## ****Objectives****

### ****1. Explain the Need and Benefits of ORM****

**Object-Relational Mapping (ORM)** is a programming technique that simplifies the interaction between object-oriented programming languages (such as Java) and relational databases. ORM tools abstract the SQL operations and allow developers to work with database records as objects.

#### ****Benefits of ORM:****

* Simplifies development by abstracting database access.
* Reduces boilerplate code.
* Automatically handles CRUD operations.
* Manages relationships and transactions.
* Enables database-independence with minimal code changes.

### ****2. Demonstrate the Need and Benefits of Spring Data JPA****

Spring Data JPA builds on top of the Java Persistence API (JPA) and provides a powerful repository abstraction layer. It simplifies data access by reducing the need to write complex boilerplate code and SQL queries.

#### ****Evolution of ORM Tools:****

* **Hibernate (XML Configuration):** Uses XML files to map entities to tables.
* **Hibernate (Annotation Configuration):** Uses annotations for configuration.
* **Spring Data JPA:** Offers repository interfaces to simplify data access.

#### ****Key Benefits:****

* Lightweight and open-source.
* Reduces code complexity and improves maintainability.
* Supports multiple databases seamlessly.
* Eliminates the need for writing most SQL queries manually.

### ****3. Explain Core Objects of the Hibernate Framework****

Hibernate provides several key components to manage database interactions:

* **SessionFactory:** A heavyweight object used to create Session instances.
* **Session:** Represents a single unit of work; manages the persistence of objects.
* **Transaction:** Wraps a series of operations into a single unit of work to ensure data consistency.
* **TransactionFactory:** Produces Transaction instances.
* **ConnectionProvider:** Manages JDBC connections behind the scenes.

These objects work together to handle data operations efficiently and in a standardized way.

### ****4. Explain ORM Implementation using Hibernate (XML and Annotations)****

#### ****Using XML Configuration:****

* Define persistence class (POJO).
* Create mapping XML files to describe object-table relationships.
* Create configuration XML to set up the database connection and Hibernate properties.
* Use SessionFactory to open sessions and interact with the database.

#### ****Using Annotation Configuration:****

* Use annotations like @Entity, @Table, @Id, and @Column in the entity class.
* Define Hibernate configuration XML for database setup.
* Load configuration and perform operations using Hibernate session methods such as:
  + Open session
  + Begin transaction
  + Persist entity
  + Commit transaction
  + Close session

### ****5. Explain the Difference Between JPA, Hibernate, and Spring Data JPA****

| **Feature** | **JPA** | **Hibernate** | **Spring Data JPA** |
| --- | --- | --- | --- |
| Type | Specification (API) | Implementation (Framework) | Abstraction Framework |
| Function | Defines ORM standards | Implements ORM features | Simplifies and extends JPA/Hibernate |
| Boilerplate Code | Required | Moderate | Minimal |
| Query Writing | Developer-defined | Developer-defined | Auto-generated and declarative |
| Vendor Dependency | None | Specific (Hibernate) | Built on top of JPA and Hibernate |

* **JPA** is a specification that defines how ORM should work.
* **Hibernate** is one of the most popular implementations of the JPA specification and adds additional features.
* **Spring Data JPA** is a framework that abstracts JPA/Hibernate and automates repository creation.

### ****6. Demonstrate Implementation of DML using Spring Data JPA****

Spring Data JPA provides built-in support for data manipulation (DML) operations through its repository interface.

#### ****Common Repository Methods:****

* save(entity) – Saves or updates an entity.
* findById(id) – Finds an entity by primary key.
* deleteById(id) – Deletes an entity by primary key.

#### ****Defining Query Methods:****

* Use method names that reflect the query logic (e.g., findByEmail, findByLastName).
* Spring Data JPA automatically parses method names to generate SQL queries.

#### ****Configuration Notes:****

* spring.jpa.show-sql=true – Enables SQL logging in the console.
* spring.jpa.hibernate.ddl-auto=update – Automatically updates the schema based on entities.

This approach allows rapid development with less code while still providing the flexibility to define custom queries if needed.

## ****Hands-on 1****

### ****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 an Eclipse Project using Spring Initializr****

1. Open Spring Initializr.
2. Set the following project details:
   * **Group:** com.cognizant
   * **Artifact Id:** orm-learn
   * **Description:** Demo project for Spring Data JPA and Hibernate
3. Select dependencies:
   * Spring Boot DevTools
   * Spring Data JPA
   * MySQL Driver
4. Click **Generate** and download the project as a zip file.
5. Extract the zip file into your Eclipse workspace root directory.
6. Import the project in Eclipse:
   * Go to File > Import > Maven > Existing Maven Projects
   * Click **Browse**, select the extracted folder, then click **Finish**
7. Create a new schema in MySQL:

sql

CopyEdit

mysql -u root -p

create schema ormlearn;

### ****Configure application.properties****

In src/main/resources/application.properties, add the following:

properties

# 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****

Run the following command from the terminal:

**bash**

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

### ****Add Logging in Main Method****

In OrmLearnApplication.java, add logging:

**java**

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");

}

Run the application and check the logs for the Inside main message.

### ****Walkthrough by SME (Subject Matter Expert)****

1. src/main/java – Folder with application code.
2. src/main/resources – Folder with configuration files like application.properties.
3. src/test/java – Folder containing unit and integration tests.
4. OrmLearnApplication.java – Entry point for the Spring Boot application.
5. @SpringBootApplication – Enables component scan, auto-configuration, and Spring Boot configuration.
6. pom.xml
   * Review Maven dependencies and plugins.
   * Open Dependency Hierarchy tab in Eclipse to see dependency tree.

### ****Country Table Creation****

In MySQL Workbench or client:

**sql**

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');

### ****Create Persistence Class****

**Path:** com.cognizant.ormlearn.model

**Filename:** Country.java

**Java**

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;

// Getters and setters

@Override

public String toString() {

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

}

}

**Notes:**

* @Entity marks the class as a database entity.
* @Table maps it to a specific table.
* @Id marks the primary key.
* @Column maps class fields to table columns.

### ****Create Repository Interface****

**Path:** com.cognizant.ormlearn.repository

**Filename:** CountryRepository.java

**java**

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> {

}

### ****Create Service Class****

**Path:** com.cognizant.ormlearn.service

**Filename:** CountryService.java

**java**

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();

}

}

### ****Test the Application****

In OrmLearnApplication.java, test service logic:

**java**

import com.cognizant.ormlearn.service.CountryService;

import com.cognizant.ormlearn.model.Country;

import org.springframework.context.ApplicationContext;

import java.util.List;

private static CountryService countryService;

private static void testGetAllCountries() {

LOGGER.info("Start");

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

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

LOGGER.info("End");

}

public static void main(String[] args) {

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

countryService = context.getBean(CountryService.class);

testGetAllCountries();

}

Run the application and confirm that the country data is retrieved from the ormlearn database and printed in the logs.

### ****Hands-on 2: Hibernate XML Config Implementation****

**1. Object to Relational Database Mapping in XML**

* Mapping is done using a .hbm.xml file.
* It includes <hibernate-mapping> → <class> → <id> and <property> tags.
* Maps a Java class to a table, and fields to columns.

**Example:**

**xml**

<hibernate-mapping>

<class name="Employee" table="EMPLOYEE">

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

<generator class="native"/>

</id>

<property name="firstName" column="first\_name"/>

<property name="lastName" column="last\_name"/>

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

</class>

</hibernate-mapping>

**2. Explanation of Hibernate Components**

* SessionFactory: Singleton factory to create Session objects. Configured once.
* Session: Represents a single unit of work with DB.
* Transaction: Encapsulates the DB transaction.
* beginTransaction(): Begins a database transaction.
* commit(): Commits the transaction to DB.
* rollback(): Reverts changes if an error occurs.
* session.save(): Saves a new entity to the DB.
* session.createQuery().list(): Executes HQL and retrieves result list.
* session.get(): Fetches a record by primary key.
* session.delete(): Deletes the given object from DB.

### ****Hands-on 3: Hibernate Annotation Config Implementation****

**1. Object to Relational Mapping with Annotations in Entity Class**

**Example – Employee.java**

java

CopyEdit

@Entity

@Table(name = "EMPLOYEE")

public class Employee {

@Id

@GeneratedValue

private int id;

@Column(name = "first\_name")

private String firstName;

@Column(name = "last\_name")

private String lastName;

@Column(name = "salary")

private int salary;

}

**2. Annotations Explained**

* @Entity: Declares the class as an entity mapped to a DB table.
* @Table: Maps class to the specific table.
* @Id: Denotes primary key.
* @GeneratedValue: Auto-generates the ID.
* @Column: Maps field to DB column.

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

xml

CopyEdit

<hibernate-configuration>

<session-factory>

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

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

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

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

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

</session-factory>

</hibernate-configuration>

### ****Hands-on 4: Difference Between JPA, Hibernate, Spring Data JPA****

**Java Persistence API (JPA)**

* Specification defined in JSR 338.
* Abstract layer to manage relational data.
* Requires an implementation like Hibernate.

**Hibernate**

* ORM tool that implements JPA.
* Handles low-level JDBC tasks.
* Offers both native and JPA-based APIs.

**Spring Data JPA**

* Abstraction layer on top of JPA.
* Reduces boilerplate with repositories.
* Handles CRUD and query generation automatically.

**Code Comparison**

Hibernate Example

**java**

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();

} finally {

session.close();

}

return employeeID;

}

Spring Data JPA Example

**java**

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

}

@Service

public class EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

@Transactional

public void addEmployee(Employee employee) {

employeeRepository.save(employee);

}

}

### ****Hands-on 5: Implement Services for Managing Country****

**Service Features Required**

* Find by country code
* Add new country
* Update country
* Delete country
* Find by partial country name

**DDL Auto Configuration**

properties

# Options: create, create-drop, update, validate

spring.jpa.hibernate.ddl-auto=validate

**DDL Auto Options**

* create: Drops and recreates all tables.
* create-drop: Creates at start, drops at end.
* update: Updates schema without deleting data.
* validate: Verifies schema matches mappings.

**Populate Country Table**

* Run SQL script to insert country records.
* Delete existing records before running insert queries.
* Use data population script as provided.

### ****Hands-on 6: Find a Country Based on Country Code****

**1. Create Exception Class**

* Package: com.cognizant.spring-learn.service.exception

**java**

public class CountryNotFoundException extends Exception {

public CountryNotFoundException(String message) {

super(message);

}

}

**2. Add Method in CountryService**

**java**

@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();

}

}

**3. Add Test Method in OrmLearnApplication**

**java**

private static void getAllCountriesTest() {

LOGGER.info("Start");

Country country = null;

try {

country = countryService.findCountryByCode("IN");

} catch (CountryNotFoundException e) {

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

}

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

LOGGER.info("End");

}

**4. Call Method from main()**

**java**

getAllCountriesTest();

**Note:**

* @Transactional: Ensures that the Hibernate session and transaction are automatically managed by Spring when the service method is executed.

### ****Hands-on 7: Add a New Country****

**1. Add Method in CountryService**

**java**

@Transactional

public void addCountry(Country country) {

countryRepository.save(country);

}

**2. Add Test Method in OrmLearnApplication**

**java**

private static void testAddCountry() {

LOGGER.info("Start");

Country newCountry = new Country();

newCountry.setCode("JP");

newCountry.setName("Japan");

countryService.addCountry(newCountry);

try {

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

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

} catch (CountryNotFoundException e) {

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

}

LOGGER.info("End");

}

**3. Call Method from main()**

java

CopyEdit

testAddCountry();

**4. Verify in DB**

* Check if JP exists with name Japan.

### ****Hands-on 8: Update a Country Based on Code****

**1. Add Method in CountryService**

**java**

@Transactional

public void updateCountry(String code, String name) throws CountryNotFoundException {

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

if (!result.isPresent()) {

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

}

Country country = result.get();

country.setName(name);

countryRepository.save(country);

}

**2. Add Test Method in OrmLearnApplication**

**java**

private static void testUpdateCountry() {

LOGGER.info("Start");

try {

countryService.updateCountry("JP", "Nippon");

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

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

} catch (CountryNotFoundException e) {

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

}

LOGGER.info("End");

}

**3. Call Method from main()**

java

CopyEdit

testUpdateCountry();

**4. Verify in DB**

* Check if JP name is updated to Nippon.

### ****Hands-on 9: Delete a Country Based on Code****

**1. Add Method in CountryService**

**java**

@Transactional

public void deleteCountry(String code) {

countryRepository.deleteById(code);

}

**2. Add Test Method in OrmLearnApplication**

**java**

private static void testDeleteCountry() {

LOGGER.info("Start");

countryService.deleteCountry("JP");

try {

countryService.findCountryByCode("JP");

} catch (CountryNotFoundException e) {

LOGGER.error("Country Not Found After Deletion: {}", e.getMessage());

}

LOGGER.info("End");

}

**3. Call Method from main()**

**Java**

testDeleteCountry();

**4. Verify in DB**

* Ensure that country with code JP is no longer present.