**JPA vs Hibernate vs Spring Data JPA - Complete Comparison**

**Overview**

This document provides a comprehensive comparison between Java Persistence API (JPA), Hibernate, and Spring Data JPA, explaining their relationships, differences, and practical usage with code examples.

**Table of Contents**

1. [Understanding Each Technology](https://claude.ai/chat/a8329b13-e780-43c2-8673-e763dcb34741#understanding-each-technology)
2. [Architecture and Relationships](https://claude.ai/chat/a8329b13-e780-43c2-8673-e763dcb34741#architecture-and-relationships)
3. [Code Comparison Examples](https://claude.ai/chat/a8329b13-e780-43c2-8673-e763dcb34741#code-comparison-examples)
4. [Detailed Feature Comparison](https://claude.ai/chat/a8329b13-e780-43c2-8673-e763dcb34741#detailed-feature-comparison)
5. [When to Use What](https://claude.ai/chat/a8329b13-e780-43c2-8673-e763dcb34741#when-to-use-what)
6. [Best Practices](https://claude.ai/chat/a8329b13-e780-43c2-8673-e763dcb34741#best-practices)

**Understanding Each Technology**

**Java Persistence API (JPA)**

**JPA** is a **specification** (JSR 338) that defines how to persist, read, and manage data from Java objects to relational databases.

**Key Characteristics:**

* **Specification, not implementation**: JPA is just a set of interfaces and annotations
* **Vendor-neutral**: Provides a standard way to work with ORM tools
* **JSR 338 Standard**: Officially standardized by Java Community Process
* **No concrete implementation**: Requires an implementation provider

**Core JPA Components:**

* **EntityManager**: Interface for database operations
* **Persistence Context**: Environment where entities are managed
* **JPQL**: Java Persistence Query Language
* **Annotations**: @Entity, @Table, @Id, @Column, etc.

// Pure JPA Example

@Entity

@Table(name = "employees")

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Integer id;

@Column(name = "first\_name")

private String firstName;

@Column(name = "last\_name")

private String lastName;

// Getters and setters

}

**Hibernate**

**Hibernate** is a **concrete implementation** of the JPA specification and also provides additional features beyond JPA.

**Key Characteristics:**

* **ORM Framework**: Object-Relational Mapping tool
* **JPA Implementation**: Implements all JPA specifications
* **Extended Features**: Provides features beyond JPA standard
* **Mature Framework**: Been around since 2001
* **Hibernate-specific APIs**: SessionFactory, Session, Query, etc.

**Hibernate Components:**

* **SessionFactory**: Factory for creating Session objects
* **Session**: First-level cache and transaction management
* **Transaction**: Database transaction management
* **HQL**: Hibernate Query Language (superset of JPQL)
* **Criteria API**: Programmatic query building

// Hibernate-specific Example

public class EmployeeDAO {

private SessionFactory sessionFactory;

public void saveEmployee(Employee employee) {

Session session = sessionFactory.openSession();

Transaction tx = null;

try {

tx = session.beginTransaction();

session.save(employee);

tx.commit();

} catch (HibernateException e) {

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

throw e;

} finally {

session.close();

}

}

}

**Spring Data JPA**

**Spring Data JPA** is a **higher-level abstraction** that simplifies JPA usage by reducing boilerplate code.

**Key Characteristics:**

* **Abstraction Layer**: Built on top of JPA implementations
* **No JPA Implementation**: Uses existing JPA providers (like Hibernate)
* **Boilerplate Reduction**: Eliminates repetitive DAO code
* **Convention over Configuration**: Follows naming conventions
* **Automatic Transaction Management**: Handles transactions automatically

**Spring Data JPA Components:**

* **Repository Interfaces**: JpaRepository, CrudRepository, PagingAndSortingRepository
* **Query Methods**: Automatic query generation from method names
* **Custom Queries**: @Query annotation for custom JPQL/SQL
* **Transaction Management**: @Transactional annotation

// Spring Data JPA Example

@Repository

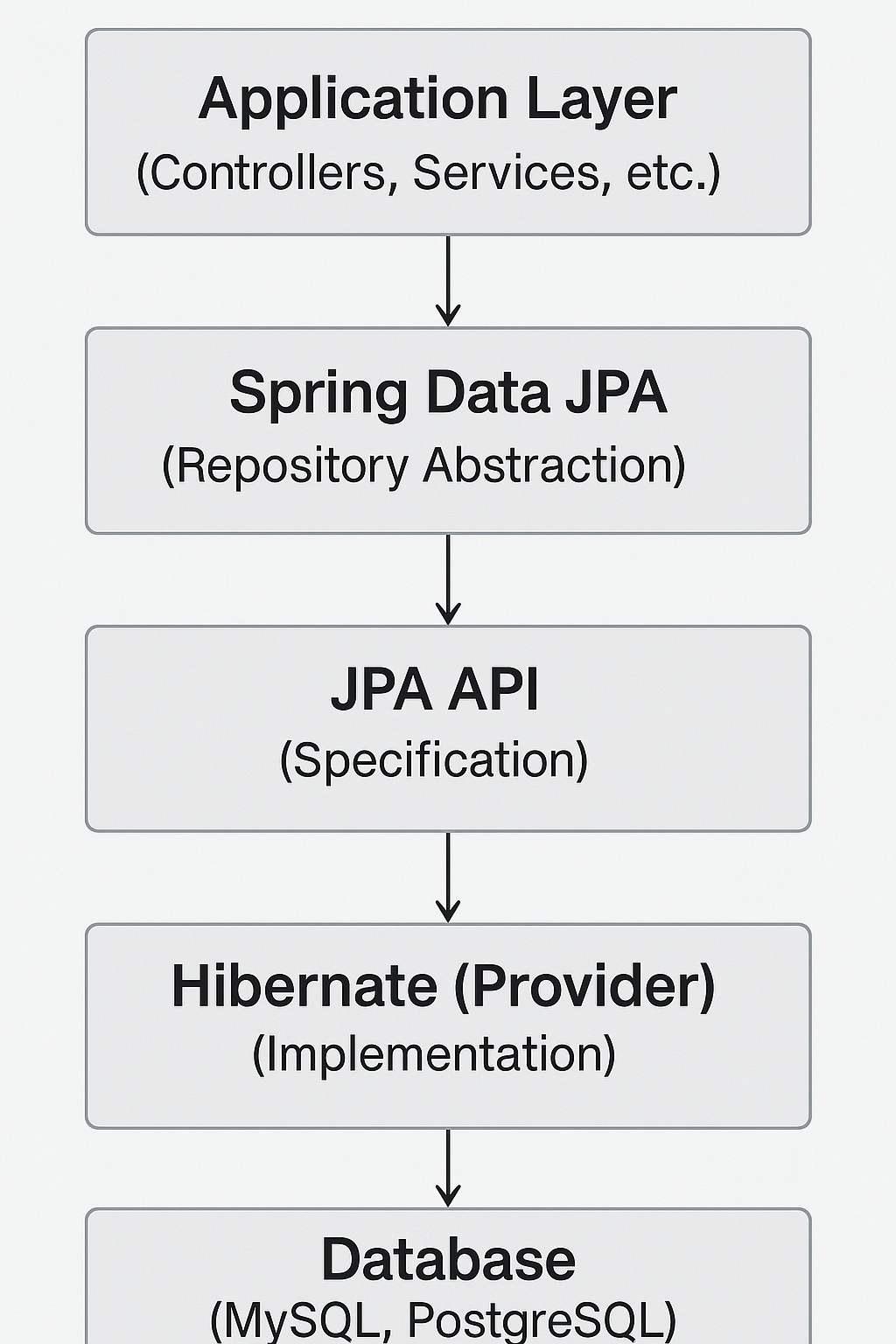
public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

// No implementation needed - Spring generates it automatically

List<Employee> findByFirstName(String firstName);

List<Employee> findByLastNameContaining(String lastName);

}

****

**Relationship Summary:**

* **JPA**: Defines the contract (interfaces and annotations)
* **Hibernate**: Implements the JPA contract + additional features
* **Spring Data JPA**: Provides a higher-level abstraction over JPA

**Code Comparison Examples**

Let's see how the same operations are implemented across all three approaches:

**1. Entity Definition**

All three use the same entity definition since they all follow JPA annotations:

@Entity

@Table(name = "employees")

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Integer id;

@Column(name = "first\_name")

private String firstName;

@Column(name = "last\_name")

private String lastName;

@Column(name = "salary")

private BigDecimal salary;

// Constructors, getters, and setters

public Employee() {}

public Employee(String firstName, String lastName, BigDecimal salary) {

this.firstName = firstName;

this.lastName = lastName;

this.salary = salary;

}

// Getters and setters...

}

**2. CREATE Operation**

**Pure JPA Approach:**

@Repository

public class EmployeeJpaDAO {

@PersistenceContext

private EntityManager entityManager;

@Transactional

public void addEmployee(Employee employee) {

entityManager.persist(employee);

}

}

**Hibernate Approach:**

@Repository

public class EmployeeHibernateDAO {

@Autowired

private SessionFactory sessionFactory;

public Integer addEmployee(Employee employee) {

Session session = sessionFactory.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 Approach:**

// Repository Interface

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

// No implementation needed!

}

// Service Class

@Service

public class EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

@Transactional

public Employee addEmployee(Employee employee) {

return employeeRepository.save(employee);

}

}

**3. READ Operations**

**Pure JPA Approach:**

@Repository

public class EmployeeJpaDAO {

@PersistenceContext

private EntityManager entityManager;

public Employee findById(Integer id) {

return entityManager.find(Employee.class, id);

}

public List<Employee> findAll() {

return entityManager.createQuery("SELECT e FROM Employee e", Employee.class)

.getResultList();

}

public List<Employee> findByFirstName(String firstName) {

return entityManager.createQuery(

"SELECT e FROM Employee e WHERE e.firstName = :firstName", Employee.class)

.setParameter("firstName", firstName)

.getResultList();

}

}

**Hibernate Approach:**

@Repository

public class EmployeeHibernateDAO {

@Autowired

private SessionFactory sessionFactory;

public Employee findById(Integer id) {

Session session = sessionFactory.openSession();

Transaction tx = null;

Employee employee = null;

try {

tx = session.beginTransaction();

employee = session.get(Employee.class, id);

tx.commit();

} catch (HibernateException e) {

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

e.printStackTrace();

} finally {

session.close();

}

return employee;

}

public List<Employee> findAll() {

Session session = sessionFactory.openSession();

Transaction tx = null;

List<Employee> employees = null;

try {

tx = session.beginTransaction();

employees = session.createQuery("FROM Employee", Employee.class).list();

tx.commit();

} catch (HibernateException e) {

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

e.printStackTrace();

} finally {

session.close();

}

return employees;

}

}

**Spring Data JPA Approach:**

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

// Built-in methods (no implementation needed):

// findById(Integer id)

// findAll()

// save(Employee employee)

// deleteById(Integer id)

// Custom query methods (auto-generated):

List<Employee> findByFirstName(String firstName);

List<Employee> findByLastName(String lastName);

List<Employee> findByFirstNameAndLastName(String firstName, String lastName);

List<Employee> findByFirstNameContaining(String firstName);

List<Employee> findBySalaryGreaterThan(BigDecimal salary);

// Custom queries using @Query annotation:

@Query("SELECT e FROM Employee e WHERE e.salary > :minSalary")

List<Employee> findHighSalaryEmployees(@Param("minSalary") BigDecimal minSalary);

@Query(value = "SELECT \* FROM employees WHERE salary BETWEEN ?1 AND ?2",

nativeQuery = true)

List<Employee> findBySalaryRange(BigDecimal minSalary, BigDecimal maxSalary);

}

**4. UPDATE Operations**

**Pure JPA Approach:**

@Transactional

public void updateEmployee(Employee employee) {

entityManager.merge(employee);

}

**Hibernate Approach:**

public void updateEmployee(Employee employee) {

Session session = sessionFactory.openSession();

Transaction tx = null;

try {

tx = session.beginTransaction();

session.update(employee);

tx.commit();

} catch (HibernateException e) {

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

e.printStackTrace();

} finally {

session.close();

}

}

**Spring Data JPA Approach:**

@Transactional

public Employee updateEmployee(Employee employee) {

return employeeRepository.save(employee); // save() handles both insert and update

}

**5. DELETE Operations**

**Pure JPA Approach:**

@Transactional

public void deleteEmployee(Integer id) {

Employee employee = entityManager.find(Employee.class, id);

if (employee != null) {

entityManager.remove(employee);

}

}

**Hibernate Approach:**

public void deleteEmployee(Integer id) {

Session session = sessionFactory.openSession();

Transaction tx = null;

try {

tx = session.beginTransaction();

Employee employee = session.get(Employee.class, id);

if (employee != null) {

session.delete(employee);

}

tx.commit();

} catch (HibernateException e) {

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

e.printStackTrace();

} finally {

session.close();

}

}

**Spring Data JPA Approach:**

@Transactional

public void deleteEmployee(Integer id) {

employeeRepository.deleteById(id);

}

**Detailed Feature Comparison**

| **Feature** | **JPA** | **Hibernate** | **Spring Data JPA** |
| --- | --- | --- | --- |
| **Type** | Specification | Implementation | Abstraction Layer |
| **Boilerplate Code** | High | High | Very Low |
| **Learning Curve** | Medium | High | Low |
| **Transaction Management** | Manual | Manual | Automatic |
| **Query Methods** | Manual JPQL | Manual HQL | Auto-generated |
| **Custom Queries** | @NamedQuery | HQL/SQL | @Query annotation |
| **Caching** | Basic | Advanced | Inherited from JPA provider |
| **Batch Operations** | Limited | Extensive | Limited |
| **Lazy Loading** | Standard | Advanced | Standard |
| **Vendor Independence** | High | Low | High |
| **Performance Tuning** | Limited | Extensive | Limited |
| **Community Support** | Good | Excellent | Good |

**When to Use What**

**Use Pure JPA When:**

* You need **vendor independence** across different JPA providers
* Working with **Java EE** environments
* You want to **minimize dependencies**
* You need **fine-grained control** over persistence operations
* Working with **legacy systems** that require JPA compatibility

**Use Hibernate When:**

* You need **advanced features** beyond JPA specification
* **Performance optimization** is critical
* You require **complex batch operations**
* You need **advanced caching strategies**
* Working with **legacy Hibernate applications**
* You need **custom data types** or **user types**

**Use Spring Data JPA When:**

* You're building **Spring/Spring Boot** applications
* You want to **reduce boilerplate code**
* **Rapid development** is a priority
* You need **automatic transaction management**
* You want **convention-over-configuration** approach
* You're building **CRUD-heavy applications**
* You need **repository pattern** implementation

**Best Practices**

**General Guidelines:**

1. **Choose Based on Requirements**:
   * **Rapid Development**: Spring Data JPA
   * **Advanced Features**: Hibernate
   * **Vendor Independence**: Pure JPA
2. **Combine Approaches**:
3. @Repository
4. public interface EmployeeRepository extends JpaRepository<Employee, Integer> {
5. // Spring Data JPA methods
6. List<Employee> findByFirstName(String firstName);
8. // Custom implementation when needed
9. List<Employee> findEmployeesWithComplexCriteria(EmployeeSearchCriteria criteria);
10. }
11. @Repository
12. public class EmployeeRepositoryImpl {
13. @PersistenceContext
14. private EntityManager entityManager;
16. public List<Employee> findEmployeesWithComplexCriteria(EmployeeSearchCriteria criteria) {
17. // Use pure JPA or Hibernate for complex queries
18. CriteriaBuilder cb = entityManager.getCriteriaBuilder();
19. CriteriaQuery<Employee> query = cb.createQuery(Employee.class);
20. // Complex criteria building...
21. return entityManager.createQuery(query).getResultList();
22. }
23. }
24. **Transaction Management**:
25. @Service
26. @Transactional
27. public class EmployeeService {
28. // All methods are transactional by default
30. @Transactional(readOnly = true)
31. public List<Employee> getAllEmployees() {
32. // Read-only transaction for better performance
33. return employeeRepository.findAll();
34. }
36. @Transactional(rollbackFor = Exception.class)
37. public void saveEmployee(Employee employee) {
38. // Rollback on any exception
39. employeeRepository.save(employee);
40. }
41. }
42. **Performance Optimization**:
43. @Repository
44. public interface EmployeeRepository extends JpaRepository<Employee, Integer> {
45. @Query("SELECT e FROM Employee e JOIN FETCH e.department")
46. List<Employee> findAllWithDepartment();
48. @Query("SELECT e FROM Employee e WHERE e.salary > :salary")
49. Page<Employee> findBySalaryGreaterThan(@Param("salary") BigDecimal salary,
50. Pageable pageable);
51. }

**Configuration Examples:**

**Spring Boot with Spring Data JPA:**

# application.properties

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

spring.datasource.username=root

spring.datasource.password=password

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect

**Pure JPA Configuration:**

@Configuration

@EnableJpaRepositories

public class JpaConfig {

@Bean

public LocalContainerEntityManagerFactoryBean entityManagerFactory() {

LocalContainerEntityManagerFactoryBean em = new LocalContainerEntityManagerFactoryBean();

em.setDataSource(dataSource());

em.setPackagesToScan("com.example.entity");

em.setJpaVendorAdapter(new HibernateJpaVendorAdapter());

return em;

}

@Bean

public PlatformTransactionManager transactionManager() {

JpaTransactionManager transactionManager = new JpaTransactionManager();

transactionManager.setEntityManagerFactory(entityManagerFactory().getObject());

return transactionManager;

}

}

**Conclusion**

* **JPA** provides the foundation and standardization
* **Hibernate** offers powerful implementation with advanced features
* **Spring Data JPA** simplifies development with conventions and automation

Choose the approach that best fits your project requirements, team expertise, and performance needs. In many cases, you can combine approaches within the same application to leverage the strengths of each.

**Hibernate XML Configuration Implementation Walkthrough**

**Overview**

This document provides a comprehensive walkthrough of Hibernate XML configuration and demonstrates the core concepts including SessionFactory, Session, Transaction management, and CRUD operations.

**Table of Contents**

1. [Object-Relational Mapping with XML Configuration](https://claude.ai/chat/a8329b13-e780-43c2-8673-e763dcb34741#object-relational-mapping-with-xml-configuration)
2. [Core Hibernate Components](https://claude.ai/chat/a8329b13-e780-43c2-8673-e763dcb34741#core-hibernate-components)
3. [Transaction Management](https://claude.ai/chat/a8329b13-e780-43c2-8673-e763dcb34741#transaction-management)
4. [CRUD Operations](https://claude.ai/chat/a8329b13-e780-43c2-8673-e763dcb34741#crud-operations)
5. [Complete Example Implementation](https://claude.ai/chat/a8329b13-e780-43c2-8673-e763dcb34741#complete-example-implementation)

**Object-Relational Mapping with XML Configuration**

**1. Hibernate Configuration File (hibernate.cfg.xml)**

The main configuration file that defines database connection properties and mapping files:

<?xml version='1.0' encoding='utf-8'?>

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD//EN"

"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

<!-- Database connection settings -->

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

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

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

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

<!-- JDBC connection pool (use built-in) -->

<property name="connection.pool\_size">10</property>

<!-- SQL dialect -->

<property name="dialect">org.hibernate.dialect.MySQL8Dialect</property>

<!-- Enable Hibernate's automatic session context management -->

<property name="current\_session\_context\_class">thread</property>

<!-- Disable the second-level cache -->

<property name="cache.provider\_class">org.hibernate.cache.internal.NoCacheProvider</property>

<!-- Echo all executed SQL to stdout -->

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

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

<!-- Drop and re-create the database schema on startup -->

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

<!-- Mapping files -->

<mapping resource="com/example/Employee.hbm.xml"/>

<mapping resource="com/example/Department.hbm.xml"/>

</session-factory>

</hibernate-configuration>

**2. Entity Mapping File (Employee.hbm.xml)**

XML mapping file that defines how Java objects map to database tables:

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

<!DOCTYPE hibernate-mapping PUBLIC

"-//Hibernate/Hibernate Mapping DTD//EN"

"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

<class name="com.example.Employee" table="EMPLOYEE">

<!-- Primary Key -->

<id name="id" type="int" column="id">

<generator class="native"/>

</id>

<!-- Properties -->

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

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

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

<!-- Many-to-One Relationship -->

<many-to-one name="department"

class="com.example.Department"

column="dept\_id"

cascade="save-update"/>

</class>

</hibernate-mapping>

**Key Mapping Elements Explained:**

* **<class>**: Maps Java class to database table
* **<id>**: Defines primary key mapping
* **<generator>**: Specifies primary key generation strategy
* **<property>**: Maps Java properties to table columns
* **<many-to-one>**: Defines foreign key relationships
* **cascade**: Defines cascade operations (save, update, delete)

**Core Hibernate Components**

**1. SessionFactory**

The SessionFactory is a heavyweight object that's typically created once per application. It's responsible for creating Session objects.

public class HibernateUtil {

private static final SessionFactory sessionFactory;

static {

try {

// Create the SessionFactory from hibernate.cfg.xml

Configuration configuration = new Configuration();

configuration.configure("hibernate.cfg.xml");

ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()

.applySettings(configuration.getProperties())

.build();

sessionFactory = configuration.buildSessionFactory(serviceRegistry);

} catch (Throwable ex) {

System.err.println("Initial SessionFactory creation failed." + ex);

throw new ExceptionInInitializerError(ex);

}

}

public static SessionFactory getSessionFactory() {

return sessionFactory;

}

public static void shutdown() {

// Close caches and connection pools

getSessionFactory().close();

}

}

**Key Characteristics:**

* Thread-safe and immutable
* Expensive to create
* Should be created once per application
* Contains compiled mapping metadata
* Factory for Session objects

**2. Session**

The Session represents a single unit of work with the database. It's not thread-safe and should be used in a single-threaded manner.

// Getting a Session from SessionFactory

Session session = HibernateUtil.getSessionFactory().openSession();

// Or using getCurrentSession() for thread-local sessions

Session session = HibernateUtil.getSessionFactory().getCurrentSession();

**Key Characteristics:**

* Not thread-safe
* Lightweight object
* Should be closed after use
* Manages persistent objects
* Acts as a first-level cache

**3. Transaction**

Transaction allows the application to define units of work while maintaining abstraction from the underlying transaction implementation.

Transaction transaction = session.beginTransaction();

**Key Characteristics:**

* Represents a unit of work
* Can be committed or rolled back
* Associated with a Session
* Maintains ACID properties

**Transaction Management**

**1. beginTransaction()**

Begins a unit of work and returns the associated Transaction object.

Session session = HibernateUtil.getSessionFactory().openSession();

Transaction transaction = session.beginTransaction();

**Purpose:**

* Starts a new transaction
* Returns Transaction object for management
* Required for write operations

**2. commit()**

Commits the current transaction, making all changes permanent.

try {

// Perform database operations

session.save(employee);

// Commit the transaction

transaction.commit();

} catch (Exception e) {

transaction.rollback();

throw e;

} finally {

session.close();

}

**Purpose:**

* Makes changes permanent in database
* Releases locks
* Ends the transaction

**3. rollback()**

Rolls back the current transaction, discarding all changes.

try {

// Database operations

session.save(employee);

// Some error occurs

if (someErrorCondition) {

transaction.rollback();

return;

}

transaction.commit();

} catch (Exception e) {

if (transaction != null) {

transaction.rollback();

}

throw e;

} finally {

session.close();

}

**Purpose:**

* Discards all changes made in current transaction
* Releases locks
* Restores data to pre-transaction state

**CRUD Operations**

**1. session.save()**

Persists a new object to the database.

public void saveEmployee(Employee employee) {

Session session = HibernateUtil.getSessionFactory().openSession();

Transaction transaction = null;

try {

transaction = session.beginTransaction();

// Save the employee object

Serializable id = session.save(employee);

System.out.println("Employee saved with ID: " + id);

transaction.commit();

} catch (Exception e) {

if (transaction != null) {

transaction.rollback();

}

e.printStackTrace();

} finally {

session.close();

}

}

**Key Points:**

* Assigns generated ID to the object
* Returns the generated primary key
* Object becomes persistent
* Changes are flushed to database on commit

**2. session.get()**

Retrieves an object by its primary key.

public Employee getEmployee(int employeeId) {

Session session = HibernateUtil.getSessionFactory().openSession();

Transaction transaction = null;

Employee employee = null;

try {

transaction = session.beginTransaction();

// Get employee by ID

employee = (Employee) session.get(Employee.class, employeeId);

transaction.commit();

} catch (Exception e) {

if (transaction != null) {

transaction.rollback();

}

e.printStackTrace();

} finally {

session.close();

}

return employee;

}

**Key Points:**

* Returns null if object not found
* Eager loading by default
* Object becomes persistent
* Uses primary key for retrieval

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

Executes HQL queries and returns results as a list.

public List<Employee> getAllEmployees() {

Session session = HibernateUtil.getSessionFactory().openSession();

Transaction transaction = null;

List<Employee> employees = null;

try {

transaction = session.beginTransaction();

// HQL Query

Query query = session.createQuery("FROM Employee");

employees = query.list();

transaction.commit();

} catch (Exception e) {

if (transaction != null) {

transaction.rollback();

}

e.printStackTrace();

} finally {

session.close();

}

return employees;

}

// Query with parameters

public List<Employee> getEmployeesByDepartment(String deptName) {

Session session = HibernateUtil.getSessionFactory().openSession();

Transaction transaction = null;

List<Employee> employees = null;

try {

transaction = session.beginTransaction();

Query query = session.createQuery(

"FROM Employee e WHERE e.department.name = :deptName");

query.setParameter("deptName", deptName);

employees = query.list();

transaction.commit();

} catch (Exception e) {

if (transaction != null) {

transaction.rollback();

}

e.printStackTrace();

} finally {

session.close();

}

return employees;

}

**Key Points:**

* Uses HQL (Hibernate Query Language)
* Returns List of objects
* Supports parameterized queries
* Can handle complex joins and conditions

**4. session.delete()**

Removes an object from the database.

public void deleteEmployee(int employeeId) {

Session session = HibernateUtil.getSessionFactory().openSession();

Transaction transaction = null;

try {

transaction = session.beginTransaction();

// First get the employee

Employee employee = (Employee) session.get(Employee.class, employeeId);

if (employee != null) {

// Delete the employee

session.delete(employee);

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

}

transaction.commit();

} catch (Exception e) {

if (transaction != null) {

transaction.rollback();

}

e.printStackTrace();

} finally {

session.close();

}

}

**Key Points:**

* Requires object to be persistent
* Object must be retrieved first
* Cascading deletes are supported
* Physical deletion from database

**Complete Implementation**

**Employee.java (Entity Class)**

package com.example;

public class Employee {

private int id;

private String firstName;

private String lastName;

private int salary;

private Department department;

// Constructors

public Employee() {}

public Employee(String firstName, String lastName, int salary) {

this.firstName = firstName;

this.lastName = lastName;

this.salary = salary;

}

// Getters and Setters

public int getId() { return id; }

public void setId(int id) { this.id = id; }

public String getFirstName() { return firstName; }

public void setFirstName(String firstName) { this.firstName = firstName; }

public String getLastName() { return lastName; }

public void setLastName(String lastName) { this.lastName = lastName; }

public int getSalary() { return salary; }

public void setSalary(int salary) { this.salary = salary; }

public Department getDepartment() { return department; }

public void setDepartment(Department department) { this.department = department; }

@Override

public String toString() {

return "Employee{" +

"id=" + id +

", firstName='" + firstName + '\'' +

", lastName='" + lastName + '\'' +

", salary=" + salary +

", department=" + department +

'}';

}

}

**EmployeeDAO.java (Data Access Object)**

package com.example;

import org.hibernate.Session;

import org.hibernate.Transaction;

import org.hibernate.query.Query;

import java.util.List;

public class EmployeeDAO {

// Create operation

public void saveEmployee(Employee employee) {

Session session = HibernateUtil.getSessionFactory().openSession();

Transaction transaction = null;

try {

transaction = session.beginTransaction();

session.save(employee);

transaction.commit();

System.out.println("Employee saved successfully");

} catch (Exception e) {

if (transaction != null) {

transaction.rollback();

}

e.printStackTrace();

} finally {

session.close();

}

}

// Read operation

public Employee getEmployee(int employeeId) {

Session session = HibernateUtil.getSessionFactory().openSession();

Transaction transaction = null;

Employee employee = null;

try {

transaction = session.beginTransaction();

employee = session.get(Employee.class, employeeId);

transaction.commit();

} catch (Exception e) {

if (transaction != null) {

transaction.rollback();

}

e.printStackTrace();

} finally {

session.close();

}

return employee;

}

// Update operation

public void updateEmployee(Employee employee) {

Session session = HibernateUtil.getSessionFactory().openSession();

Transaction transaction = null;

try {

transaction = session.beginTransaction();

session.update(employee);

transaction.commit();

System.out.println("Employee updated successfully");

} catch (Exception e) {

if (transaction != null) {

transaction.rollback();

}

e.printStackTrace();

} finally {

session.close();

}

}

// Delete operation

public void deleteEmployee(int employeeId) {

Session session = HibernateUtil.getSessionFactory().openSession();

Transaction transaction = null;

try {

transaction = session.beginTransaction();

Employee employee = session.get(Employee.class, employeeId);

if (employee != null) {

session.delete(employee);

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

}

transaction.commit();

} catch (Exception e) {

if (transaction != null) {

transaction.rollback();

}

e.printStackTrace();

} finally {

session.close();

}

}

// List all employees

public List<Employee> getAllEmployees() {

Session session = HibernateUtil.getSessionFactory().openSession();

Transaction transaction = null;

List<Employee> employees = null;

try {

transaction = session.beginTransaction();

Query<Employee> query = session.createQuery("FROM Employee", Employee.class);

employees = query.list();

transaction.commit();

} catch (Exception e) {

if (transaction != null) {

transaction.rollback();

}

e.printStackTrace();

} finally {

session.close();

}

return employees;

}

}

**Main Application (Testing)**

package com.example;

import java.util.List;

public class MainApp {

public static void main(String[] args) {

EmployeeDAO employeeDAO = new EmployeeDAO();

// Create employees

Employee emp1 = new Employee("John", "Doe", 50000);

Employee emp2 = new Employee("Jane", "Smith", 60000);

// Save employees

employeeDAO.saveEmployee(emp1);

employeeDAO.saveEmployee(emp2);

// Retrieve all employees

List<Employee> employees = employeeDAO.getAllEmployees();

System.out.println("All Employees:");

for (Employee emp : employees) {

System.out.println(emp);

}

// Get specific employee

Employee retrievedEmp = employeeDAO.getEmployee(1);

System.out.println("Retrieved Employee: " + retrievedEmp);

// Update employee

if (retrievedEmp != null) {

retrievedEmp.setSalary(55000);

employeeDAO.updateEmployee(retrievedEmp);

}

// Delete employee

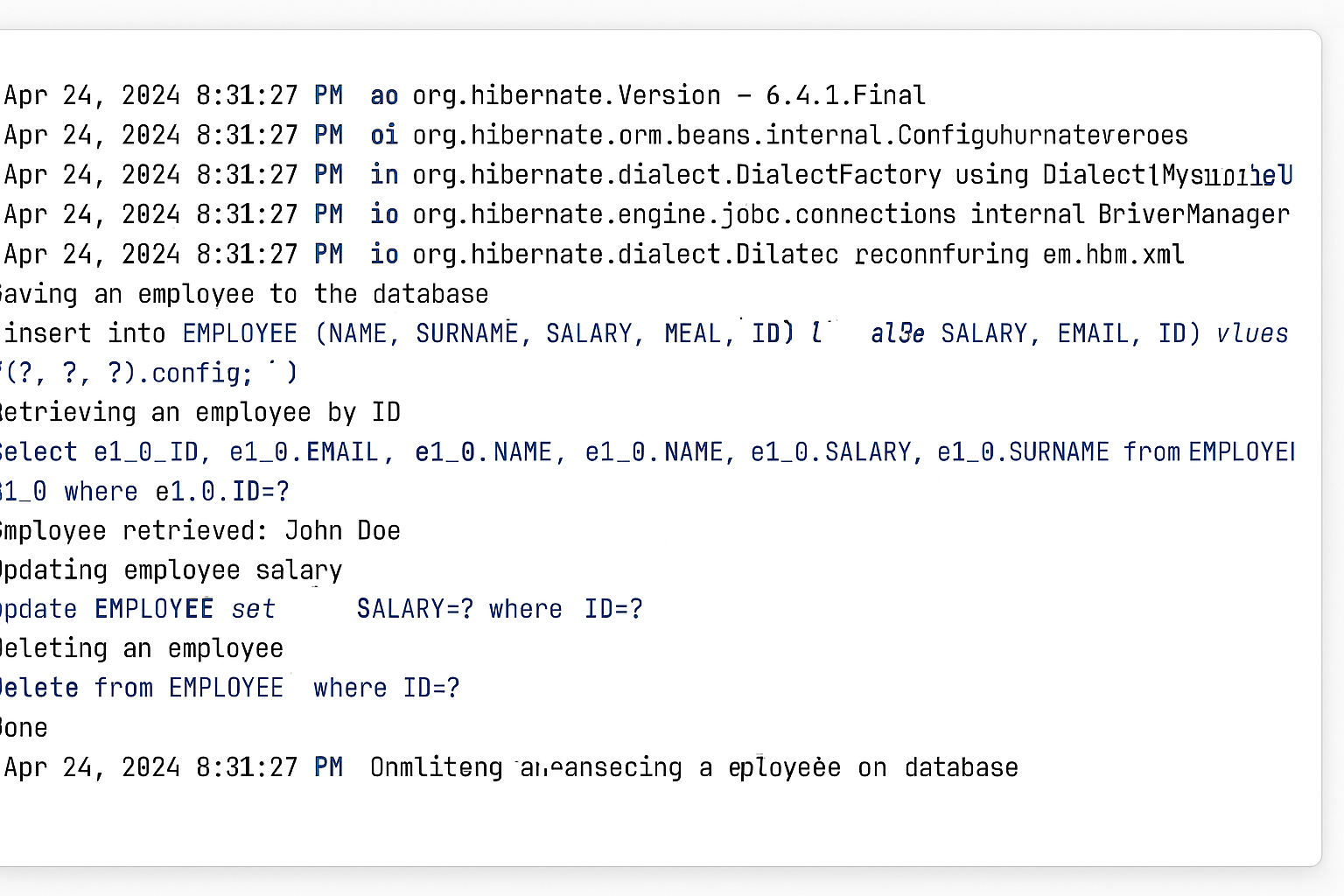
employeeDAO.deleteEmployee(2);

// Close SessionFactory

HibernateUtil.shutdown();

}

}

****

**Spring Data JPA - Quick Start Guide**

**Overview**

This hands-on tutorial demonstrates how to create a Spring Boot application with Spring Data JPA and Hibernate for database operations using MySQL.

**Software Prerequisites**

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

**Project Setup**

**1. Create Eclipse Project using Spring Initializr**

1. Navigate to https://start.spring.io/
2. Configure project settings:
   * **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 zip
5. Extract the zip to your Eclipse Workspace
6. Import the project in Eclipse:
   * File → Import → Maven → Existing Maven Projects
   * Browse and select extracted folder → Finish

**2. Database Schema Setup**

Create a new schema in MySQL database:

-- Connect to MySQL

mysql -u root -p

-- Create schema

mysql> create schema ormlearn;

**3. Application Configuration**

Open src/main/resources/application.properties and add the following 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

**4. Build the Project**

Execute the following Maven command:

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

**5. Add Logging to Main Method**

Update OrmLearnApplication.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");

}

**Project Structure Overview**

* **src/main/java**: Application source code
* **src/main/resources**: Application configuration files
* **src/test/java**: Test code
* **OrmLearnApplication.java**: Main application class with @SpringBootApplication
* **pom.xml**: Maven configuration with dependency management

**Database Table Creation**

Create the country table with sample data:

-- Create table

CREATE TABLE country(

co\_code VARCHAR(2) PRIMARY KEY,

co\_name VARCHAR(50)

);

-- Insert sample records

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

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

**Implementation**

**1. Entity Class - Country.java**

Create package: com.cognizant.ormlearn.model

package 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;

// Constructors

public Country() {}

public Country(String code, String name) {

this.code = code;

this.name = name;

}

// Getters and Setters

public String getCode() {

return code;

}

public void setCode(String code) {

this.code = code;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

// toString method

@Override

public String toString() {

return "Country{" +

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

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

'}';

}

}

**Key Annotations:**

* @Entity: Marks the class as a JPA entity
* @Table: Maps the entity to a database table
* @Id: Defines the primary key
* @Column: Maps fields to table columns

**2. Repository Interface - CountryRepository.java**

Create package: com.cognizant.ormlearn.repository

package 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> {

// JpaRepository provides basic CRUD operations

// Additional custom methods can be defined here

}

**3. Service Class - CountryService.java**

Create package: com.cognizant.ormlearn.service

package com.cognizant.ormlearn.service;

import java.util.List;

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

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import com.cognizant.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();

}

}

**4. Testing in Main Application**

Update OrmLearnApplication.java:

package com.cognizant.ormlearn;

import java.util.List;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

import com.cognizant.ormlearn.model.Country;

import com.cognizant.ormlearn.service.CountryService;

@SpringBootApplication

public class OrmLearnApplication {

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

private static CountryService countryService;

public static void main(String[] args) {

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

countryService = context.getBean(CountryService.class);

LOGGER.info("Inside main");

testGetAllCountries();

}

private static void testGetAllCountries() {

LOGGER.info("Start");

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

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

LOGGER.info("End");

}

}

**Running the Application**

1. Ensure MySQL server is running
2. Verify that the ormlearn schema exists and contains the country table with sample data
3. Run the OrmLearnApplication class
4. Check the console logs to verify:
   * Database connection is established
   * SQL queries are executed (due to Hibernate logging)
   * Country data is retrieved and displayed

**Output**

