**1.Spring Data JPA with Spring Boot, Hibernate**

**Hands on 1 Question**

**Spring Data JPA - Quick Example**   
  
**Software Pre-requisites**

* MySQL Server 8.0
* MySQL Workbench 8
* Eclipse IDE for Enterprise Java Developers 2019-03 R
* Maven 3.6.2

**Create a Eclipse Project using Spring Initializr**

* Go to <https://start.spring.io/>
* Change Group as “com.cognizant”
* Change Artifact Id as “orm-learn”
* In Options > Description enter "Demo project for Spring Data JPA and Hibernate"
* Click on menu and select "Spring Boot DevTools", "Spring Data JPA" and "MySQL Driver"
* Click Generate and download the project as zip
* Extract the zip in root folder to Eclipse Workspace
* Import the project in Eclipse "File > Import > Maven > Existing Maven Projects > Click Browse and select extracted folder > Finish"
* Create a new schema "ormlearn" in MySQL database. Execute the following commands to open MySQL client and create schema.

>mysql -u root -p

mysql> create schema ormlearn;

* In orm-learn Eclipse project, open src/main/resources/application.properties and include the below database and log configuration.

# Spring Framework and application log

logging.level.org.springframework=info

logging.level.com.cognizant=debug

# Hibernate logs for displaying executed SQL, input and output

logging.level.org.hibernate.SQL=trace

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

# Log pattern

logging.pattern.console=%d{dd-MM-yy} %d{HH:mm:ss.SSS} %-20.20thread %5p %-25.25logger{25} %25M %4L %m%n

# Database configuration

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=root

# Hibernate configuration

spring.jpa.hibernate.ddl-auto=validate

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

* Build the project using ‘mvn clean package -Dhttp.proxyHost=proxy.cognizant.com -Dhttp.proxyPort=6050 -Dhttps.proxyHost=proxy.cognizant.com -Dhttps.proxyPort=6050 -Dhttp.proxyUser=123456’ command in command line
* Include logs for verifying if main() method is called.

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

public static void main(String[] args) {

SpringApplication.run(OrmLearnApplication.class, args);

  LOGGER.info("Inside main");

}

* Execute the OrmLearnApplication and check in log if main method is called.

SME to walk through the following aspects related to the project created:

1. src/main/java - Folder with application code
2. src/main/resources - Folder for application configuration
3. src/test/java - Folder with code for testing the application
4. OrmLearnApplication.java - Walkthrough the main() method.
5. Purpose of @SpringBootApplication annotation
6. pom.xml
   1. Walkthrough all the configuration defined in XML file
   2. Open 'Dependency Hierarchy' and show the dependency tree.

**Country table creation**

* Create a new table country with columns for code and name. For sample, let us insert one country with values 'IN' and 'India' in this table.

create table country(co\_code varchar(2) primary key, co\_name varchar(50));

* Insert couple of records into the table

insert into country values ('IN', 'India');

insert into country values ('US', 'United States of America');

**Persistence Class - com.cognizant.orm-learn.model.Country**

* Open Eclipse with orm-learn project
* Create new package com.cognizant.orm-learn.model
* Create Country.java, then generate getters, setters and toString() methods.
* Include @Entity and @Table at class level
* Include @Column annotations in each getter method specifying the column name.

importjavax.persistence.Column;

importjavax.persistence.Entity;

importjavax.persistence.Id;

importjavax.persistence.Table;

@Entity

@Table(name="country")

public class Country {

  @Id

    @Column(name="code")

    private String code;

    @Column(name="name")

    private String name;

// getters and setters

  // toString()

}

*Notes:*

* @Entity is an indicator to Spring Data JPA that it is an entity class for the application
* @Table helps in defining the mapping database table
* @Id helps is defining the primary key
* @Column helps in defining the mapping table column

**Repository Class - com.cognizant.orm-learn.CountryRepository**

* Create new package com.cognizant.orm-learn.repository
* Create new interface named CountryRepository that extends JpaRepository<Country, String>
* Define @Repository annotation at class level

importorg.springframework.data.jpa.repository.JpaRepository;

importorg.springframework.stereotype.Repository;

importcom.cognizant.ormlearn.model.Country;

@Repository

public interface CountryRepository extends JpaRepository<Country, String> {

}

**Service Class - com.cognizant.orm-learn.service.CountryService**

* Create new package com.cognizant.orm-learn.service
* Create new class CountryService
* Include @Service annotation at class level
* AutowireCountryRepository in CountryService
* Include new method getAllCountries() method that returns a list of countries.
* Include @Transactional annotation for this method
* In getAllCountries() method invoke countryRepository.findAll() method and return the result

**Testing in OrmLearnApplication.java**

* Include a static reference to CountryService in OrmLearnApplication class

private static CountryServicecountryService;

* Define a test method to get all countries from service.

    private static void testGetAllCountries() {

        LOGGER.info("Start");

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

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

        LOGGER.info("End");

    }

* Modify SpringApplication.run() invocation to set the application context and the CountryService reference from the application context.

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

        countryService = context.getBean(CountryService.class);

        testGetAllCountries();

* Execute main method to check if data from ormlearn database is retrieved.

**Hands-on Spring Data JPA - Comprehensive Guide with Deep Dive and Full Example**

This comprehensive guide explores in great depth how to set up and work with Spring Data JPA and Hibernate in a Java-based application using the Spring Boot framework. We’ll go step-by-step through installation, configuration, development, and testing of a complete persistence layer using MySQL. The intent is not just to provide a hands-on example but also to develop a deep understanding of how various layers of a Spring Data JPA application interact and how best practices can be applied to enterprise-level applications.

**1. Software Pre-requisites and Environment Setup**

To begin working on this application, ensure the following software tools and configurations are available and set up correctly:

* **MySQL Server 8.0**: A reliable open-source relational database management system used for storing persistent application data.
* **MySQL Workbench 8**: A visual database design tool that helps run SQL scripts, manage schemas, and visually design data models.
* **Eclipse IDE for Enterprise Java Developers (2019-03 R)**: A powerful integrated development environment that offers built-in Maven support and project management.
* **Maven 3.6.2**: A build automation tool used for managing project dependencies, building, and packaging.

Also ensure:

* **Java Development Kit (JDK 1.8 or above)** is installed and configured.
* Set environment variables like JAVA\_HOME, MAVEN\_HOME, and add them to the PATH.
* Verify by running java -version and mvn -version in the terminal.

**2. Project Creation Using Spring Initializr**

**Step-by-Step Instructions:**

1. Go to [Spring Initializr](https://start.spring.io/), the web-based Spring Boot project generator.
2. Configure the following metadata:
   * **Group**: com.cognizant
   * **Artifact**: orm-learn
   * **Name**: orm-learn
   * **Description**: Demo project for Spring Data JPA and Hibernate
   * **Packaging**: Jar
   * **Java**: 8 or above
3. Add Dependencies:
   * Spring Boot DevTools (for hot reload and live refresh)
   * Spring Data JPA (for data persistence abstraction)
   * MySQL Driver (JDBC connector for MySQL)
4. Click **Generate**, download the ZIP, and extract it into your preferred Eclipse workspace.
5. In Eclipse:
   * Go to File > Import > Maven > Existing Maven Projects
   * Browse the extracted folder
   * Finish the import process

You now have a ready-to-develop Spring Boot application base.

**3. Creating and Configuring MySQL Schema**

**Using MySQL Client or MySQL Workbench:**

Open your terminal or MySQL Workbench client and execute:

mysql -u root -p

Then execute:

CREATE SCHEMA ormlearn;

This will set up the schema that the Spring Boot application will connect to for executing queries.

If using MySQL Workbench:

* Open a new SQL tab.
* Paste and execute the CREATE SCHEMA command.

**4. Adding Properties to application.properties**

Navigate to: src/main/resources/application.properties and configure the following:

# General logging configuration

logging.level.org.springframework=info

logging.level.com.cognizant=debug

# Hibernate-specific SQL and data logging

logging.level.org.hibernate.SQL=trace

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

# Console output format for logs

logging.pattern.console=%d{dd-MM-yy} %d{HH:mm:ss.SSS} %-20.20thread %5p %-25.25logger{25} %25M %4L %m%n

# Database properties

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=root

# Hibernate configuration

spring.jpa.hibernate.ddl-auto=validate

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

This file tells Spring Boot how to connect to the database, what level of logs to generate, and how Hibernate should interact with the database schema.

**5. Building the Project Using Maven**

Navigate to your project folder in the terminal and execute:

mvn clean package -Dhttp.proxyHost=proxy.cognizant.com -Dhttp.proxyPort=6050 -Dhttps.proxyHost=proxy.cognizant.com -Dhttps.proxyPort=6050 -Dhttp.proxyUser=123456

This command:

* Cleans existing target directory
* Resolves and downloads all Maven dependencies
* Builds the final .jar file inside the target directory
* Ensures proper proxy configuration if behind a corporate firewall

**6. Logging Application Startup in main()**

Edit OrmLearnApplication.java in com.cognizant.ormlearn package:

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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);

public static void main(String[] args) {

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

LOGGER.info("Inside main");

}

}

This ensures that the application starts correctly and that your logs are configured properly.

**7. Understanding Project Structure**

**src/main/java**

Contains all Java classes:

* **model**: Domain models/entities like Country.java
* **repository**: Spring Data JPA interfaces
* **service**: Business logic implementations
* **controller**: (Optional) REST APIs or UI interaction logic

**src/main/resources**

* Holds configuration files like application.properties
* May contain static files for frontend (HTML, JS, CSS) if using Spring MVC

**src/test/java**

Contains JUnit test cases to validate logic and DB operations

**pom.xml**

Defines all project dependencies and plugins. Key dependencies:

<dependency>

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

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

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

Use the Eclipse "Dependency Hierarchy" tab to visualize transitive dependencies.

**8. Significance of @SpringBootApplication Annotation**

This is a combination of three essential annotations:

* @Configuration: Declares the class as a configuration class
* @EnableAutoConfiguration: Enables Spring Boot's auto-configuration mechanism
* @ComponentScan: Scans the current package and sub-packages for annotated components

Without this, your application won’t recognize service/repository/beans.

**9. Creating a Database Table country**

Execute the following SQL commands:

CREATE TABLE country (

co\_code VARCHAR(2) PRIMARY KEY,

co\_name VARCHAR(50)

);

INSERT INTO country VALUES ('IN', 'India');

INSERT INTO country VALUES ('US', 'United States of America');

These commands set up the country table with two sample entries.

**10. Entity Class: Country.java**

Create in com.cognizant.ormlearn.model:

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Table;

@Entity

@Table(name = "country")

public class Country {

@Id

@Column(name = "co\_code")

private String code;

@Column(name = "co\_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 + "]";

}

}

This is the basic entity class mapped to the country table using JPA annotations.

**11. Repository Interface: CountryRepository.java**

Create in com.cognizant.ormlearn.repository:

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

import org.springframework.stereotype.Repository;

import com.cognizant.ormlearn.model.Country;

@Repository

public interface CountryRepository extends JpaRepository<Country, String> {

}

This interface now gives you access to all basic CRUD operations without writing implementation.

**12. Service Layer: CountryService.java**

Create in com.cognizant.ormlearn.service:

import java.util.List;

import javax.transaction.Transactional;

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

import org.springframework.stereotype.Service;

import com.cognizant.ormlearn.model.Country;

import com.cognizant.ormlearn.repository.CountryRepository;

@Service

public class CountryService {

@Autowired

private CountryRepository countryRepository;

@Transactional

public List<Country> getAllCountries() {

return countryRepository.findAll();

}

}

This service class acts as a layer between controller (or application logic) and the data repository.

**13. Testing Service Logic in main()**

In OrmLearnApplication.java, update the logic:

import com.cognizant.ormlearn.service.CountryService;

import com.cognizant.ormlearn.model.Country;

import java.util.List;

private static CountryService countryService;

private static void testGetAllCountries() {

LOGGER.info("Start");

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

for (Country country : countries) {

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

}

LOGGER.info("End");

}

public static void main(String[] args) {

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

countryService = context.getBean(CountryService.class);

testGetAllCountries();

LOGGER.info("Inside main");

}

**14. Validating Output in Console**

Expected log output should resemble:

Inside main

Start

Country: Country [code=IN, name=India]

Country: Country [code=US, name=United States of America]

End

**Hands on 4**

**Difference between JPA, Hibernate and Spring Data JPA**   
  
Java Persistence API (JPA)

* JSR 338 Specification for persisting, reading and managing data from Java objects
* Does not contain concrete implementation of the specification
* Hibernate is one of the implementation of JPA

Hibernate

* ORM Tool that implements JPA

Spring Data JPA

* Does not have JPA implementation, but reduces boiler plate code
* This is another level of abstraction over JPA implementation provider like Hibernate
* Manages transactions

**Refer code snippets below on how the code compares between Hibernate and Spring Data JPA  
Hibernate**

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

   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();

      }

      returnemployeeID;

   }

**Spring Data JPA**  
EmployeeRespository.java

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

}

EmployeeService.java

@Autowire

  privateEmployeeRepositoryemployeeRepository;

@Transactional

public void addEmployee(Employee employee) {

  employeeRepository.save(employee);

  }

​​​​​​​   
  
**Reference Links:**   
<https://dzone.com/articles/what-is-the-difference-between-hibernate-and-sprin-1>   
<https://www.javaworld.com/article/3379043/what-is-jpa-introduction-to-the-java-persistence-api.html>

### Hands-on Spring Data JPA - Comprehensive Guide with Deep Dive and Full Example

[...existing content remains unchanged above this point...]

## 16. Difference Between JPA, Hibernate, and Spring Data JPA

Understanding the differences between JPA, Hibernate, and Spring Data JPA is foundational to mastering modern Java persistence. These technologies, while closely related, play distinct roles in the ecosystem. Below, we explore their historical evolution, technical design, integration patterns, performance implications, and real-world usage in great detail.

### Historical Context

**JPA (Java Persistence API)** was introduced as part of the Java EE specification (JSR 220) with EJB 3.0. Prior to JPA, developers used complex and heavyweight EJB entity beans for ORM. These beans were hard to manage and tested poorly. To simplify persistence, JPA was introduced in 2006 to offer a standard for mapping Java objects to database tables. Over time, JPA evolved—most notably in JPA 2.0 and 2.1—to include features like criteria queries and support for more advanced data modeling.

**Hibernate** predates JPA and was created as an open-source project in the early 2000s by Gavin King. It quickly gained popularity because it offered a powerful, flexible ORM solution at a time when standard Java APIs lacked viable options. Hibernate was so successful that much of JPA’s design was inspired by Hibernate’s capabilities.

**Spring Data JPA** emerged from the Spring ecosystem, aiming to radically simplify the development of data access layers. Introduced around 2011 as part of the larger Spring Data project, Spring Data JPA offered automatic query generation, built-in pagination, and seamless integration with Spring Boot.

### Architectural Differences

* **JPA** provides only interfaces and annotations, such as @Entity, @Id, and EntityManager, to define how Java objects relate to database records. It does not contain any implementation logic.
* **Hibernate** is a complete ORM framework that provides the implementation for JPA. It adds features like lazy loading, first-level and second-level caching, and support for native SQL.
* **Spring Data JPA** sits on top of JPA and Hibernate (or another JPA implementation). It abstracts repetitive CRUD operations by providing interfaces like JpaRepository, eliminating the need for boilerplate code.

### API and Coding Comparison

**JPA (Raw)**:

EntityManager em = entityManagerFactory.createEntityManager();

EntityTransaction tx = em.getTransaction();

try {

tx.begin();

em.persist(new Employee("John", "Doe"));

tx.commit();

} catch (Exception e) {

if (tx.isActive()) tx.rollback();

} finally {

em.close();

}

**Hibernate (Native)**:

Session session = sessionFactory.openSession();

Transaction tx = session.beginTransaction();

try {

session.save(new Employee("Jane", "Smith"));

tx.commit();

} catch (HibernateException e) {

tx.rollback();

} finally {

session.close();

}

**Spring Data JPA**:

@Service

public class EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

public void saveEmployee(Employee emp) {

employeeRepository.save(emp);

}

}

### Performance Comparison

* **JPA** performance depends entirely on the underlying implementation. It provides no caching or tuning mechanisms by itself.
* **Hibernate** provides advanced features such as:
  + First-level cache (session scope)
  + Second-level cache (shared across sessions)
  + Connection pooling and batch processing
  + SQL logging and custom interceptors
* **Spring Data JPA** can inherit Hibernate’s tuning options, but since it hides most of the details, developers need to be cautious when it comes to performance bottlenecks. Overusing auto-generated queries without inspection can lead to N+1 query issues.

### Use Cases and Adoption

* **JPA** is ideal when you want portability between different ORM providers or frameworks.
* **Hibernate** is preferred in complex projects needing fine-grained control over the ORM behavior.
* **Spring Data JPA** is excellent for rapid development of enterprise applications where boilerplate code needs to be minimized.

**Real-World Example**:

* E-commerce application:
  + **JPA** defines entities such as Product, User, and Order.
  + **Hibernate** handles fetching of product catalogs using optimized queries with caching.
  + **Spring Data JPA** simplifies the creation of repositories for managing users and orders.

### Advanced Capabilities

* **JPA**:
  + Criteria API
  + JPQL
  + Entity lifecycle callbacks (@PrePersist, @PostLoad)
* **Hibernate**:
  + Multi-tenancy
  + Custom SQL dialects
  + Enhanced mapping annotations (@Where, @Filter)
* **Spring Data JPA**:
  + Method name-based query derivation (findByLastName)
  + Projection interfaces for partial field selection
  + Custom @Query annotation
  + Support for Specifications API

### Integration and Tooling

* **JPA** is supported by all major IDEs with schema validation, code completion, and reverse engineering.
* **Hibernate** has dedicated tools in IntelliJ IDEA and Eclipse.
* **Spring Data JPA** integrates well with Spring Boot Actuator, Spring DevTools, and Spring Initializr, making setup and monitoring seamless.

### Extended Summary Table

| **Feature** | **JPA** | **Hibernate** | **Spring Data JPA** |
| --- | --- | --- | --- |
| Type | Specification | Implementation | Abstraction over JPA Implementation |
| Introduced in | 2006 (Java EE 5) | 2001 | 2011 |
| Boilerplate Code | High | Moderate | Minimal |
| Built-in Caching | No | Yes | Delegated to JPA provider |
| Query Language | JPQL | HQL, Native SQL | JPQL, Native SQL, derived queries |
| Learning Curve | Moderate | Steep | Beginner friendly |
| Community Support | Standard-level | Very Active | Extensive via Spring Ecosystem |
| Use Case | Portability-focused ORM | Performance and fine control | Rapid CRUD and prototyping |
| Integration Complexity | Medium | High | Low (with Spring Boot) |
| IDE Tooling | IntelliJ, Eclipse JPA support | IntelliJ Hibernate plugin | Spring Tool Suite, Spring Boot DevTools |
| Popular Libraries | EclipseLink, OpenJPA | Hibernate ORM | Spring Boot, Spring Data |
| Declarative Transactions | With EJB or Spring | Supported | Fully integrated via @Transactional |
| Testability | Requires setup | More complex | Easily testable with Spring Boot Test |

### Final Thoughts

JPA, Hibernate, and Spring Data JPA are not competitors but complementary technologies. A typical Spring Boot application will use JPA annotations, rely on Hibernate for implementation, and use Spring Data JPA to abstract repetitive repository logic. Understanding when and how to leverage each component is key to building maintainable and scalable Java applications.

In the next section, we will look at code practices and advanced usage patterns in Spring Data JPA that leverage these core technologies effectively.

Understanding the distinction between JPA, Hibernate, and Spring Data JPA is crucial for developing and maintaining a clean and scalable persistence layer in any enterprise Java application.

### Java Persistence API (JPA)

#### Historical Evolution of JPA

Before JPA, enterprise-level Java persistence was handled through Enterprise JavaBeans (EJB) 2.x, which included entity beans as a mechanism to persist data. Entity beans were tightly coupled to the EJB container, difficult to test, and required verbose boilerplate code for remote interfaces and object serialization. As developers began to favor lighter, more maintainable solutions, frameworks like Hibernate, iBatis, and TopLink emerged, offering more practical object-relational mapping (ORM) strategies.

To standardize these approaches and reduce the heavy dependencies of EJB, the Java Community Process (JCP) introduced JPA as part of the EJB 3.0 specification under JSR 220. This was a major shift toward simplifying persistence and promoting POJO (Plain Old Java Object) development. With JPA, developers could now define entities using annotations without being tied to heavyweight container services.

JPA 1.0, introduced in Java EE 5 (2006), laid the groundwork by supporting basic ORM concepts. Subsequent versions, such as JPA 2.0 and 2.1, introduced critical features like:

* The Criteria API for type-safe, programmatic queries
* Support for collection-valued parameters in JPQL
* Element collections and embeddable types
* Better support for caching and locking

JPA 2.2, released with Java EE 8, added Java 8 enhancements like support for Stream, Optional, and Date/Time API.

With the move from Java EE to Jakarta EE (under the Eclipse Foundation), JPA continues to evolve and remains the core standard for ORM in modern enterprise applications.

#### Definition and Purpose

* **Definition**: JPA is a Java specification (JSR 338) that defines a standard for object-relational mapping (ORM) between Java objects and relational databases.
* **Nature**: It provides a metadata-driven approach to ORM using annotations and XML but no actual implementation.
* **Purpose**: JPA simplifies persistence by allowing developers to map, query, and manage relational data using Java objects, reducing the need to write complex SQL or JDBC code directly.

#### Key Features

* Declarative object-relational mapping via annotations (@Entity, @Table, @Column, etc.)
* Relationship mapping (@OneToMany, @ManyToOne, @OneToOne, @ManyToMany)
* Entity lifecycle management (@PrePersist, @PostLoad, etc.)
* Support for JPQL (Java Persistence Query Language)
* Support for criteria-based dynamic queries
* Pluggable persistence providers (e.g., Hibernate, EclipseLink)
* Integration with Java EE and Spring environments

#### Implementation Providers

JPA is only an API specification. It must be implemented by a concrete ORM provider. Common implementations include:

* **Hibernate**: The most widely adopted and feature-rich implementation
* **EclipseLink**: The reference implementation for JPA
* **Apache OpenJPA**: A robust alternative with enterprise support
* **DataNucleus**: Supports not just relational DBs but also NoSQL and object-based databases

#### Annotations Provided

JPA uses annotations to define mappings and behaviors for persistence:

* @Entity: Marks a class as a JPA entity (a table mapping)
* @Id: Declares the primary key
* @GeneratedValue: Defines the strategy for primary key generation
* @Column: Maps a field to a table column
* @Table: Allows customization of the table name and constraints
* @JoinColumn, @JoinTable: Specify join strategies for relationships

JPA has drastically influenced how developers design enterprise applications, bringing simplicity, portability, and standardization to Java persistence layers. It remains a cornerstone of the Java EE and Spring ecosystems and continues to play a vital role in data-driven application architectures.

* **Definition**: JPA is a Java specification (JSR 338) that defines a standard for object-relational mapping (ORM) between Java objects and relational databases.
* **Nature**: It is just a set of interfaces and annotations. It provides no implementation itself.
* **Purpose**: Allows developers to interact with relational data using Java objects rather than SQL queries.
* **Annotations Provided**:
  + @Entity, @Table, @Id, @Column, @ManyToOne, @OneToMany, etc.
* **Implementation Providers**: JPA must be implemented by a concrete ORM tool such as:
  + Hibernate
  + EclipseLink
  + OpenJPA

### Hibernate

* **Definition**: Hibernate is one of the most popular and widely used implementations of JPA.
* **Features**:
  + Full support for JPA
  + Extended ORM capabilities beyond JPA spec (e.g., @Filter, @SQLDelete, Second-level cache, etc.)
  + Automatic dirty checking
  + Lazy/eager fetching strategies
  + Criteria API and HQL (Hibernate Query Language)
* **Stand-alone Capability**: Hibernate can be used directly without JPA, with its own native APIs and configurations.
* **Example Usage Without Spring**:

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

* **Definition**: Spring Data JPA is a part of the Spring Data umbrella project that provides an abstraction over JPA and a convenient way to interact with databases with minimal boilerplate code.
* **Not a JPA Implementation**: Instead, it builds on top of an existing JPA provider (typically Hibernate).
* **Key Benefits**:
  + Eliminates the need for boilerplate code
  + Simplifies data access with JpaRepository, CrudRepository, PagingAndSortingRepository
  + Auto-generates implementations of interface methods (like findByName, findByEmail) based on method names
  + Integrates seamlessly with Spring’s dependency injection and transaction management
* **Built-in Features**:
  + Pagination and Sorting
  + Query creation from method names
  + Query DSL and Specifications
  + Native queries and JPQL support
* **Spring Data Projections**:
  + Allows you to fetch partial data from an entity without retrieving the entire object.
  + Types:
    - **Interface-based projections**: Interfaces with getters that match field names.
    - **Class-based (DTO) projections**: Map query results directly into a custom POJO.
    - **Dynamic projections**: Choose the projection type dynamically at runtime.

public interface EmployeeNameView {

String getFirstName();

String getLastName();

}

List<EmployeeNameView> findByDepartment(String dept);

* **Custom Repositories**:
  + Extend JpaRepository with custom methods.
  + Implement these methods in a class suffixed with Impl.

public interface EmployeeRepositoryCustom {

List<Employee> findEmployeesWithHighSalary();

}

public class EmployeeRepositoryImpl implements EmployeeRepositoryCustom {

@PersistenceContext

private EntityManager em;

public List<Employee> findEmployeesWithHighSalary() {

return em.createQuery("FROM Employee e WHERE e.salary > 100000", Employee.class)

.getResultList();

}

}

* **Integration with Spring Boot Auto Configuration**:
  + Spring Boot detects spring-boot-starter-data-jpa and auto-configures:
    - EntityManagerFactory
    - DataSource
    - TransactionManager
    - Repository interfaces
  + No manual configuration is required for common setups.
  + Application properties can fine-tune JPA behavior:

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

spring.datasource.username=root

spring.datasource.password=root

spring.jpa.show-sql=true

spring.jpa.hibernate.ddl-auto=update

This tight integration makes it very convenient to get started quickly and scale applications with robust persistence features.

* **Definition**: Spring Data JPA is a part of the Spring Data umbrella project that provides an abstraction over JPA and a convenient way to interact with databases with minimal boilerplate code.
* **Not a JPA Implementation**: Instead, it builds on top of an existing JPA provider (typically Hibernate).
* **Key Benefits**:
  + Eliminates the need for boilerplate code
  + Simplifies data access with JpaRepository, CrudRepository, PagingAndSortingRepository
  + Auto-generates implementations of interface methods (like findByName, findByEmail) based on method names
  + Integrates seamlessly with Spring’s dependency injection and transaction management
* **Built-in Features**:
  + Pagination and Sorting
  + Query creation from method names
  + Query DSL and Specifications
  + Native queries and JPQL support
* **Example with Spring Data JPA**:
  + **Repository**:

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

}

* **Service Layer**:

@Autowired

private EmployeeRepository employeeRepository;

@Transactional

public void addEmployee(Employee employee) {

employeeRepository.save(employee);

}

### Summary of Differences

| **Feature** | **JPA** | **Hibernate** | **Spring Data JPA** |
| --- | --- | --- | --- |
| Type | Specification | Implementation | Abstraction Framework |
| Provides Implementation | No | Yes | No (depends on Hibernate, EclipseLink, etc.) |
| Reduces Boilerplate | No | Partially | Yes |
| Auto Query Methods | No | No | Yes |
| Stand-alone Usage | No | Yes | No (requires Spring environment) |
| Used in Spring Boot | Yes (via Hibernate, etc.) | Yes (as JPA provider) | Yes |
| Performance Tuning Options | Limited, depends on implementation | Extensive tuning options available | Inherits capabilities from JPA provider |
| Learning Curve | Moderate | Steep | Low to moderate |
| Integration Complexity | Medium | High | Low |
| Tooling Support | Good (via IDEs and JPA plugins) | Excellent (Hibernate-specific tools) | Excellent (Spring Boot + Spring Data tools) |
| Feature | JPA | Hibernate | Spring Data JPA |
| ------------------------ | ------------------------------------- | ---------------------------------------- | -------------------------------------------------- |
| Type | Specification | Implementation | Abstraction Framework |
| Provides Implementation | No | Yes | No (depends on Hibernate, EclipseLink, etc.) |
| Reduces Boilerplate | No | Partially | Yes |
| Auto Query Methods | No | No | Yes |
| Stand-alone Usage | No | Yes | No (requires Spring environment) |
| Used in Spring Boot | Yes (via Hibernate, etc.) | Yes (as JPA provider) | Yes |

## 17. References

* [DZone - Hibernate vs Spring Data JPA](https://dzone.com/articles/what-is-the-difference-between-hibernate-and-sprin-1)
* [JavaWorld - Introduction to JPA](https://www.javaworld.com/article/3379043/what-is-jpa-introduction-to-the-java-persistence-api.html)

**Hands on 5**

**Implement services for managing Country**   
  
An application requires for features to be implemented with regards to country. These features needs to be supported by implementing them as service using Spring Data JPA.

* Find a country based on country code
* Add new country
* Update country
* Delete country
* Find list of countries matching a partial country name

Before starting the implementation of the above features, there are few configuration and data population that needs to be incorporated. Please refer each topic below and implement the same.   
  
**Explanation for Hibernate table creation configuration**

* Moreover the ddl-auto defines how hibernate behaves if a specific table or column is not present in the database.
  + create - drops existing tables data and structure, then creates new tables
  + validate - check if the table and columns exist or not, throws an exception if a matching table or column is not found
  + update - if a table does not exists, it creates a new table; if a column does not exists, it creates a new column
  + create-drop - creates the table, once all operations are completed, the table is dropped

# Hibernate ddl auto (create, create-drop, update, validate)

spring.jpa.hibernate.ddl-auto=validate

Populate country table

* Delete all the records in Country table and then use the below script to create the actual list of all countries in our world.

insert into country (co\_code, co\_name) values ("AF", "Afghanistan");

insert into country (co\_code, co\_name) values ("AL", "Albania");

insert into country (co\_code, co\_name) values ("DZ", "Algeria");

insert into country (co\_code, co\_name) values ("AS", "American Samoa");

insert into country (co\_code, co\_name) values ("AD", "Andorra");

insert into country (co\_code, co\_name) values ("AO", "Angola");

insert into country (co\_code, co\_name) values ("AI", "Anguilla");

insert into country (co\_code, co\_name) values ("AQ", "Antarctica");

insert into country (co\_code, co\_name) values ("AG", "Antigua and Barbuda");

insert into country (co\_code, co\_name) values ("AR", "Argentina");

insert into country (co\_code, co\_name) values ("AM", "Armenia");

insert into country (co\_code, co\_name) values ("AW", "Aruba");

insert into country (co\_code, co\_name) values ("AU", "Australia");

insert into country (co\_code, co\_name) values ("AT", "Austria");

insert into country (co\_code, co\_name) values ("AZ", "Azerbaijan");

insert into country (co\_code, co\_name) values ("BS", "Bahamas");

insert into country (co\_code, co\_name) values ("BH", "Bahrain");

insert into country (co\_code, co\_name) values ("BD", "Bangladesh");

insert into country (co\_code, co\_name) values ("BB", "Barbados");

insert into country (co\_code, co\_name) values ("BY", "Belarus");

insert into country (co\_code, co\_name) values ("BE", "Belgium");

insert into country (co\_code, co\_name) values ("BZ", "Belize");

insert into country (co\_code, co\_name) values ("BJ", "Benin");

insert into country (co\_code, co\_name) values ("BM", "Bermuda");

insert into country (co\_code, co\_name) values ("BT", "Bhutan");

insert into country (co\_code, co\_name) values ("BO", "Bolivia, Plurinational State of");

insert into country (co\_code, co\_name) values ("BQ", "Bonaire, Sint Eustatius and Saba");

insert into country (co\_code, co\_name) values ("BA", "Bosnia and Herzegovina");

insert into country (co\_code, co\_name) values ("BW", "Botswana");

insert into country (co\_code, co\_name) values ("BV", "Bouvet Island");

insert into country (co\_code, co\_name) values ("BR", "Brazil");

insert into country (co\_code, co\_name) values ("IO", "British Indian Ocean Territory");

insert into country (co\_code, co\_name) values ("BN", "Brunei Darussalam");

insert into country (co\_code, co\_name) values ("BG", "Bulgaria");

insert into country (co\_code, co\_name) values ("BF", "Burkina Faso");

insert into country (co\_code, co\_name) values ("BI", "Burundi");

insert into country (co\_code, co\_name) values ("KH", "Cambodia");

insert into country (co\_code, co\_name) values ("CM", "Cameroon");

insert into country (co\_code, co\_name) values ("CA", "Canada");

insert into country (co\_code, co\_name) values ("CV", "Cape Verde");

insert into country (co\_code, co\_name) values ("KY", "Cayman Islands");

insert into country (co\_code, co\_name) values ("CF", "Central African Republic");

insert into country (co\_code, co\_name) values ("TD", "Chad");

insert into country (co\_code, co\_name) values ("CL", "Chile");

insert into country (co\_code, co\_name) values ("CN", "China");

insert into country (co\_code, co\_name) values ("CX", "Christmas Island");

insert into country (co\_code, co\_name) values ("CC", "Cocos (Keeling) Islands");

insert into country (co\_code, co\_name) values ("CO", "Colombia");

insert into country (co\_code, co\_name) values ("KM", "Comoros");

insert into country (co\_code, co\_name) values ("CG", "Congo");

insert into country (co\_code, co\_name) values ("CD", "Congo, the Democratic Republic of the");

insert into country (co\_code, co\_name) values ("CK", "Cook Islands");

insert into country (co\_code, co\_name) values ("CR", "Costa Rica");

insert into country (co\_code, co\_name) values ("HR", "Croatia");

insert into country (co\_code, co\_name) values ("CU", "Cuba");

insert into country (co\_code, co\_name) values ("CW", "Curaçao");

insert into country (co\_code, co\_name) values ("CY", "Cyprus");

insert into country (co\_code, co\_name) values ("CZ", "Czech Republic");

insert into country (co\_code, co\_name) values ("CI", "Côte d'Ivoire");

insert into country (co\_code, co\_name) values ("DK", "Denmark");

insert into country (co\_code, co\_name) values ("DJ", "Djibouti");

insert into country (co\_code, co\_name) values ("DM", "Dominica");

insert into country (co\_code, co\_name) values ("DO", "Dominican Republic");

insert into country (co\_code, co\_name) values ("EC", "Ecuador");

insert into country (co\_code, co\_name) values ("EG", "Egypt");

insert into country (co\_code, co\_name) values ("SV", "El Salvador");

insert into country (co\_code, co\_name) values ("GQ", "Equatorial Guinea");

insert into country (co\_code, co\_name) values ("ER", "Eritrea");

insert into country (co\_code, co\_name) values ("EE", "Estonia");

insert into country (co\_code, co\_name) values ("ET", "Ethiopia");

insert into country (co\_code, co\_name) values ("FK", "Falkland Islands (Malvinas)");

insert into country (co\_code, co\_name) values ("FO", "Faroe Islands");

insert into country (co\_code, co\_name) values ("FJ", "Fiji");

insert into country (co\_code, co\_name) values ("FI", "Finland");

insert into country (co\_code, co\_name) values ("FR", "France");

insert into country (co\_code, co\_name) values ("GF", "French Guiana");

insert into country (co\_code, co\_name) values ("PF", "French Polynesia");

insert into country (co\_code, co\_name) values ("TF", "French Southern Territories");

insert into country (co\_code, co\_name) values ("GA", "Gabon");

insert into country (co\_code, co\_name) values ("GM", "Gambia");

insert into country (co\_code, co\_name) values ("GE", "Georgia");

insert into country (co\_code, co\_name) values ("DE", "Germany");

insert into country (co\_code, co\_name) values ("GH", "Ghana");

insert into country (co\_code, co\_name) values ("GI", "Gibraltar");

insert into country (co\_code, co\_name) values ("GR", "Greece");

insert into country (co\_code, co\_name) values ("GL", "Greenland");

insert into country (co\_code, co\_name) values ("GD", "Grenada");

insert into country (co\_code, co\_name) values ("GP", "Guadeloupe");

insert into country (co\_code, co\_name) values ("GU", "Guam");

insert into country (co\_code, co\_name) values ("GT", "Guatemala");

insert into country (co\_code, co\_name) values ("GG", "Guernsey");

insert into country (co\_code, co\_name) values ("GN", "Guinea");

insert into country (co\_code, co\_name) values ("GW", "Guinea-Bissau");

insert into country (co\_code, co\_name) values ("GY", "Guyana");

insert into country (co\_code, co\_name) values ("HT", "Haiti");

insert into country (co\_code, co\_name) values ("HM", "Heard Island and McDonald Islands");

insert into country (co\_code, co\_name) values ("VA", "Holy See (Vatican City State)");

insert into country (co\_code, co\_name) values ("HN", "Honduras");

insert into country (co\_code, co\_name) values ("HK", "Hong Kong");

insert into country (co\_code, co\_name) values ("HU", "Hungary");

insert into country (co\_code, co\_name) values ("IS", "Iceland");

insert into country (co\_code, co\_name) values ("IN", "India");

insert into country (co\_code, co\_name) values ("ID", "Indonesia");

insert into country (co\_code, co\_name) values ("IR", "Iran, Islamic Republic of");

insert into country (co\_code, co\_name) values ("IQ", "Iraq");

insert into country (co\_code, co\_name) values ("IE", "Ireland");

insert into country (co\_code, co\_name) values ("IM", "Isle of Man");

insert into country (co\_code, co\_name) values ("IL", "Israel");

insert into country (co\_code, co\_name) values ("IT", "Italy");

insert into country (co\_code, co\_name) values ("JM", "Jamaica");

insert into country (co\_code, co\_name) values ("JP", "Japan");

insert into country (co\_code, co\_name) values ("JE", "Jersey");

insert into country (co\_code, co\_name) values ("JO", "Jordan");

insert into country (co\_code, co\_name) values ("KZ", "Kazakhstan");

insert into country (co\_code, co\_name) values ("KE", "Kenya");

insert into country (co\_code, co\_name) values ("KI", "Kiribati");

insert into country (co\_code, co\_name) values ("KP", "Democratic People's Republic of Korea");

insert into country (co\_code, co\_name) values ("KR", "Republic of Korea");

insert into country (co\_code, co\_name) values ("KW", "Kuwait");

insert into country (co\_code, co\_name) values ("KG", "Kyrgyzstan");

insert into country (co\_code, co\_name) values ("LA", "Lao People's Democratic Republic");

insert into country (co\_code, co\_name) values ("LV", "Latvia");

insert into country (co\_code, co\_name) values ("LB", "Lebanon");

insert into country (co\_code, co\_name) values ("LS", "Lesotho");

insert into country (co\_code, co\_name) values ("LR", "Liberia");

insert into country (co\_code, co\_name) values ("LY", "Libya");

insert into country (co\_code, co\_name) values ("LI", "Liechtenstein");

insert into country (co\_code, co\_name) values ("LT", "Lithuania");

insert into country (co\_code, co\_name) values ("LU", "Luxembourg");

insert into country (co\_code, co\_name) values ("MO", "Macao");

insert into country (co\_code, co\_name) values ("MK", "Macedonia, the Former Yugoslav Republic of");

insert into country (co\_code, co\_name) values ("MG", "Madagascar");

insert into country (co\_code, co\_name) values ("MW", "Malawi");

insert into country (co\_code, co\_name) values ("MY", "Malaysia");

insert into country (co\_code, co\_name) values ("MV", "Maldives");

insert into country (co\_code, co\_name) values ("ML", "Mali");

insert into country (co\_code, co\_name) values ("MT", "Malta");

insert into country (co\_code, co\_name) values ("MH", "Marshall Islands");

insert into country (co\_code, co\_name) values ("MQ", "Martinique");

insert into country (co\_code, co\_name) values ("MR", "Mauritania");

insert into country (co\_code, co\_name) values ("MU", "Mauritius");

insert into country (co\_code, co\_name) values ("YT", "Mayotte");

insert into country (co\_code, co\_name) values ("MX", "Mexico");

insert into country (co\_code, co\_name) values ("FM", "Micronesia, Federated States of");

insert into country (co\_code, co\_name) values ("MD", "Moldova, Republic of");

insert into country (co\_code, co\_name) values ("MC", "Monaco");

insert into country (co\_code, co\_name) values ("MN", "Mongolia");

insert into country (co\_code, co\_name) values ("ME", "Montenegro");

insert into country (co\_code, co\_name) values ("MS", "Montserrat");

insert into country (co\_code, co\_name) values ("MA", "Morocco");

insert into country (co\_code, co\_name) values ("MZ", "Mozambique");

insert into country (co\_code, co\_name) values ("MM", "Myanmar");

insert into country (co\_code, co\_name) values ("NA", "Namibia");

insert into country (co\_code, co\_name) values ("NR", "Nauru");

insert into country (co\_code, co\_name) values ("NP", "Nepal");

insert into country (co\_code, co\_name) values ("NL", "Netherlands");

insert into country (co\_code, co\_name) values ("NC", "New Caledonia");

insert into country (co\_code, co\_name) values ("NZ", "New Zealand");

insert into country (co\_code, co\_name) values ("NI", "Nicaragua");

insert into country (co\_code, co\_name) values ("NE", "Niger");

insert into country (co\_code, co\_name) values ("NG", "Nigeria");

insert into country (co\_code, co\_name) values ("NU", "Niue");

insert into country (co\_code, co\_name) values ("NF", "Norfolk Island");

insert into country (co\_code, co\_name) values ("MP", "Northern Mariana Islands");

insert into country (co\_code, co\_name) values ("NO", "Norway");

insert into country (co\_code, co\_name) values ("OM", "Oman");

insert into country (co\_code, co\_name) values ("PK", "Pakistan");

insert into country (co\_code, co\_name) values ("PW", "Palau");

insert into country (co\_code, co\_name) values ("PS", "Palestine, State of");

insert into country (co\_code, co\_name) values ("PA", "Panama");

insert into country (co\_code, co\_name) values ("PG", "Papua New Guinea");

insert into country (co\_code, co\_name) values ("PY", "Paraguay");

insert into country (co\_code, co\_name) values ("PE", "Peru");

insert into country (co\_code, co\_name) values ("PH", "Philippines");

insert into country (co\_code, co\_name) values ("PN", "Pitcairn");

insert into country (co\_code, co\_name) values ("PL", "Poland");

insert into country (co\_code, co\_name) values ("PT", "Portugal");

insert into country (co\_code, co\_name) values ("PR", "Puerto Rico");

insert into country (co\_code, co\_name) values ("QA", "Qatar");

insert into country (co\_code, co\_name) values ("RO", "Romania");

insert into country (co\_code, co\_name) values ("RU", "Russian Federation");

insert into country (co\_code, co\_name) values ("RW", "Rwanda");

insert into country (co\_code, co\_name) values ("RE", "Réunion");

insert into country (co\_code, co\_name) values ("BL", "Saint Barthélemy");

insert into country (co\_code, co\_name) values ("SH", "Saint Helena, Ascension and Tristan da Cunha");

insert into country (co\_code, co\_name) values ("KN", "Saint Kitts and Nevis");

insert into country (co\_code, co\_name) values ("LC", "Saint Lucia");

insert into country (co\_code, co\_name) values ("MF", "Saint Martin (French part)");

insert into country (co\_code, co\_name) values ("PM", "Saint Pierre and Miquelon");

insert into country (co\_code, co\_name) values ("VC", "Saint Vincent and the Grenadines");

insert into country (co\_code, co\_name) values ("WS", "Samoa");

insert into country (co\_code, co\_name) values ("SM", "San Marino");

insert into country (co\_code, co\_name) values ("ST", "Sao Tome and Principe");

insert into country (co\_code, co\_name) values ("SA", "Saudi Arabia");

insert into country (co\_code, co\_name) values ("SN", "Senegal");

insert into country (co\_code, co\_name) values ("RS", "Serbia");

insert into country (co\_code, co\_name) values ("SC", "Seychelles");

insert into country (co\_code, co\_name) values ("SL", "Sierra Leone");

insert into country (co\_code, co\_name) values ("SG", "Singapore");

insert into country (co\_code, co\_name) values ("SX", "Sint Maarten (Dutch part)");

insert into country (co\_code, co\_name) values ("SK", "Slovakia");

insert into country (co\_code, co\_name) values ("SI", "Slovenia");

insert into country (co\_code, co\_name) values ("SB", "Solomon Islands");

insert into country (co\_code, co\_name) values ("SO", "Somalia");

insert into country (co\_code, co\_name) values ("ZA", "South Africa");

insert into country (co\_code, co\_name) values ("GS", "South Georgia and the South Sandwich Islands");

insert into country (co\_code, co\_name) values ("SS", "South Sudan");

insert into country (co\_code, co\_name) values ("ES", "Spain");

insert into country (co\_code, co\_name) values ("LK", "Sri Lanka");

insert into country (co\_code, co\_name) values ("SD", "Sudan");

insert into country (co\_code, co\_name) values ("SR", "Suriname");

insert into country (co\_code, co\_name) values ("SJ", "Svalbard and Jan Mayen");

insert into country (co\_code, co\_name) values ("SZ", "Swaziland");

insert into country (co\_code, co\_name) values ("SE", "Sweden");

insert into country (co\_code, co\_name) values ("CH", "Switzerland");

insert into country (co\_code, co\_name) values ("SY", "Syrian Arab Republic");

insert into country (co\_code, co\_name) values ("TW", "Taiwan, Province of China");

insert into country (co\_code, co\_name) values ("TJ", "Tajikistan");

insert into country (co\_code, co\_name) values ("TZ", "Tanzania, United Republic of");

insert into country (co\_code, co\_name) values ("TH", "Thailand");

insert into country (co\_code, co\_name) values ("TL", "Timor-Leste");

insert into country (co\_code, co\_name) values ("TG", "Togo");

insert into country (co\_code, co\_name) values ("TK", "Tokelau");

insert into country (co\_code, co\_name) values ("TO", "Tonga");

insert into country (co\_code, co\_name) values ("TT", "Trinidad and Tobago");

insert into country (co\_code, co\_name) values ("TN", "Tunisia");

insert into country (co\_code, co\_name) values ("TR", "Turkey");

insert into country (co\_code, co\_name) values ("TM", "Turkmenistan");

insert into country (co\_code, co\_name) values ("TC", "Turks and Caicos Islands");

insert into country (co\_code, co\_name) values ("TV", "Tuvalu");

insert into country (co\_code, co\_name) values ("UG", "Uganda");

insert into country (co\_code, co\_name) values ("UA", "Ukraine");

insert into country (co\_code, co\_name) values ("AE", "United Arab Emirates");

insert into country (co\_code, co\_name) values ("GB", "United Kingdom");

insert into country (co\_code, co\_name) values ("US", "United States");

insert into country (co\_code, co\_name) values ("UM", "United States Minor Outlying Islands");

insert into country (co\_code, co\_name) values ("UY", "Uruguay");

insert into country (co\_code, co\_name) values ("UZ", "Uzbekistan");

insert into country (co\_code, co\_name) values ("VU", "Vanuatu");

insert into country (co\_code, co\_name) values ("VE", "Venezuela, Bolivarian Republic of");

insert into country (co\_code, co\_name) values ("VN", "Viet Nam");

insert into country (co\_code, co\_name) values ("VG", "Virgin Islands, British");

insert into country (co\_code, co\_name) values ("VI", "Virgin Islands, U.S.");

insert into country (co\_code, co\_name) values ("WF", "Wallis and Futuna");

insert into country (co\_code, co\_name) values ("EH", "Western Sahara");

insert into country (co\_code, co\_name) values ("YE", "Yemen");

insert into country (co\_code, co\_name) values ("ZM", "Zambia");

insert into country (co\_code, co\_name) values ("ZW", "Zimbabwe");

insert into country (co\_code, co\_name) values ("AX", "Åland Islands");

Refer subsequent hands on exercises to implement the features related to country.

## 18. Hands-on 5: Country Management Service with Spring Data JPA

This hands-on guides you through building a Spring Boot-based application for managing countries using Spring Data JPA.

### ✅ Objective

Implement a CRUD-based service to manage country records stored in a database, using Spring Boot, Spring Data JPA, and optionally an in-memory H2 database.

### 📦 Step-by-Step Implementation

#### 1. Create Spring Boot Project

* Use [Spring Initializr](https://start.spring.io/)
* Name: country-service
* Dependencies:
  + Spring Web
  + Spring Data JPA
  + H2 Database (or MySQL/PostgreSQL)
  + Lombok (optional)

#### 2. Entity Class

@Entity

@Table(name = "country")

public class Country {

@Id

private String coCode;

private String coName;

public Country() {}

public Country(String coCode, String coName) {

this.coCode = coCode;

this.coName = coName;

}

// Getters and Setters

}

#### 3. Repository Interface

public interface CountryRepository extends JpaRepository<Country, String> {

List<Country> findByCoNameContainingIgnoreCase(String partialName);

}

#### 4. Service Layer

public interface CountryService {

Country findByCode(String code);

Country addCountry(Country country);

Country updateCountry(Country country);

void deleteCountry(String code);

List<Country> searchCountries(String namePart);

}

#### 5. Service Implementation

@Service

public class CountryServiceImpl implements CountryService {

@Autowired

private CountryRepository countryRepository;

@Override

public Country findByCode(String code) {

return countryRepository.findById(code)

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

}

@Override

public Country addCountry(Country country) {

return countryRepository.save(country);

}

@Override

public Country updateCountry(Country country) {

if (!countryRepository.existsById(country.getCoCode())) {

throw new RuntimeException("Cannot update non-existing country");

}

return countryRepository.save(country);

}

@Override

public void deleteCountry(String code) {

countryRepository.deleteById(code);

}

@Override

public List<Country> searchCountries(String namePart) {

return countryRepository.findByCoNameContainingIgnoreCase(namePart);

}

}

#### 6. Controller Layer

@RestController

@RequestMapping("/countries")

public class CountryController {

@Autowired

private CountryService countryService;

@GetMapping("/{code}")

public Country getCountry(@PathVariable String code) {

return countryService.findByCode(code);

}

@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);

}

@GetMapping("/search")

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

return countryService.searchCountries(name);

}

}

#### 7. application.properties

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

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=

spring.jpa.hibernate.ddl-auto=create

spring.jpa.show-sql=true

spring.h2.console.enabled=true

spring.h2.console.path=/h2-console

Replace with MySQL/PostgreSQL configuration for production.

#### 8. SQL Initialization (optional)

Add data.sql to src/main/resources/:

insert into country(co\_code, co\_name) values('IN','India');

insert into country(co\_code, co\_name) values('US','United States');

insert into country(co\_code, co\_name) values('UK','United Kingdom');

insert into country(co\_code, co\_name) values('JP','Japan');

-- Add more as required

#### 9. API Testing

| **Method** | **Endpoint** | **Description** |
| --- | --- | --- |
| GET | /countries/IN | Get country by code |
| POST | /countries | Add new country |
| PUT | /countries | Update existing country |
| DELETE | /countries/IN | Delete country by code |
| GET | /countries/search?name=land | Search by name part |

**Hands on 6**

**Find a country based on country code** 

* Create new exception class CountryNotFoundException in com.cognizant.spring-learn.service.exception
* Create new method findCountryByCode() in CountryService with @Transactional annotation
* In findCountryByCode() method, perform the following steps:
  + Method signature

@Transactional

public Country findCountryByCode(String countryCode) throws CountryNotFoundException

* Get the country based on findById() built in method

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

* From the result, check if a country is found. If not found, throw CountryNotFoundException

if (!result.isPresent())

* Use get() method to return the country fetched.

Country country = result.get();

* Include new test method in OrmLearnApplication to find a country based on country code and compare the country name to check if it is valid.

    private static void getAllCountriesTest() {

        LOGGER.info("Start");

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

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

        LOGGER.info("End");

    }

* Invoke the above method in main() method and test it.

**NOTE:** SME to explain the importance of @Transactional annotation. Spring takes care of creating the Hibernate session and manages the transactionality when executing the service method

### Hands-on Spring Data JPA - Comprehensive Guide with Deep Dive and Full Example

[...existing content remains unchanged above this point...]

## 18. Hands-on 5: Country Management Service with Spring Data JPA

This hands-on guides you through building a Spring Boot-based application for managing countries using Spring Data JPA.

### ✅ Objective

Implement a CRUD-based service to manage country records stored in a database, using Spring Boot, Spring Data JPA, and optionally an in-memory H2 database.

### 📦 Step-by-Step Implementation

#### 1. Create Spring Boot Project

* Use [Spring Initializr](https://start.spring.io/)
* Name: country-service
* Dependencies:
  + Spring Web
  + Spring Data JPA
  + H2 Database (or MySQL/PostgreSQL)
  + Lombok (optional)

#### 2. Entity Class

@Entity

@Table(name = "country")

public class Country {

@Id

private String coCode;

private String coName;

public Country() {}

public Country(String coCode, String coName) {

this.coCode = coCode;

this.coName = coName;

}

// Getters and Setters

}

#### 3. Repository Interface

public interface CountryRepository extends JpaRepository<Country, String> {

List<Country> findByCoNameContainingIgnoreCase(String partialName);

}

#### 4. Service Layer

public interface CountryService {

Country findByCode(String code);

Country addCountry(Country country);

Country updateCountry(Country country);

void deleteCountry(String code);

List<Country> searchCountries(String namePart);

}

#### 5. Service Implementation

@Service

public class CountryServiceImpl implements CountryService {

@Autowired

private CountryRepository countryRepository;

@Override

public Country findByCode(String code) {

return countryRepository.findById(code)

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

}

@Override

public Country addCountry(Country country) {

return countryRepository.save(country);

}

@Override

public Country updateCountry(Country country) {

if (!countryRepository.existsById(country.getCoCode())) {

throw new RuntimeException("Cannot update non-existing country");

}

return countryRepository.save(country);

}

@Override

public void deleteCountry(String code) {

countryRepository.deleteById(code);

}

@Override

public List<Country> searchCountries(String namePart) {

return countryRepository.findByCoNameContainingIgnoreCase(namePart);

}

}

#### 6. Controller Layer

@RestController

@RequestMapping("/countries")

public class CountryController {

@Autowired

private CountryService countryService;

@GetMapping("/{code}")

public Country getCountry(@PathVariable String code) {

return countryService.findByCode(code);

}

@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);

}

@GetMapping("/search")

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

return countryService.searchCountries(name);

}

}

#### 7. application.properties

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

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=

spring.jpa.hibernate.ddl-auto=create

spring.jpa.show-sql=true

spring.h2.console.enabled=true

spring.h2.console.path=/h2-console

Replace with MySQL/PostgreSQL configuration for production.

#### 8. SQL Initialization (optional)

Add data.sql to src/main/resources/:

insert into country(co\_code, co\_name) values('IN','India');

insert into country(co\_code, co\_name) values('US','United States');

insert into country(co\_code, co\_name) values('UK','United Kingdom');

insert into country(co\_code, co\_name) values('JP','Japan');

-- Add more as required

#### 9. API Testing

| **Method** | **Endpoint** | **Description** |
| --- | --- | --- |
| GET | /countries/IN | Get country by code |
| POST | /countries | Add new country |
| PUT | /countries | Update existing country |
| DELETE | /countries/IN | Delete country by code |
| GET | /countries/search?name=land | Search by name part |

## 19. Hands-on 6: Find Country by Country Code with Exception Handling

### 🧩 Objective

Enhance the CountryService to handle the case where a country code does not exist by throwing a custom exception.

### 📂 Step-by-Step

#### 1. Create CountryNotFoundException

package com.cognizant.springlearn.service.exception;

public class CountryNotFoundException extends RuntimeException {

public CountryNotFoundException(String message) {

super(message);

}

}

#### 2. Modify Service Method with @Transactional

@Service

public class CountryServiceImpl implements CountryService {

@Autowired

private CountryRepository countryRepository;

@Override

@Transactional

public Country findByCode(String countryCode) {

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

if (!result.isPresent()) {

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

}

return result.get();

}

}

📌 @Transactional ensures Spring manages transaction boundaries and the Hibernate session lifecycle automatically.

#### 3. Update CountryService Interface

public interface CountryService {

Country findByCode(String countryCode) throws CountryNotFoundException;

// other methods remain unchanged

}

#### 4. Add Test Method in OrmLearnApplication

private static void getAllCountriesTest() {

LOGGER.info("Start");

try {

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

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

} catch (CountryNotFoundException e) {

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

}

LOGGER.info("End");

}

#### 5. Modify Main Method

public static void main(String[] args) {

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

countryService = context.getBean(CountryService.class);

getAllCountriesTest();

}

**Hands on 7**

**Add a new country** 

* Create new method in CountryService.

@Transactional

public void addCountry(Country country)

* Invoke save() method of repository to get the country added.

countryRepository.save(country)

* Include new testAddCountry() method in OrmLearnApplication. Perform steps below:
  + Create new instance of country with a new code and name
  + Call countryService.addCountry() passing the country created in the previous step.
  + Invoke countryService.findCountryByCode() passing the same code used when adding a new country
  + Check in the database if the country is added

### Hands-on Spring Data JPA - Comprehensive Guide with Deep Dive and Full Example

## 18. Hands-on 5: Country Management Service with Spring Data JPA

This hands-on guides you through building a Spring Boot-based application for managing countries using Spring Data JPA.

### ✅ Objective

Implement a CRUD-based service to manage country records stored in a database, using Spring Boot, Spring Data JPA, and optionally an in-memory H2 database.

### 📦 Step-by-Step Implementation

#### 1. Create Spring Boot Project

* Use [Spring Initializr](https://start.spring.io/)
* Name: country-service
* Dependencies:
  + Spring Web
  + Spring Data JPA
  + H2 Database (or MySQL/PostgreSQL)
  + Lombok (optional)

#### 2. Entity Class

@Entity

@Table(name = "country")

public class Country {

@Id

private String coCode;

private String coName;

public Country() {}

public Country(String coCode, String coName) {

this.coCode = coCode;

this.coName = coName;

}

// Getters and Setters

}

#### 3. Repository Interface

public interface CountryRepository extends JpaRepository<Country, String> {

List<Country> findByCoNameContainingIgnoreCase(String partialName);

}

#### 4. Service Layer

public interface CountryService {

Country findByCode(String code);

Country addCountry(Country country);

Country updateCountry(Country country);

void deleteCountry(String code);

List<Country> searchCountries(String namePart);

}

#### 5. Service Implementation

@Service

public class CountryServiceImpl implements CountryService {

@Autowired

private CountryRepository countryRepository;

@Override

public Country findByCode(String code) {

return countryRepository.findById(code)

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

}

@Override

public Country addCountry(Country country) {

return countryRepository.save(country);

}

@Override

public Country updateCountry(Country country) {

if (!countryRepository.existsById(country.getCoCode())) {

throw new RuntimeException("Cannot update non-existing country");

}

return countryRepository.save(country);

}

@Override

public void deleteCountry(String code) {

countryRepository.deleteById(code);

}

@Override

public List<Country> searchCountries(String namePart) {

return countryRepository.findByCoNameContainingIgnoreCase(namePart);

}

}

#### 6. Controller Layer

@RestController

@RequestMapping("/countries")

public class CountryController {

@Autowired

private CountryService countryService;

@GetMapping("/{code}")

public Country getCountry(@PathVariable String code) {

return countryService.findByCode(code);

}

@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);

}

@GetMapping("/search")

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

return countryService.searchCountries(name);

}

}

#### 7. application.properties

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

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=

spring.jpa.hibernate.ddl-auto=create

spring.jpa.show-sql=true

spring.h2.console.enabled=true

spring.h2.console.path=/h2-console

Replace with MySQL/PostgreSQL configuration for production.

#### 8. SQL Initialization (optional)

Add data.sql to src/main/resources/:

insert into country(co\_code, co\_name) values('IN','India');

insert into country(co\_code, co\_name) values('US','United States');

insert into country(co\_code, co\_name) values('UK','United Kingdom');

insert into country(co\_code, co\_name) values('JP','Japan');

-- Add more as required

#### 9. API Testing

| **Method** | **Endpoint** | **Description** |
| --- | --- | --- |
| GET | /countries/IN | Get country by code |
| POST | /countries | Add new country |
| PUT | /countries | Update existing country |
| DELETE | /countries/IN | Delete country by code |
| GET | /countries/search?name=land | Search by name part |

## 19. Hands-on 6: Find Country by Country Code with Exception Handling

### 🧩 Objective

Enhance the CountryService to handle the case where a country code does not exist by throwing a custom exception.

### 📂 Step-by-Step

#### 1. Create CountryNotFoundException

package com.cognizant.springlearn.service.exception;

public class CountryNotFoundException extends RuntimeException {

public CountryNotFoundException(String message) {

super(message);

}

}

#### 2. Modify Service Method with @Transactional

@Service

public class CountryServiceImpl implements CountryService {

@Autowired

private CountryRepository countryRepository;

@Override

@Transactional

public Country findByCode(String countryCode) {

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

if (!result.isPresent()) {

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

}

return result.get();

}

@Override

@Transactional

public Country addCountry(Country country) {

return countryRepository.save(country);

}

// other methods remain unchanged

}

📌 @Transactional ensures Spring manages transaction boundaries and the Hibernate session lifecycle automatically.

#### 3. Update CountryService Interface

public interface CountryService {

Country findByCode(String countryCode) throws CountryNotFoundException;

void addCountry(Country country);

// other methods remain unchanged

}

#### 4. Add Test Method in OrmLearnApplication

private static void testAddCountry() {

LOGGER.info("Start");

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

countryService.addCountry(newCountry);

Country country = countryService.findByCode("ZZ");

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

LOGGER.info("End");

}

#### 5. Modify Main Method

public static void main(String[] args) {

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

countryService = context.getBean(CountryService.class);

getAllCountriesTest();

testAddCountry();

}