService payrollService;

public PayRollController(PayrollService payrollService) {
   this.payrollService = payrollService;
}
@PostMapping("/employee/{employeeId}/payroll")
public ResponseEntity<Payroll> postPayroll(@PathVariable Long employeeId, @RequestBody Payroll
payroll)
{
   Payroll obj=payrollService.postPayroll(employeeId,payroll);
   return new ResponseEntity<Payroll>(obj,HttpStatus.CREATED);
}
@GetMapping("employee/{employeeId}/payroll")
public ResponseEntity<Payroll> getPayroll(@PathVariable Long employeeId)
{
   Payroll pay=payrollService.getPayroll(employeeId);
   return new ResponseEntity<Payroll>(pay,HttpStatus.OK);
}
}
```

```
package com.examly.springapp.model;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import com.fasterxml.jackson.annotation.JsonManagedReference;

@Entity
public class Employee {
    @Id
    private Long employeeId;
    private String employeeNmae;
    private int age;
    private Long mobile;
    @OneToOne(mappedBy = "employee" ,cascade = CascadeType.ALL)
   @JsonManagedReference
    private  Payroll payroll;

public Employee()
{

}
public Employee(Long employeeId, String employeeNmae, int age, Long mobile, Payroll payroll) {
    this.employeeId = employeeId;
    this.employeeNmae = employeeNmae;
    this.age = age;
    this.mobile = mobile;
    this.payroll = payroll;
}
public Long getEmployeeId() {
    return employeeId;
}

public void setEmployeeId(Long employeeId) {
    this.employeeId = employeeId;
}
```

```
public String getEmployeeNmae() {
   return employeeNmae;
}

public void setEmployeeNmae(String employeeNmae) {
   this.employeeNmae = employeeNmae;
}

public int getAge() {
   return age;
}

public void setAge(int age) {
   this.age = age;
}

public Payroll getPayroll() {
   return payroll;
}

public void setPayroll(Payroll payroll) {
   this.payroll = payroll;
}

public Long getMobile() {
   return mobile;
}

public void setMobile(Long mobile) {
   this.mobile = mobile;
}
}
```

```
package com.examly.springapp.model;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.OneToOne;

import com.fasterxml.jackson.annotation.JsonBackReference;

@Entity
public class Payroll {
   @Id
    private Long payrollId;
   private double amount;
   private int noOfDaysWorked;
   @OneToOne
   @JsonBackReference
   private Employee employee;
  public Payroll()
  {

  }
public Payroll(Long payrollId, double amount, int noOfDaysWorked, Employee employee) {
    this.payrollId = payrollId;
    this.amount = amount;
    this.noOfDaysWorked = noOfDaysWorked;
    this.employee = employee;
}

public Long getPayrollId() {
    return payrollId;
}
public void setPayrollId(Long payrollId) {
    this.payrollId = payrollId;
}
public double getAmount() {
    return amount;
}
```

```
public void setAmount(double amount) {
   this.amount = amount;
}
public int getNoOfDaysWorked() {
   return noOfDaysWorked;
}
public void setNoOfDaysWorked(int noOfDaysWorked) {
   this.noOfDaysWorked = noOfDaysWorked;
}

public Employee getEmployee() {
   return employee;
}
public void setEmployee(Employee employee) {
   this.employee = employee;
}

}
```

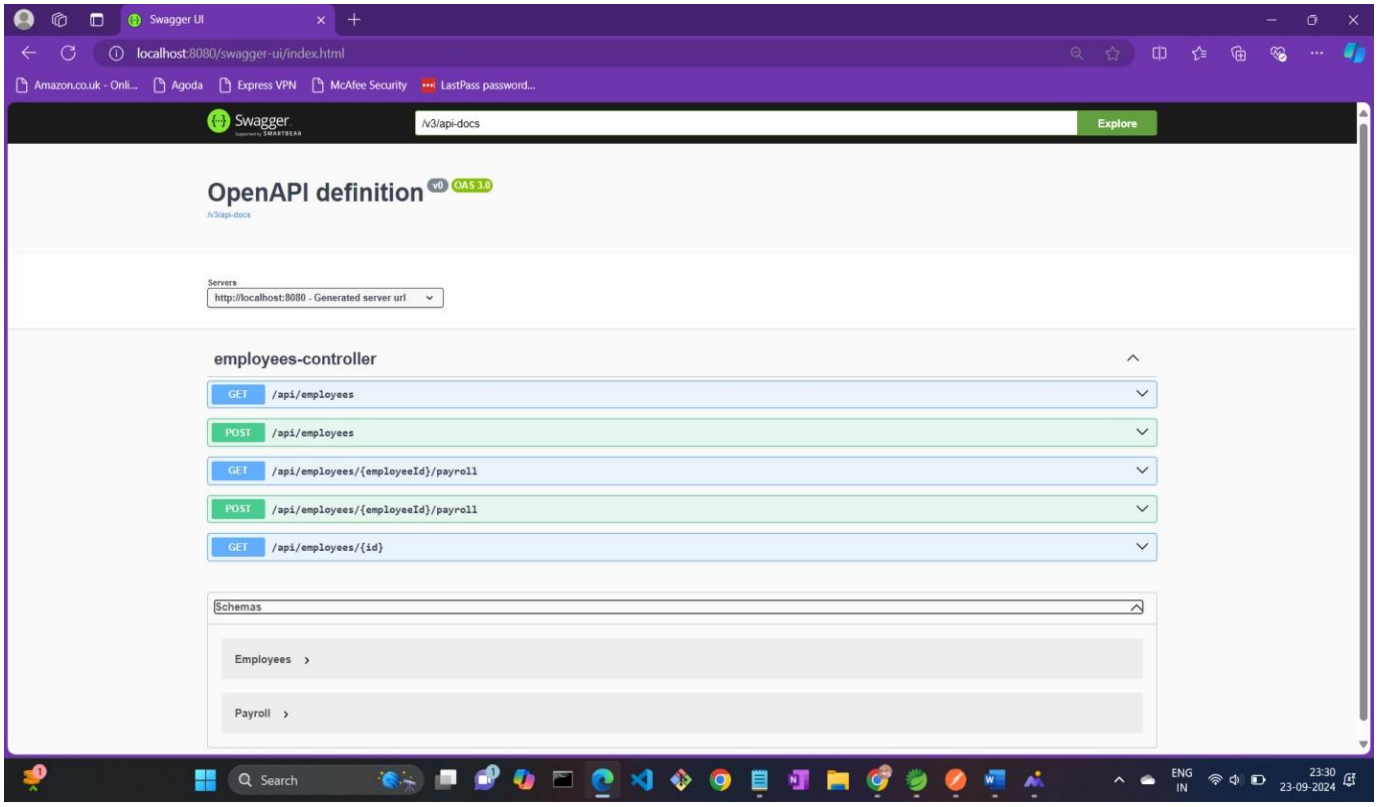
```
package com.examly.springapp.repository;

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

import com.examly.springapp.model.Employee;

@Repository
public interface EmployeeRepo extends JpaRepository<Employee,Long>{

}
```

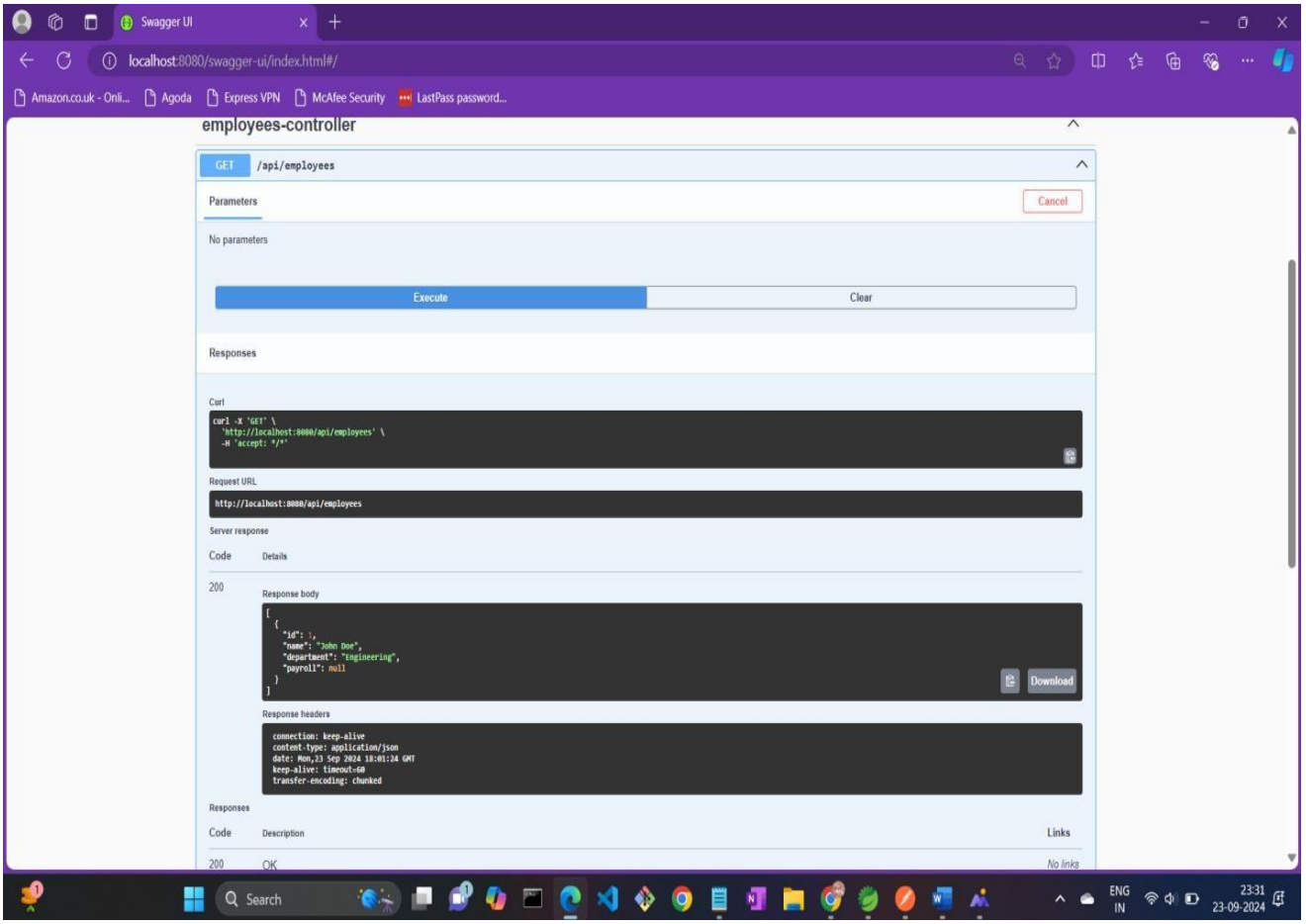
```
package com.examly.springapp.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.examly.springapp.model.Payroll;

@Repository
public interface PayrollRepo extends JpaRepository<Payroll,Long>{
}
```

```
package com.examly.springapp.service;

import java.util.HashMap;
import java.util.Map;
import org.springframework.stereotype.Service;
import com.examly.springapp.model.Payroll;
import com.examly.springapp.repository.PayrollRepo;

@Service
public class PayrollService {
    private PayrollRepo payrollRepo;

   public PayrollService(PayrollRepo payrollRepo) {
        this.payrollRepo = payrollRepo;
    }
private Map<Long,Payroll> map=new HashMap<>();
public Payroll postPayroll(Long employeeId,Payroll payroll)
{
    map.put(employeeId,payroll);
    return payroll;
}
public Payroll getPayroll(Long employeeId)
{
    return payrollRepo.findById(employeeId).orElse(null);
}
}
```

```
package com.examly.springapp.service;

import java.util.List;
import org.springframework.stereotype.Service;
import com.examly.springapp.model.Employee;
import com.examly.springapp.repository.EmployeeRepo;

@Service
public class EmployeeService {
    private EmployeeRepo employeeRepo;

    public EmployeeService(EmployeeRepo employeeRepo) {
        this.employeeRepo = employeeRepo;
    }

  public Employee postEmployee(Employee obj)
  {
    return employeeRepo.save(obj);
  }
  public List<Employee >getEmployees()
  {
    return employeeRepo.findAll();
  }
  public Employee getEmployeeById(Long employeeId)
  {
    return employeeRepo.findById(employeeId).orElse(null);
  }
}
```

OUTPUT:


Swagger UI
localhost:8080/swagger-ui/index.html
Amazon.co.uk - Onli...
Agoda
Express VPN
McAfee Security
LastPass password...
Swagger
/v3/api-docs
Explore
OpenAPI definition
v0
OAS 3.0
/v3/api-docs
Servers
http://localhost:8080 - Generated server url
employees-controller
GET /api/employees
POST /api/employees
GET /api/employees/{employeeId}/payroll
POST /api/employees/{employeeId}/payroll
GET /api/employees/{id}
Schemas
Employees
Payroll
Search
ENG
IN
23:30
23-09-2024



Problem understanding and Design (3 Marks)	
Code Implementation (3 Marks)	
Output & Viva Explanation (4 Marks)	
Total (10 Marks)	

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.

Ex-16 Managing Person details by integrating comprehensive logging capabilities

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)

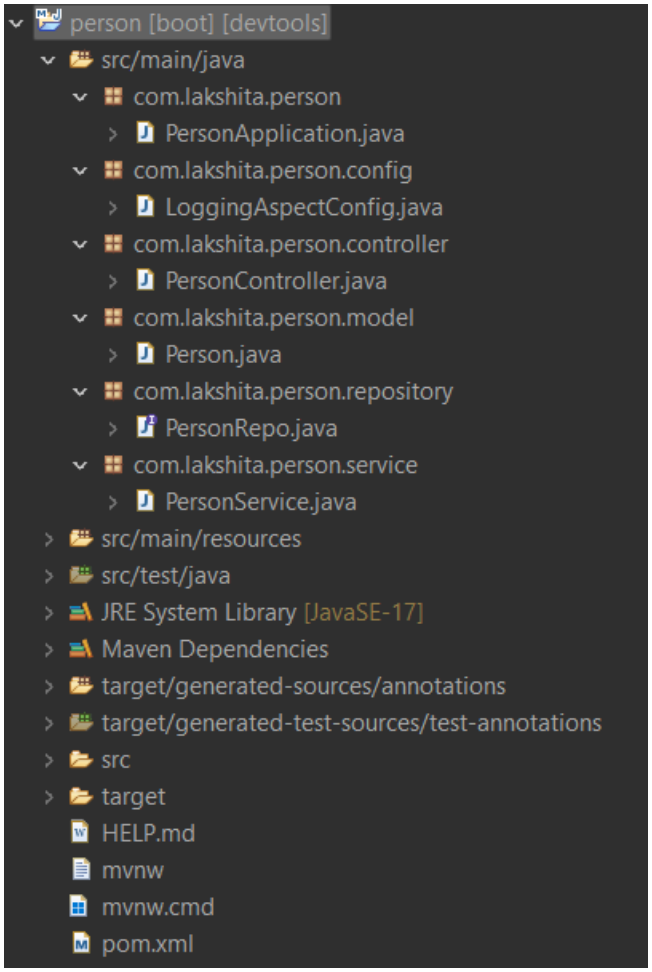
logging.level.root=INFO

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)

PROGRAM:

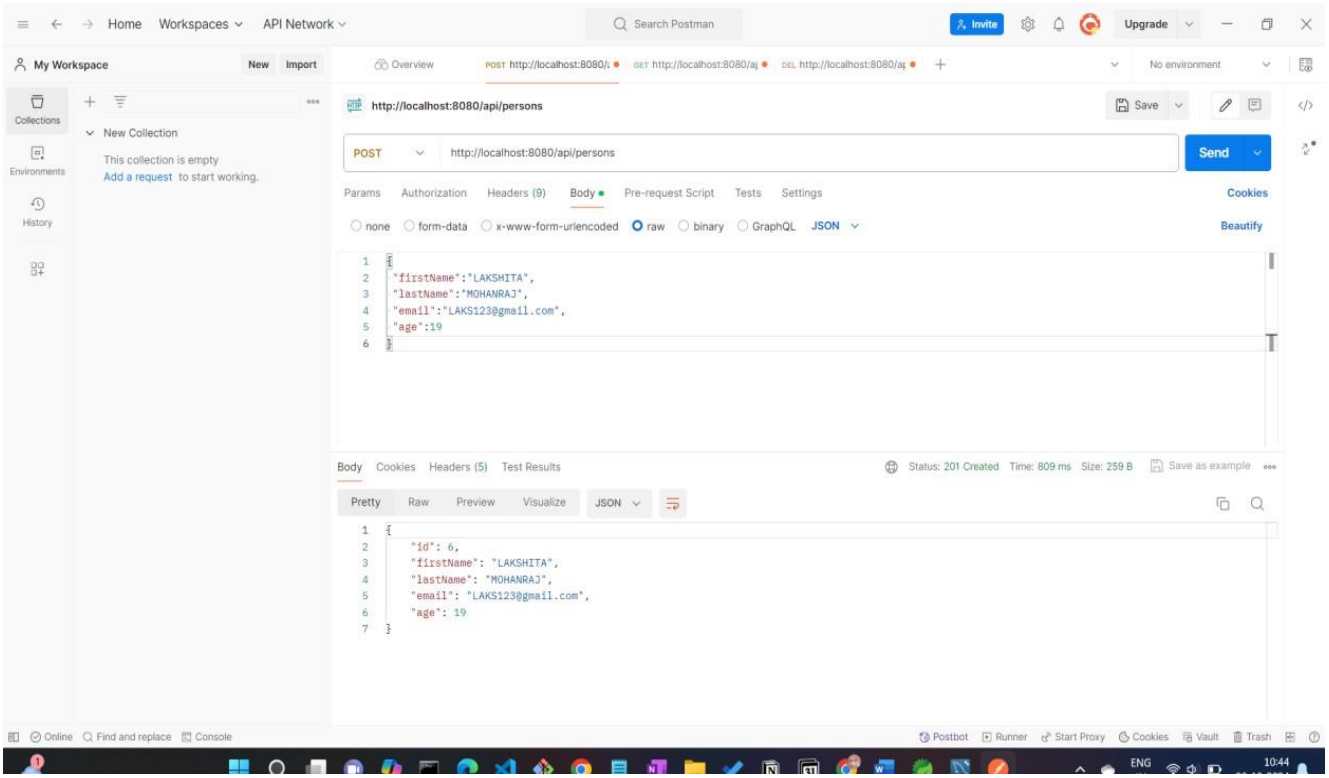
```
package com.examly.springapp.model;

import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Person {
    @Id
    private int id;
    private String firstName,lastName;
    public Person()
    {

    }
    public Person(int id, String firstName, String lastName) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

```
package com.examly.springapp.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.examly.springapp.model.Person;

@Repository
public interface PersonRepository extends JpaRepository <Person,Integer> {

}


package com.examly.springapp.service;

import java.util.List;
import org.springframework.stereotype.Service;
import com.examly.springapp.model.Person;
import com.examly.springapp.repository.PersonRepository;

@Service
public class PersonService {
   private PersonRepository personRepo;

   public PersonService(PersonRepository personRepo) {
      this.personRepo = personRepo;
   }
   public Person postPerson(Person p)
   {
      return personRepo.save(p);
   }
   public List<Person> getAllPersons()
   {
      return personRepo.findAll();
   }
}
```

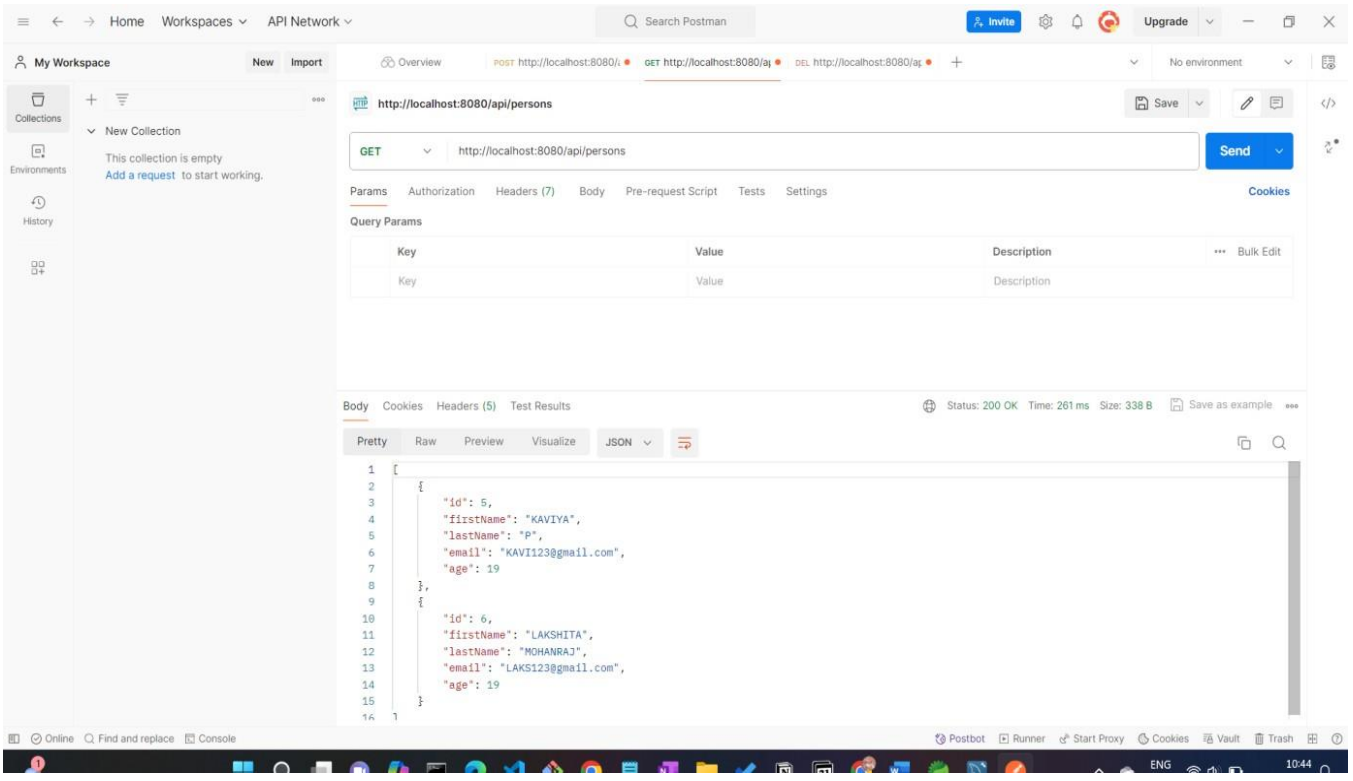
```
package com.examly.springapp.controller;

import java.util.List;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RestController;
import com.examly.springapp.model.Person;
import com.examly.springapp.service.PersonService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@RestController
public class PersonController {
    private static final Logger lobj=LoggerFactory.getLogger(PersonController.class);
    private PersonService personService;

    public PersonController(PersonService personService) {
        this.personService = personService;
    }

    @PostMapping("/persons")
    public ResponseEntity<Person> postPerson(Person p)
    {
     try {
        lobj.info("POST Request received for /persons");
        return new ResponseEntity<>(personService.postPerson(p),HttpStatus.OK);
     } catch (Exception e) {
     return new ResponseEntity<>(HttpStatus.OK);
     }
    }

    @GetMapping("/persons")
    public ResponseEntity<List<Person>> getAllPersons()
    {
        try {
            lobj.info("GET Request received for /persons");
            return new ResponseEntity<>(personService.getAllPersons(),HttpStatus.OK);
        } catch (Exception e) {
           return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }
}
```

OUTPUT:

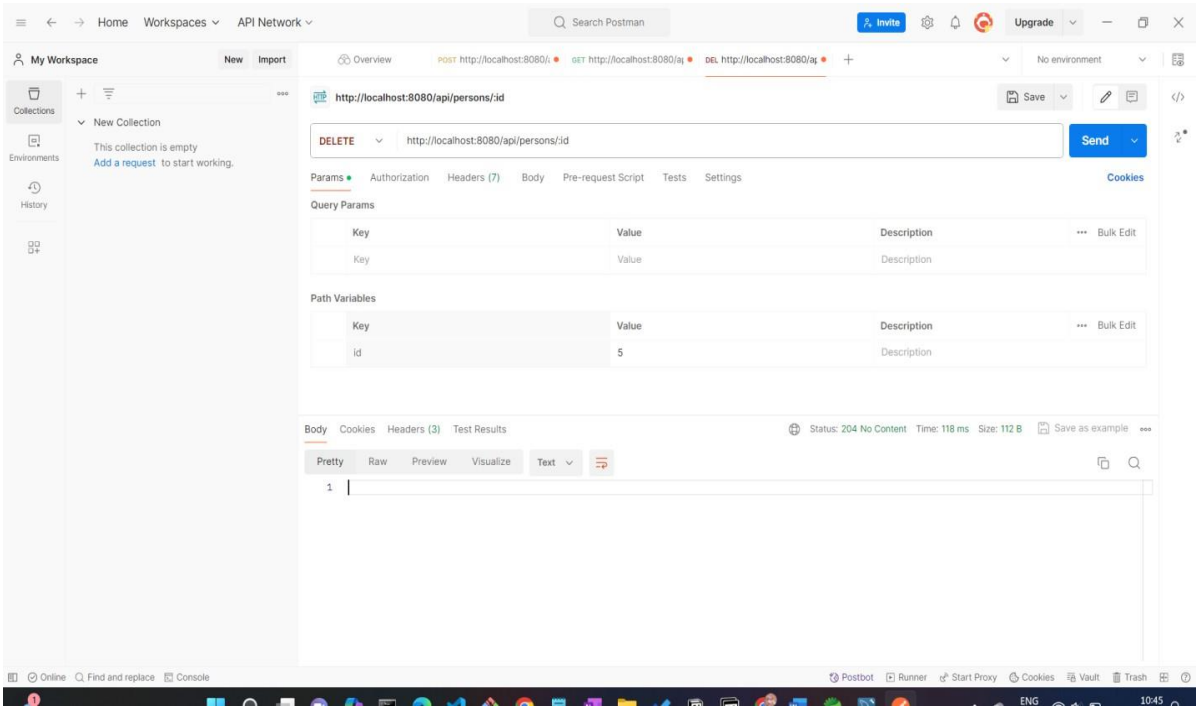
Postman for API Calls Test - Post

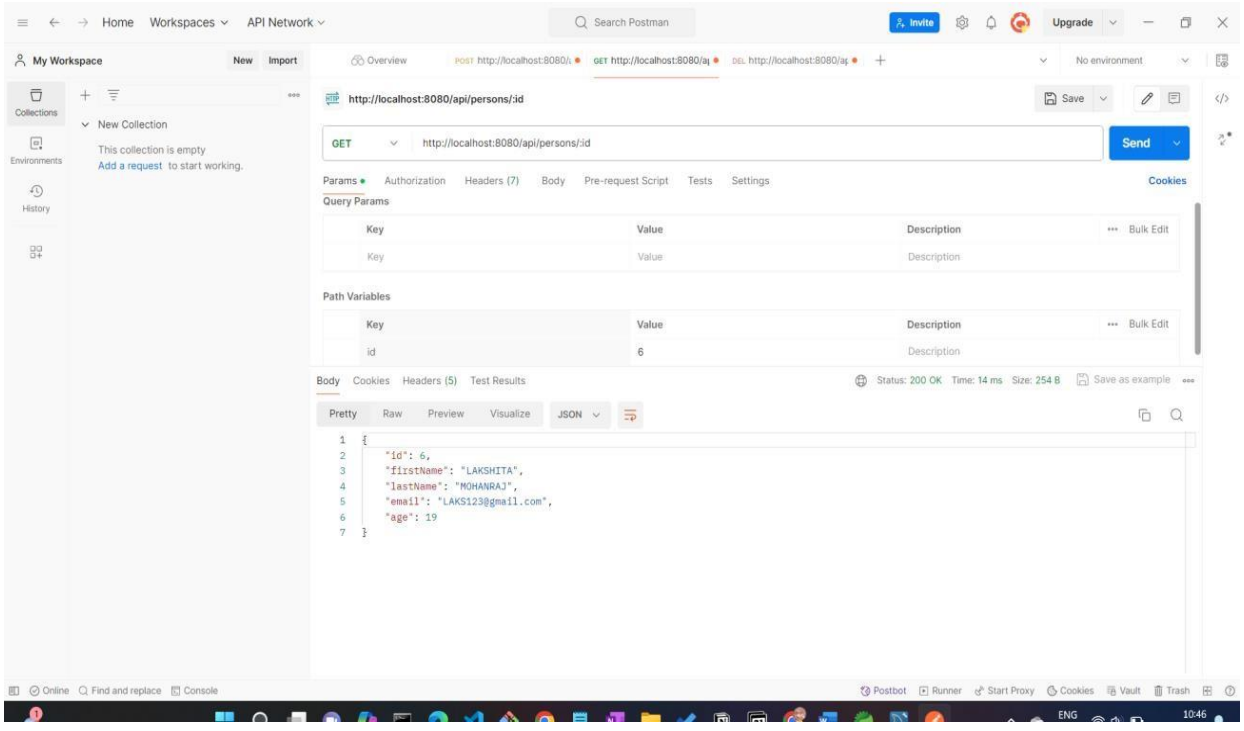


Postman for API Calls Test - Get



Postman for API Calls Test - Delete





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=?

Problem understanding and Design (3 Marks)	
Code Implementation (3 Marks)	
Output & Viva Explanation (4 Marks)	
Total (10 Marks)	

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.

Ex-17 Implementation of AOP

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

PROGRAM :

MyAspect.java

```
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

@Component
@Aspect
public class MyAspect {

   @Before("execution(* MyService.doSomething(..))")
   public void beforeAdvice() {
      System.out.println("Before advice: Executing before doSomething()");
   }

   @After("execution(* MyService.doSomething(..))")
   public void afterAdvice() {
      System.out.println("After advice: Executing after doSomething()");
   }

   @Around("execution(* MyService.doSomething(..))")
   public Object aroundAdvice(ProceedingJoinPoint joinPoint) throws Throwable {
      System.out.println("Around advice: Executing before doSomething()");
      Object result = joinPoint.proceed(); // Execute the target method
      System.out.println("Around advice: Executing after doSomething()");
      return result;
   }
}
```

MyService.java

```
import org.springframework.stereotype.Service;

@Service
public class MyService {
    public void doSomething() {
        System.out.println("Inside MyService.doSomething()");
    }
}
```

App.java

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class App {
    public static void main(String[] args) {
        // Initialize the Spring application context
        try (AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class)) {
            // Retrieve the MyService bean
            MyService myService = context.getBean(MyService.class);
            // Call the doSomething method
            myService.doSomething();
        }
    }
}
```

AppConfig.java

```
import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.EnableAspectJAutoProxy;


@Configuration
@ComponentScan(basePackages = " com.examly.springapp")

@EnableAspectJAutoProxy

public class AppConfig {

}
```

Sample Output:

```
System.out.println("Before advice: Executing before doSomething()");
System.out.println("Around advice: Executing before doSomething()");
System.out.println("Around advice: Executing after doSomething()");
System.out.println("After advice: Executing after doSomething()");
```

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)
```

Problem understanding and Design (3 Marks)	
Code Implementation (3 Marks)	
Output & Viva Explanation (4 Marks)	
Total (10 Marks)	

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.