# Spring Data JPA with Spring Boot, Hibernate

## 1. ORM (Object-Relational Mapping) and Its Benefits

ORM is a programming technique that allows developers to map Java objects to relational database tables. It abstracts the complexity of database access, enabling developers to perform CRUD operations without writing SQL directly.

Benefits of ORM:

* • Eliminates boilerplate JDBC code
* • Provides database independence
* • Handles transaction management automatically
* • Improves productivity and maintainability

## 2. Spring Data JPA – Need and Benefits

Spring Data JPA is a part of the Spring Data project. It provides an abstraction layer over JPA providers like Hibernate. It reduces boilerplate code by generating repository implementations at runtime.

Key Benefits:

* • Simplifies CRUD implementation
* • Supports derived query methods
* • Integrates with Spring Boot for rapid development

## 3. Core Objects of Hibernate Framework

* • SessionFactory – Creates and manages Session objects
* • Session – Interface to perform CRUD operations
* • Transaction – Manages unit of work
* • Query – Allows HQL/SQL execution

## 4. Hands On – Country Service with Spring Data JPA

Step 1: Create a Spring Boot project using Spring Initializr with dependencies:

* • Spring Web
* • Spring Data JPA
* • MySQL Driver

Step 2: Configure application.properties:

spring.datasource.url=jdbc:mysql://localhost:3306/ormlearn  
spring.datasource.username=root  
spring.datasource.password=root  
spring.jpa.hibernate.ddl-auto=validate  
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect  
logging.level.org.hibernate.SQL=trace  
logging.level.org.hibernate.type.descriptor.sql=trace

Step 3: Create Country Entity:

package com.cognizant.ormlearn.model;  
  
import jakarta.persistence.\*;  
  
@Entity  
@Table(name="country")  
public class Country {  
 @Id  
 @Column(name="co\_code")  
 private String code;  
  
 @Column(name="co\_name")  
 private String name;  
  
 // Getters and setters  
}

Step 4: Create CountryRepository:

package com.cognizant.ormlearn.repository;  
  
import org.springframework.data.jpa.repository.JpaRepository;  
import com.cognizant.ormlearn.model.Country;  
  
public interface CountryRepository extends JpaRepository<Country, String> {  
}

Step 5: Create CountryService with @Transactional methods for CRUD operations.

package com.cognizant.ormlearn.service;  
  
import org.springframework.stereotype.Service;  
import org.springframework.transaction.annotation.Transactional;  
import com.cognizant.ormlearn.model.Country;  
import com.cognizant.ormlearn.repository.CountryRepository;  
import java.util.\*;  
  
@Service  
public class CountryService {  
 private final CountryRepository repo;  
  
 public CountryService(CountryRepository repo) { this.repo = repo; }  
  
 @Transactional  
 public List<Country> getAllCountries() {  
 return repo.findAll();  
 }  
  
 @Transactional  
 public Country findCountryByCode(String code) {  
 return repo.findById(code).orElseThrow();  
 }  
  
 @Transactional  
 public void addCountry(Country country) {  
 repo.save(country);  
 }  
  
 @Transactional  
 public void updateCountry(String code, String name) {  
 Country c = repo.findById(code).orElseThrow();  
 c.setName(name);  
 repo.save(c);  
 }  
  
 @Transactional  
 public void deleteCountry(String code) {  
 repo.deleteById(code);  
 }  
}

Step 6: Main Application to Test Methods:

package com.cognizant.ormlearn;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.context.ApplicationContext;  
import com.cognizant.ormlearn.service.CountryService;  
import com.cognizant.ormlearn.model.Country;  
import java.util.\*;  
  
@SpringBootApplication  
public class OrmLearnApplication {  
 public static void main(String[] args) {  
 ApplicationContext ctx = SpringApplication.run(OrmLearnApplication.class, args);  
 CountryService service = ctx.getBean(CountryService.class);  
  
 System.out.println(service.getAllCountries());  
 }  
}

# 5. Spring Data JPA Query Methods and ORM Relationships

## Objectives

• Demonstrate implementation of Query Methods feature of Spring Data JPA

• Demonstrate implementation of Object/Relational Mapping with annotations like @ManyToOne, @OneToMany, @ManyToMany, FetchType, and mappedBy

## Hands On 1: Write queries on country table using Query Methods

Example methods in CountryRepository:

List<Country> findByNameContaining(String text);  
List<Country> findByNameContainingOrderByNameAsc(String text);  
List<Country> findByNameStartingWith(String prefix);

Testing in OrmLearnApplication:

private static void testSearchCountries() {  
 LOGGER.info("Start");  
 List<Country> list = countryRepository.findByNameContaining("ou");  
 LOGGER.debug("Countries:{}", list);  
 LOGGER.info("End");  
}

## Hands On 2: Write queries on stock table using Query Methods

SQL table creation:

CREATE TABLE stock (  
 st\_id INT AUTO\_INCREMENT PRIMARY KEY,  
 st\_code VARCHAR(10),  
 st\_date DATE,  
 st\_open NUMERIC(10,2),  
 st\_close NUMERIC(10,2),  
 st\_volume NUMERIC  
);

StockRepository methods:

List<Stock> findByCodeAndDateBetween(String code, LocalDate start, LocalDate end);  
List<Stock> findByCodeAndCloseGreaterThan(String code, BigDecimal price);  
List<Stock> findTop3ByOrderByVolumeDesc();  
List<Stock> findTop3ByCodeOrderByCloseAsc(String code);

## Hands On 3: Create payroll tables and bean mapping

Entities: Employee, Department, Skill

* • Each entity annotated with @Entity and @Table  
  • Primary key with @Id and @GeneratedValue(strategy = GenerationType.IDENTITY)  
  • Fields with @Column  
  • Getters, setters, and toString()

## Hands On 4: Implement many-to-one relationship between Employee and Department

Employee.java:

@ManyToOne  
@JoinColumn(name = "em\_dp\_id")  
private Department department;

## Hands On 5: Implement one-to-many relationship between Employee and Department

Department.java:

@OneToMany(mappedBy = "department", fetch = FetchType.EAGER)  
private Set<Employee> employeeList;

## Hands On 6: Implement many-to-many relationship between Employee and Skill

Employee.java:

@ManyToMany(fetch = FetchType.EAGER)  
@JoinTable(name = "employee\_skill",  
 joinColumns = @JoinColumn(name = "es\_em\_id"),  
 inverseJoinColumns = @JoinColumn(name = "es\_sk\_id"))  
private Set<Skill> skillList;

Skill.java:

@ManyToMany(mappedBy = "skillList")  
private Set<Employee> employeeList;

# 6. HQL, JPQL, Native Query and Criteria Query

## Objectives

• Demonstrate writing Hibernate Query Language and Native Query

o HQL stands for Hibernate Query Language, JPQL stands for Java Persistence Query Language, Compare HQL and JPQL, @Query annotation, HQL fetch keyword, aggregate functions in HQL, Native Query, nativeQuery attribute

Reference - https://docs.jboss.org/hibernate/orm/4.3/devguide/en-US/html/ch11.html

Features of JPA Query - https://www.baeldung.com/spring-data-jpa-query

• Explain the need and benefit of Criteria Query

o Scenarios where Criteria Query helps, CriteriaBuilder, Criteria Query, Root, TypedQuery

Reference - https://docs.oracle.com/javaee/6/tutorial/doc/gjrij.html

## Hands On 1: Introduction to HQL and JPQL

• HQL stands for Hibernate Query Language

• JPQL stands for Java Persistence Query Language

• Both HQL and JPQL are object focused query language similar to SQL

• JPQL is a subset of HQL

• All JPQL queries are valid HQL query, but the reverse is not true

• Both HQL and JPQL allows SELECT, UPDATE and DELETE

• HQL additionally allows INSERT statement

Reference - https://docs.jboss.org/hibernate/orm/4.3/devguide/en-US/html/ch11.html

## Hands On 2: Get all permanent employees using HQL

Using HQL get all permanent employees. Retrieve respective department and skill list as well.

@Query(value="SELECT e FROM Employee e WHERE e.permanent = 1")  
List<Employee> getAllPermanentEmployees();

Service method and test method in OrmLearnApplication:

public static void testGetAllPermanentEmployees() {  
 LOGGER.info("Start");  
 List<Employee> employees = employeeService.getAllPermanentEmployees();  
 LOGGER.debug("Permanent Employees:{}", employees);  
 employees.forEach(e -> LOGGER.debug("Skills:{}", e.getSkillList()));  
 LOGGER.info("End");  
}

Optimizing by removing EAGER fetch and using fetch keyword in HQL:

@Query(value="SELECT e FROM Employee e left join fetch e.department d left join fetch e.skillList WHERE e.permanent = 1")

IMPORTANT: 'join' links the table, but 'fetch' populates the beans.

## Hands On 3: Fetch quiz attempt details using HQL

Scenario: Fetch username, attempted date, all questions, list of options, correct answer flag, and score for correct answer from quiz attempt schema.

Steps: Create entities for user, question, options, attempt, attempt\_question, attempt\_option.  
Define O/R mapping.  
Repository method: public Attempt getAttempt(int userId, int attemptId)  
Use HQL joins with fetch for one-to-many and many-to-many relations.  
Display output in quiz format.

## Hands On 4: Get average salary using HQL

@Query(value="SELECT AVG(e.salary) FROM Employee e where e.department.id = :id")  
double getAverageSalary(@Param("id") int id);

Notes: Use aggregate functions like AVG(), COUNT(), SUM(), MIN(), MAX(). @Param binds method parameters to query parameters.

## Hands On 5: Get all employees using Native Query

@Query(value="SELECT \* FROM employee", nativeQuery = true)  
List<Employee> getAllEmployeesNative();

Native queries use direct SQL; prefer minimal use for portability.

## Hands On 6: Criteria Query

Scenario: Dynamic filtering like Amazon product search.  
Criteria Query allows programmatic construction of queries with dynamic where clauses.

Example:

CriteriaBuilder cb = entityManager.getCriteriaBuilder();  
CriteriaQuery<Product> cq = cb.createQuery(Product.class);  
Root<Product> product = cq.from(Product.class);  
Predicate ramPredicate = cb.equal(product.get("ram"), 8);  
cq.where(ramPredicate);  
TypedQuery<Product> query = entityManager.createQuery(cq);  
List<Product> results = query.getResultList();

**Exercise 1: Project Setup**

**Application Structure:**

text

src/main/java/com/example/ems/

EmployeeManagementSystemApplication.java

entity/

Employee.java

Department.java

repository/

EmployeeRepository.java

DepartmentRepository.java

controller/

EmployeeController.java

DepartmentController.java

src/main/resources/application.properties

**File:** src/main/resources/application.properties

text

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

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=password

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.jpa.hibernate.ddl-auto=update

spring.h2.console.enabled=true

spring.jpa.show-sql=true

**Exercise 2: Creating Entities**

**File:** entity/Department.java

java

**package** com.example.ems.entity;

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

**import** lombok.Data;

**import** java.util.List;

@Entity

@Table(name = "departments")

@Data

**public** **class** Department {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

**private** Long id;

**private** String name;

@OneToMany(mappedBy = "department")

**private** List<Employee> employees;

}

**File:** entity/Employee.java

java

**package** com.example.ems.entity;

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

**import** lombok.Data;

@Entity

@Table(name = "employees")

@Data

**public** **class** Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

**private** Long id;

**private** String name;

**private** String email;

@ManyToOne

@JoinColumn(name = "department\_id")

**private** Department department;

}

**Exercise 3: Creating Repositories**

**File:** repository/DepartmentRepository.java

java

**package** com.example.ems.repository;

**import** com.example.ems.entity.Department;

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

**public** **interface** DepartmentRepository **extends** JpaRepository<Department, Long> {

Department findByName(String name);

}

**File:** repository/EmployeeRepository.java

java

**package** com.example.ems.repository;

**import** com.example.ems.entity.Employee;

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

**import** java.util.List;

**public** **interface** EmployeeRepository **extends** JpaRepository<Employee, Long> {

List<Employee> findByName(String name);

List<Employee> findByDepartmentId(Long departmentId);

}

**Exercise 4: Implementing CRUD Operations**

**File:** controller/DepartmentController.java

java

**package** com.example.ems.controller;

**import** com.example.ems.entity.Department;

**import** com.example.ems.repository.DepartmentRepository;

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

**import** java.util.List;

@RestController

@RequestMapping("/departments")

**public** **class** DepartmentController {

**private** **final** DepartmentRepository repo;

**public** DepartmentController(DepartmentRepository repo) { **this**.repo = repo; }

@GetMapping

**public** List<Department> getAll() { **return** repo.findAll(); }

@PostMapping

**public** Department create(@RequestBody Department department) { **return** repo.save(department); }

@PutMapping("/{id}")

**public** Department update(@PathVariable Long id, @RequestBody Department department) {

department.setId(id);

**return** repo.save(department);

}

@DeleteMapping("/{id}")

**public** **void** delete(@PathVariable Long id) { repo.deleteById(id); }

}

**File:** controller/EmployeeController.java

java

**package** com.example.ems.controller;

**import** com.example.ems.entity.Employee;

**import** com.example.ems.repository.EmployeeRepository;

**import** org.springframework.data.domain.Page;

**import** org.springframework.data.domain.Pageable;

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

**import** java.util.List;

@RestController

@RequestMapping("/employees")

**public** **class** EmployeeController {

**private** **final** EmployeeRepository repo;

**public** EmployeeController(EmployeeRepository repo) { **this**.repo = repo; }

@GetMapping

**public** List<Employee> getAll() { **return** repo.findAll(); }

@GetMapping("/page")

**public** Page<Employee> getPage(Pageable pageable) {

**return** repo.findAll(pageable);

}

@PostMapping

**public** Employee create(@RequestBody Employee employee) { **return** repo.save(employee); }

@PutMapping("/{id}")

**public** Employee update(@PathVariable Long id, @RequestBody Employee employee) {

employee.setId(id);

**return** repo.save(employee);

}

@DeleteMapping("/{id}")

**public** **void** delete(@PathVariable Long id) { repo.deleteById(id); }

}

**Exercise 5: Defining Query Methods**

Inside EmployeeRepository.java:

java

List<Employee> findByNameContainingIgnoreCase(String name);

List<Employee> findByDepartment\_Name(String departmentName);

@Query("SELECT e FROM Employee e WHERE e.email = :email")

Employee findByEmail(@Param("email") String email);

**Exercise 6: Pagination and Sorting**

Spring Data JPA supports pagination & sorting directly:

java

Page<Employee> findAll(Pageable pageable);

**Exercise 7: Enabling Entity Auditing**

**File:** entity/Auditable.java

java

**import** org.springframework.data.annotation.\*;

**import** org.springframework.data.jpa.domain.support.AuditingEntityListener;

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

**import** java.util.Date;

@MappedSuperclass

@EntityListeners(AuditingEntityListener.**class**)

**public** **abstract** **class** Auditable {

@CreatedBy

**protected** String createdBy;

@LastModifiedBy

**protected** String lastModifiedBy;

@CreatedDate

@Temporal(TemporalType.TIMESTAMP)

**protected** Date createdDate;

@LastModifiedDate

@Temporal(TemporalType.TIMESTAMP)

**protected** Date lastModifiedDate;

}

**Enable Auditing:**

java

@SpringBootApplication

@EnableJpaAuditing

**public** **class** EmployeeManagementSystemApplication { ... }

**Exercise 8: Creating Projections**

**Interface-based Projection:**

java

**public** **interface** EmployeeNameView {

String getName();

}

List<EmployeeNameView> findAllBy();

**Class-based Projection:**

java

**public** **class** EmployeeSummary {

**private** **final** String name;

**private** **final** String department;

**public** EmployeeSummary(String name, String department) {

**this**.name = name;

**this**.department = department;

}

}

**Exercise 9: Customizing Data Source Configuration**

Example for multiple data sources in application.properties:

text

spring.datasource.primary.url=...

spring.datasource.secondary.url=...

**Exercise 10: Hibernate-Specific Features**

java

@org.hibernate.annotations.BatchSize(size = 10)

@OneToMany(mappedBy = "department")

**private** List<Employee> employees;

**Main Application**

**File:** EmployeeManagementSystemApplication.java

java

**package** com.example.ems;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

**import** org.springframework.data.jpa.repository.config.EnableJpaAuditing;

@SpringBootApplication

@EnableJpaAuditing

**public** **class** EmployeeManagementSystemApplication {

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

SpringApplication.run(EmployeeManagementSystemApplication.**class**, args);

}

}

**Output**

* H2 console available at /h2-console.
* Endpoints:
  + /employees for employee CRUD
  + /departments for department CRUD