PART - A

1.	Construct the code snippet for performing a Select, Update, and Delete operation using JPQL in JPA.
2.	Illustrate how to execute SQL queries using the @Query annotation in JPA.
3.	Infer the purpose of establishing a unidirectional OneToOne relationship mapping with JPA.
4.	Define lazy loading in the context of JPA.
5.	Mention the significance of establishing a OneToMany relationship with JPA.
6.	Write the steps involved in inserting a record into a database table that has a OneToMany relationship using JPA.
7.	Enumerate the role of Swagger UI in Spring Boot applications and its impact on API documentation and testing.
8.	Evaluate the importance of logging in Spring Boot application.
9.	Demonstrate the process of logging request and response JSON payloads in a Spring Boot application.
10.	Provide a real-world scenario of AOP (Aspect-Oriented Programming) to show how it is implemented in Spring Boot applications.
11.	Mention the significance of using JPQL for executing queries in JPA.
12.	Solve a JPQL query to fetch products whose names start with the letter 'A' and another JPQL query to retrieve customers whose email addresses end with '@iamneo.com.
13.	Construct an example for join queries in JPA.
14.	Analyze the significance of establishing a bidirectional OneToOne relationship with JPA.
15.	Illustrate the process of inserting a record into a database table that has a OneToOne relationship using JPA.
16.	Mention the role of the cascade attribute in JPA when working with OneToOne and OneToMany relationships.
17.	State how OpenUI is utilized in a Spring Boot application.
18.	Enumerate how to change the log level in a Spring Boot application.
19.	Correlate the use of logging properties in configuring logging behavior in a Spring Boot application.
20.	Infer how method parameters are utilized in BeforeAdvice in Aspect-Oriented Programming (AOP).

PART – B

Demonstrate POST and GET operations for managing Person details using JPA methods via RESTful APIs.

Functional Requirements:
create a class named "**PersonController**".
create a class named "**Person**" with the following attributes:

1. personId - int
2. firstName - String
3. lastName - String

4. age - int
5. gender - String

Implement getters, setters and constructors for the corresponding attributes.
create an interface named "**PersonRepo**".
create a class named "**PersonService**".

API:
POST - "/api/person" - Returns response status 201 with person object on successful creation or else 500
GET - "/api/person" - Returns response status 200 with List<Person> objects on successful retrieval or else 500
GET - "/api/person/byAge" - Returns response status 200 with List<Person> objects filtered by age based on the request parameter key as age on successful retrieval or else 500.

2. Illustrate POST and GET operations for managing Course details using JPA methods via RESTful APIs.

Functional Requirements:
create a class named "**CourseController".**
create a class named **"Course"** with the following attributes:

1. courseId - int
2. courseName - String
3. creditHours - int
4. instructor - String
5. preRequisites - String
6. maxCapacity - int
7. enrolledStudents - String

Implement getters**,** setters and constructors for the corresponding attributes.
create an interface named **"CourseRepo".**
create a class file named **"CourseService".**

API Endpoints:
POST - "/api/course" - Returns response status 201 with course object on successful creation or else 500.
GET - "/api/course" - Returns response status 200 with List<Course> object on successful retrieval or else 500.
GET - "/api/course/{courseName}" - Returns a response status of 200 with a List<Course> object filtered by the courseName passed in the request parameter upon successful retrieval or else 500.

3. Build a web application that facilitates POST and GET operations for managing Door details using JPA methods via RESTful APIs.

Functional Requirements:
create a class named "**DoorController".**
create a class named "**Door"** with the following attributes:

1. doorId - int
2. location - String
3. accessCode - String
4. accessType - String

Implement getters, setters and constructors for the corresponding attributes.
Inside repository folder, create an interface named **"DoorRepo".**
Inside service folder, create a class file named **"DoorService".**

API:
POST - "/api/door" - Returns response status 201 with door object on successful creation or else 500
GET - "/api/door" - Returns response status 200 with List <Door> objects on successful retrieval or else 500
GET - "/api/door/{doorId}" - Returns response status 200 with door object on successful retrieval or else 500
GET - "/api/door/accesstype/{accessType}" - Returns response status 200 with List<Door> object filtered by accessType on successful retrieval or else 500.

4. Build a web application that facilitates POST and GET operations for managing Car details using JPA methods via RESTful APIs.
Functional Requirements:
create a class named "**CarController".**
create a class named "**Car"** with the following attributes:

1. carId - int
2. carName - String
3. carType - String
4. owners - int
5. currentOwnerName - String
6. mobile - long
7. address - String

Implement getters, setters and constructors for the corresponding attributes.
create an interface named **"CarRepo".**
create a class file named **"CarService".**
API Endpoints:
POST - "/api/car" - Returns response status 201 with car objects on successful creation or else 500.
GET - "/api/car" - Returns response status 200 with List<Car> objects on successful retrieval or else 500.
GET - "/api/car/distinct" - Returns a response status of 200 with the distinct car object filtered by currentOwnerName and address, which are passed as request parameters, upon successful retrieval or else 500.

5. Retrieve product details using JPQL with conditions for names starting or ending with specific patterns.

Functional Requirements:
create a class named "Product" with the following attributes:

1. id - int
2. productName - String
3. price - int
4. category - String

Implement getters, setters and constructors for these attributes.
create an interface named "ProductRepo".
create a class named "ProductService".

create a class named "ProductController".

API Endpoints:
POST - "/product" - Returns response status 201 with product object on successful creation or else 500.
GET - "/product/distinct/startwith/{name}" - Retrieves distinct List<String> that start with a specified letter and have a status code of 200 or else 500.
GET - "/product/distinct/endwith/{name}" - Retrieves distinct List<String> that ends with a specified letter and have a status code of 200 or else 500.
GET - "/product/sortBy/price" - Retrieve List<Product> objects sorted by price in ascending order with status code 200 or else 500.

6. Describe the process of executing queries using JPA and JPQL in a Spring Boot application. Discuss the advantages of using JPQL over native SQL queries and provide examples of JPQL queries for common CRUD operations. Explain how to use the @Query annotation to define custom JPQL queries in repository methods. Compare and contrast JPQL queries with criteria queries in terms of flexibility and readability.

7. Build a web application that facilitates POST and GET operations for managing Recipe details using JPQL via RESTful APIs.
Functional Requirements:
create a class named "**RecipeController**".
create a class named "**Recipe**" with the following attributes:

1. recipeId - int
2. title - String
3. ingredients - String
4. instructions - String
5. recipeName - String

Implement getters, setters and constructors for the corresponding attributes.
create an interface named "**RecipeRepo**".
create a class named "**RecipeService**".

API Endpoint :
POST - /api/recipe- Returns response status 201 with recipe object on successful creation or else 500.
GET - /api/recipe/byname - Returns response status 200 with List<Recipe> objects filtered by recipeName which is passed in the request parameter key as 'recipeName' on successful retrieval or else 505.
GET - /api/recipe/{recipeId} - Returns response status 200 with recipe object on successful retrieval or else 404.

15. Build a web application that facilitates CRUD operations for managing door details using JPQL via REST APIs.
Functional Requirements:
Create a class named **"DoorController"**.
create a class named **"Door"** with the following attributes:

1. doorId - int
2. doorType - String
3. material - String
4. color - String
5. price - int

Implement getters, setters and constructors for the corresponding attributes.
create an interface named **"DoorRepo"**.
Inside the DoorRepo, utilize the @Query annotation to implement the GET, PUT, and DELETE operations.
create a class file named **"DoorService"**.

API Endpoint :
POST -/api/door - Returns response status 201 with door object on successful creation or else 500.
GET -/api/door/bycolor/{color} - Returns response status 200 with List<Door> object filtered by color on successful retrieval or else 404.
PUT -/api/door/{doorId} - Returns response status 200 with updated door object, where color to be updated is passed in the request parameter key as 'color' on successful updation or else 404.
DELETE -/api/door/{doorId} - Returns response status 200 with String "Door deleted successfully." on successful deletion or else 404.
GET -/api/door/bydoortype - Returns response status 200 with List<Door> object filtered by doorType, where the door type is passed in the request parameter key as 'doorType' on successful retrieval or else 404.
GET -/api/door - Returns response status 200 with List<Door> object on successful retrieval or else 404.
GET -/api/door/{doorId} - Returns response status 200 with door object on successful retrieval or else 404.

16. Build a web application that facilitates CRUD operations for managing customer details using JPQL via REST APIs.

Functional Requirements:
create a class named **"CustomerController"**.
create a class named **"Customer"** with the following attributes:

1. customerId - int
2. customerName - String
3. city - String
4. address - String
5. pincode - int

Implement getters, setters and constructors for the corresponding attributes.
Inside repository folder, create an interface named **"CustomerRepo"**.
In CustomerRepo, utilize @Qwery annotation to implement GET, PUT and DELETE operations.
Inside service folder, create a class file named **“CustomerService”**.

API Endpoint :
POST -/api/customer- Returns response status 201 with customer object on successful creation or else 500.
GET -/api/customer/bycity/{city} - Returns response status 200 with List<Customer> object on successful retrieval filtered by city or else 404.
GET -/api/customer - Returns response status 200 with List<Customer> object on successful retrieval or else 404.
GET -/api/customer/{customerId} - Returns response status 200 with customer object on successful retrieval or else 404.

PUT - /api/customer/{customerId}/{pincode} - Returns response status 200 with updated customer object on successful updation of pincode or else 404.
DELETE - /api/customer/{customerId} - Returns response status 200 with string as "Customer deleted successfully." on successful deletion or else 404.

17. Create a Spring Boot application with two entities: "Person" and "Address". A person can have only one address. Implement a one-to-one mapping between these entities using Spring JPA

Functional Requirements:
Create classes named **"PersonController"** and **"AddressController".**
Create a class named **"Person"** with the following attributes

1. id - Long (GeneratedValue)
2. name - String
3. email - String
4. phoneNumber - String
5. nationality - String
6. address - Address (OneToOne)

Create another class named **"Address"** with the following attributes

1. id - Long (GeneratedValue)
2. street - String
3. city - String
4. zipCode - String

Implement getters, setters and constructors for the Person and Address entities.
create interfaces named **"PersonRepository"** and **"AddressRepository"**.
create classes named **"PersonService"** and **"AddressService"**.

API ENDPOINTS:
POST "/person" - Returns response status 201 with person object on successful creation or else 500.
POST "/address/person/{personId}" - Returns response status 201 with address object on successfully mapping the address to the personId or else 500.
GET "/person" - Returns response status 200 with List<Person> object, which includes details of the address on successful retrieval or else 404.
GET "/person/{personId}" - Returns response status 200 with person object, which includes details of the address on successful retrieval or else 404.

18. Elaborate on the implementation of one-to-many relationship mapping with JPA in Spring Boot applications. Discuss the use cases for one-to-many relationships and their impact on database schema design and entity modeling. Provide examples demonstrating the configuration of one-to-many associations using JPA annotations. Analyze the strategies for fetching related entities lazily and eagerly and discuss their implications on application performance.

19. Implement logging with Spring Boot to monitor application behavior and diagnose issues effectively. Discuss the different log levels available in Spring Boot, including DEBUG, INFO, WARN, and ERROR, and their respective use cases. Provide examples of how to configure log levels for specific packages or classes in the application.properties file, ensuring appropriate logging granularity for different components of the application.

20. Develop a Spring Boot application with a Logging filter for player management.

Functional Requirements:
create a class named "**Player**" with the following attributes:

1. id - int
2. playerName - String
3. team - String
4. age - int

Implement getters, setters and constructors for the corresponding attributes.
create an interface named "**PlayerRepo**".
create a class named "**PlayerService**".
create a class named "**PlayerController**".
create a class named "**LoggingFilter**".
Inside the LoggingFilter file extend **OncePerRequestFilter** and annotate it with **@Component**, then implement the **doFilterInternal** method within it to log request and response details.
Log the following information:

- HTTP method
- Request URI
- Request payload
- Response code
- Response
- Time taken for processing

The logging information should be exactly like this:
"FINISHED PROCESSING : METHOD={}; REQUESTURI={}; REQUEST PAYLOAD={}; RESPONSE CODE={}; RESPONSE={}; TIME TAKEN={}"

API ENDPOINTS :
POST - "/players" - Returns response status 201 with a player object on successful creation or else 500.
GET - "/players" - Returns response status 200 with List<Player> object on successful retrieval or else 500.
GET - "/players/{id}" - Returns response status 200 with player object filtered by 'id' on successful retrieval or else 500.

21. Create an AOP aspect in a Spring Boot application that logs method execution time for all service layer methods. Demonstrate its effectiveness with sample output.

22. Explain the role of @AfterReturning advice in Aspect-Oriented Programming (AOP). Describe the functionality of @Around advice in AOP.