Document: Implementing Country Management Using Spring Data JPA

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Introduction

This document describes the end-to-end implementation of a Country Management module using Spring Boot and Spring Data JPA. The goal is to provide CRUD operations and search functionality for a Country entity stored in a MySQL database.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Objective

To implement RESTful services for the following features:

• Find a country based on country code

• Add a new country

• Update an existing country

• Delete a country

• Find a list of countries matching a partial country name

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Technology Stack

• Spring Boot

• Spring Data JPA

• MySQL

• Maven

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Project Setup

Create a new Spring Boot project named CountryManagement using Spring Initializr with the following dependencies:

• Spring Web

• Spring Data JPA

• MySQL Driver

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2. Configuration - application.properties

spring.datasource.url=jdbc:mysql://localhost:3306/your\_database\_name

spring.datasource.username=root

spring.datasource.password=your\_password

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

3. Country Entity

package com.example.country.model;

import jakarta.persistence.\*;

@Entity

@Table(name = "country")

public class Country {

@Id

@Column(name = "co\_code")

private String code;

@Column(name = "co\_name")

private String name;

// Getters and Setters

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; }

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

4. Repository Interface

package com.example.country.repository;

import com.example.country.model.Country;

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

import java.util.List;

public interface CountryRepository extends JpaRepository<Country, String> {

List<Country> findByNameContainingIgnoreCase(String name);

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

5. Service Layer

package com.example.country.service;

import com.example.country.model.Country;

import com.example.country.repository.CountryRepository;

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

import org.springframework.stereotype.Service;

import java.util.\*;

@Service

public class CountryService {

@Autowired

private CountryRepository countryRepository;

public Optional<Country> findByCode(String code) {

return countryRepository.findById(code);

}

public List<Country> searchByName(String name) {

return countryRepository.findByNameContainingIgnoreCase(name);

}

public Country addCountry(Country country) {

return countryRepository.save(country);

}

public Country updateCountry(Country country) {

return countryRepository.save(country);

}

public void deleteCountry(String code) {

countryRepository.deleteById(code);

}

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

6. REST Controller

package com.example.country.controller;

import com.example.country.model.Country;

import com.example.country.service.CountryService;

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

import org.springframework.web.bind.annotation.\*;

import java.util.\*;

@RestController

@RequestMapping("/countries")

public class CountryController {

@Autowired

private CountryService countryService;

@GetMapping("/{code}")

public Optional<Country> getCountryByCode(@PathVariable String code) {

return countryService.findByCode(code);

}

@GetMapping("/search")

public List<Country> searchCountries(@RequestParam String name) {

return countryService.searchByName(name);

}

@PostMapping

public Country addCountry(@RequestBody Country country) {

return countryService.addCountry(country);

}

@PutMapping

public Country updateCountry(@RequestBody Country country) {

return countryService.updateCountry(country);

}

@DeleteMapping("/{code}")

public void deleteCountry(@PathVariable String code) {

countryService.deleteCountry(code);

}

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

7. Testing Using Postman

• GET /countries/IN → Returns country with code IN

• GET /countries/search?name=ind → Returns countries with “ind” in the name

• POST /countries with JSON body → Adds a country

• PUT /countries with JSON body → Updates a country

• DELETE /countries/IN → Deletes country with code IN

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

8. Populating the Country Table

Use the provided SQL insert statements to populate the country table. Ensure that the ddl-auto=validate once data is populated to avoid accidental schema change.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Conclusion

This implementation demonstrates how to use Spring Boot and Spring Data JPA to build a REST API for managing country data. The configuration ensures smooth integration with MySQL and offers robust methods for CRUD operations and search functionality.

**Exercise: Find a Country Based on Country Code Using Spring Data JPA**

**Introduction:** This exercise focuses on implementing a feature in a Spring Boot application that retrieves a country based on its country code using Spring Data JPA. The implementation includes proper exception handling and transaction management.

**Objective:** To implement the method findCountryByCode(String countryCode) in a service class to retrieve a country using Spring Data JPA. If the country is not found, a custom exception should be thrown.

**Steps:**

1. **Create the Exception Class:**
   * Package: com.cognizant.spring\_learn.service.exception
   * Class Name: CountryNotFoundException

**package com**.**cognizant**.**spring\_learn**.**service**.**exception**;  
  
**public** **class** CountryNotFoundException **extends** Exception {  
 **public** CountryNotFoundException(String message) {  
 **super**(message);  
 }  
}

1. **Update the Service Class:**
   * Method Name: findCountryByCode
   * Signature:

@Transactional  
**public** Country findCountryByCode(String countryCode) **throws** CountryNotFoundException {  
 Optional<Country> result = countryRepository.findById(countryCode);  
 **if** (!result.isPresent()) {  
 **throw** **new** CountryNotFoundException("Country not found with code: " + countryCode);  
 }  
 **return** result.get();  
}

1. **Usage of @Transactional:**
   * The @Transactional annotation ensures that the method executes within a transactional context.
   * Spring handles the opening/closing of the Hibernate session and ensures data consistency.
2. **Add Test Method:**
   * In the OrmLearnApplication class:

**private** static void getAllCountriesTest() {  
 LOGGER.info("Start");  
 Country country = countryService.findCountryByCode("IN");  
 LOGGER.debug("Country:{}", country);  
 LOGGER.info("End");  
}

1. **Invoke Test Method in main():**

**public** static void main(String[] args) {  
 SpringApplication.run(OrmLearnApplication.class, args);  
 getAllCountriesTest();  
}

**Conclusion:** By completing this exercise, we implemented a robust service layer method to fetch a country by its code. We ensured the method is transactional and used custom exception handling to manage error cases, adhering to clean and maintainable Spring Boot practices.

**Document: Adding a New Country using Spring Data JPA**

**Introduction:** This document outlines the step-by-step implementation of adding a new country record to the database using Spring Data JPA in a Spring Boot application. This includes service-level logic, repository interaction, and testing.

**Objective:** To implement a feature that allows the addition of a new country to the database using the CountryService class with proper transactional support and testing.

**Technologies Used:**

* Java
* Spring Boot
* Spring Data JPA
* H2/MySQL Database
* SLF4J (Logging)

**Step-by-Step Implementation:**

1. Update CountryService to Add New Country

Create a method named addCountry() in the CountryService class:

**import** **org**.**springframework**.**transaction**.**annotation**.**Transactional**;  
  
@Service  
**public** **class** CountryService {  
  
 @Autowired  
 **private** CountryRepository countryRepository;  
  
 @Transactional  
 **public** void addCountry(Country country) {  
 countryRepository.save(country);  
 }  
}

2. Create a Test Method in OrmLearnApplication

Add a new test method to the main application class (e.g., OrmLearnApplication) to validate the add operation:

**private** static void testAddCountry() {  
 LOGGER.info("Start");  
  
 Country newCountry = **new** Country();  
 newCountry.setCode("ZZ");  
 newCountry.setName("Zootopia");  
  
 countryService.addCountry(newCountry);  
  
 **try** {  
 Country retrieved = countryService.findCountryByCode("ZZ");  
 LOGGER.debug("Country Retrieved: {}", retrieved);  
 } **catch** (CountryNotFoundException e) {  
 LOGGER.error("Country not found after add operation");  
 }  
  
 LOGGER.info("End");  
}

3. Invoke testAddCountry() in the main() method:

**public** static void main(String[] args) {  
 ApplicationContext context = SpringApplication.run(OrmLearnApplication.class, args);  
 countryService = context.getBean(CountryService.class);  
  
 testAddCountry();  
}

**Expected Outcome:**

* A new record is added to the country table with the code ZZ and name Zootopia.
* The log displays the newly added country confirming the successful save operation.

**Conclusion:** This exercise demonstrates the implementation of adding a new country using Spring Data JPA. Using @Transactional, we ensure that the operation is properly managed within a transactional context. The addition is verified through logging and service-layer retrieval.

**Title: Implementation of Query Methods using Spring Data JPA**

**Introduction:** Spring Data JPA provides a powerful mechanism to create queries by defining method names in repository interfaces. This eliminates the need to write boilerplate SQL/JPQL. Query methods enhance productivity and support various filters like text matching, range queries, and sorting. This document demonstrates the implementation of Spring Data JPA query methods.

**Objective:** To implement and demonstrate various query methods in a Spring Boot application using Spring Data JPA.

**Technologies Used:**

* Java
* Spring Boot
* Spring Data JPA
* H2 Database

**Steps:**

**1. Define Entity Class (Country.java):**

@Entity  
**public** **class** Country {  
 @Id  
 **private** String code;  
 **private** String name;  
 **private** LocalDate dateOfIndependence; *// Optional field for date-related queries*  
 **private** Long population;  
 *// Getters and Setters*  
}

**2. Create Repository Interface (CountryRepository.java):**

@Repository  
**public** **interface** CountryRepository **extends** JpaRepository<Country, String> {  
 List<Country> findByNameContaining(String keyword);  
 List<Country> findByNameStartingWith(String prefix);  
 List<Country> findByDateOfIndependenceBetween(LocalDate startDate, LocalDate endDate);  
 List<Country> findByPopulationGreaterThan(Long population);  
 List<Country> findByPopulationLessThan(Long population);  
 List<Country> findTop3ByOrderByPopulationDesc();  
 List<Country> findTop3ByOrderByPopulationAsc();  
}

**3. Implement Service Layer (CountryService.java):**

@Service  
**public** **class** CountryService {  
 @Autowired  
 **private** CountryRepository countryRepository;  
  
 **public** List<Country> getCountriesContaining(String keyword) {  
 **return** countryRepository.findByNameContaining(keyword);  
 }  
  
 **public** List<Country> getCountriesStartingWith(String prefix) {  
 **return** countryRepository.findByNameStartingWith(prefix);  
 }  
  
 **public** List<Country> getCountriesBetweenDates(LocalDate start, LocalDate end) {  
 **return** countryRepository.findByDateOfIndependenceBetween(start, end);  
 }  
  
 **public** List<Country> getCountriesWithPopulationGreaterThan(Long pop) {  
 **return** countryRepository.findByPopulationGreaterThan(pop);  
 }  
  
 **public** List<Country> getCountriesWithPopulationLessThan(Long pop) {  
 **return** countryRepository.findByPopulationLessThan(pop);  
 }  
  
 **public** List<Country> getTop3ByHighestPopulation() {  
 **return** countryRepository.findTop3ByOrderByPopulationDesc();  
 }  
  
 **public** List<Country> getTop3ByLowestPopulation() {  
 **return** countryRepository.findTop3ByOrderByPopulationAsc();  
 }  
}

**4. Controller for Testing (CountryController.java):**

@RestController  
@RequestMapping("/countries")  
**public** **class** CountryController {  
  
 @Autowired  
 **private** CountryService countryService;  
  
 @GetMapping("/search")  
 **public** List<Country> search(@RequestParam String keyword) {  
 **return** countryService.getCountriesContaining(keyword);  
 }  
  
 @GetMapping("/start")  
 **public** List<Country> start(@RequestParam String prefix) {  
 **return** countryService.getCountriesStartingWith(prefix);  
 }  
  
 @GetMapping("/between")  
 **public** List<Country> between(@RequestParam String start, @RequestParam String end) {  
 **return** countryService.getCountriesBetweenDates(LocalDate.parse(start), LocalDate.parse(end));  
 }  
  
 @GetMapping("/greater")  
 **public** List<Country> greater(@RequestParam Long pop) {  
 **return** countryService.getCountriesWithPopulationGreaterThan(pop);  
 }  
  
 @GetMapping("/less")  
 **public** List<Country> less(@RequestParam Long pop) {  
 **return** countryService.getCountriesWithPopulationLessThan(pop);  
 }  
  
 @GetMapping("/top3high")  
 **public** List<Country> topHigh() {  
 **return** countryService.getTop3ByHighestPopulation();  
 }  
  
 @GetMapping("/top3low")  
 **public** List<Country> topLow() {  
 **return** countryService.getTop3ByLowestPopulation();  
 }  
}

**5. Sample** ``**:**

spring.datasource.url=jdbc:h2:mem:testdb  
spring.datasource.driver-class-name=org.h2.Driver  
spring.datasource.username=sa  
spring.datasource.password=  
spring.jpa.show-sql=true  
spring.jpa.hibernate.ddl-auto=create  
spring.h2.console.enabled=true

**Conclusion:** This document showcases how Spring Data JPA simplifies querying using method names. It demonstrates text search, pattern matching, date range filtering, and population-based queries using just method signatures without writing SQL or JPQL.

**Exercise: Demonstrate Implementation of O/R Mapping Using Spring Data JPA**

**Introduction:** Object-Relational Mapping (O/R Mapping) is a critical concept in building enterprise applications using Spring Data JPA. It helps map Java objects to relational database tables. In this exercise, we demonstrate various relationships such as @ManyToOne, @OneToMany, @ManyToMany, along with key annotations like @JoinColumn, @JoinTable, mappedBy, and different fetch types (EAGER, LAZY).

**1. Project Setup:**

* Create a Spring Boot project using Spring Initializr with the following dependencies:
  + Spring Web
  + Spring Data JPA
  + H2 Database (for testing)

**2. Define Entities and Relationships:**

**a. Entity: Department**

@Entity  
**public** **class** Department {  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 **private** Long id;  
 **private** String name;  
  
 @OneToMany(mappedBy = "department", fetch = FetchType.LAZY)  
 **private** List<Employee> employees;  
}

**b. Entity: Employee**

@Entity  
**public** **class** Employee {  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 **private** Long id;  
 **private** String name;  
  
 @ManyToOne(fetch = FetchType.EAGER)  
 @JoinColumn(name = "department\_id")  
 **private** Department department;  
}

**c. Entity: Course**

@Entity  
**public** **class** Course {  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 **private** Long id;  
 **private** String title;  
  
 @ManyToMany(mappedBy = "courses")  
 **private** List<Student> students;  
}

**d. Entity: Student**

@Entity  
**public** **class** Student {  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 **private** Long id;  
 **private** String name;  
  
 @ManyToMany  
 @JoinTable(  
 name = "student\_course",  
 joinColumns = @JoinColumn(name = "student\_id"),  
 inverseJoinColumns = @JoinColumn(name = "course\_id")  
 )  
 **private** List<Course> courses;  
}

**3. Explanation of Annotations Used:**

* @ManyToOne: Employee to Department.
* @OneToMany: Department to list of Employees using mappedBy to indicate ownership.
* @JoinColumn: Used to specify the foreign key column.
* @ManyToMany: Student and Course with @JoinTable for a join table.
* mappedBy: Tells JPA which side owns the relationship.
* FetchType.EAGER: Loads related entities immediately.
* FetchType.LAZY: Loads related entities on access.

**4. Testing the Relationships:**

* Use CommandLineRunner or REST controllers to test the insert and retrieval of data.

@Bean  
CommandLineRunner runner(StudentRepository studentRepo, CourseRepository courseRepo) {  
 **return** args -> {  
 Course course = **new** Course();  
 course.setTitle("Mathematics");  
 courseRepo.save(course);  
  
 Student student = **new** Student();  
 student.setName("Alice");  
 student.setCourses(Arrays.asList(course));  
 studentRepo.save(student);  
 };  
}

**5. Output Verification:**

* Verify via H2 Console or application logs that relationships are created and maintained.

**Conclusion:** This exercise provides hands-on understanding of setting up different entity relationships using Spring Data JPA. You’ve learned how to model @ManyToOne, @OneToMany, and @ManyToMany relationships with appropriate annotations and fetch types, which are essential for designing real-world enterprise systems.

**Exercise: Demonstrate Hibernate Query Language and Native Query in Spring Data JPA**

Introduction

This exercise demonstrates the use of Hibernate Query Language (HQL), Java Persistence Query Language (JPQL), and native SQL queries in a Spring Data JPA application. These querying techniques are essential for retrieving data in a more controlled, optimized, and flexible manner than standard CRUD operations.

Technologies Used

* Java
* Spring Boot
* Spring Data JPA
* Hibernate
* H2/MySQL Database

HQL and JPQL

* **JPQL** is defined in the JPA specification and operates on the entity object model.
* **HQL** is a Hibernate-specific implementation that also supports entity querying.
* Both use entity class names instead of table names.

Example: Fetch All Countries Using JPQL

@Query("FROM Country")  
List<Country> findAllCountries();

Fetch by Condition (HQL/JPQL)

@Query("SELECT c FROM Country c WHERE c.name = :name")  
Country findByName(@Param("name") String name);

Use of fetch Keyword

@Query("SELECT c FROM Country c JOIN FETCH c.states")  
List<Country> findAllCountriesWithStates();

Aggregate Functions in JPQL

@Query("SELECT COUNT(c) FROM Country c")  
long countAllCountries();

@Query("SELECT MAX(c.population) FROM Country c")  
long getMaxPopulation();

Native Query Example

@Query(value = "SELECT \* FROM country WHERE co\_name = ?1", nativeQuery = **true**)  
Country findByCountryNameNative(String name);

Comparison of JPQL vs Native Query

| Feature | JPQL/HQL | Native SQL |
| --- | --- | --- |
| Portability | Portable across databases | DB-specific |
| Syntax | Based on entity names | Based on table names |
| Complex Queries | Limited to JPQL capabilities | Full power of SQL |
| Performance | Optimized for ORM | Can leverage DB-specific indexes |

Spring Data JPA Annotations Used

* @Query for custom queries
* @Param for binding query parameters

Conclusion

By leveraging JPQL, HQL, and native queries, Spring Data JPA provides powerful querying capabilities while maintaining abstraction over underlying SQL. Use JPQL/HQL for portability and native queries for fine-tuned performance and DB-specific features.