CRUD Operations Using Spring Data JPA

Aashish Kulkarni

Spring Data JPA is part of the larger Spring Data family, providing an abstraction layer for JPA. It allows you to interact with a relational database through simple methods without having to write complex SQL queries. Spring Data JPA also handles CRUD operations for you using repositories.

Setting Up Spring Data JPA

1. Add dependencies:

Choose Maven or Gradle.

2. Configure Database Connection:

In application.properties or application.yml, set your database connection.

Creating the Entity Class

An entity class represents a table in the database.

Example Person entity:

```
@Entity
public class Person {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String firstName;
   private String lastName;

   // Getters and Setters

}
```

Creating the Repository Interface

The repository interface allows you to perform CRUD operations without implementing methods manually. It extends JpaRepository or CrudRepository.

```
import org.springframework.data.jpa.repository.JpaRepository;


public interface PersonRepository extends JpaRepository<Person,
                                                         Long> {
   // You can define custom queries here if needed
}
```

- **JpaRepository<T, ID> provides the main CRUD functionality and additional methods like pagination and sorting.**
- **CrudRepository<T, ID> provides basic CRUD methods.**

Performing CRUD Operations

1. Create an Entity

To create and save a new entity in the database, you can use the save() method of the repository.

```
@RestController
public class PersonController {
   @Autowired
   private PersonRepository personRepository;

   @PostMapping("/persons")
   public Person createPerson(@RequestBody Person person) {
      return personRepository.save(person);
   }
}
```

- **The save() method returns the entity, with its generated id field if it's newly created.**

2. Read Entities

To retrieve data from the database, you can use methods like findById(), findAll(), and custom queries.

- **Find by ID:**

```
@GetMapping("/persons/{id}")
public Person getPersonById(@PathVariable Long id) {
   return personRepository.findById(id).orElseThrow(() -> new
                  PersonNotFoundException("Person not found"));
}
```

- **Find all entities:**

```
@GetMapping("/persons")
public List<Person> getAllPersons() {
   return personRepository.findAll();
}
```

- **Custom Queries:**

  You can define custom query methods in the repository interface.

```
List<Person> findByLastName(String lastName);
```

Spring Data JPA automatically implements this method based on the name of the method. It will generate a query like SELECT * FROM person WHERE lastName = ?.

3. Update an Entity

To update an existing entity, you simply call save() again on the entity. If the entity already exists, it will update the existing record which is identified by its primary key.

```
@PutMapping("/persons/{id}")
public Person updatePerson(@PathVariable Long id,
                           @RequestBody Person personDetails) {
   Person person = personRepository.findById(id)
                        .orElseThrow(() -> new
                  PersonNotFoundException("Person not found"));

   person.setFirstName(personDetails.getFirstName());
   person.setLastName(personDetails.getLastName());

   return personRepository.save(person);  // Saves updated entity

}
```

4. Delete an Entity

To delete an entity by its ID, use deleteById():

```
@DeleteMapping("/persons/{id}")
public ResponseEntity<?> deletePerson(@PathVariable Long id) {
   Person person = personRepository.findById(id)
                        .orElseThrow(() -> new
                  PersonNotFoundException("Person not found"));

   personRepository.delete(person);
   // or
  personRepository.deleteById(id);

   return ResponseEntity.ok().build();
}
```

Additional Features in Spring Data JPA

1. Custom Queries:

You can use @Query annotations to write custom SQL or JPQL queries:

```
@Query("SELECT p FROM Person p WHERE p.lastName =
                                                :lastName")
List<Person> findByLastName(@Param("lastName")
                                                String lastName);
```

2. Pagination and Sorting:

Spring Data JPA supports pagination and sorting with Pageable and Sort.

```
Page<Person> findAll(Pageable pageable);
                        // Pageable used to paginate results
List<Person> findAll(Sort sort);        // Sort results based on a field
```

3. Transaction Management:

Spring Data JPA integrates with Spring's transaction management. By default, Spring handles transactions automatically with the @Transactional annotation.

Example of CRUD Operations in a Controller

```
@RestController
@RequestMapping("/persons")
public class PersonController {

  @Autowired
  private PersonRepository personRepository;

  @PostMapping
  public Person createPerson(@RequestBody Person person) {
    return personRepository.save(person);
  }
```

Code continue on Next Page ->

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

@PutMapping("/{id}")
public Person updatePerson(@PathVariable Long id,
                           @RequestBody Person personDetails) {
   Person person = personRepository.findById(id)
                       .orElseThrow(() -> new
                      RuntimeException("Person not found"));

   person.setFirstName(personDetails.getFirstName());
   person.setLastName(personDetails.getLastName());
   return personRepository.save(person);
}
```

Code continue on Next Page ->

```
    @DeleteMapping("/{id}")
    public ResponseEntity<?> deletePerson(@PathVariable Long id)
{
        personRepository.deleteById(id);
        return ResponseEntity.ok().build();
    }
}
```