Additinoal Important Hands On for spring data jpa

## 1. Demonstrate O/R Mapping

Object-Relational Mapping (ORM) is a technique for converting data between incompatible type systems, in this case, between Java objects and a relational database. JPA (Java Persistence API) is the standard specification for ORM in Java.

The following Country.java class demonstrates ORM. It maps a Java Country object to a countries table in a database using JPA annotations.

@Entity: Marks this class as a database entity.

@Table(name = "countries"): Specifies that this entity maps to the countries table.

@Id: Declares the id field as the primary key.

@GeneratedValue: Configures the primary key generation strategy.

@Column: Maps the fields code and name to specific columns in the table, defining constraints like uniqueness and nullability.

import jakarta.persistence.Column;

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.GenerationType;

import jakarta.persistence.Id;

import jakarta.persistence.Table;

@Entity

@Table(name = "countries")

public class Country {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column(name = "country\_code", unique = true, nullable = false, length = 2)

private String code;

@Column(name = "name", nullable = false)

private String name;

// Constructors, Getters, and Setters

public Country() {}

public Country(String code, String name) {

this.code = code;

this.name = name;

}

// Getters and Setters ...

public Long getId() { return id; }

public void setId(Long id) { this.id = id; }

public String getCode() { return code; }

public void setCode(String code) { this.code = code; }

public String getName() { return name; }

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

}

## 2. Demonstrate Spring Data JPA Query Methods

Spring Data JPA's Query Methods feature allows you to create database queries automatically by simply defining method signatures in a repository interface. Spring Data derives the query from the method's name.

The CountryRepository interface below demonstrates this.

findByCode(String code): Spring Data understands this method name and automatically generates a query equivalent to SELECT c FROM Country c WHERE c.code = ?1.

findByNameStartingWith(String prefix): This method will generate a query to find all countries whose names start with the given prefix, like SELECT c FROM Country c WHERE c.name LIKE ?1%

.

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

import org.springframework.stereotype.Repository;

import java.util.List;

import java.util.Optional;

@Repository

public interface CountryRepository extends JpaRepository<Country, Long> {

/\*\*

\* Query method: Finds a Country by its 'code' field.

\* Spring Data JPA generates the query from the method name.

\*/

Optional<Country> findByCode(String code);

/\*\*

\* Query method: Finds all countries where the name starts with the given prefix.

\*/

List<Country> findByNameStartingWith(String prefix);

## }

## 3. Implement Services for Managing Country

The service layer contains the application's business logic. It acts as an intermediary between the controller (or API endpoint) and the data access layer (repository). This promotes separation of concerns.

The CountryService class below is responsible for orchestrating operations related to countries. It is injected with the CountryRepository to perform database operations.

java

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Service;

**import** java.util.Optional;

@Service

**public** **class** CountryService {

**private** **final** CountryRepository countryRepository; *// Dependency Injection of the repository via the constructor*

@Autowired

**public** CountryService(CountryRepository countryRepository) { **this**.countryRepository = countryRepository; } */\*\* \* Business logic to add a new country. \* @param country The country to save. \* @return The saved country entity. \*/*

**public** Country addNewCountry(Country country) { *// You could add business validation here, e.g., check if code is uppercase*

**return** countryRepository.save(country); } */\*\* \* Business logic to find a country by its code. \* @param code The two-letter country code. \* @return An Optional containing the country if found. \*/*

**public** Optional<Country> findCountryByCode(String code) {

**return** countryRepository.findByCode(code); }}

## 4. Add a New Country

This service method demonstrates how to add a new Country to the database. It uses the save() method, which is provided by the JpaRepository that our CountryRepository extends. The save() method can be used for both creating a new record (if the entity is new) and updating an existing one.

This logic is implemented within the CountryService.

**Import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Service;

@Service

**public** **class** CountryService {

**private** **final** CountryRepository countryRepository; @Autowired

**public** CountryService(CountryRepository countryRepository) { **this**.countryRepository = countryRepository; } */\*\* \* Adds a new country to the database by calling the repository's save() method. \* The save() method persists the given entity. \* \* @param country The Country object to be saved. \* @return The saved Country object, which now includes the database-generated ID. \*/*

**public** Country addNewCountry(Country country) { *// Example usage:* *// Country newCountry = new Country("IN", "India");* *// Country savedCountry = countryService.addNewCountry(newCountry);*

**return** countryRepository.save(country); }}

## 5. Find a Country Based on Country Code

This service method finds a Country using its unique two-letter code. It calls the findByCode(String code) query method defined in our CountryRepository. This illustrates how the service layer uses the repository to fetch data without needing to know the underlying query details.

This logic is also implemented within the CountryService.

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Service;

**import** java.util.Optional;

@Service

**public** **class** CountryService {

**private** **final** CountryRepository countryRepository; @Autowired

**public** CountryService(CountryRepository countryRepository) { **this**.countryRepository = countryRepository; } */\*\* \* Finds a country based on its two-letter country code. \* This method delegates the call to the repository's derived query method. \* \* @param code The two-letter country code (e.g., "US", "IN"). \* @return An Optional containing the found Country, or an empty Optional if not found. \*/*

**public** Optional<Country> findCountryByCode(String code) {

**return** countryRepository.findByCode(code); }}

## 6. Demonstrate HQL and Native Query

For more complex or specific queries, you can use the @Query annotation in your repository. This allows you to write custom queries using either Hibernate Query Language (HQL/JPQL) or native SQL.

## Hibernate Query Language (HQL)

HQL is an object-oriented query language, similar to SQL, but it operates on persistent objects (entities) and their properties rather than database tables and columns. This makes it database-independent.

The method findByCountryNameHQL uses an HQL query to find countries by name.

**import** org.springframework.data.jpa.repository.Query;

**import** org.springframework.data.repository.query.Param;

*// ... other imports*

**public** **interface** CountryRepository **extends** JpaRepository<Country, Long> { */\*\* \* Demonstrates using Hibernate Query Language (HQL). \* The query is written in terms of JPA entities ("Country") and fields ("name"). \* This is portable across different relational databases. \*/* @Query("SELECT c FROM Country c WHERE c.name = :countryName") List<Country> findByCountryNameHQL(@Param("countryName") String countryName);}

## Native Query

A native query uses the specific SQL dialect of your target database. This is useful for leveraging database-specific features or for complex queries that are difficult to express in HQL. You must set the nativeQuery = true flag.

The method findByCountryCodeNative uses a native SQL query. Notice it refers to the table countries and column country\_code directly.

**import** org.springframework.data.jpa.repository.Query;

**import** org.springframework.data.repository.query.Param;

*// ... other imports*

**public** **interface** CountryRepository **extends** JpaRepository<Country, Long> { */\*\* \* Demonstrates a native SQL query. \* The 'nativeQuery = true' flag tells Spring to execute this as a raw SQL statement. \* Note that it uses the actual table name 'countries' and column name 'country\_code'. \*/*

@Query(value = "SELECT \* FROM countries c WHERE c.country\_code = :countryCode", nativeQuery = true) Optional<Country> findByCountryCodeNative(@Param("countryCode") String countryCode);}