01 - Spring Data JPA Introduction

Spring Data JPA integrates with Spring Framework and provides a consistent programming model for interacting with databases using Java Persistence API (JPA).

Key Features of Spring Data JPA:

- ➢ **Simplified Data Access**: It abstracts common database operations, eliminating the need to write complex SQL queries or implement repetitive boilerplate code for basic CRUD operations.
- ➢ **Repository Pattern**: It follows the Repository pattern, providing an interface-based approach where we can define interfaces for data operations, and Spring Data JPA automatically generates the necessary implementation at runtime.
- ➢ **Automatic Query Generation**: By following simple naming conventions in repository method names (e.g., findByName), Spring Data JPA can automatically generate and execute SQL queries without the need for explicit JPQL.
- ➢ **Pagination and Sorting**: Spring Data JPA supports pagination and sorting, enabling efficient handling of large datasets by slicing results into pages and sorting them based on defined criteria.
- ➢ **Custom Queries**: In cases where automatic query generation is not enough, we can still define custom JPQL, and native SQL queries, or use the **@Query** annotation directly on repository methods.
- ➢ **Integration with Spring Boot**: When used with Spring Boot, Spring Data JPA can automatically configure itself based on the application's properties, making setup quick and easy.
- ➢ **Support for Various Databases**: Spring Data JPA works with a wide variety of databases, including relational databases like MySQL, PostgreSQL, Oracle, etc., through JPA-compliant ORM frameworks such as Hibernate.

02 - What is ORM and JPA

👉**ORM (Object-Relational Mapping):**

- Object-relational mapping (ORM) is a programming technique used to map objects in object-oriented programming languages to relational databases.
- It allows developers to work with databases using objects rather than writing SQL queries directly.
- Relational databases store data in tables, while objects in programming languages contain attributes and methods. ORM helps to bridge the gap between these two by mapping tables to objects and vice versa.

👉 **Key ORM Tools:**

➢ Hibernate
➢ TopLink
➢ EclipseLink

👉 **JPA (Java Persistence API):**

- JPA is a specification (not a framework) that defines how Java objects should be persisted in relational databases.
- It standardizes the mapping of objects to relational databases and provides a common interface for ORM tools like Hibernate.
- Hibernate, as one of the popular ORM frameworks, implements the JPA specification.

👉 **Why Use ORM & JPA in Spring Applications?**

- Spring ORM and Hibernate provide ORM capabilities but require significant boilerplate code for data persistence.
- Spring Data JPA further simplifies working with Hibernate and Spring ORM by abstracting and automating database operations.

03 - Creating Table and Inserting Data

Create a Spring Boot Maven Project. Add Spring Data JPA and PostgreSQL Driver dependency.

👉 **Repository Creation:**

Create a repository interface that extends JpaRepository. This interface automatically provides methods like save(), findAll(), findById(), deleteById(), etc., without needing to write their implementations.

The JpaRepository<Type, ID> takes two parameters:

1. Type: The class (entity) that represents a database table.
2. ID: The data type of the primary key of the table.

```
@Repository
public interface StudentRepo extends JpaRepository<Student, Integer> {

}
```

- Here, the StudentRepo interface is a repository for the Student entity, where the primary key type is Integer.

👉 **Entity Definition:**

Entities represent the data model or table structure in the database. Spring Data JPA uses these entities to map Java objects to database tables.

```
@Component
@Scope("prototype")
@Entity
public class Student {

	@Id
	private int rollNo;
	private String name;
	private int marks;

	public int getRollNo() {
		return rollNo;
	}

	public void setRollNo(int rollNo) {
		this.rollNo = rollNo;
	}

	public String getName() {
		return name;
	}

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

	public int getMarks() {
		return marks;
	}

	public void setMarks(int marks) {

		this.marks = marks;
	}
}
```

- In this, **@Entity** tells Spring Data JPA that this class is linked to a database table, and **@Id** marks rollNo as the primary key of the Student table.

👉 **Spring Boot Application Class:**

It initializes the Spring Application Context and loads all the beans.

- Create a Student object using the Spring context.
- Set values for the Student object.
- Use the StudentRepo to save the Student object into the database.

```
@SpringBootApplication
public class SpringDataJpaExApplication {
	public static void main(String[] args) {
	ApplicationContext context=
	SpringApplication.run(SpringDataJpaExApplication.class, args);
	  Student s1= context.getBean(Student.class);
	  Student s2=context.getBean(Student.class);
	  Student s3=context.getBean(Student.class);
	  StudentRepo repo=context.getBean(StudentRepo.class);

	  s1.setRollNo(101);
	  s1.setName("Navin");
	  s1.setMarks(75);

	  s2.setRollNo(102);
	  s2.setName("Kiran");
	  s2.setMarks(80);

	  s3.setRollNo(103);
	  s3.setName("Harsh");
	  s3.setMarks(70);

	  repo.save(s1);
	  repo.save(s2);
	  repo.save(s3);
	}
}
```

- Here, we retrieve the Student bean and StudentRepo bean from the container, then use repo.save() to persist the data.

👉 **Database Configuration:**

We need to configure in the application.properties file the connection to the database so that Spring Data JPA can automatically create and manage tables, and perform operations.

```
spring.datasource.url=jdbc:postgresql://localhost:5432/telusko

spring.datasource.username=postgres

spring.datasource.password=root

spring.datasource.driver-class-name=org.postgresql.Driver

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true
```

Code Link :
https://github.com/navinreddy20/spring6-course/tree/c6690e4f2c70d8f530d70623f13d14ff0ffd7e7d/9%20Spring%20DataJPA/9.3%20Creating%20Table%20And%20Inserting%20Data

04 - Findall

The findAll() method in Spring Data JPA is provided by the JpaRepository interface, retrieving all the records from the database associated with the entity.

Example:

```
@SpringBootApplication
public class SpringDataJpaExApplication {

	public static void main(String[] args) {
	ApplicationContext context=
	SpringApplication.run(SpringDataJpaExApplication.class, args);
	  Student s1= context.getBean(Student.class);
	  Student s2=context.getBean(Student.class);
	  Student s3=context.getBean(Student.class);

	  StudentRepo repo=context.getBean(StudentRepo.class);

	  System.out.println(repo.findAll());
	}
}
```

Code Link :
https://github.com/navinreddy20/spring6-course/tree/c6690e4f2c70d8f530d70623f13d14ff0ffd7e7d/9%20Spring%20DataJPA/9.4%20Findall

05 - FindById

The **findById()** method fetches a single record from the database based on its primary key.
It returns an **Optional** object that provides a flexible way of handling cases where a record may or may not exist in the database.

- If the record is found, the Optional contains the entity.
- If the record is not found, the Optional is empty, preventing NullPointerException by offering safe ways to handle absent values.

Using methods like orElse(), we can safely handle the absence of data without null checks.

Example:

```
@SpringBootApplication
public class SpringDataJpaExApplication {

	public static void main(String[] args) {
	ApplicationContext context=
	SpringApplication.run(SpringDataJpaExApplication.class, args);
	  Student s1= context.getBean(Student.class);
	  Student s2=context.getBean(Student.class);
	  Student s3=context.getBean(Student.class);

	  StudentRepo repo=context.getBean(StudentRepo.class);

	Optional<Student> s= repo.findById(103);
	System.out.println(s.orElse(new Student()));
	}
}
```

Code Link :
https://github.com/navinreddy20/spring6-course/tree/c6690e4f2c70d8f530d70623f13d14ff0ffd7e7d/9%20Spring%20DataJPA/9.5%20Findbyid

06 - Query DSL

In JPQL (Java Persistence Query Language), queries are based on the **entity class** names and their fields, not on database table names or column names.

- JPA provides a powerful query method DSL that defines queries based on method names such as findBy, findAllBy, findByName, findByMarksGreaterThan, etc.
- If more complex queries are needed beyond what is automatically generated by the method names, **@Query** annotation can be used to define custom JPQL queries.

Example:
StudentRepo.java

```
@Repository
public interface StudentRepo extends JpaRepository<Student, Integer> {

//	@Query("select s from Student s where s.name=?1")
//	List<Student> findByName(String name);
//	List<Student> findByMarks(int marks);
	List<Student> findByMarksGreaterThan(int marks);
}
```

SpringDataJpaExApplication.java

```
@SpringBootApplication
public class SpringDataJpaExApplication {

	public static void main(String[] args) {
	ApplicationContext context =
			SpringApplication.run(SpringDataJpaExApplication.class, args);
	  Student s1= context.getBean(Student.class);
	  Student s2=context.getBean(Student.class);
	  Student s3=context.getBean(Student.class);

	  StudentRepo repo=context.getBean(StudentRepo.class);
	 //System.out.println(repo.findByName("Navin"));
	//System.out.println(repo.findByMarks(80));
	 System.out.println(repo.findByMarksGreaterThan(72));
	 }
}
```

Code Link :
https://github.com/navinreddy20/spring6-course/tree/c6690e4f2c70d8f530d70623f13d14ff0ffd7e7d/9%20Spring%20DataJPA/9.6%20Query%20Dsl

07 - Update and Delete

Update Using save() Method:

The save() method in Spring Data JPA can be used to update records in the database.

Before updating a record:

- A select query is first executed to check if the data exists based on the provided entity's primary key.
- If the data exists, it fires an update query.
- If the data does not exist, it fires a create query to insert a new record.

Delete Using delete() Method:

- The delete() method is used to delete an entity from the database.
- The method takes the entity object as a parameter, and the deletion is performed based on the primary key of the object.
- Before deletion, a select query is executed to verify if the entity exists. If the record is found, a delete query is then issued to remove the entity from the database.

Example:

```
@SpringBootApplication
public class SpringDataJpaExApplication {
	public static void main(String[] args) {
	ApplicationContext context=
	  SpringApplication.run(SpringDataJpaExApplication.class, args);
	  Student s1= context.getBean(Student.class);

	  Student s2=context.getBean(Student.class);
	  Student s3=context.getBean(Student.class);
	  StudentRepo repo=context.getBean(StudentRepo.class);

	  //System.out.println(repo.save(s2));
	  repo.delete(s2);
	}
}
```

Code Link :

https://github.com/navinreddy20/spring6-course/tree/c6690e4f2c70d8f530d70623f13d14ff0ffd7e7d/9%20Spring%20DataJPA/9.7%20Update%20And%20Delete

08 - JPA in Job App

Setting up JPA in a Job Application:
To implement JPA with PostgreSQL in a Job Management Application, the following steps involve configuring the project dependencies, repository, service, and REST API endpoints to handle CRUD operations for job postings.

1. Adding Dependencies:

Include JPA and PostgreSQL dependencies in the project's pom.xml file.

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
   <groupId>org.postgresql</groupId>
   <artifactId>postgresql</artifactId>
   <scope>runtime</scope>
</dependency>
```

2. Job Repository Interface:

Create a repository interface JobRepo that extends JpaRepository. This interface will automatically provide CRUD operations for JobPost entities.

```
@Repository
public interface JobRepo extends JpaRepository<JobPost, Integer> {

}
```

3. JobRestController for Handling HTTP Requests:

- JobRestController leverages JPA through JobRepo, a repository extending JpaRepository, to handle CRUD operations without SQL.
- JPA automatically translates method calls (like save(), findById(), findAll(), and deleteById()) into SQL queries, making database interactions straightforward without needing to write SQL manually.
- JPA's built-in CRUD methods (findAll(), findById(), save(), deleteById()) allow for straightforward data handling and reduce boilerplate code.
- JPA manages the persistence context, when JobRepo.save() is called in JobRestController, JPA decides whether to insert a new record or update an existing one based on the primary key.

```
@RestController
@CrossOrigin
public class JobRestController {
	@Autowired
	private JobService service;

	@GetMapping("jobPosts")
	public List<JobPost> getAllJobs() {
		return service.getAllJobs();
	}

	@GetMapping("/jobPost/{postId}")
	public JobPost getJob(@PathVariable int postId) {
		return service.getJob(postId);
	}

	@PostMapping("jobPost")
	public JobPost addJob(@RequestBody JobPost jobPost) {
		service.addJob(jobPost);
		return service.getJob(jobPost.getPostId());
	}

	@PutMapping("jobPost")
	public JobPost updateJob(@RequestBody JobPost jobPost) {
		service.updateJob(jobPost);
		return service.getJob(jobPost.getPostId());
	}

	@DeleteMapping("jobPost/{postId}")
	public String deleteJob(@PathVariable int postId)
	{
		service.deleteJob(postId);
		return "Deleted";
	}
```

```
	@GetMapping("load")
	public String loadData() {
		service.load();
		return "success";
	}
}
```

4. JobService Layer:

- The JobService class handles the core business logic and communicates with JobRepo for database interactions.
- Each method in JobService corresponds to a controller method in JobRestController, such as fetching all jobs, getting a job by ID, adding, updating, or deleting job posts.

```
@Service
public class JobService {

   @Autowired
   public JobRepo repo;

   public List<JobPost> getAllJobs() {
      return repo.findAll();
   }

    public void addJob(JobPost jobPost) {
      repo.save(jobPost);
   }

   public JobPost getJob(int postId) {
      return repo.findById(postId).orElse(new JobPost());
   }

   public void updateJob(JobPost jobPost) {
      repo.save(jobPost);
   }

   public void deleteJob(int postId) {
      repo.deleteById(postId);
   }
```

```
    public void load() {
        List<JobPost> jobs = new ArrayList<>(List.of(
            new JobPost(1, "Software Engineer", "Exciting opportunity for a skilled software
            engineer.", 3, List.of("Java", "Spring", "SQL")),
            new JobPost(2, "Data Scientist", "Join our data science team and work on cutting-edge
            projects.", 5, List.of("Python", "Machine Learning", "TensorFlow")),
            new JobPost(3, "Frontend Developer", "Create amazing user interfaces with our talented
            frontend team.", 2, List.of("JavaScript", "React", "CSS")),
            new JobPost(4, "Network Engineer", "Design and maintain our robust network
            infrastructure.", 4, List.of("Cisco", "Routing", "Firewalls")),
            new JobPost(5, "UX Designer", "Shape the user experience with your creative design
            skills.", 3, List.of("UI/UX Design", "Adobe XD", "Prototyping"))

        ));
        repo.saveAll(jobs);
    }
}
```

Code Link :
https://github.com/navinreddy20/spring6-course/tree/c6690e4f2c70d8f530d70623f13d14ff0ffd7e7d/9%20Spring%20DataJPA/9.8%20Jpa%20In%20Job%20App

09 - Loading Data and Entity

👉 **Entity Creation with JobPost:**

- **@Entity:** Specifies that the JobPost class is an entity mapped to a database table.
- **@Id:** Identifies the primary key of the entity, which in this case is the postId field.
- **@Data:** Generates getters, setters, equals(), hashCode(), toString(), and other methods automatically.
- **@NoArgsConstructor and @AllArgsConstructor:** Provide a default constructor and an all-arguments constructor, simplifying object creation and injection.

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
public class JobPost {

	@Id
	private int postId;
	private String postProfile;
	private String postDesc;
	private Integer reqExperience;
	private List<String> postTechStack;

}
```

👉 **Database Configuration in application.properties:**

This configuration file connects the application to a PostgreSQL database and manages database properties.

```
spring.datasource.url=jdbc:postgresql://localhost:5432/telusko
spring.datasource.username=postgres
spring.datasource.password=Lumia@540
spring.datasource.driver-class-name=org.postgresql.Driver
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Code Link :
https://github.com/navinreddy20/spring6-course/tree/c6690e4f2c70d8f530d70623f13d14ff0ffd7e7d/9%20Spring%20DataJPA/9.9%20Loading%20Data%20And%20Entity

10 - Search By Keyword

The following approach uses Spring Data JPA to create a repository with methods that search specific fields of the JobPost entity for the presence of the keyword.

1. Search Endpoint in JobRestController:

A new GET endpoint /jobPosts/keyword/{keyword} is added to the JobRestController to enable keyword-based searches.

```
@GetMapping("jobPosts/keyword/{keyword}")
public List<JobPost> searchByKeyword(@PathVariable("keyword") String keyword){
    return service.search(keyword);
}
```

- @PathVariable("keyword") String keyword: Captures the keyword from the URL path.
- The controller calls service.search(keyword), which performs the keyword search in the JobPost repository.

2. Custom Query in JobRepo Interface:

Spring Data JPA enables automatic query generation through methods based on naming conventions, referred to as Domain-Specific Language (DSL).

```
@Repository
public interface JobRepo extends JpaRepository<JobPost, Integer> {

    List<JobPost> findByPostProfileContainingOrPostDescContaining(String postProfile,
    String postDesc);

}
```

- the findByPostProfileContainingOrPostDescContaining method is defined to retrieve JobPost records where either the postProfile or postDesc field contains the specified keyword.
- The Containing keyword indicates that the search should check for partial matches within the specified fields.

3. Service Layer Method search():

It ensures that any JobPost records with the specified keyword in either field are returned to the controller.

```
public List<JobPost> search(String keyword) {
    return repo.findByPostProfileContainingOrPostDescContaining(keyword, keyword);
}
```

- The search() method in JobService calls repo.findByPostProfileContainingOrPostDescContaining(keyword, keyword), passing the keyword to both postProfile and postDesc fields to find matches.

👉 Summary of Workflow:

- A request is sent to the endpoint /jobPosts/keyword/{keyword}, passing a keyword as a path variable.
- The JobRestController invokes the JobService's search() method, which in turn calls the custom findByPostProfileContainingOrPostDescContaining method in JobRepo.
- The repository method performs a search for JobPost records containing the keyword in either postProfile or postDesc, returning matching results.

Code Link :
https://github.com/navinreddy20/spring6-course/tree/c6690e4f2c70d8f530d70623f13d14ff0ffd7e7d/9%20Spring%20DataJPA/9.10%20Search%20By%20Keyword/spring-boot-rest

11 - React UI for Search

In this, we develop a React frontend to enable search functionality for job posts, leveraging the searchByKeyword method from the backend.

Backend API Endpoint: The search functionality is exposed through a REST API endpoint defined in Java as:

```
@GetMapping("jobPosts/keyword/{keyword}")
public List<JobPost> searchByKeyword(@PathVariable("keyword") String keyword) {
   return service.search(keyword);
}
```

React Frontend Workflow: A text field in React triggers the search when a user enters a keyword. The component fetches job posts matching the keyword from the backend and displays them.

- **State and Initial Data Fetching:**
  - **useEffect** is used to load all job posts initially when the component mounts.
  - axios.get is used to make a GET request to http://localhost:8080/jobPosts.

```
useEffect(() => {
   const fetchInitialPosts = async () => {
      const response = await axios.get("http://localhost:8080/jobPosts");
      setPost(response.data);
   };
   fetchInitialPosts();
}, []);
```

- **Search Functionality:**
  - A text input captures the keyword and calls the backend searchByKeyword endpoint dynamically.
  - When the keyword is updated, another axios.get request is sent to the URL http://localhost:8080/jobPosts/keyword/{keyword} to retrieve matching posts.

Code Link :
https://github.com/navinreddy20/spring6-course/tree/c6690e4f2c70d8f530d70623f13d14ff0ffd7e7d/React%20UI

12 - React UI for Delete And Update

In this, we implement update and delete functionality for job posts on the React frontend, connecting with backend endpoints.

Backend API Endpoint for Update: @PutMapping("jobPost") is used to handle update requests for job posts.-

- **Edit Component Setup:**
  - Fetches the job post data by ID from the backend using axios.get and populates the form with the data retrieved.
  - useLocation(): Captures the current post ID from the location state.
  - useNavigate(): Allows redirection after successful submission if needed.
  - useState(): Manages form data and stores the current post ID.

```
const location = useLocation();
const navigate = useNavigate();
const [form, setForm] = useState(initial);
const [currId] = useState(location.state.id);

useEffect(() => {
   const fetchInitialPosts = async (id) => {
      const response = await axios.get(`http://localhost:8080/jobPost/${id}`);
      console.log(response.data);
      setForm(response.data);
  };

 fetchInitialPosts(currId);
}, [currId]);
```

- **Update Functionality:**
    - Sends a POST request to http://localhost:8080/jobPost, passing the form data that submits updated form data to the backend
    - Logs the response or catches any errors for debugging.

```
const handleSubmit = (e) => {
   e.preventDefault();
   axios
     .post("http://localhost:8080/jobPost", form)
     .then((resp) => {
        console.log(resp.data);
     })
     .catch((error) => {
        console.log(error);
     });
};
```

- **Handling Form Changes:**
    - It Updates the form state based on input changes.
    - Adds the new value to postTechStack in the form state dynamically.

```
const handleChange = (e) => {
   setForm({ ...form, postTechStack: [...form.postTechStack, e.target.value] });
};
```

- **Delete Functionality:**
    - It Deletes a specific job post by ID and reloads the page.
    - Calls a delete request to the backend endpoint and triggers a page reload.

```
const handleDelete = (id) => {
   async function deletePost() {
      await axios.delete(`http://localhost:8080/jobPost/${id}`);
      console.log("Delete");
   }
   deletePost();

  window.location.reload();
};
```

Code Link :
https://github.com/navinreddy20/spring6-course/tree/c6690e4f2c70d8f530d70623f13d14ff0ffd7e7d/React%20UI