Spring Boot + Microservices2402

Spring Data JPA : Working on Advance stuffs

Spring Data JPA offers multiple ways to write custom methods in your JPA repositories.

1. Derived Queries (Method Naming Conventions)

   This is the most convenient approach for simple queries.
   Spring Data JPA automatically translates method names into JPA queries based on conventions:

   ```
     findBy<FieldName>: Find by a specific field (e.g., findBy

   findBy<FieldName1>And<FieldName2>: Find by multiple fields

   countBy<FieldName>: Count entities by a specific field.

   existsBy<FieldName>: Check if an entity exists by a field.
   ```

2. Query methods

   For more complex queries that don't fit the naming conventions, use the @Query annotation on your repository method.

   You can directly write the JPQL or native SQL query within the annotation:

```
@Query("SELECT c FROM Customer c WHERE c.name = :name AND c
    List<Customer> findByNameAndActive(@Param("name") String

    native queries:
    @Query(value = "SELECT * FROM customers WHERE name = ?1 A
    List<Customer> findByNameAndActiveNative(String name, boo
```

3. Custom Implementation with Own Logic

   For highly customized logic or complex criteria, you can define separate class and us Entity manager to write custom logic.

```
@Component
    public class CustomerRepositoryImpl {

    @PersistenceContext
    private EntityManager em;

   @Override
    public List<Customer> findCustomersByCriteria(String name
    CriteriaBuilder cb = em.getCriteriaBuilder();
    CriteriaQuery<Customer> cq = cb.createQuery(Customer.clas
    Root<Customer> root = cq.from(Customer.class);
    Predicate namePredicate = cb.like(root.get("name"), "%" +
    Predicate cityPredicate = cb.equal(root.get("city"), city
    cq.where(cb.and(namePredicate, cityPredicate));
    return em.createQuery(cq).getResultList();
   }
   }
```

Example 1:

Here's how to write a custom JPQL query in a Spring Data JPA repository to join `Product` and `Category` entities and retrieve products along with their categories:

Here's how to write a custom JPQL query in a Spring Data JPA repository to join `Product` and `Category` entities and retrieve products along with their categories:

```
@Entity
public class Product {

  @Id
  private Long id;

  private String name;
  private String description;

  @ManyToMany(mappedBy = "products")
  private Set<Category> categories;

  // Getters, setters, and other methods
}

@Entity
public class Category {

  @Id
  private Long id;

  private String name;
  private String description;

  @ManyToMany(cascade = CascadeType.ALL)
  @JoinTable(name = "product_category",
      joinColumns = @JoinColumn(name = "category_id", referen
      inverseJoinColumns = @JoinColumn(name = "product_id", 
  private Set<Product> products;
  // Getters, setters, and other methods
}
```

Create the Repository Interface:

Extend `JpaRepository<Product, Long>` for the `ProductRepository`. Here, we'll define a custom method using the `@Query` annotation:

```
public interface ProductRepository extends JpaRepository<Prod

  @Query("SELECT p FROM Product p JOIN FETCH p.categories c W
  Product findProductWithCategories(Long productId);
}
```

`JOIN FETCH p.categories c`: Joins the `Category` entity with `p` through the `categories` collection, using `FETCH` to eagerly load the categories along with the product.

Example2:

Here's an example of a JPQL join query in a Spring Data JPA repository that retrieves custom data from joined entities:

Entity Classes:

Let's assume we have entities `Order` and `Customer`, with a One-to-Many relationship (a customer can have multiple orders). We want to retrieve a custom object containing order details and customer name:

```
@Entity
public class Order {

  @Id
  private Long id;

  private String productName;
  private double price;
  @ManyToOne
  private Customer customer;

  // Getters, setters, and other methods
}

@Entity
public class Customer {

  @Id
  private Long id;

  private String name;
  private String email;

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

  // Getters, setters, and other methods
}
```

Custom DTO Class:

```
public class OrderDetailsWithCustomerName {
private Long orderId;
private String productName;
private double price;
private String customerName;

// Getters and setters
}
```

Repository Interface:

Extend `JpaRepository<Order, Long>` for the `OrderRepository`. Define a custom method using `@Query` and a constructor expression to create the DTO object:

```
public interface OrderRepository extends JpaRepository<Order,
  @Query("SELECT NEW com.yourpackage.OrderDetailsWithCustomer
         "FROM Order o JOIN o.customer c WHERE o.id = :orderI
  OrderDetailsWithCustomerName findOrderDetailsWithCustomerNa
}
```

SELECT NEW com.yourpackage.OrderDetailsWithCustomerName(...): Uses a constructor expression to create a new instance of OrderDetailsWithCustomerName during the query execution.

JPQL/HQL

JPQL query in a Spring Data JPA repository to join `Product` and `Category` entities and retrieve products along with their categories:

Here's a breakdown of key JPQL features:

1. Basic Structure:

A JPQL query typically consists of the following clauses:

- **SELECT clause:** Specifies the data to be retrieved (e.g., select all fields, specific columns).
- **FROM clause:** Defines the root entity you're querying from.
- **WHERE clause (optional):** Filters the results based on conditions involving entity properties and comparison operators.
- **JOIN clause (optional):** Enables joining multiple entities based on relationships.
- **ORDER BY clause (optional):** Sorts the results based on specified criteria.

2. Selecting Data:

- You can select entire entities or specific properties:
    - `SELECT p FROM Product p`: Selects all `Product` entities.
    - `SELECT p.name, p.price FROM Product p`: Selects only name and price properties from `Product` entities.

3. Filtering with WHERE Clause:

- Use comparison operators and entity properties to create conditions:
    - `SELECT p FROM Product p WHERE p.id = 10`: Finds products with ID 10.
    - `SELECT c FROM Customer c WHERE c.name LIKE '%John%'`: Finds customers with names containing "John".

4. Joining Entities:

- JPQL supports various join types (INNER JOIN, LEFT JOIN, etc.) to retrieve data from related entities:
    - `SELECT o FROM Order o JOIN o.customer c`: Joins `Order` and `Customer` entities based on the one-to-many relationship.

5. Sorting Results:

- Use the `ORDER BY` clause with entity properties and optional ascending/descending order:
    - `SELECT p FROM Product p ORDER BY p.price DESC`: Sorts products by price in descending order.

Join Examples:

1. Inner Join (Fetching All Related Entities):

This query retrieves all `Order` entities and eagerly fetches their associated `Customer`:

```
SELECT o FROM Order o JOIN FETCH o.customer c
```

2. Left Join (Including Orders Without Customers):

```
SELECT o FROM Order o LEFT JOIN o.customer c
```

3. Join with Conditions:

This query finds `Order` entities where the product name contains "Laptop" and joins with the `Customer`:

```
SELECT o FROM Order o JOIN o.customer c WHERE o.productName LIKE
```

Criteria API

The Criteria API in JPA offers a programmatic way to construct database queries.

provides a type-safe and flexible alternative to JPQL (Java Persistence Query Language).

1. Building Blocks:

The Criteria API provides several key components for building queries:

- **CriteriaBuilder:** This object helps you construct the criteria query itself. You can obtain it from the `EntityManager`.
- **CriteriaQuery:** This represents the overall query you're building, specifying the selection clause, root entity, and optional filtering criteria.
- **Root:** This defines the starting point of the query, typically the entity you're querying for.
- **Criteria:** These represent restrictions on the results, such as predicates and conjunctions.
- **Predicate:** This defines a condition that must be true for a result to be included.

2. Constructing a Criteria Query:

Here's a basic structure for building a Criteria query to find all `Product` entities:

```
EntityManager em = ...; // Get the entity manager
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Product> cq = cb.createQuery(Product.class);
Root<Product> root = cq.from(Product.class); // Set the root ent

// Select all fields (optional)
cq.select(root);

// Build and execute the query
List<Product> products = em.createQuery(cq).getResultList();
```

3. Adding Filtering Criteria (Predicates):

You can use various criteria builders like `cb.equal`, `cb.like`, and `cb.gt` to create predicates based on different comparison operators. Here's an example to find products with a price greater than 100:

```
Predicate priceGreaterThan100 = cb.gt(root.get("price"), 100.0)
cq.where(priceGreaterThan100);
```

4. Combining Predicates:

You can combine multiple predicates using `and` and `or` methods from the `CriteriaBuilder`:

```
Predicate nameStartsWithA = cb.like(root.get("name"), "A%");
Predicate priceLessThan50 = cb.lt(root.get("price"), 50.0);
cq.where(cb.and(nameStartsWithA, priceLessThan50));
```

5. Sorting:

The Criteria API allows sorting results using the `orderBy` method on the `CriteriaQuery`:

```
cq.orderBy(cb.asc(root.get("price"))); // Sort by price in ascer
```

6. Advantages of Criteria API:

- **Type Safety:** Eliminates syntax errors common with JPQL strings.
- **Flexibility:** Allows for complex query construction with various criteria.
- **Integration with Metamodel API:** Provides compile-time type checking for field names (optional).

When to Use Criteria API:

Consider using the Criteria API when:

- You need complex queries with dynamic filtering criteria.
- You want type safety and compile-time validation.

Transaction Management in JPA

1. Understanding Transactions:

- A transaction is a unit of work that ensures data consistency in your database.
- All database operations within a transaction are treated as a whole.
- If any operation fails, the entire transaction is rolled back, reverting all changes.

2. Using `@Transactional` Annotation:

The `@Transactional` annotation is applied at the method or class level in your service or repository classes. It tells Spring to manage the transaction automatically.

```
@Service
public class ProductService {

  @Autowired
  private ProductRepository productRepository;

  @Transactional
  public Product saveProduct(Product product) {
    Product savedProduct = productRepository.save(product);
```

```java
        // Other logic that modifies data (e.g., updating stock)
        return savedProduct;
    }
}
```