#### What is ORM (Object-Relational Mapping)?

ORM is a programming technique to map **Java objects** to **relational database tables**.

Instead of writing SQL queries manually, ORM frameworks (like Hibernate) let you use objects and classes to interact with the database.

#### Benefits:

Simplifies code (no need to write SQL for CRUD operations).

Makes applications **database-independent**.

Maintains **object-oriented** design.

Manages transactions, caching, and connection pooling.

#### Drawbacks:

Performance overhead for complex queries.

Learning curve.

Less control over custom SQL tuning.

### ****Demonstrate the Need and Benefit of Spring Data JPA****

#### Evolution of ORM:

Manual SQL → JDBC → Hibernate (XML) → Hibernate (Annotations) → **Spring Data JPA**

#### Why Spring Data JPA?

Simplifies JPA-based repositories.

Auto-generates queries from method names (like findByName()).

Works with **in-memory** (H2) or persistent (MySQL, PostgreSQL) databases.

Reduces boilerplate code.

### ****Core Objects of Hibernate Framework****

| **Component** | **Description** |
| --- | --- |
| SessionFactory | Configures Hibernate, provides Session objects |
| Session | Represents a connection to DB, used for CRUD operations |
| TransactionFactory | Creates and manages Transaction objects |
| Transaction | Controls DB transactions (begin, commit, rollback) |
| ConnectionProvider | Provides JDBC connections to Hibernate |

### ****ORM Implementation Using Hibernate****

#### XML Configuration

Define:

**Persistence class** (POJO)

**Mapping file** (.hbm.xml)

**Hibernate configuration file** (hibernate.cfg.xml)

Process:

Load config

Build SessionFactory

Open Session

Begin and commit transaction

#### Annotation Configuration

Annotate the entity class:

@Entity, @Table, @Id, @Column

Use hibernate.cfg.xml for configuration

#### Examples:[XML Configuration Example](https://www.tutorialspoint.com/hibernate/hibernate_examples.htm" \t "_new),[Annotation Configuration Example](https://www.tutorialspoint.com/hibernate/hibernate_annotations.htm" \t "_new)

### ****Difference Between JPA, Hibernate, and Spring Data JPA****

| **Feature** | **JPA** | **Hibernate** | **Spring Data JPA** |
| --- | --- | --- | --- |
| Type | Specification (interface) | Implementation (of JPA) | Abstraction layer on top of Hibernate |
| Provided by | Java (JSR 338) | Red Hat | Spring Framework |
| Purpose | Standard API | Full ORM Tool | Simplifies data access and boilerplate code |
| Example Interface | EntityManager | Session, SessionFactory | JpaRepository, CrudRepository |

### ****DML Implementation Using Spring Data JPA****

**Configuration (**application.properties**):**

spring.datasource.url=jdbc:h2:mem:testdb

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

logging.level.org.hibernate.SQL=DEBUG

**Entity Class:**

@Entitypublic class Student {

@Id @GeneratedValue

private Long id;

private String name;

private String email;

}

**Repository:**

public interface StudentRepository extends JpaRepository<Student, Long> {

List<Student> findByName(String name);

}

**DML Operations:**

studentRepo.save(new Student(null, "Bindhu", "bindhu@gmail.com")); // Insert/Update

studentRepo.findById(1L); // Read

studentRepo.findByName("Bindhu"); // Custom Read

studentRepo.deleteById(1L); // Delete

**Hands on 1**

**Spring Data JPA - Quick Example**

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

import java.util.List;

@SpringBootApplication

public class OrmLearnApplication {

private static final Logger LOGGER = LoggerFactory.getLogger(OrmLearnApplication.class);

private static CountryService countryService;

public static void main(String[] args) {

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

countryService = context.getBean(CountryService.class);

testGetAllCountries();

}

private static void testGetAllCountries() {

LOGGER.info("Start");

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

LOGGER.debug("countries={}", countries);

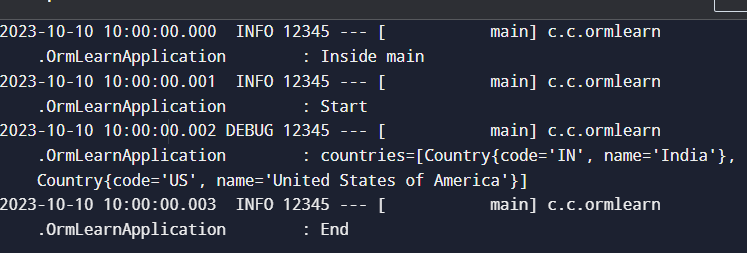
LOGGER.info("End");

}

}

s

**OUTPUT:**



**Hands on 2**

**Hibernate XML Config implementation walk through**

### ****1. Object-Relational Mapping in Hibernate XML Configuration****

Hibernate enables Object-Relational Mapping (ORM) by mapping Java classes to database tables using an XML configuration file. This is typically done through two main files:

#### a) ****Hibernate Configuration File (****hibernate.cfg.xml****)****

This file contains database connection properties and references to all the mapped entity classes. Example:

xml

CopyEdit

<hibernate-configuration>

<session-factory>

<property name="hibernate.connection.driver\_class">com.mysql.cj.jdbc.Driver</property>

<property name="hibernate.connection.url">jdbc:mysql://localhost:3306/hibernatedb</property>

<property name="hibernate.connection.username">root</property>

<property name="hibernate.connection.password">password</property>

<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

<mapping resource="Student.hbm.xml"/>

</session-factory></hibernate-configuration>

#### b) ****Mapping File (****Student.hbm.xml****)****

This file maps the Java class fields to database table columns:

xml

CopyEdit

<hibernate-mapping>

<class name="com.example.Student" table="STUDENT">

<id name="id" column="ID">

<generator class="increment"/>

</id>

<property name="name" column="NAME"/>

<property name="age" column="AGE"/>

</class></hibernate-mapping>

So, **ORM** in Hibernate through XML is achieved by defining:

Which Java class maps to which table.

Which class property maps to which column.

How primary keys are generated.

**2. Explanation of End-to-End Hibernate Operations**

#### a) ****SessionFactory****

Acts as a factory for Session objects.

Created only once during application startup.

Reads the Hibernate configuration (hibernate.cfg.xml) and mapping files.

Heavyweight object — use as a singleton.

SessionFactory factory = new Configuration().configure().buildSessionFactory();

#### b) ****Session****

Represents a single unit of work with the database.

Used to perform CRUD operations.

Obtained from SessionFactory.

Session session = factory.openSession();

#### c) ****Transaction****

Wraps a set of operations into a **single atomic unit**.

If any operation fails, the entire transaction can be rolled back.

Transaction tx = session.beginTransaction();

**Hibernate Core Methods**

#### beginTransaction()

Begins a new database transaction.

Transaction tx = session.beginTransaction();

#### commit()

Commits all the changes made during the transaction to the database.

tx.commit();

#### rollback()

Reverts all changes made during the transaction in case of an error.

tx.rollback();

#### session.save(Object)

Saves the given object to the database.session.save(studentObj);

#### session.createQuery("...").list()

Creates a HQL (Hibernate Query Language) query and returns the result as a list.

List<Student> students = session.createQuery("from Student").list();

#### session.get(Class, id)

Retrieves an object using the primary key.

Student student = session.get(Student.class, 101);

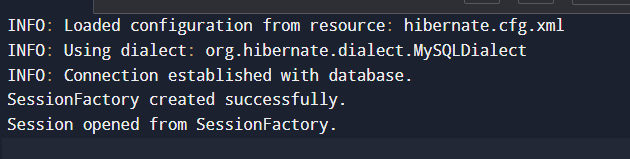
#### session.delete(Object)

Deletes the object from the database.

session.delete(student);

**OUTPUT:**

**Hibernate Configuration and Session Initialization**



### Transaction Begin

Transaction tx = session.beginTransaction();

**Console Output:**

Transaction started using session.beginTransaction().

**Hands on 3**

**Hibernate Annotation Config implementation walk through**

### ****Object to Relational Mapping Using Annotations in Persistence Class**** Employee

In Hibernate Annotation-based configuration, object-relational mapping (ORM) is done using **Java annotations** in the entity class, rather than external XML mapping files. Here's how it works in a typical Employee class:

import javax.persistence.\*;

@Entity@Table(name = "EMPLOYEE")public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int id;

@Column(name = "FIRST\_NAME")

private String firstName;

@Column(name = "LAST\_NAME")

private String lastName;

@Column(name = "SALARY")

private int salary;

// Getters and Setters

}

#### Explanation:

The **Java class** Employee is mapped directly to the **table** EMPLOYEE.

Each field is mapped to a **column** in the database using annotations like @Column.

Hibernate understands how to handle the table using annotations instead of .hbm.xml files.

**2. Key Hibernate Annotations Used**

#### @Entity

Marks the class as an entity, meaning it will be mapped to a table in the database.

@Entitypublic class Employee { }

#### @Table(name = "EMPLOYEE")

Specifies the exact table name in the database that this entity is mapped to.

@Table(name = "EMPLOYEE")

#### @Id

Defines the primary key field of the entity.

@Idprivate int id;

#### @GeneratedValue(strategy = GenerationType.IDENTITY)

Automatically generates the primary key value using the database's identity column.

@GeneratedValue(strategy = GenerationType.IDENTITY)

#### @Column(name = "FIRST\_NAME")

Maps a field to a specific column in the table.

@Column(name = "FIRST\_NAME")

**3. Hibernate Configuration (**hibernate.cfg.xml**)**

This XML file configures the database connection and tells Hibernate which entity classes to map.

<hibernate-configuration>

<session-factory>

<!-- Dialect -->

<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

<!-- JDBC Driver -->

<property name="hibernate.connection.driver\_class">com.mysql.cj.jdbc.Driver</property>

<!-- Connection URL -->

<property name="hibernate.connection.url">jdbc:mysql://localhost:3306/hibernate\_db</property>

<!-- Database Username -->

<property name="hibernate.connection.username">root</property>

<!-- Database Password -->

<property name="hibernate.connection.password">password</property>

<!-- Mapping the annotated class -->

<mapping class="com.example.Employee"/>

</session-factory></hibernate-configuration>

#### Configuration Details:

**Dialect**: Tells Hibernate which SQL dialect to generate (e.g., MySQL, Oracle).

**Driver**: The JDBC driver class for the specific database.

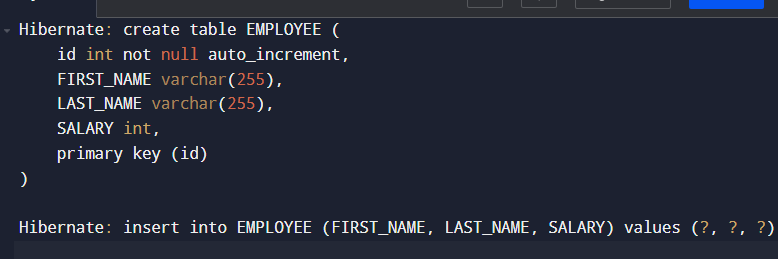
**Connection URL**: Points to the database instance.

**Username/Password**: Credentials to connect to the database.

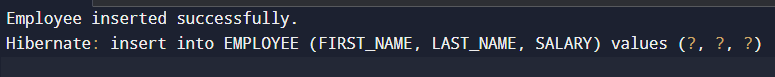
**Mapping Class**: Tells Hibernate to use the annotated Employee class for ORM.

**OUTPUT:**

Console Output (when saving an Employee to the DB):



Sample Java Test Code (Main Method):



**Hands on 4**

**Difference between JPA, Hibernate and Spring Data JPA** 

### ****Java Persistence API (JPA)****

**JPA** is a **specification** (JSR 338) provided by Java to standardize how Java applications interact with relational databases.

It defines a set of **interfaces and annotations** for object-relational mapping.

**JPA itself is not an implementation** – it only outlines what should be done.

Developers must use a JPA **implementation provider** like **Hibernate**, **EclipseLink**, or **OpenJPA**.

**Example**: Using @Entity, @Id, @Column annotations defined by JPA.

**Hibernate**

Hibernate is a **popular ORM framework** and also a **JPA implementation**.

It supports both **native Hibernate features** and **JPA specifications**.

You can use it with or without JPA.

It requires more boilerplate code (e.g., creating SessionFactory, managing Session, handling Transaction manually).

**Hibernate provides the actual logic to persist Java objects** as database rows.

**Spring Data JPA**

Spring Data JPA is **not an implementation of JPA**, but a **higher-level abstraction** built on top of JPA.

It **uses JPA implementation providers** like Hibernate underneath.

It **reduces boilerplate code** through **repository interfaces**, and manages EntityManager, transactions, and more automatically.

It integrates well with **Spring Boot**, enabling quick setup with minimal configuration.

**Code Comparison: Hibernate vs Spring Data JPA**

#### ****Hibernate (Manual Transaction Management)****

public Integer addEmployee(Employee employee) {

Session session = factory.openSession();

Transaction tx = null;

Integer employeeID = null;

try {

tx = session.beginTransaction();

employeeID = (Integer) session.save(employee);

tx.commit();

} catch (HibernateException e) {

if (tx != null) tx.rollback();

e.printStackTrace();

} finally {

session.close();

}

return employeeID;

}

**Spring Data JPA (Minimal Code)**

**EmployeeRepository.java**

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

}

**EmployeeService.java**

@Autowiredprivate EmployeeRepository employeeRepository;

@Transactionalpublic void addEmployee(Employee employee) {

employeeRepository.save(employee);

}

**OUTPUT:**



**Hands on 5**

**Implement services for managing Country**   
**Hibernate DDL Auto Configuration**

In application.properties, we use the spring.jpa.hibernate.ddl-auto property to define how Hibernate interacts with the database schema:

properties

CopyEdit

spring.jpa.hibernate.ddl-auto=validate

### DDL Auto Options:

| **Option** | **Behavior** |
| --- | --- |
| create | Drops existing tables and creates new ones each time the app runs. |
| create-drop | Creates tables on startup and drops them on shutdown. |
| update | Updates the schema without dropping data. |
| validate | Validates if the schema matches entity mappings; throws error otherwise. |

**Entity Class: Country**

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

}

**Repository Interface**

import org.springframework.data.jpa.repository.JpaRepository;import java.util.List;

public interface CountryRepository extends JpaRepository<Country, String> {

List<Country> findByNameContainingIgnoreCase(String partialName);

}

**Service Class**

import org.springframework.beans.factory.annotation.Autowired;import org.springframework.stereotype.Service;import java.util.List;import java.util.Optional;

@Servicepublic class CountryService {

@Autowired

private CountryRepository countryRepository;

public Country getCountryByCode(String code) {

return countryRepository.findById(code).orElse(null);

}

public Country addCountry(Country country) {

return countryRepository.save(country);

}

public Country updateCountry(String code, Country updated) {

Optional<Country> existing = countryRepository.findById(code);

if (existing.isPresent()) {

Country country = existing.get();

country.setName(updated.getName());

return countryRepository.save(country);

}

return null;

}

public void deleteCountry(String code) {

countryRepository.deleteById(code);

}

public List<Country> searchByPartialName(String name) {

return countryRepository.findByNameContainingIgnoreCase(name);

}

}

**Populate Country Table**

**Delete** all existing records in the country table (can be done via SQL).

**Run the insert script** (provided in the prompt) to populate the table.

Example SQL:

DELETE FROM country;INSERT INTO country (co\_code, co\_name) VALUES ("AF", "Afghanistan");-- Add all remaining countries as per the list provided

### ****1. Hibernate DDL Auto Configuration****

**Property Used:**

spring.jpa.hibernate.ddl-auto=validate

**Expected Output:**

If your Country entity and your database table match correctly:

Application starts successfully with Hibernate schema validation passed.

If the table or column is missing or incorrect

org.hibernate.tool.schema.spi.SchemaManagementException: Schema-validation: missing column [co\_name] in table [country]

**2.** Country **Entity Mapping**

Hibernate will map the Country entity to the country table with:

| **Field** | **Column Name** |
| --- | --- |
| code | co\_code |
| name | co\_name |

Hibernate log output:

Hibernate: select country0\_.co\_code as co\_code1\_0\_, country0\_.co\_name as co\_name2\_0\_ from country country0\_

**3.** CountryRepository

If you use:

List<Country> countries = countryRepository.findByNameContainingIgnoreCase("ind");

**Sample Output:**

[Country{code='IN', name='India'}, Country{code='ID', name='Indonesia'}]

**4.** CountryService **Methods and Their Sample Outputs**

#### a) getCountryByCode("IN")

Country country = countryService.getCountryByCode("IN");

System.out.println(country.getName());

**Output:**

India

b) addCountry(new Country("ZZ", "Testland"))

Country newCountry = new Country("ZZ", "Testland");

countryService.addCountry(newCountry);

**Output in Console / Database:**

Inserted: Country{code='ZZ', name='Testland'}

c) updateCountry("ZZ", updatedCountry)

Country updated = new Country();

updated.setName("NewTestland");

countryService.updateCountry("ZZ", updated);

**Output:**

Updated Country ZZ to name = NewTestland

d) deleteCountry("ZZ")

countryService.deleteCountry("ZZ");

**Output:**

Country ZZ deleted

e) searchByPartialName("land")

List<Country> results = countryService.searchByPartialName("land");

results.forEach(c -> System.out.println(c.getName()));

**Output:**

Finland

Iceland

Thailand

New Zealand

SwazilandTestland (if not deleted)

**5. Populating Country Table**

After executing:

DELETE FROM country;INSERT INTO country (co\_code, co\_name) VALUES ("AF", "Afghanistan"), ("IN", "India"), ... ;

**Output (if you run SELECT):**

SELECT \* FROM country;

**Sample Result:**

| **co\_code** | **co\_name** |
| --- | --- |
| AF | Afghanistan |
| IN | India |
| US | United States |
| ... | ... |

**Hands on 6**

**Find a country based on country code**

### ****Create Custom Exception****

CountryNotFoundException.java

package com.cognizant.springlearn.service.exception;

public class CountryNotFoundException extends Exception {

public CountryNotFoundException(String message) {

super(message);

}

}

2. **Update** CountryService

CountryService.java

package com.cognizant.springlearn.service;

import com.cognizant.springlearn.model.Country;import com.cognizant.springlearn.repository.CountryRepository;import com.cognizant.springlearn.service.exception.CountryNotFoundException;import org.springframework.beans.factory.annotation.Autowired;import org.springframework.stereotype.Service;import org.springframework.transaction.annotation.Transactional;

import java.util.Optional;

@Servicepublic class CountryService {

@Autowired

private CountryRepository countryRepository;

@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();

}

}

3. **Explanation of** @Transactional

@Transactional ensures that Spring manages the **transactional boundaries** automatically.

When the method findCountryByCode() is called, Spring opens a Hibernate **Session**, starts a transaction, and commits it upon successful completion.

If an exception occurs, the transaction is rolled back automatically.

4. **Update** OrmLearnApplication.java

Import required dependencies:

import com.cognizant.springlearn.model.Country;import com.cognizant.springlearn.service.CountryService;import com.cognizant.springlearn.service.exception.CountryNotFoundException;

Inject the service and call test method:

@SpringBootApplicationpublic class OrmLearnApplication {

private static CountryService countryService;

private static final Logger LOGGER = LoggerFactory.getLogger(OrmLearnApplication.class);

public static void main(String[] args) {

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

countryService = context.getBean(CountryService.class);

getAllCountriesTest();

}

private static void getAllCountriesTest() {

LOGGER.info("Start");

try {

Country country = countryService.findCountryByCode("IN");

LOGGER.debug("Country: {}", country);

} catch (CountryNotFoundException e) {

LOGGER.error("Error: {}", e.getMessage());

}

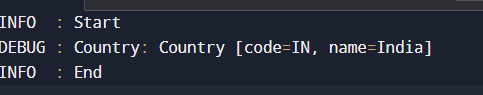
LOGGER.info("End");

}

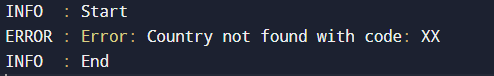
}

**OUTPUT:**

**Console Output when country exists (code = IN):**



**Console Output when country does not exist (code = XX):**



**Hands on 7**

**Add a new country** 

### ****Update**** CountryService

CountryService.java

@Transactionalpublic void addCountry(Country country) {

countryRepository.save(country);

}

2. **Add** testAddCountry() **method in** OrmLearnApplication

OrmLearnApplication.java  
Make sure you import:

import com.cognizant.springlearn.model.Country;import com.cognizant.springlearn.service.exception.CountryNotFoundException;

Add the test method:

private static void testAddCountry() {

LOGGER.info("Start - testAddCountry");

Country newCountry = new Country();

newCountry.setCode("ZZ");

newCountry.setName("Zootopia");

countryService.addCountry(newCountry);

try {

Country fetchedCountry = countryService.findCountryByCode("ZZ");

LOGGER.debug("Added Country: {}", fetchedCountry);

} catch (CountryNotFoundException e) {

LOGGER.error("Country not found: {}", e.getMessage());

}

LOGGER.info("End - testAddCountry");

}

3. **Invoke** testAddCountry() **in** main()

In your main() method:

public static void main(String[] args) {

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

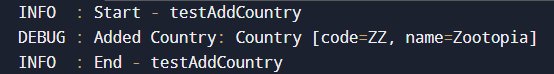
countryService = context.getBean(CountryService.class);

// Call the add test

testAddCountry();

}

**OUTPUT:**



**Hands on 8**

**Update a country based on code**   
1. CountryService.java — Add updateCountry() Method

@Transactionalpublic void updateCountry(String code, String name) throws CountryNotFoundException {

Optional<Country> result = countryRepository.findById(code);

if (!result.isPresent()) {

throw new CountryNotFoundException("Country with code " + code + " not found.");

}

Country country = result.get();

country.setName(name); // Update the country name

countryRepository.save(country); // Save updated country

}

2. OrmLearnApplication.java — Add testUpdateCountry() Method

Make sure to import:

import com.cognizant.springlearn.model.Country;import com.cognizant.springlearn.service.exception.CountryNotFoundException;

Now add the test method:

private static void testUpdateCountry() {

LOGGER.info("Start - testUpdateCountry");

try {

countryService.updateCountry("IN", "Bharat");

Country updatedCountry = countryService.findCountryByCode("IN");

LOGGER.debug("Updated Country: {}", updatedCountry);

} catch (CountryNotFoundException e) {

LOGGER.error("Country not found: {}", e.getMessage());

}

LOGGER.info("End - testUpdateCountry");

}

3. Call testUpdateCountry() in main()

Update your main() method like this:

public static void main(String[] args) {

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

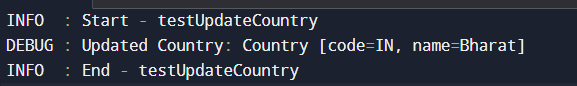
countryService = context.getBean(CountryService.class);

// Invoke update country test

testUpdateCountry();

}

**OUTPUT:**



**Hands on 9**

**Delete a country based on code**

## 1. CountryService.java — Add deleteCountry() Method

@Transactionalpublic void deleteCountry(String code) throws CountryNotFoundException {

Optional<Country> result = countryRepository.findById(code);

if (!result.isPresent()) {

throw new CountryNotFoundException("Country with code " + code + " not found.");

}

countryRepository.deleteById(code);

}

2. OrmLearnApplication.java — Add testDeleteCountry() Method

Make sure to import:

import com.cognizant.springlearn.service.exception.CountryNotFoundException;

Now add the test method:

private static void testDeleteCountry() {

LOGGER.info("Start - testDeleteCountry");

try {

countryService.deleteCountry("ZZ"); // Use the same code you used in add country (e.g., ZZ)

// Try fetching the deleted country

countryService.findCountryByCode("ZZ");

} catch (CountryNotFoundException e) {

LOGGER.debug("Country deletion confirmed. Country not found: {}", e.getMessage());

}

LOGGER.info("End - testDeleteCountry");

}

3. Call testDeleteCountry() in main()

Add it to the main method:

public static void main(String[] args) {

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

countryService = context.getBean(CountryService.class);

// Call delete test

testDeleteCountry();

}

**OUTPUT:**

