**WEEK 3**

**ADDITIONAL DEMONSTRATION-HANDS ON**

**Demonstrate implementation of Query Methods feature of Spring Data JPA**

**Hands-on 8: Demonstrate Query Methods in Spring Data JPA**

Spring Data JPA provides **Query Methods** to define data access patterns through method names alone, without writing SQL or JPQL explicitly. These methods are automatically implemented by Spring Data JPA based on naming conventions.

**What Are Query Methods?**

Query methods are derived from method names that follow a naming strategy recognized by Spring Data JPA. Based on the method name, Spring creates the actual query at runtime.

**Step-by-Step Demonstration**

**1. Add Query Methods in CountryRepository**

**Location:** com.cognizant.ormlearn.repository.CountryRepository

@Repository

public interface CountryRepository extends JpaRepository<Country, String> {

// Find countries by name containing specific text (case-sensitive)

List<Country> findByNameContaining(String keyword);

// Find countries with names starting with a prefix

List<Country> findByNameStartingWith(String prefix);

// Find top 3 countries ordered by name ascending

List<Country> findTop3ByOrderByNameAsc();

// Find all countries ordered by name descending

List<Country> findAllByOrderByNameDesc();

}

**2. Add New Test Methods in OrmLearnApplication.java**

private static void testFindByNameContaining() {

LOGGER.info("Start: testFindByNameContaining");

List<Country> countries = countryService.findByNameContaining("land");

countries.forEach(c -> LOGGER.debug("Country: {}", c));

LOGGER.info("End");

}

private static void testFindByNameStartingWith() {

LOGGER.info("Start: testFindByNameStartingWith");

List<Country> countries = countryService.findByNameStartingWith("A");

countries.forEach(c -> LOGGER.debug("Country: {}", c));

LOGGER.info("End");

}

private static void testFindTop3ByOrderByNameAsc() {

LOGGER.info("Start: testFindTop3ByOrderByNameAsc");

List<Country> countries = countryService.findTop3ByOrderByNameAsc();

countries.forEach(c -> LOGGER.debug("Country: {}", c));

LOGGER.info("End");

}

private static void testFindAllByOrderByNameDesc() {

LOGGER.info("Start: testFindAllByOrderByNameDesc");

List<Country> countries = countryService.findAllByOrderByNameDesc();

countries.forEach(c -> LOGGER.debug("Country: {}", c));

LOGGER.info("End");

}

**3. Expose Repository Methods via CountryService**

@Transactional

public List<Country> findByNameContaining(String keyword) {

return countryRepository.findByNameContaining(keyword);

}

@Transactional

public List<Country> findByNameStartingWith(String prefix) {

return countryRepository.findByNameStartingWith(prefix);

}

@Transactional

public List<Country> findTop3ByOrderByNameAsc() {

return countryRepository.findTop3ByOrderByNameAsc();

}

@Transactional

public List<Country> findAllByOrderByNameDesc() {

return countryRepository.findAllByOrderByNameDesc();

}

**4. Modify main() to Run All Query Method Tests**

public static void main(String[] args) {

ApplicationContext context = SpringApplication.run(OrmLearnApplication.class, args);

countryService = context.getBean(CountryService.class);

testFindByNameContaining();

testFindByNameStartingWith();

testFindTop3ByOrderByNameAsc();

testFindAllByOrderByNameDesc();

}

**Expected Output (Sample Logs)**

Start: testFindByNameContaining

Country: Country [code=IS, name=Iceland]

Country: Country [code=TH, name=Thailand]

End

Start: testFindByNameStartingWith

Country: Country [code=AF, name=Afghanistan]

Country: Country [code=AL, name=Albania]

...

End

Start: testFindTop3ByOrderByNameAsc

Country: Country [name=Afghanistan]

Country: Country [name=Albania]

Country: Country [name=Algeria]

End

**Notes**

* Spring automatically implements the repository methods at runtime using proxy and reflection.
* You do **not** need to write custom queries if the method name follows conventions.
* These query methods are ideal for **search**, **filter**, **pagination**, and **sorting**.

**Demonstrate implementation of O/R Mapping**

**Hands-on 9: Demonstrating O/R Mapping in Spring Data JPA**

In this hands-on, we will demonstrate how to establish relationships between entities using annotations provided by JPA for **Object/Relational (O/R) Mapping**.

**Types of Relationships in JPA**

**1. @ManyToOne and @JoinColumn**

**Example: Employee and Department (Many employees belong to one department)**

@Entity

public class Employee {

@Id

private int id;

private String name;

@ManyToOne(fetch = FetchType.EAGER) // default fetch type

@JoinColumn(name = "dept\_id") // foreign key column

private Department department;

// Getters and setters

}

@Entity

public class Department {

@Id

private int id;

private String name;

// Getters and setters

}

**2. @OneToMany with mappedBy**

**Example: Department has multiple Employees**

@Entity

public class Department {

@Id

private int id;

private String name;

@OneToMany(mappedBy = "department", fetch = FetchType.LAZY)

private List<Employee> employees;

// Getters and setters

}

* mappedBy = "department" means the department field in Employee owns the relationship.
* Use FetchType.LAZY to load employees only when accessed.

**3. @ManyToMany and @JoinTable**

**Example: Student and Course have many-to-many relationship**

@Entity

public class Student {

@Id

private int id;

private String name;

@ManyToMany

@JoinTable(name = "student\_course",

joinColumns = @JoinColumn(name = "student\_id"),

inverseJoinColumns = @JoinColumn(name = "course\_id"))

private Set<Course> courses;

// Getters and setters

}

@Entity

public class Course {

@Id

private int id;

private String title;

@ManyToMany(mappedBy = "courses")

private Set<Student> students;

// Getters and setters

}

* @JoinTable defines the join (junction) table.
* mappedBy on the other side avoids duplication of the relationship table.

**Fetch Strategies**

| **Annotation** | **Description** |
| --- | --- |
| FetchType.EAGER | Loads the related entity **immediately** with the parent |
| FetchType.LAZY | Loads the related entity **only when accessed**, improves performance |

**Key Annotations Summary**

| **Annotation** | **Purpose** |
| --- | --- |
| @ManyToOne | Many records in current entity map to one record in target entity |
| @OneToMany | One record in current entity maps to many in target entity |
| @ManyToMany | Many-to-many mapping between both entities |
| @JoinColumn | Defines foreign key column |
| @JoinTable | Defines mapping table for many-to-many relationships |
| mappedBy | Declares the owning side of a bidirectional relationship |

**Demonstrate writing Hibernate Query Language and Native Query**

**Hands-on 10: Writing Hibernate Query Language (HQL) and Native Queries**

This section focuses on writing custom queries using **HQL/JPQL** and **Native SQL** in Spring Data JPA with @Query annotation.

**What is HQL and JPQL?**

| **Feature** | **HQL (Hibernate Query Language)** | **JPQL (Java Persistence Query Language)** |
| --- | --- | --- |
| Purpose | Hibernate’s own query language | Standard defined by JPA specification |
| Entity-based | Yes | Yes |
| Vendor-specific | Yes (Hibernate only) | No (Portable across JPA providers) |
| Querying Tables | Uses entity names | Uses entity names |
| Database portability | Limited | High |

**Note**: In Spring Data JPA, JPQL is generally used and referred to as HQL interchangeably when using Hibernate.

**Using @Query Annotation**

Custom queries can be defined using the @Query annotation in the repository interface.

**🔹 Syntax for JPQL (HQL)**

@Query("SELECT c FROM Country c WHERE c.name LIKE %:keyword%")

List<Country> searchCountriesByName(@Param("keyword") String keyword);

* Uses entity name and field names.
* Portable across databases.

**Aggregate Functions in HQL**

You can use aggregate functions like COUNT, SUM, AVG, MAX, MIN.

@Query("SELECT COUNT(c) FROM Country c")

long countAllCountries();

**HQL fetch Keyword**

Used to fetch associated entities eagerly (overriding LAZY behavior).

@Query("SELECT d FROM Department d JOIN FETCH d.employees")

List<Department> findAllDepartmentsWithEmployees();

* JOIN FETCH forces the loading of child collection/entities.

**Native Queries in Spring Data JPA**

You can also write **native SQL** queries using nativeQuery = true.

**🔹 Example: Native Query**

@Query(value = "SELECT \* FROM country WHERE co\_name LIKE %:keyword%", nativeQuery = true)

List<Country> findCountriesUsingNativeQuery(@Param("keyword") String keyword);

* **Pros**: Flexibility and full access to SQL features
* **Cons**: Tightly coupled with specific DB (non-portable)

**Comparing JPQL (HQL) vs Native Query**

| **Feature** | **JPQL / HQL** | **Native Query** |
| --- | --- | --- |
| Portability | High | Low |
| Syntax | Based on entities and fields | Based on actual DB tables and columns |
| Use case | Preferred for most cases | Complex queries, performance tuning |
| Performance tuning | May be limited | Allows full control |

**Example Use Cases in Repository Interface**

@Repository

public interface CountryRepository extends JpaRepository<Country, String> {

// JPQL - Find country by code

@Query("SELECT c FROM Country c WHERE c.code = :code")

Country getByCode(@Param("code") String code);

// Native Query - Retrieve all countries

@Query(value = "SELECT \* FROM country", nativeQuery = true)

List<Country> getAllNative();

// JPQL - Count total countries

@Query("SELECT COUNT(c) FROM Country c")

long getTotalCount();

// JPQL - Find countries that start with given prefix

@Query("SELECT c FROM Country c WHERE c.name LIKE :prefix%")

List<Country> findByNamePrefix(@Param("prefix") String prefix);

}