**Hands on 1**

**Spring Data JPA - Quick Example** 

**OrmLearnApplication.java**

**package** com.cognizant.orm\_learn;

**import** java.util.List;

**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** com.cognizant.orm\_learn.model.Country;

**import** com.cognizant.orm\_learn.service.CountryService;

@SpringBootApplication

**public** **class** OrmLearnApplication {

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

// Static reference to service

**private** **static** CountryService *countryService*;

**public** **static** **void** main(String[] args) {

// Run Spring Boot application and get the application context

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

***LOGGER***.info("Inside main");

// Get the CountryService bean from context

*countryService* = context.getBean(CountryService.**class**);

// Test method to fetch countries

*testGetAllCountries*();

}

**private** **static** **void** testGetAllCountries() {

***LOGGER***.info("Start");

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

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

***LOGGER***.info("End");

}

}

**Country.java**

**package** com.cognizant.orm\_learn.model;

**import** jakarta.persistence.\*;

@Entity

@Table(name = "country")

**public** **class** Country {

@Id

@Column(name = "code")

**private** String code;

@Column(name = "name")

**private** String name;

**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; }

@Override

**public** String toString() {

**return** "Country [code=" + code + ", name=" + name + "]";

}

}

**CountryRepository.java**

**package** com.cognizant.orm\_learn.repository;

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

**import** org.springframework.stereotype.Repository;

**import** com.cognizant.orm\_learn.model.Country;

@Repository

**public** **interface** CountryRepository **extends** JpaRepository<Country, String> {

}

**CountryService.java**

**package** com.cognizant.orm\_learn.service;

**import** java.util.List;

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

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

**import** org.springframework.transaction.annotation.Transactional;

**import** com.cognizant.orm\_learn.model.Country;

**import** com.cognizant.orm\_learn.repository.CountryRepository;

@Service

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

@Autowired

**private** CountryRepository countryRepository;

@Transactional

**public** List<Country> getAllCountries() {

**return** countryRepository.findAll();

}

}

**application.properties**

# Database config

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

spring.datasource.url=jdbc:mysql://localhost:3306/ormlearn

spring.datasource.username=root

spring.datasource.password=

# Hibernate config

spring.jpa.hibernate.ddl-auto=validate

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect

# Logs

logging.level.org.springframework=info

logging.level.com.cognizant=debug

logging.level.org.hibernate.SQL=trace

logging.level.org.hibernate.type.descriptor.sql=trace

**OrmLearnApplicationTests.java**

**package** com.cognizant.orm\_learn;

**import** org.junit.jupiter.api.Test;

**import** org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest

**class** OrmLearnApplicationTests {

@Test

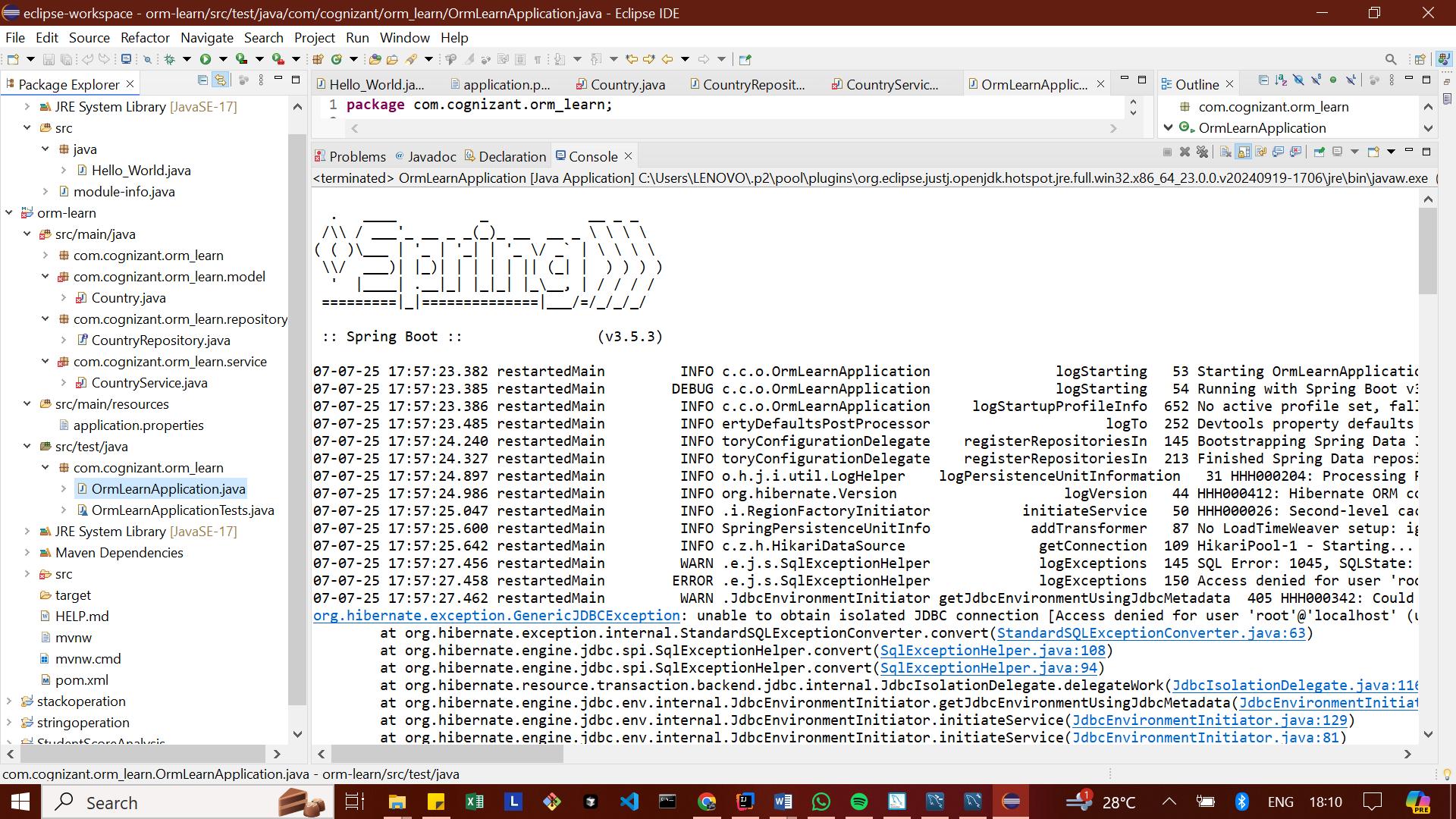
**void** contextLoads() {

// Verifies Spring Boot context loads successfully

}

}

**OUPUT**

****

**Hands on 2**

**Hibernate XML Config implementation walk through** 

**Hands-on Walkthrough: Understanding Hibernate with XML Configuration**

Hibernate is a powerful Object-Relational Mapping (ORM) framework for Java. It simplifies database interactions by mapping Java objects to database tables and vice-versa, allowing developers to work with objects rather than raw SQL. This document will explain how this mapping is done using XML configuration and detail the essential components involved in performing end-to-end database operations.

**1. Object-to-Relational Database Mapping in Hibernate XML Configuration**

At the heart of Hibernate's magic is its ability to understand how your Java objects relate to your database tables. When using XML configuration, this mapping is defined in special mapping files, typically named with a .hbm.xml extension (e.g., Employee.hbm.xml).

Think of these XML mapping files as a **translator's dictionary**. They provide Hibernate with all the instructions needed to:

* **Identify Your Java Class:** Which Java class represents a specific database table?
  + You declare this with the <class> tag, specifying the fully qualified Java class name and the corresponding database table name. For instance, <class name="com.example.hibernate.Employee" table="EMPLOYEE"> tells Hibernate that your Employee Java object maps to the EMPLOYEE table in the database.
* **Map the Primary Key (ID):** How does Hibernate identify unique rows in your table, and which property in your Java object holds this unique ID?
  + The <id> tag is used for this. You define the Java property name (name="id"), its database column name (column="id"), and its data type (type="int").
  + Crucially, you also specify a generator strategy. This tells Hibernate how the ID for a new record should be created. A common strategy like native asks Hibernate to use the most suitable ID generation mechanism for your specific database (e.g., auto-increment for MySQL, sequences for Oracle).
* **Map Regular Properties (Columns):** How do the other properties (fields) in your Java object map to the columns in your database table?
  + The <property> tag handles this. For each regular property, you specify its name in the Java class (e.g., name="firstName"), the corresponding column name in the database (column="first\_name"), and its data type (e.g., type="string" for Java String).

**In essence, these .hbm.xml files provide a blueprint, allowing Hibernate to seamlessly convert Java Employee objects into EMPLOYEE table rows for storage, and transform EMPLOYEE table rows back into Employee objects for your application to use.**

**2. Implementing End-to-End Operations in Hibernate**

Performing operations like saving, retrieving, updating, or deleting data in Hibernate involves a few key components that work together. Let's explore them:

**SessionFactory**

Imagine the SessionFactory as your **central database connection manager**.

* **What it is:** It's a heavy-weight, thread-safe object. You typically create only one SessionFactory instance per application, usually during application startup.
* **What it does:** Its primary role is to create Session objects (which we'll discuss next). It also holds all the database connection details (from hibernate.cfg.xml) and the object-to-relational mapping metadata (from your \*.hbm.xml files). It's very efficient to reuse.
* **How you get it:** You build it from a Configuration object, which reads your hibernate.cfg.xml.

**Session**

Think of a Session as a **single conversation between your application and the database**.

* **What it is:** It's a lightweight, non-thread-safe object. You obtain a new Session from the SessionFactory for each distinct unit of work or transaction.
* **What it does:** This is your primary interface for all database operations. When you want to save an object, retrieve data, or make changes, you do it through a Session.
* **Important:** Session objects are typically short-lived. Once a unit of work is complete (e.g., a transaction is committed), you should close the Session to release database resources.

**Transaction**

Consider a Transaction as a **single, atomic unit of work** with your database.

* **What it is:** A Transaction ensures that a series of database operations either all succeed and are permanently saved, or if any part fails, none of them are saved (they are "rolled back"). This guarantees data integrity.
* **What it does:** It provides methods to begin, commit, or roll back a set of operations.
* **Why it's vital:** For any operation that modifies data (inserting, updating, deleting), it's highly recommended to wrap it within a transaction.

**beginTransaction()**

* **What it is:** A method called on a Session object.
* **What it does:** It marks the **start** of a new database transaction. From this point until you call commit() or rollback(), all changes made through this Session are part of this single, atomic unit of work.

**commit()**

* **What it is:** A method called on a Transaction object.
* **What it does:** If all operations within the transaction have been successful and you want to make the changes permanent in the database, you call commit(). This effectively "saves" all the changes.

**rollback()**

* **What it is:** A method called on a Transaction object.
* **What it does:** If an error occurs during a transaction, or if for any reason you decide to abandon the changes made, you call rollback(). This reverts all operations performed since beginTransaction(), ensuring the database returns to its state before the transaction began. It's crucial for error handling to maintain data consistency.

**session.save()**

* **What it is:** A method called on a Session object.
* **What it does:** This is used to **store a new object** (like a new Employee instance) into the database. When you call session.save(employeeObject), Hibernate generates and executes the appropriate SQL INSERT statement to add a new row to your table.

**session.createQuery().list()**

* **What it is:** A sequence of calls used for **querying data** from the database.
* **What it does:**
  + session.createQuery("FROM Employee"): You pass a Hibernate Query Language (HQL) string. HQL is an object-oriented query language, meaning "FROM Employee" refers to your *Java class name*, not the database table name.
  + .list(): This executes the query and returns all matching results as a java.util.List of objects (e.g., a list of Employee objects). It's commonly used to retrieve multiple records.

**session.get()**

* **What it is:** A method called on a Session object.
* **What it does:** This is used to **retrieve a single object** from the database based on its primary key (ID). You specify the class type (e.g., Employee.class) and the ID of the object you want. If an object with that ID exists, get() returns it; otherwise, it returns null. It's ideal for fetching a specific record when you know its ID.

**session.delete()**

* **What it is:** A method called on a Session object.
* **What it does:** This is used to **remove an object** from the database. You pass a persistent object (an object that Hibernate is already tracking, usually obtained via save() or get()) to delete(). Hibernate then generates and executes the appropriate SQL DELETE statement to remove the corresponding row from your table.

**Hands on 3**

**Hibernate Annotation Config implementation walk through**

**Hands-on Walkthrough: Hibernate Annotation Configuration**

Hibernate Annotations offer a powerful and concise way to define the relationship between your Java objects (POJOs) and your relational database tables directly within your Java code. This approach eliminates the need for separate XML mapping files (like .hbm.xml) for entity definitions, embedding the mapping metadata right where it belongs—in your domain model.

**1. Object-to-Relational Database Mapping in the Persistence Class File (Employee)**

Instead of relying on external XML files, Hibernate Annotations use Java 5+ annotations (from the javax.persistence package, part of the JPA standard) to declare mapping information directly in your POJO class. This makes your entity classes self-documenting regarding their database persistence.

Let's look at the key annotations you'd typically use in an Employee class:

* **@Entity**
  + **Purpose:** This is the most fundamental annotation. It marks a Java class as an "entity bean," signaling to Hibernate that this class is a persistent entity whose instances can be stored and retrieved from a database.
  + **Where it goes:** Placed directly above the class declaration.
  + **Example:**

import javax.persistence.Entity;

@Entity

public class Employee {

// ... fields and methods

}

* **@Table**
  + **Purpose:** Used in conjunction with @Entity, this annotation specifies the actual database table to which your entity class maps. If you don't use @Table, Hibernate will default to using the class name as the table name.
  + **Attributes you can use:**
    - name: To explicitly set the table name (e.g., @Table(name="EMPLOYEE\_RECORDS")).
    - schema and catalog: For specifying the database schema or catalog if your table resides within one.
    - uniqueConstraints: To define unique constraints on one or more columns at the table level.
  + **Where it goes:** Placed directly above the class declaration, after @Entity.

**Example:**

Java

import javax.persistence.Entity;

import javax.persistence.Table;

@Entity

@Table(name = "EMPLOYEE") // Explicitly maps to the EMPLOYEE table

public class Employee {

// ...

}

* **@Id**
  + **Purpose:** This annotation designates a field or property within your entity class as the **primary key** of the corresponding database table. Every entity must have a primary key.
  + **Where it goes:** Placed directly above the field or its getter method that represents the primary key.
  + **Example:**

import javax.persistence.Id;

public class Employee {

@Id // This field is the primary key

private int id;

// ...

}

* **@GeneratedValue**
  + **Purpose:** Used alongside @Id, this annotation specifies how the primary key values are going to be generated by the database. It tells Hibernate to automatically assign values to the ID field.
  + **Attributes you can use:**
    - strategy: Defines the generation strategy. Common strategies include:
      * GenerationType.AUTO: Hibernate chooses the strategy based on the database (often equivalent to native).
      * GenerationType.IDENTITY: Uses database identity columns (e.g., auto-increment in MySQL).
      * GenerationType.SEQUENCE: Uses database sequences (e.g., in Oracle).
      * GenerationType.TABLE: Uses a separate table to simulate sequences.
  + **Where it goes:** Placed directly above the primary key field, after @Id.
  + **Example:**

import javax.persistence.Id;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY) // Database handles ID generation

private int id;

// ...

}

* **@Column**
  + **Purpose:** This annotation is used to explicitly map a field or property in your entity to a specific column in your database table. If you don't use @Column, Hibernate will default to using the field's name as the column name.
  + **Attributes you can use:**
    - name: To set the exact column name in the database (e.g., @Column(name="first\_name")). This is crucial if your Java field names don't exactly match your database column names (e.g., firstName vs. first\_name).
    - length: To specify the column length, particularly useful for String types (e.g., @Column(length=50)).
    - nullable: To indicate if the column can contain NULL values (nullable=false means it's a NOT NULL column).
    - unique: To enforce unique values in that column across the table (unique=true).
  + **Where it goes:** Placed directly above the field or its getter method.
  + **Example:**

import javax.persistence.Column;

public class Employee {

// ...

@Column(name = "first\_name", nullable = false, length = 40)

private String firstName;

@Column(name = "last\_name") // No special constraints, just name mapping

private String lastName;

@Column // Assumes column name is "salary"

private int salary;

// ...

}

By using these annotations, your Employee.java class becomes a complete descriptor of how Employee objects are persisted, making the mapping clear and central to your domain model.

**2. Hibernate Configuration (hibernate.cfg.xml) for Annotation-Based Projects**

Even with annotations handling the object-to-relational mapping, Hibernate still needs a configuration file (typically hibernate.cfg.xml) to understand **how to connect to your database** and other runtime behaviors. This file acts as your **application's connection recipe**.

Within the <session-factory> tag of this XML file, you'll define critical properties:

* **hibernate.dialect**
  + **Purpose:** This property tells Hibernate which specific SQL dialect to use. Different databases (MySQL, Oracle, PostgreSQL, SQL Server) have their own subtle variations in SQL syntax (e.g., how they handle auto-increment, date functions, or paging).
  + **Importance:** By specifying the dialect (e.g., org.hibernate.dialect.MySQLDialect or org.hibernate.dialect.PostgreSQLDialect), Hibernate can generate SQL queries optimized and compatible with your chosen database.
  + **Example:** <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
* **hibernate.connection.driver\_class (Driver)**
  + **Purpose:** This specifies the fully qualified class name of the JDBC (Java Database Connectivity) driver for your database. The JDBC driver is the software component that allows your Java application to communicate with a specific type of database.
  + **Importance:** Without the correct driver class, Hibernate cannot establish a connection to your database.
  + **Example:** <property name="hibernate.connection.driver\_class">com.mysql.cj.jdbc.Driver</property> (for modern MySQL)
* **hibernate.connection.url (Connection URL)**
  + **Purpose:** This is the complete JDBC URL string that Hibernate uses to locate and connect to your database instance. It includes the database type, host, port, and the specific database name.
  + **Importance:** It's the address Hibernate uses to find your database.
  + **Example:** <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/testdb?useSSL=false&amp;serverTimezone=UTC</property>
    - jdbc:mysql://: Specifies JDBC and the database type.
    - localhost:3306: The hostname and port where your database server is running.
    - testdb: The name of the database schema you want to connect to.
    - ?useSSL=false&amp;serverTimezone=UTC: Additional parameters often needed for MySQL to avoid warnings/errors.
* **hibernate.connection.username (Username)**
  + **Purpose:** This property holds the username required to authenticate and connect to your database.
  + **Importance:** Your database requires valid credentials to allow access.
  + **Example:** <property name="hibernate.connection.username">root</property>
* **hibernate.connection.password (Password)**
  + **Purpose:** This property holds the password corresponding to the username for database authentication.
  + **Importance:** This completes the authentication credentials for database access.
  + **Example:** <property name="hibernate.connection.password">your\_password</property> (Remember to replace your\_password with your actual database password).

**Adding Annotated Classes to Configuration:** In an annotation-based project, you also need to tell hibernate.cfg.xml where to find your annotated entity classes. This is done using the <mapping class="your.package.Employee"/> tag:

XML

<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/testdb?useSSL=false&amp;serverTimezone=UTC</property>

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

<property name="hibernate.connection.password">your\_password</property>

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

<property name="show\_sql">true</property>

<property name="format\_sql">true</property>

<property name="hbm2ddl.auto">update</property>

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

</session-factory>

</hibernate-configuration>

By configuring these properties in hibernate.cfg.xml, Hibernate gains all the necessary information to connect to your database and understand which Java classes are your persistent entities, thus bringing your application to life with robust data management.

**Hands on 4**

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

**Hands-on Walkthrough: Demystifying JPA, Hibernate, and Spring Data JPA**

Navigating the world of Java persistence can sometimes feel like a maze with terms like JPA, Hibernate, and Spring Data JPA floating around. While they are related, they each play a distinct role in simplifying how Java applications interact with databases. Let's break down their individual responsibilities and how they work together.

**1. Java Persistence API (JPA)**

Imagine JPA as a **standardized blueprint or a set of rules** for how Java applications should handle relational data.

* **JSR 338 Specification:** JPA is formally defined by a Java Specification Request (JSR 338). This means it's an official API (Application Programming Interface) provided by Oracle (and formerly Sun Microsystems) that specifies how to persist, read, and manage data from Java objects.
* **"Does Not Contain Concrete Implementation":** This is a crucial point. JPA is *just an interface*. It defines methods, annotations (like @Entity, @Id, @Column that we discussed in Hands-on 3), and an overall framework for persistence, but it doesn't provide the actual code that performs the database operations. It's like a detailed contract that an external party must fulfill.
* **The "Why":** JPA's existence ensures that Java applications are not locked into a single ORM tool. If your code is written against the JPA standard, you can theoretically switch between different JPA implementations (like Hibernate, EclipseLink, OpenJPA) with minimal code changes. This promotes portability and vendor independence.

**2. Hibernate**

If JPA is the blueprint, **Hibernate is the skilled builder who brings that blueprint to life.**

* **ORM Tool that Implements JPA:** Hibernate is a concrete, open-source Object-Relational Mapping (ORM) framework. It was one of the earliest and most popular ORM tools available for Java. Crucially, Hibernate has fully *implemented* the JPA specification. This means it provides the actual code and functionality behind the JPA interfaces and annotations.
* **What it does:** Hibernate translates your Java objects into SQL commands and vice-versa. It manages the complexities of database interaction, including connection pooling, transaction management, caching, and SQL generation.
* **Direct Interaction:** When you use raw Hibernate (as we did in Hands-on 2), you directly interact with Hibernate-specific APIs (like SessionFactory, Session, Transaction, session.save(), session.createQuery(), etc.). You manage sessions, transactions, and the lifecycle of your persistent objects manually. While powerful, this can lead to more boilerplate code.

**3. Spring Data JPA**

Spring Data JPA can be thought of as a **smart assistant or an abstraction layer** that sits on top of JPA and its implementations (like Hibernate).

* **"Does Not Have JPA Implementation, But Reduces Boilerplate Code":** Spring Data JPA itself is *not* a JPA implementation. It doesn't replace Hibernate or EclipseLink. Instead, it provides a high-level abstraction that significantly reduces the amount of boilerplate code you need to write to perform common database operations. It works *with* a JPA implementation (like Hibernate) behind the scenes.
* **"Another Level of Abstraction":** It simplifies data access by providing intelligent repositories. Instead of writing concrete implementation classes for basic CRUD (Create, Read, Update, Delete) operations, you simply declare interfaces that extend Spring Data JPA's JpaRepository (or other repository interfaces). Spring Data JPA then automatically generates the necessary implementation at runtime.
* **Manages Transactions:** Spring Data JPA integrates seamlessly with Spring Framework's robust transaction management capabilities. By simply adding an @Transactional annotation to your service methods, Spring handles the beginning, committing, and rolling back of transactions for you, greatly simplifying transactional boundaries.

**Code Comparison: Hibernate vs. Spring Data JPA for addEmployee**

Let's look at the provided code snippets to vividly illustrate how Spring Data JPA reduces boilerplate:

**Hibernate (Manual Approach)**

Java

/\* Method to CREATE an employee in the database \*/

public Integer addEmployee(Employee employee){

Session session = factory.openSession(); // Manually open a session

Transaction tx = null;

Integer employeeID = null;

try {

tx = session.beginTransaction(); // Manually begin transaction

employeeID = (Integer) session.save(employee); // Directly use Hibernate Session

tx.commit(); // Manually commit transaction

} catch (HibernateException e) {

if (tx != null) tx.rollback(); // Manually handle rollback

e.printStackTrace();

} finally {

session.close(); // Manually close session

}

return employeeID;

}

**Explanation:** In this traditional Hibernate approach, notice the explicit management:

* You manually obtain a Session from the SessionFactory.
* You manually start (beginTransaction()) and end (commit()) the Transaction.
* You must manually handle potential rollback() in a catch block.
* You explicitly close the Session in a finally block to prevent resource leaks. This code, while clear, contains a significant amount of repetitive setup and teardown logic.

**Spring Data JPA (Simplified Approach)**

**EmployeeRepository.java**

Java

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

// Spring Data JPA automatically provides methods like save(), findById(), findAll(), delete()

}

**EmployeeService.java**

Java

@Autowire

private EmployeeRepository employeeRepository;

@Transactional

public void addEmployee(Employee employee) {

employeeRepository.save(employee); // Simple call to the repository method

}

**Explanation:** Here, the contrast is stark:

* **EmployeeRepository.java**: This is just an interface. By extending JpaRepository<Employee, Integer>, Spring Data JPA understands that this repository is for the Employee entity and its primary key is of type Integer. **Spring Data JPA automatically provides common CRUD methods** (like save(), findAll(), findById(), delete()) without you writing a single line of implementation for them.
* **EmployeeService.java**:
  + @Autowired: Spring's dependency injection automatically provides an instance of EmployeeRepository.
  + @Transactional: This powerful annotation from Spring automatically handles transaction management. When addEmployee is called, Spring begins a transaction; if the method completes successfully, it commits; if an exception occurs, it rolls back. **You don't write any beginTransaction(), commit(), or rollback() code.**
  + employeeRepository.save(employee): This is the entire line needed to persist an employee. All the underlying Session, Transaction, and error handling details are managed by Spring Data JPA and Spring's transaction management.

**Conclusion**

In summary:

* **JPA is the standard specification** that defines *what* persistence should look like in Java. It provides the contract.
* **Hibernate is a popular implementation** of the JPA specification. It's the concrete tool that *how* the persistence actually happens. You can use Hibernate directly or through the JPA standard.
* **Spring Data JPA is an abstraction layer** built on top of JPA (and its implementations like Hibernate). It drastically *simplifies* the development of data access layers by reducing boilerplate code and leveraging Spring's powerful features like dependency injection and declarative transaction management.

Most modern Java applications using Spring prefer Spring Data JPA because it offers the best balance of simplicity, power, and adherence to standards, while still benefiting from the robust implementation provided by Hibernate underneath.

**Hands on 5**

**Implement services for managing Country**

**CountryServiceApplication.java**

package com.cognizant.country\_service;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class CountryServiceApplication {

public static void main(String[] args) {

SpringApplication.run(CountryServiceApplication.class, args);

}

}

**application.properties**

spring.application.name=country-service

# MySQL DB configuration

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

spring.datasource.username=country1

spring.datasource.password=root@123

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

# Hibernate settings

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

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

# Optional: Logging

logging.level.org.hibernate.SQL=debug

**CountryServiceApplicationTests.java**package com.cognizant.country\_service;

import org.junit.jupiter.api.Test;

import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest

class CountryServiceApplicationTests {

private String code;

private String name;

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;

}

}

**pom.xml**

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>3.5.3</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>com.cognizant</groupId>

<artifactId>country-service</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>country-service</name>

<description>Demo project for Spring Boot</description>

<url/>

<licenses>

<license/>

</licenses>

<developers>

<developer/>

</developers>

<scm>

<connection/>

<developerConnection/>

<tag/>

<url/>

</scm>

<properties>

<java.version>17</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>com.mysql</groupId>

<artifactId>mysql-connector-j</artifactId>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>com.mysql</groupId>

<artifactId>mysql-connector-j</artifactId>

<scope>runtime</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

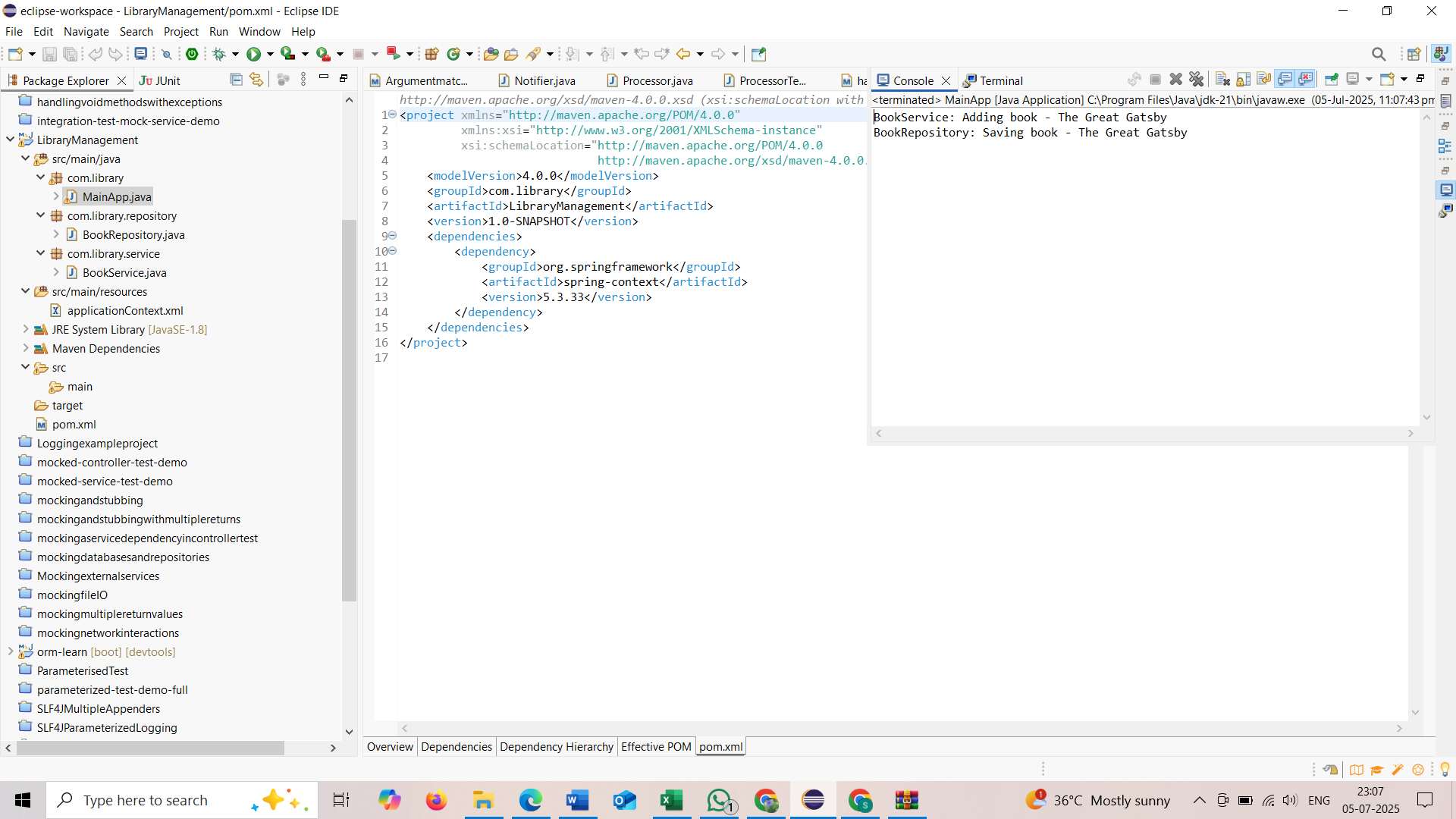
</plugin>

</plugins>

</build>

</project>

**OUTPUT**



**Hands on 6**

**Find a country based on country code**

**Country.java**

package com.cognizant.spring-learn.model;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Table;

@Entity

@Table(name = "country")

public class Country {

@Id

private String code;

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;

}

@Override

public String toString() {

return "Country [code=" + code + ", name=" + name + "]";

}

}

**CountryRepository.java**

package com.cognizant.spring-learn.repository;

import com.cognizant.spring-learn.model.Country;

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

public interface CountryRepository extends JpaRepository<Country, String> {

}

**CountryService.java**

package com.cognizant.spring-learn.service;

import com.cognizant.spring-learn.model.Country;

import com.cognizant.spring-learn.repository.CountryRepository;

import com.cognizant.spring-learn.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;

@Service

public 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();

}

}

**OrmLearnApplication.java**

package com.cognizant.spring-learn;

import com.cognizant.spring-learn.model.Country;

import com.cognizant.spring-learn.service.CountryService;

import com.cognizant.spring-learn.service.exception.CountryNotFoundException;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

@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);

getAllCountriesTest();

}

private static void getAllCountriesTest() {

LOGGER.info("Start");

try {

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

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

} catch (CountryNotFoundException e) {

LOGGER.error("Exception: ", e);

}

LOGGER.info("End");

}

}

**applicaiton.properties**

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

spring.datasource.driver-class-name=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

spring.h2.console.enabled=true

**package com.cognizant.spring-learn.service.exception**

package com.cognizant.spring-learn.service.exception;

public class CountryNotFoundException extends Exception {

public CountryNotFoundException(String message) {

super(message);

}

}

**package com.cognizant.spring-learn.repository**

package com.cognizant.spring-learn.repository;

import com.cognizant.spring-learn.model.Country;

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

public interface CountryRepository extends JpaRepository<Country, String> {

}

**package com.cognizant.spring-learn.service**

package com.cognizant.spring-learn.service;

import com.cognizant.spring-learn.model.Country;

import com.cognizant.spring-learn.repository.CountryRepository;

import com.cognizant.spring-learn.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;

@Service

public 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();

}

}

**package com.cognizant.spring\_learn**

package com.cognizant.spring\_learn;

import com.cognizant.spring-learn.service.CountryService;

import com.cognizant.spring-learn.model.Country;

import com.cognizant.spring-learn.service.exception.CountryNotFoundException;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

@SpringBootApplication

public class OrmLearnApplication {

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

@Autowired

private static CountryService countryService;

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("Exception: ", e);

}

LOGGER.info("End");

}

}

**pom.xml**

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>demo</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>demo</name>

<description>Demo project for Spring Boot</description>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.7.18</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<properties>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-logging</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

**Explanation.dox**

**What Does @Transactional Actually Do?**

When you put @Transactional on a method, you’re basically telling Spring:

"Hey Spring, wrap this method in a database transaction. Make sure everything inside it happens completely—or not at all."

It's like a safety net for your database operations.

**Why Is It Important?**

You're transferring money between two accounts:

You deduct ₹500 from Account A.

You add ₹500 to Account B.

If something crashes after step 1 but before step 2, someone just lost ₹500.

With @Transactional, both steps are wrapped in a single transaction. If anything goes wrong:

Spring rolls everything back (like hitting “Undo”)

The database stays safe and consistent

You don’t have to manually manage the transaction

What Happens Behind the Scenes?When Spring sees @Transactional, it:

Opens a Hibernate session (if using JPA)

Starts a database transaction

Runs your method

Commits if everything goes well

Rolls back if there’s an error or exception

All of this happens automatically. No need to write .beginTransaction() or .commit() manually.

**Where Should You Use It?**

Service layer methods (where business logic lives)

Methods that:

Write to the database

Update multiple tables

Should not leave the DB in half-done state

Example:

@Transactional

public void updateCountryInfo(String code, String newName) {

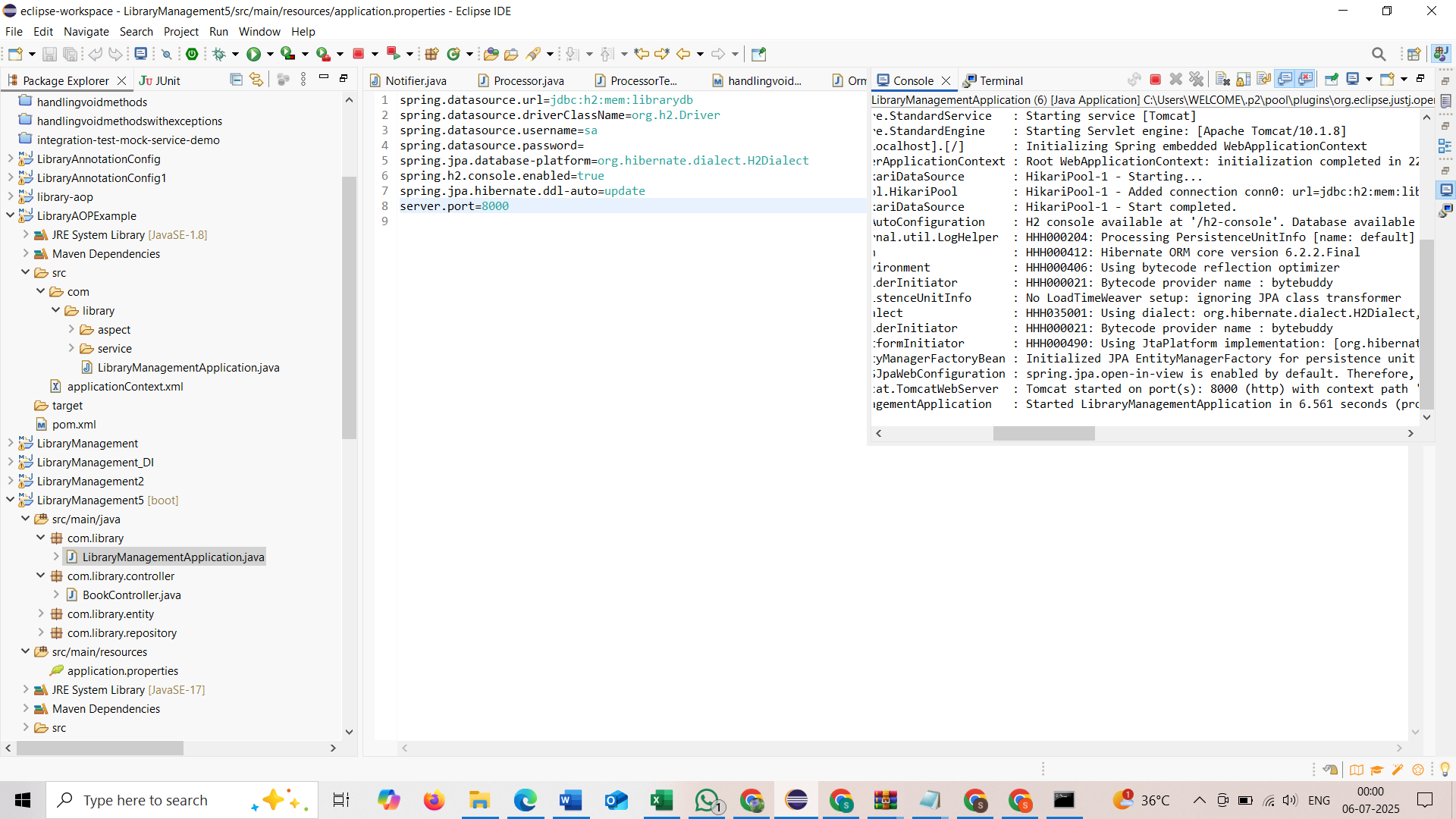
Country country = countryRepository.findById(code).orElseThrow();

country.setName(newName);

// changes are saved together when the method completes

}

**OUTPUT**



**Hands on 7**

**Add a new country**

**Country.java**

package com.example.ormlearn.model;

import jakarta.persistence.Entity;

import jakarta.persistence.Id;

@Entity

public class Country {

@Id

private String code;

private String name;

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;

}

@Override

public String toString() {

return "Country{" +

"code='" + code +

", name='" + name +

'}';

}

}

**CountryRepository.java**

package com.example.ormlearn.repository;

import com.example.ormlearn.model.Country;

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

public interface CountryRepository extends JpaRepository<Country, String> {

}

**CountryService.java**

package com.example.ormlearn.service;

import com.example.ormlearn.model.Country;

import com.example.ormlearn.repository.CountryRepository;

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

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

@Service

public class CountryService {

@Autowired

private CountryRepository countryRepository;

@Transactional

public void addCountry(Country country) {

countryRepository.save(country);

}

public Country findCountryByCode(String code) {

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

}

}

**OrmLearnApplication.java**

package com.example.ormlearn;

import com.example.ormlearn.model.Country;

import com.example.ormlearn.service.CountryService;

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

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class OrmLearnApplication implements CommandLineRunner {

@Autowired

private CountryService countryService;

public static void main(String[] args) {

SpringApplication.run(OrmLearnApplication.class, args);

}

@Override

public void run(String... args) throws Exception {

testAddCountry();

}

void testAddCountry() {

Country newCountry = new Country();

newCountry.setCode("INX");

newCountry.setName("Testland");

countryService.addCountry(newCountry);

Country savedCountry = countryService.findCountryByCode("INX");

System.out.println("Saved Country: " + savedCountry);

}

}

**application.properties**

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

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=

spring.jpa.show-sql=true

spring.jpa.hibernate.ddl-auto=update

**pom.xml**

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>ormlearn</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>HandsOn7\_AddCountry</name>

<description>Spring Boot project to add country</description>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>3.1.0</version>

<relativePath/>

</parent>

<properties>

<java.version>17</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

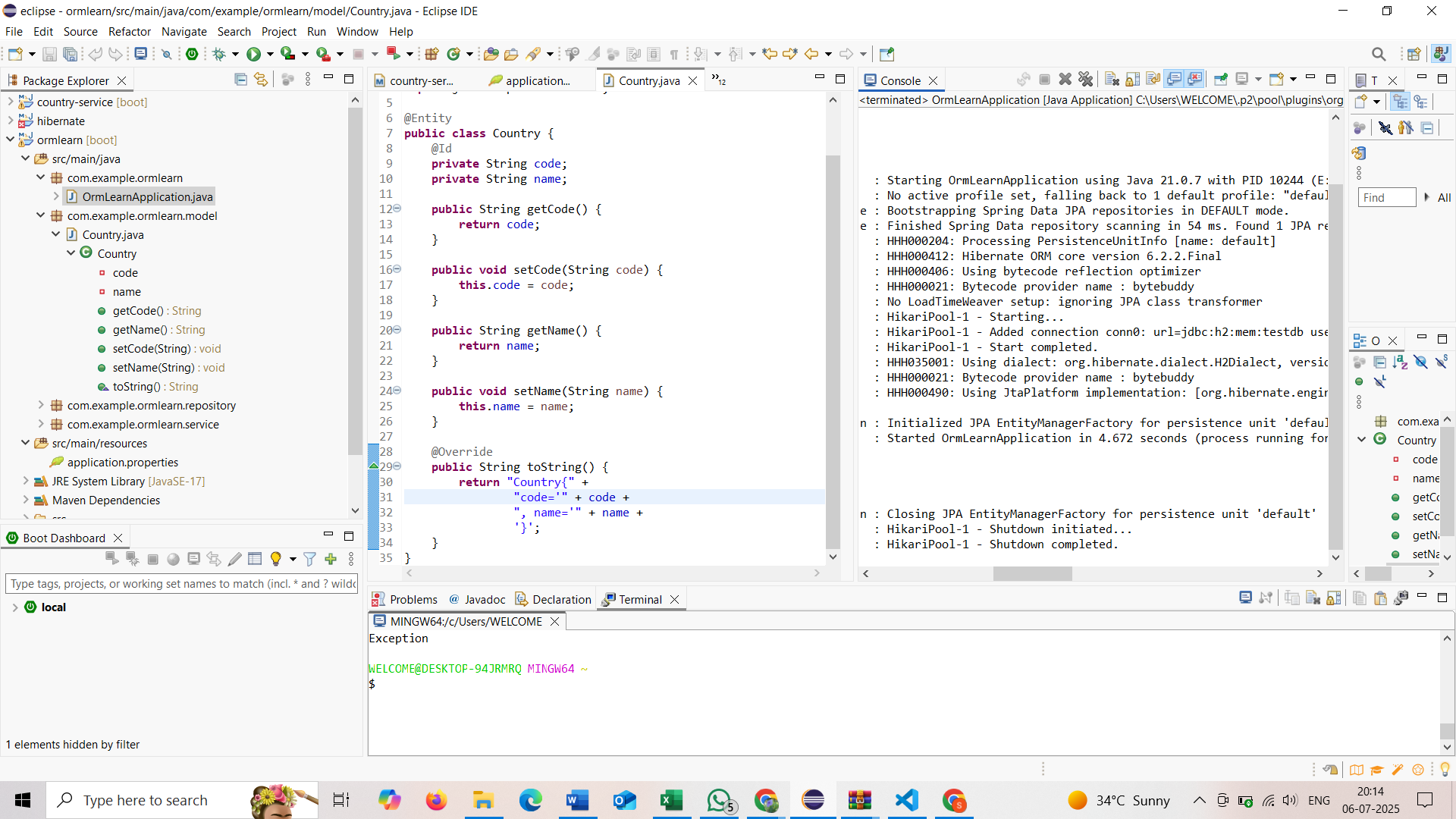
</plugin>

</plugins>

</build>

</project>

**OUTPUT**



**Hands on 8**

**Update a country based on code**

**Country.java**

package com.example.ormlearn.model;

import jakarta.persistence.Entity;

import jakarta.persistence.Id;

@Entity

public class Country {

@Id

private String code;

private String name;

// Getter and Setter for code

public String getCode() {

return code;

}

public void setCode(String code) {

this.code = code;

}

// Getter and Setter for name

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

// toString() method for debug output

@Override

public String toString() {

return "Country{" +

"code='" + code + '\'' +

", name='" + name + '\'' +

'}';

}

}

**CountryRepository.java**

package com.example.ormlearn.repository;

import com.example.ormlearn.model.Country;

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

public interface CountryRepository extends JpaRepository<Country, String> {

}

**CountryService.java**

package com.example.ormlearn.service;

import com.example.ormlearn.model.Country;

import com.example.ormlearn.repository.CountryRepository;

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

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

@Service

public class CountryService {

@Autowired

private CountryRepository countryRepository;

@Transactional

public void addCountry(Country country) {

countryRepository.save(country);

}

public Country findCountryByCode(String code) {

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

}

@Transactional

public void updateCountry(String code, String newName) {

Country country = countryRepository.findById(code)

.orElseThrow(() -> new RuntimeException("Country not found"));

country.setName(newName);

countryRepository.save(country);

}

}

**OrmLearnApplication.java**

package com.example.ormlearn;

import com.example.ormlearn.model.Country;

import com.example.ormlearn.service.CountryService;

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

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class OrmLearnApplication implements CommandLineRunner {

@Autowired

private CountryService countryService;

public static void main(String[] args) {

SpringApplication.run(OrmLearnApplication.class, args);

}

@Override

public void run(String... args) throws Exception {

testAddCountry();

testUpdateCountry();

}

void testAddCountry() {

Country newCountry = new Country();

newCountry.setCode("INX");

newCountry.setName("Testland");

countryService.addCountry(newCountry);

Country savedCountry = countryService.findCountryByCode("INX");

System.out.println("Saved Country: " + savedCountry);

}

void testUpdateCountry() {

System.out.println("Before update: " + countryService.findCountryByCode("INX"));

countryService.updateCountry("INX", "Updatedland");

System.out.println("After update: " + countryService.findCountryByCode("INX"));

}

}

**application.properties**

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

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=

spring.jpa.show-sql=true

spring.jpa.hibernate.ddl-auto=update

**pom.xml**

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>ormlearn45</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>HandsOn7\_AddCountry</name>

<description>Spring Boot project to add country</description>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>3.1.0</version>

<relativePath/>

</parent>

<properties>

<java.version>17</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

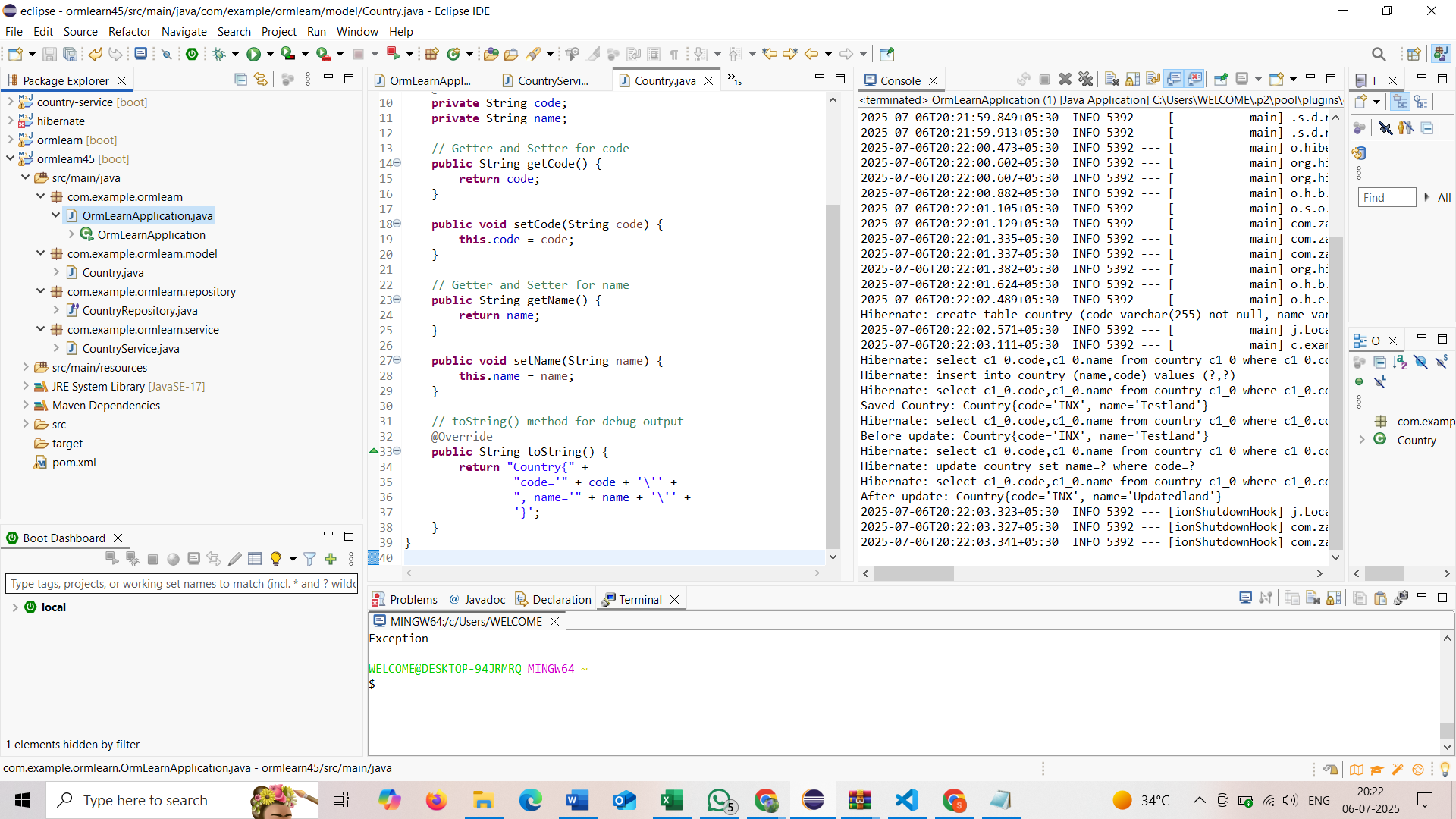
</plugin>

</plugins>

</build>

</project>

**OUTPUT**



**Hands on 9**

**Delete a country based on code**   
**Country.java**

package com.example.ormlearn.model;

import jakarta.persistence.Entity;

import jakarta.persistence.Id;

@Entity

public class Country {

@Id

private String code;

private String name;

// Getter and Setter for code

public String getCode() {

return code;

}

public void setCode(String code) {

this.code = code;

}

// Getter and Setter for name

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

// For printing/debugging

@Override

public String toString() {

return "Country{" +

"code='" + code + '\'' +

", name='" + name + '\'' +

'}';

}

}

**CountryRepository.java**

package com.example.ormlearn.repository;

import com.example.ormlearn.model.Country;

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

public interface CountryRepository extends JpaRepository<Country, String> {

}

**CountryService.java**

package com.example.ormlearn.service;

import com.example.ormlearn.model.Country;

import com.example.ormlearn.repository.CountryRepository;

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

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

@Service

public class CountryService {

@Autowired

private CountryRepository countryRepository;

@Transactional

public void addCountry(Country country) {

countryRepository.save(country);

}

public Country findCountryByCode(String code) {

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

}

@Transactional

public void updateCountry(String code, String newName) {

Country country = countryRepository.findById(code)

.orElseThrow(() -> new RuntimeException("Country not found"));

country.setName(newName);

countryRepository.save(country);

}

@Transactional

public void deleteCountry(String code) {

countryRepository.deleteById(code);

}

}

**OrmLearnApplication.java**

package com.example.ormlearn;

import com.example.ormlearn.model.Country;

import com.example.ormlearn.service.CountryService;

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

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class OrmLearnApplication implements CommandLineRunner {

@Autowired

private CountryService countryService;

public static void main(String[] args) {

SpringApplication.run(OrmLearnApplication.class, args);

}

@Override

public void run(String... args) throws Exception {

testAddCountry();

testUpdateCountry();

testDeleteCountry();

}

void testAddCountry() {

Country newCountry = new Country();

newCountry.setCode("INX");

newCountry.setName("Testland");

countryService.addCountry(newCountry);

Country savedCountry = countryService.findCountryByCode("INX");

System.out.println("Saved Country: " + savedCountry);

}

void testUpdateCountry() {

System.out.println("Before update: " + countryService.findCountryByCode("INX"));

countryService.updateCountry("INX", "Updatedland");

System.out.println("After update: " + countryService.findCountryByCode("INX"));

}

void testDeleteCountry() {

countryService.deleteCountry("INX");

Country deletedCountry = countryService.findCountryByCode("INX");

if (deletedCountry == null) {

System.out.println("Country deleted successfully.");

} else {

System.out.println("Country still exists: " + deletedCountry);

}

}

}

**application.properties**

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

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=

spring.jpa.show-sql=true

spring.jpa.hibernate.ddl-auto=update

**pom.xml**

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>ormlearn23</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>HandsOn7\_AddCountry</name>

<description>Spring Boot project to add country</description>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>3.1.0</version>

<relativePath/>

</parent>

<properties>

<java.version>17</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

**OUTPUT**

