Introduction: **Java Persistence Query Language (JPQL)** is a powerful tool that allows developers to interact with databases using **object-oriented concepts.**

In conjunction with the Spring Boot framework and Spring Data JPA, JPQL empowers developers to write concise and expressive queries, making database operations more efficient and maintainable.

Understanding JPQL: At its core, JPQL is a query language defined as part of the Java Persistence API (JPA) specification.

It offers a way to perform database operations using entity objects and their relationships, abstracting away the intricacies of SQL. JPQL queries are written in terms of entity classes and their attributes, providing a platform-independent approach to database access.

Using JPQL in Spring Boot: Spring Boot simplifies the development of Java applications by providing conventions and auto-configurations out of the box.

When combined with Spring Data JPA, developers can seamlessly integrate JPQL into their applications.

Spring Data JPA offers various mechanisms for defining JPQL queries, including method naming conventions, query annotations, and query derivation from method signatures.

JPQL is like SQL, but instead of querying tables, it queries Java entities.

- **SQL:** SELECT * FROM user WHERE email = 'abc@gmail.com'
- **JPQL:** SELECT u FROM User u WHERE u.email = 'abc@gmail.com'

JPQL works with:

- **Entity classes (e.g., User)**
- **Entity fields (e.g., email, username)**

```
@Query("SELECT u FROM User u WHERE u.username = :username")
User findByUsername(@Param("username") String username);
```

- **User** = **entity class name**
- **u.username** = **field in entity**
- **:username** = **parameter**

Example: Consider a scenario where we have an e-commerce application with a Product entity representing products in our database. Let's say we want to retrieve products based on their category. We can achieve this using JPQL in Spring Boot with the following code snippet:

```
import java.util.List;

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

```
@Repository
public interface ProductRepository extends JpaRepository<Product,
Long> {

    @Query("SELECT p FROM Product p WHERE p.category = ?1")
    List<Product> findByCategory(String category);
}
```

In this example, we define a method named findByCategory in the ProductRepository interface. The method is annotated with @Query, where we specify the JPQL query to select products based on the provided category.

Common JPQL Examples

1. Select All Users

```
@Query("SELECT u FROM User u")
List<User> getAllUsers();
```

2. Find by Email

```
@Query("SELECT u FROM User u WHERE u.email = :email")
User getUserByEmail(@Param("email") String email);
```

3. Search Users by Name (LIKE)

```
@Query("SELECT u FROM User u WHERE u.username LIKE %:name%")
List<User> searchByUsername(@Param("name") String name);
```

LIKE %:name% allows searching **anywhere in the username**.

4. Order By (Descending)

```
@Query("SELECT u FROM User u ORDER BY u.id DESC")
List<User> getAllUsersDesc();
```

5. Count Users

```
@Query("SELECT COUNT(u) FROM User u")
Long countAllUsers();
```

6. Check Existence

```
@Query("SELECT COUNT(u) > 0 FROM User u WHERE u.email = :email")
boolean existsByEmail(@Param("email") String email);
```

Conclusion: Java Persistence Query Language (JPQL) is a valuable asset for Java developers working with databases. When combined with Spring Boot and Spring Data JPA, JPQL offers a streamlined approach to database access, enhancing productivity and code maintainability. By mastering JPQL, developers can write efficient and expressive queries, leading to robust and scalable applications.