Hibernate

Introduction

What is ORM?

Object Relational Mapping (ORM) is a technique that helps developers interact with databases using objects rather than writing manual SQL queries. With ORM, Java classes map to database tables and class fields map to table columns. Instead of manually writing SQL like INSERT INTO student ..., you work with objects and let the framework generate the SQL. This reduces boilerplate code, increases productivity, and makes applications portable across databases.

A simple example without ORM using JDBC looks like this:

```
PreparedStatement ps = con.prepareStatement("INSERT INTO student (id, name, age) VALUES (?, ?, ?)");
ps.setInt(1, student.getId());
ps.setString(2, student.getName());
ps.setInt(3, student.getAge());
ps.executeUpdate();
```

Here, the developer must handle SQL queries and object-field mapping manually.

With ORM, the same operation is simpler.

```
Session session = sessionFactory.openSession();
session.beginTransaction();
session.save(student);
session.getTransaction().commit();
session.close();
```

In this case, saving the object is enough. ORM takes care of generating the SQL and persisting it in the database.

Introduction to Hibernate

Hibernate is a widely used Java ORM framework. It automates the mapping between Java classes and database tables. Hibernate eliminates repetitive JDBC code, supports transactions, provides caching, and allows database-independent applications. It also implements the JPA (Java Persistence API) specification, which makes it a standard solution in enterprise applications.

Features and Advantages of Hibernate

- Removes boilerplate JDBC code.
- Automatically maps Java classes to database tables.
- Works with multiple databases such as MySQL, Oracle, PostgreSQL, and H2 without code changes.
- Provides HQL (Hibernate Query Language), an object-oriented query language similar to SQL.

- Supports caching mechanisms for performance improvements.
- Manages relationships like One-to-One, One-to-Many, and Many-to-Many.
- Provides built-in transaction management and concurrency control.
- Portable, flexible, and widely used in industry.

Hibernate vs JDBC

Aspect	**JDBC (Traditional)**	**Hibernate (ORM Framework)**
Code	Requires manual SQL and boilerplate handling	Works directly with objects, minimal code
Portability	SQL is database-specific	Database-independent
Mappings	Manual handling of tables and objects	Automatic object-to-table mapping
Caching	No support	First-level and second-level caching
Transactions	Must be handled manually	Built-in transaction support
Queries	SQL only	SQL, HQL, and Criteria API

Hibernate Setup

Hibernate Architecture

Hibernate architecture is built on several layers that handle persistence, transactions, and communication with the database.

Main Components:

- **Configuration**: Reads settings from XML or annotations to build SessionFactory.
- **SessionFactory**: A heavyweight object created once per application, used to create Session objects.
- **Session**: Represents a single unit of work with the database. It is lightweight and not thread-safe.
- **Transaction**: Ensures that a set of operations execute atomically.
- **Query**: Used to fetch data using HQL, SQL, or Criteria API.
- **Persistent Objects**: Java classes mapped to database tables.

Flow: Application → Configuration → SessionFactory → Session → Transaction → Database.

Hibernate Configuration (XML & Annotations)

Hibernate needs configuration for database connection and mapping.

XML-based configuration (hibernate.cfg.xml):

```
<hibernate-configuration>
  <session-factory>
```

```
    <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/testdb</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">root</property>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>
    <mapping class="com.example.Student"/>
  </session-factory>
</hibernate-configuration>
```

Annotation-based mapping:

```
@Entity
@Table(name = "student")
public class Student {
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private int id;
   private String name;
   private int age;
}
```

Creating First Hibernate Example

Step 1: Entity class

```
@Entity
@Table(name="student")
public class Student {
   @Id
   private int id;
   private String name;
   private int age;
   // getters and setters
}
```

Step 2: Main class

```
public class MainApp {
   public static void main(String[] args) {
      Configuration cfg = new Configuration().configure("hibernate.cfg.xml");
      SessionFactory factory = cfg.buildSessionFactory();
      Session session = factory.openSession();
      Student st = new Student();
      st.setId(1);
      st.setName("John");
      st.setAge(22);
      Transaction tx = session.beginTransaction();
      session.save(st);
      tx.commit();
      session.close();
      factory.close();
   }
}
```

Running this will insert a record into the `student` table without writing explicit SQL.

Hibernate Generator Classes (ID Strategies)

Hibernate provides strategies for automatically generating primary key values.

- **increment**: Increments the ID by one.
- **identity**: Uses database identity column (auto-increment).
- **sequence**: Uses database sequence object (supported in Oracle, PostgreSQL).
- **hilo**: Uses a high/low algorithm for ID generation.
- **native**: Lets Hibernate pick the best strategy based on database dialect.

Example using annotation:

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private int id;
```

Example using XML:

```
<id name="id">
   <generator class="increment"/>
</id>
```

Basic Mappings

Entity Class & Table Mapping

In Hibernate, a Java class can be mapped to a database table. Each instance of the class corresponds to a row in the table.

For mapping, you use `@Entity` and `@Table`.

```
import jakarta.persistence.*;

@Entity
@Table(name="student")
public class Student {
   @Id
   private int id;
   private String name;
   private int age;
}
```

Here:

- `@Entity` → Marks the class as persistent.
- `@Table(name="student")` → Specifies the table name in the database. If not provided, Hibernate uses the class name by default.

Field to Column Mapping

By default, Hibernate maps class fields to table columns with the same name. You can customize using @Column.

```
@Entity
@Table(name="student")
public class Student {
   @Id
   private int id;
   @Column(name="student_name", length=100, nullable=false)
   private String name;
  @Column(name="student_age")
   private int age;
}
```

Here:

- @Column(name="student_name") → Maps name field to student_name column.
- Attributes like nullable, unique, length allow fine control.

Data Types in Hibernate

Hibernate automatically maps Java data types to SQL types based on the dialect.

Common Mappings:

Java Type	**Hibernate/SQL Equivalent**
int, Integer	INTEGER
long, Long	BIGINT
String	VARCHAR
boolean, Boolean	BIT / BOOLEAN
float, Float	FLOAT
double, Double	DOUBLE
java.util.Date	DATE / TIMESTAMP
java.sql.Date	DATE
java.time.LocalDate	DATE
java.time.LocalDateTime	TIMESTAMP
byte[]	BLOB
char, Character	CHAR

Example with different data types:

```
@Entity
@Table(name="employee")
public class Employee {
   @Id
   private int id;

   private String name;

   private double salary;

   private boolean active;

   private LocalDate joiningDate;
}
```

When persisted, Hibernate converts these fields to appropriate SQL types.

Association Mappings (Basics)

In real-world applications, entities are often related. Hibernate allows you to define relationships between entities just like in relational databases. These relationships can be represented as **One-to-One, One-to-Many, and Many-to-Many**.

One-to-One Mapping

A **one-to-one** relationship means one entity instance is associated with exactly one other entity instance.

Annotation Example:

```
@Entity
@Table(name="passport")
public class Passport {
   @Id
   private int id;
   private String passportNumber;
}
@Entity
@Table(name="person")
public class Person {
   @Id
   private int id;
   private String name;
   @OneToOne(cascade = CascadeType.ALL)
   @JoinColumn(name="passport_id") // foreign key column
   private Passport passport;
}
```

Here:

- Each `Person` has one `Passport`.
- `@JoinColumn` specifies the foreign key in the `person` table.

XML Mapping Example (hibernate.cfg.xml style):

```
<class name="Person" table="person">
   <id name="id" column="id"/>
   <property name="name" column="name"/>
   <one-to-one name="passport" class="Passport" cascade="all"/>
</class>

<class name="Passport" table="passport">
   <id name="id" column="id"/>
   <property name="passportNumber" column="passport_number"/>
</class>
```

One-to-Many Mapping

A **one-to-many** relationship means one entity is related to multiple entities. Example: one Department has many Employees.

Annotation Example:

```
@Entity
@Table(name="department")
public class Department {
   @Id
   private int id;
   private String name;

   @OneToMany(mappedBy="department", cascade = CascadeType.ALL)
   private List<Employee> employees = new ArrayList<>();
}

@Entity
@Table(name="employee")
public class Employee {
   @Id
   private int id;
   private String name;
   @ManyToOne
   @JoinColumn(name="department_id")
   private Department department;
}
```

Here:

- A Department can have many Employees.
- In Employee, the department_id column acts as a foreign key.

XML Mapping Example:

```
<class name="Department" table="department">
   <id name="id" column="id"/>
   <property name="name" column="name"/>
   <set name="employees" cascade="all" inverse="true">
```

```
        <key column="department_id"/>
        <one-to-many class="Employee"/>
    </set>
</class>
<class name="Employee" table="employee">
    <id name="id" column="id"/>
    <property name="name" column="name"/>
    <many-to-one name="department" class="Department" column="department_id"/>
</class>
```

Many-to-Many Mapping

A **many-to-many** relationship means multiple entities of one type can be associated with multiple entities of another type. Example: a Student can enroll in many Courses, and each Course can have many Students.

Annotation Example:

```
@Entity
@Table(name="student")
public class Student {
    @Id
    private int id;
    private String name;

    @ManyToMany
    @JoinTable(

      name="student_course",
        joinColumns=@JoinColumn(name="student_id"),
        inverseJoinColumns=@JoinColumn(name="course_id")
    )
    private List<Course> courses = new ArrayList<>();
}

@Entity
@Table(name="course")
public class Course {
    @Id

   private int id;
    private String title;

    @ManyToMany(mappedBy="courses")
    private List<Student> students = new ArrayList<>();
}
```

Here:

- student_course is a join table created automatically to manage the relationship.

XML Mapping Example:

```
<class name="Student" table="student">
  <id name="id" column="id"/>
  <property name="name" column="name"/>
  <set name="courses" table="student_course" cascade="all">
    <key column="student_id"/>
    <many-to-many class="Course" column="course_id"/>
  </set>
</class>
<class name="Course" table="course">
  <id name="id" column="id"/>
  <property name="title" column="title"/>
  <set name="students" table="student_course" inverse="true">
    <key column="course_id"/>
    <many-to-many class="Student" column="student_id"/>
  </set>
</class>
```

Inheritance Mapping (Basics)

In Java, inheritance lets one class extend another. Hibernate also provides ways to map inheritance structures into relational databases. Since relational tables don't have inheritance, Hibernate provides three main strategies: **Table Per Class**, **Table Per Subclass (Joined)**, and **Table Per Concrete Class**.

Table Per Class

Each class in the inheritance hierarchy is mapped to its own table. All fields of the parent and child classes are stored together in the child's table.

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Vehicle {
  @Id
  private int id;
  private String name;
}
@Entity
public class Car extends Vehicle {
  private int speed;
}
@Entity
public class Bike extends Vehicle {
  private String type;
}
```

Generated tables:

- car → columns: id, name, speed
- bike → columns: id, name, type

Pros: Simple, no joins.
Cons: Queries across all vehicles are slower (UNION needed).

Table Per Subclass (Joined Strategy)

A base table is created for parent class fields, and each subclass has its own table with extra columns. Hibernate joins them when needed.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Vehicle {
   @Id
   private int id;
   private String name;
}
@Entity
public class Car extends Vehicle {
   private int speed;
}
@Entity
public class Bike extends Vehicle {
   private String type;
}
```

Generated tables:

- vehicle → id, name
- car → id, speed
- bike → id, type

Pros: Normalized schema, efficient for common queries.
Cons: Requires joins, slower inserts.

Table Per Concrete Class

Each concrete class has its own table, including all inherited fields. Parent class doesn't have a separate table.

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="vehicle_type", discriminatorType=DiscriminatorType.STRING)
public abstract class Vehicle {
   @Id
   private int id;
   private String name;
}
Entity
@DiscriminatorValue("CAR")
public class Car extends Vehicle {
   private int speed;
}

@Entity
@DiscriminatorValue("BIKE")
public class Bike extends Vehicle {
   private String type;
}
```

Generated table:

- vehicle → id, name, speed, type, vehicle_type (all in a single table).

Pros: Fast queries, simple.
Cons: Table may contain many null values, not normalized.

Mini example usage for all strategies:

```
Session session = factory.openSession();
Transaction tx = session.beginTransaction();

Car car = new Car();
car.setId(1);
car.setName("Audi");
car.setSpeed(200);

Bike bike = new Bike();
bike.setId(2);
bike.setName("Yamaha");
bike.setType("Sports");

session.save(car);
session.save(bike);

tx.commit();
session.close();
```

Hibernate will persist according to the chosen inheritance strategy.

Hibernate Queries

Hibernate provides multiple ways to interact with the database: **HQL (Hibernate Query Language)**, **Criteria API**, and **Named Queries**. These allow writing database-independent queries without using raw SQL.

Introduction to HQL (Hibernate Query Language)

HQL is an object-oriented query language, similar to SQL but works with entity objects instead of tables.

- SQL → Works with tables and columns
- HQL → Works with classes and properties

Example:

```
// Fetch all students
Query query = session.createQuery("from Student");
List<Student> students = query.list();
```

Here:

- `Student` is the entity class (not table name).
- Hibernate converts HQL to SQL internally.

CRUD Operations using HQL

Insert (usually done via `session.save`)

```
Student student = new Student();
student.setId(1);
student.setName("John");
student.setAge(22);

session.save(student);
```

Read (Select)

```
Query query = session.createQuery("from Student where age > :age");
query.setParameter("age", 20);
List<Student> result = query.list();
```

Update

```
Query query = session.createQuery("update Student set age = :age where name = :name");
query.setParameter("age", 25);
query.setParameter("name", "John");
int rows = query.executeUpdate();
```

Delete

```
Query query = session.createQuery("delete from Student where name = :name");
query.setParameter("name", "John");
int rows = query.executeUpdate();
```

Criteria API Basics

The Criteria API provides an **object-oriented approach** to querying, useful when queries are dynamic.

Basic Example:

```
Criteria criteria = session.createCriteria(Student.class);
criteria.add(Restrictions.gt("age", 20)); // age > 20
List<Student> students = criteria.list();
```

Projection Example (select specific fields):

```
Criteria criteria = session.createCriteria(Student.class);
criteria.setProjection(Projections.property("name"));
List<String> names = criteria.list();
```

Named Queries

Named queries are predefined queries stored in annotations or XML, reusable throughout the application.

Annotation Example:

```
@Entity
@NamedQueries({
  @NamedQuery(
    name="findStudentByName",
    query="from Student where name = :name"
  )
})
public class Student {
  @Id
  private int id;
  private String name;
  private int age;
}
```

Using Named Query:

```
Query query = session.getNamedQuery("findStudentByName");
query.setParameter("name", "John");
List<Student> result = query.list();
```

These query approaches give flexibility:

- **HQL** → SQL-like, concise
- **Criteria** → Dynamic, programmatic queries
- **Named Queries** → Predefined, reusable

Caching Basics

Caching is a key feature in Hibernate that improves performance by reducing the number of database calls. Hibernate supports multiple levels of caching.

First-Level Cache

The first-level cache is built into Hibernate and works at the **Session** level.

- Every Session object maintains a cache of persistent objects.

- If you query the same object multiple times in a single session, Hibernate returns it from cache instead of hitting the database again.

Example:

```
Session session = factory.openSession();
Transaction tx = session.beginTransaction();

// First fetch → goes to DB
Student s1 = session.get(Student.class, 1);
System.out.println(s1.getName());

// Second fetch → comes from Session cache (no DB hit)
Student s2 = session.get(Student.class, 1);
System.out.println(s2.getName());

tx.commit();
session.close();
```

Here:

- The second `get` call does not execute a SQL query.
- Once the session is closed, the cache is gone.

Introduction to Second-Level Cache

The second-level cache works at the **SessionFactory** level and can be shared across multiple sessions.

- Optional, needs configuration.
- Common providers: **Ehcache**, **Infinispan**, **OSCache**.
- Improves performance for frequently accessed data across sessions.

Basic setup for **Ehcache (example)**:

```
<hibernate-configuration>
  <session-factory>
    <!-- Second-level cache settings -->
    <property name="hibernate.cache.use_second_level_cache">true</property>
    <property
name="hibernate.cache.region.factory_class">org.hibernate.cache.ehcache.EhCacheRegionFactory</property>
  </session-factory>
</hibernate-configuration>
```

Entity with cache enabled:

```
@Entity
@Cacheable
@org.hibernate.annotations.Cache(usage = CacheConcurrencyStrategy.READ_ONLY)
public class Student {
  @Id
```

```
    private int id;
    private String name;
    private int age;
}
```

Usage:

- First session fetches from DB and stores in 2nd-level cache.
- Next session fetches the same entity directly from the cache without querying the DB.

Transactions

A transaction is a unit of work performed against the database. In Hibernate, transactions ensure **data integrity and consistency** when performing operations like insert, update, or delete.

Hibernate Transaction Management

Hibernate provides its own API for transaction management, and it can also integrate with JTA (Java Transaction API) or Spring.

Basic Example:

```
Session session = factory.openSession();
Transaction tx = null;

try {
    tx = session.beginTransaction();

    Student s = new Student();
    s.setId(1);
    s.setName("John");
    s.setAge(22);

    session.save(s); // Insert into DB

    tx.commit();  // Commit transaction
} catch (Exception e) {
    if (tx != null) tx.rollback();  // Rollback on error
    e.printStackTrace();
} finally {
    session.close();
}
```

Here:

- `beginTransaction()` starts a transaction.
- `commit()` saves changes permanently.
- `rollback()` undoes changes in case of failure.

ACID Properties in Hibernate

Hibernate transactions follow the **ACID** properties of database systems:

1. **Atomicity** – Either all operations in a transaction succeed or none are applied.
   - If an error occurs, Hibernate rolls back the transaction.
2. **Consistency** – A transaction transforms the database from one valid state to another valid state.
   - Example: Foreign key constraints remain valid after a transaction.
3. **Isolation** – Transactions are independent of each other, preventing dirty reads, non-repeatable reads, or phantom reads.
   - Hibernate can use database isolation levels (READ_COMMITTED, REPEATABLE_READ, SERIALIZABLE).
4. **Durability** – Once a transaction is committed, the changes are permanent, even in case of system failures.
   - Hibernate relies on the database for durability guarantees.

Advanced Mapping Concepts

When applications become more complex, entities often need **collections**, **bidirectional relationships**, **cascade operations**, and **embedded objects**. Hibernate provides powerful tools for these scenarios.

Collection Mapping (List, Set, Map, Bag, Array)

Hibernate allows storing collections inside an entity.

Example with List:

```
@Entity
@Table(name="student")
public class Student {
   @Id
   private int id;
   private String name;

   @ElementCollection
   @CollectionTable(name="student_courses", joinColumns=@JoinColumn(name="student_id"))
   @Column(name="course")
   private List<String> courses = new ArrayList<>();
}
```

This creates:

- student table → stores student info
- student_courses table → stores multiple courses per student

Other types:

- Set → stores unique values
- Bag → unordered list, allows duplicates
- Array → mapped to database array or serialized form
- Map → key-value pairs stored in a collection table

Bidirectional vs Unidirectional Mapping

- **Unidirectional**: Only one entity knows about the relationship.
- **Bidirectional**: Both entities know about each other.

Unidirectional One-to-Many:

```
@Entity
public class Department {
    @Id
    private int id;
    private String name;

    @OneToMany(cascade=CascadeType.ALL)
    @JoinColumn(name="department_id") // foreign key in employee table
    private List<Employee> employees = new ArrayList<>();
}
```

Bidirectional One-to-Many:

```
@Entity
public class Employee {
    @Id
    private int id;
    private String name;

    @ManyToOne
    @JoinColumn(name="department_id")
    private Department department;
}

@Entity
public class Department {
    @Id
    private int id;
    private String name;

    @OneToMany(mappedBy="department", cascade=CascadeType.ALL)
    private List<Employee> employees = new ArrayList<>();
}
```

Here, both Department and Employee are aware of the relationship.

Cascading Types (ALL, PERSIST, MERGE, REMOVE, REFRESH, DETACH)

Cascade defines how operations on one entity affect related entities.

- **ALL** → Applies all operations (persist, merge, remove, refresh, detach)
- **PERSIST** → Saving parent saves children
- **MERGE** → Updating parent updates children
- **REMOVE** → Deleting parent deletes children
- **REFRESH** → Refresh parent also refreshes children from DB
- **DETACH** → Detaching parent also detaches children

Example:

```
@OneToMany(mappedBy="department", cascade=CascadeType.ALL)
private List<Employee> employees = new ArrayList<>();
```

Now, saving or deleting a Department will apply the same operation to its Employees.

Embeddable Objects (@Embedded, @Embeddable)

Sometimes an entity contains reusable value objects that don't need their own table. Hibernate supports this via embeddables.

Example:

```
@Embeddable
public class Address {
   private String street;
   private String city;
   private String zipcode;
}

@Entity
public class Student {
   @Id
   private int id;
   private String name;

   @Embedded
   private Address address;
}
```

Here:

- `Address` is embedded into `Student`
- No separate table for `Address`
- Fields of `Address` appear as columns in the `student` table

Queries & Criteria (Deep Dive)

Hibernate provides advanced querying options beyond basic CRUD. You can use **HQL joins**, advanced **Criteria API**, the modern **JPA Criteria API**, and even **native SQL queries** for full flexibility.

HQL Joins (inner, left, right, full)

HQL supports join operations similar to SQL but works with entities and their associations.

Example Entities:

```
@Entity
public class Department {
   @Id
   private int id;
   private String name;

   @OneToMany(mappedBy="department")
   private List<Employee> employees = new ArrayList<>();
}

@Entity
public class Employee {
   @Id
   private int id;
   private String name;

   @ManyToOne
   @JoinColumn(name="department_id")
   private Department department;
}
```

Inner Join:

```
Query q = session.createQuery(
   "select d.name, e.name from Department d inner join d.employees e"
);
List<Object[]> result = q.list();
```

Left Join:

```
Query q = session.createQuery(
   "select d.name, e.name from Department d left join d.employees e"
);
```

Right Join:

```
Query q = session.createQuery(
   "select d.name, e.name from Department d right join d.employees e"
);
```

Full Join:

```
Query q = session.createQuery(
   "select d.name, e.name from Department d full join d.employees e"
);
```

Advanced Criteria API (Ordering, Pagination, Multiple Conditions)

The Criteria API (older Hibernate style) allows building queries programmatically.

Multiple Conditions:

```
Criteria criteria = session.createCriteria(Employee.class);
criteria.add(Restrictions.gt("id", 1));
criteria.add(Restrictions.like("name", "J%"));
List<Employee> employees = criteria.list();
```

Ordering:

```
criteria.addOrder(Order.asc("name")); // order by name ASC
```

Pagination:

```
criteria.setFirstResult(0);  // starting row
criteria.setMaxResults(10);  // limit
```

JPA Criteria API (modern replacement for Criteria)

The older `org.hibernate.Criteria` is deprecated. JPA Criteria API is type-safe and recommended.

```
CriteriaBuilder cb = session.getCriteriaBuilder();
CriteriaQuery<Employee> cq = cb.createQuery(Employee.class);
Root<Employee> root = cq.from(Employee.class);

// Select employees where id > 1 and order by name
cq.select(root)
  .where(cb.gt(root.get("id"), 1))
  .orderBy(cb.asc(root.get("name")));

Query<Employee> query = session.createQuery(cq);
List<Employee> employees = query.getResultList();
```

Native SQL Queries in Hibernate

Sometimes you need database-specific features. Hibernate allows direct SQL queries while still mapping results to entities.

Example:

```
String sql = "SELECT * FROM employee WHERE name = :name";
SQLQuery query = session.createSQLQuery(sql);
query.setParameter("name", "John");
query.addEntity(Employee.class); // map result to entity
List<Employee> employees = query.list();
```

You can also fetch scalar values:

```
String sql = "SELECT name, id FROM employee";
SQLQuery query = session.createSQLQuery(sql);
List<Object[]> result = query.list();
```

This chapter gives you advanced querying power:

- **HQL Joins** for relationships
- **Criteria API** for dynamic queries
- **JPA Criteria API** for type-safe queries
- **Native SQL** for database-specific operations

Caching & Performance Optimization

Caching and efficient fetching strategies are crucial for improving Hibernate performance. This chapter explores different cache levels, query caching, fetch modes, and optimization techniques like batch processing.

First-Level Cache (deep understanding)

- **Definition**: The first-level cache is associated with the Hibernate `Session` object.
- **Scope**: Transactional; exists only for the lifetime of the session.
- **Behavior**:
  - When you fetch an entity, Hibernate first checks the session cache.
  - If not found, it queries the database and stores it in the cache.
  - Any repeated call for the same entity within the same session does not hit the DB.

Example:

```
Session session = sessionFactory.openSession();

Employee e1 = session.get(Employee.class, 1); // Hits DB
Employee e2 = session.get(Employee.class, 1); // Uses cache, no DB call
```

Second-Level Cache

- **Definition**: Shared cache across sessions in a SessionFactory.
- **Scope**: Application-level cache, survives across multiple sessions.
- **Usage**: Reduces redundant DB calls for the same entity across different sessions.
- **Providers**: EhCache, Infinispan, OSCache, Hazelcast, etc.

Steps to enable:

1. Enable second-level cache in `hibernate.cfg.xml`.
2. Add caching annotations.

```
<property name="hibernate.cache.use_second_level_cache">true</property>
<property name="hibernate.cache.region.factory_class">
   org.hibernate.cache.ehcache.EhCacheRegionFactory
</property>
@Entity
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class Employee {
   @Id
   private int id;
   private String name;
}
```

Query Cache

- Stores results of queries, not entities themselves.
- Works with the second-level cache.

```
<property name="hibernate.cache.use_query_cache">true</property>
Query query = session.createQuery("from Employee where name = :name");
query.setParameter("name", "John");
query.setCacheable(true);
List<Employee> employees = query.list();
```

Lazy vs Eager Loading

- **Lazy Loading**: Associated entities are fetched only when accessed.
- **Eager Loading**: Associated entities are fetched immediately with the parent.

```
@OneToMany(mappedBy="department", fetch = FetchType.LAZY)
private List<Employee> employees;  // Default, fetch later
@OneToMany(mappedBy="department", fetch = FetchType.EAGER)
private List<Employee> employees;  // Fetch immediately
```

Fetch Strategies (select, join, batch)

1. **Select Fetching**: Default; separate query for each association.
2. **Join Fetching**: Fetches associations in a single SQL using `join fetch`.
3. Query q = session.createQuery("select d from Department d join fetch d.employees");
4. **Batch Fetching**: Loads multiple entities in batches to reduce queries.

```
<class name="Employee" batch-size="10">
   <id name="id"/>
</class>
```

N+1 Select Problem and Solutions

Problem:

When fetching parent entities, each child collection triggers an extra query.

- Example: Fetching 10 departments may cause 11 queries (1 for departments + 10 for employees).

Solutions:

- Use **join fetch** in HQL.
- Use **batch fetching**.
- Use **second-level cache** for associations.

Batch Processing

Efficient for bulk insert/update/delete operations.

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

for (int i = 0; i < 1000; i++) {
    Employee emp = new Employee(i, "Emp" + i);
    session.save(emp);

    if (i % 50 == 0) { // batch size
        session.flush();
        session.clear();
    }
}

tx.commit();
session.close();
```

Key Points

- **First-level cache** is session-specific.
- **Second-level cache** improves performance across sessions.
- **Query cache** optimizes repeated queries.
- **Lazy loading** prevents unnecessary DB hits.
- **Fetch strategies** and **batch processing** reduce the N+1 problem and improve efficiency.

Transactions & Concurrency Control

1. Transactions in Hibernate

- **Purpose:** Ensure **atomicity, consistency, isolation, and durability (ACID)** when performing database operations.

- **Types:**

    1. **Programmatic Transactions**

        You manually begin, commit, or rollback transactions in code.
        Example:

        ```
        Session session = sessionFactory.openSession();
        ```

```
Transaction tx = null;
try {
    tx = session.beginTransaction();
    // Perform operations
    session.save(entity);
    tx.commit();
} catch (Exception e) {
    if (tx != null) tx.rollback();
    e.printStackTrace();
} finally {
    session.close();
}
```

2. **Declarative Transactions**

   Managed by **Spring or JTA**, usually using @Transactional.
   The framework handles commit/rollback automatically.
   Example:

```
@Transactional
public void saveEmployee(Employee emp) {
    entityManager.persist(emp);
}
```

2. Concurrency Control

- **Purpose:** Manage simultaneous access to the same data to prevent **data inconsistencies**.

A. Optimistic Locking

- Assumes **conflicts are rare**.
- Uses a **version column** (@Version) to detect concurrent updates.
- Hibernate automatically checks the version before committing changes.
- Example:

```
@Entity
public class Employee {
    @Id
    private Long id;
        @Version
    private int version; // Managed automatically
        private String name;
    private double salary;
}
```

- **Behavior:**
  - Two transactions read the same record.
  - If both update it, the second one to commit will fail with OptimisticLockException.

B. Pessimistic Locking

- Locks the database row **immediately** to prevent others from modifying it until the transaction completes.
- Used when **conflicts are likely**.
- Methods:
  - LockMode.PESSIMISTIC_READ → Other transactions can read, but cannot write.
  - LockMode.PESSIMISTIC_WRITE → Other transactions cannot read or write.
- Example:

```
Employee emp = session.get(Employee.class, id, LockMode.PESSIMISTIC_WRITE);
emp.setSalary(emp.getSalary() + 1000);
```

Feature	**Optimistic Locking**	**Pessimistic Locking**
Conflict Probability	Low	High
Locking	No immediate lock	Locks the row immediately
Performance	High (less overhead)	Lower (more blocking)
Exception on conflict	OptimisticLockException	Waits or fails depending on DB

Hibernate with Spring

1. Integrating Hibernate with Spring ORM

- Spring provides **Hibernate support** via the Spring ORM module.
- Key Components:
  - **SessionFactory Bean**: Configured in Spring, manages Hibernate sessions.
  - **Transaction Manager**: Spring manages transactions declaratively or programmatically.
- **Configuration Example (Java-based):**

```
@Configuration
@EnableTransactionManagement
public class HibernateConfig {
  @Bean
  public LocalSessionFactoryBean sessionFactory() {
    LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
    sessionFactory.setDataSource(dataSource());
    sessionFactory.setPackagesToScan("com.example.model");
    sessionFactory.setHibernateProperties(hibernateProperties());
    return sessionFactory;
  }
  @Bean
  public PlatformTransactionManager transactionManager(SessionFactory sessionFactory) {
    HibernateTransactionManager txManager = new HibernateTransactionManager();
    txManager.setSessionFactory(sessionFactory);
    return txManager;
  }
}
```

2. Using Spring Data JPA with Hibernate

- Spring Data JPA simplifies data access by providing **CRUD operations out-of-the-box**.
- Hibernate acts as the **JPA provider** under the hood.
- **Example Repository:**

```
@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long> {
    List<Employee> findBySalaryGreaterThan(double salary);
}
```

- Benefits:
    - Less boilerplate code.
    - Supports **pagination, sorting, and custom queries**.
    - Integrates seamlessly with **Spring Transactions**.

3. Transaction Management in Spring + Hibernate

- Spring handles transactions in two ways:

  A. Declarative Transactions (Recommended)

    - Use `@Transactional` annotation on service methods.
    - Spring manages **commit/rollback** automatically.
    - Example:

```
@Service
public class EmployeeService {
    @Autowired
    private EmployeeRepository repo;
    @Transactional

    public void giveRaise(Long empId, double increment) {
        Employee emp = repo.findById(empId).orElseThrow();
        emp.setSalary(emp.getSalary() + increment);
        // No need to explicitly save, JPA flushes changes
    }
}
```

  B. Programmatic Transactions

    - Use `TransactionTemplate` or `PlatformTransactionManager` to control transactions manually.
    - Example:

```
transactionTemplate.execute(status -> {
    Employee emp = repo.findById(1L).orElseThrow();
    emp.setSalary(emp.getSalary() + 1000);
    return null;
});
```

Key Advantages of Spring + Hibernate Integration:

1. Simplifies **session and transaction management**.
2. Integrates **ORM capabilities** with **Spring DI & AOP**.
3. Supports **Spring Data repositories** for rapid development.
4. Reduces boilerplate code and improves maintainability.

Advanced Features of Hibernate

1. Interceptors in Hibernate

- **Purpose:** Intercept Hibernate operations (save, update, delete, load) to add custom behavior.
- **Use Cases:** Logging, auditing, automatic field updates.
- **Example:**

```
public class MyInterceptor extends EmptyInterceptor {
    @Override
    public boolean onSave(Object entity, Serializable id, Object[] state, String[] propertyNames, Type[]
types) {
        System.out.println("Saving entity: " + entity);
        return super.onSave(entity, id, state, propertyNames, types);
    }
}
Session session = sessionFactory.withOptions().interceptor(new MyInterceptor()).openSession();
```

2. Event Listeners

- **Purpose:** Listen to specific Hibernate events and act on them.
- **Difference from Interceptors:** More fine-grained control over events; can register multiple listeners for the same event.
- **Common Events:** pre-insert, post-insert, pre-update, post-update, pre-delete, post-delete.
- **Example:**

```
@EntityListeners(AuditListener.class)
public class Employee {
    ...
}
public class AuditListener {
    @PrePersist
    public void onPrePersist(Object entity) {
        System.out.println("About to persist: " + entity);
    }
}
```

3. Hibernate Validator (JSR 303 / JSR 380)

- **Purpose:** Enforce **Bean Validation** constraints on entity fields.
- **Annotations:** @NotNull, @Size, @Min, @Max, @Email, etc.
- **Example:**

```
@Entity
public class Employee {
   @Id
   private Long id;

   @NotNull
   @Size(min = 2, max = 50)
   private String name;
   @Min(1000)
   private double salary;
}
```

4. Hibernate Envers (Auditing & History Tracking)

- **Purpose:** Track entity changes over time for **audit/history purposes**.
- **Features:**
  - Automatically maintains revision history.
  - Query historical data easily.
- **Example:**

```
@Entity
@Audited
public class Employee {
   @Id
   private Long id;
   private String name;
   private double salary;
}
```

5. Multi-Tenancy Support

- Hibernate supports **multi-tenancy** for SaaS applications.
- **Types:**
  1. **Database-per-tenant** – Separate DB for each tenant.
  2. **Schema-per-tenant** – Same DB, different schema.
  3. **Discriminator-based** – Single table, tenant column identifies records.
- **Configuration:** Use MultiTenantConnectionProvider and CurrentTenantIdentifierResolver.

6. Hibernate with NoSQL (MongoDB, etc.)

- Hibernate OGM (Object/Grid Mapper) supports **NoSQL databases**.
- Allows using **Hibernate APIs and JPQL** with NoSQL like MongoDB, CouchDB, Neo4j.

- Example:

```
EntityManager em = emf.createEntityManager();
Employee emp = new Employee();
emp.setName("John");
em.persist(emp); // Works with MongoDB
```

Key Takeaways

- Interceptors & Event Listeners: Hooks for custom logic.
- Validator: Ensures **data integrity at the entity level**.
- Envers: Built-in **audit trail**.
- Multi-tenancy: Scalable **SaaS architecture support**.
- NoSQL integration: Extend Hibernate beyond relational DBs.

Real-Time / Interview-Oriented Topics

1. Common Hibernate Exceptions & Solutions

Exception	**Cause**	**Solution**
LazyInitializationException	Accessing lazy-loaded entity outside session	Ensure session is open, use fetch join, or initialize before closing session
NonUniqueObjectException	Two instances with same ID in session	Use merge() instead of save/update()
ConstraintViolationException	DB constraint violation	Validate data before saving, check schema constraints
StaleObjectStateException	Optimistic locking version conflict	Handle @Version conflicts properly, retry transaction
MappingException	Incorrect mapping or annotation	Check entity annotations, XML mappings, relationships

2. Best Practices in Hibernate Projects

1. Always use **session-per-transaction** pattern.
2. Prefer **Lazy loading** for associations to avoid unnecessary queries.
3. Use **Second-level cache** and **query cache** judiciously.
4. Keep **Hibernate mappings clean** and consistent.
5. Avoid **N+1 select problems** using JOIN FETCH or batch fetching.
6. Use **Spring + Hibernate** for declarative transaction management.

3. Writing Reusable DAOs / Repositories

- **Purpose:** Avoid duplicate CRUD code, provide generic operations.
- **Example Generic DAO:**

```
public interface GenericDao<T, ID extends Serializable> {
    T findById(ID id);
    List<T> findAll();
    void save(T entity);
    void update(T entity);
    void delete(T entity);
}

public class GenericDaoImpl<T, ID extends Serializable> implements GenericDao<T, ID> {
    @Autowired
    private SessionFactory sessionFactory;

    @Override
    public T findById(ID id) {
        return sessionFactory.getCurrentSession().get((Class<T>)Object.class, id);
    }
    // Implement other methods
}
```

4. Difference Between Hibernate & JPA

Feature	**Hibernate**	**JPA**
Type	ORM framework	Specification / API
Implementation	Yes	No, needs provider (Hibernate, EclipseLink)
Features	Advanced caching, Envers, Criteria API	Standardized, minimal features
Vendor Lock-in	Higher	Lower, portable across providers

5. Hibernate vs Spring Data JPA

Feature	**Hibernate**	**Spring Data JPA**
Ease of Use	Requires boilerplate code	Minimal boilerplate, repository-based
Querying	HQL, Criteria	JPQL, Method names, @Query annotation
Transactions	Manual or Spring	Managed by Spring declaratively
Caching	Built-in (first & second level)	Relies on JPA provider (Hibernate)

6. Migration from Legacy JDBC to Hibernate

- **Steps:**
  1. Create entity classes for DB tables.
  2. Configure `SessionFactory` or Spring Data JPA.
  3. Replace `Connection`, `PreparedStatement`, `ResultSet` with Hibernate sessions/repositories.
  4. Use transactions via Hibernate or Spring.
  5. Remove manual mapping code; rely on ORM mappings.

Key Takeaways

- Know common exceptions and how to solve them.
- Follow best practices for maintainable and efficient projects.
- Reusable DAOs/Repositories reduce code duplication.
- Understand differences between **Hibernate, JPA, and Spring Data JPA**.
- Migrating from JDBC is mostly about mapping tables to entities and managing sessions properly.

Projects:

Beginner Level Projects

These projects help you get familiar with **basic mappings, CRUD, and HQL**.

1. **Student Management System**
   - Features: Add, update, delete, list students; search by name or ID.
   - Concepts: Entity mapping, HQL queries, basic associations (One-to-One / One-to-Many).
2. **Library Management System**
   - Features: Manage books, authors, and borrowers.
   - Concepts: Many-to-Many relationship, CRUD operations, basic criteria queries.
3. **Simple Inventory System**
   - Features: Track products, categories, and stock levels.
   - Concepts: Entity mapping, simple transaction handling, HQL queries.

Intermediate Level Projects

These involve **associations, Spring integration, and caching**.

1. **Employee Management System with Spring + Hibernate**
   - Features: Add/update employees, departments, salaries; transactional operations.

   - Concepts: Spring ORM integration, @Transactional, One-to-Many and Many-to-One mapping.
2. **Online Course Enrollment System**
   - Features: Students enroll in courses, view progress, drop courses.
   - Concepts: Many-to-Many mapping, lazy vs eager loading, Spring Data JPA repositories.
3. **Hotel Booking System**
   - Features: Manage rooms, bookings, and customers.
   - Concepts: Cascading operations, optimistic locking, criteria API, query optimization.

Advanced Level Projects

These demonstrate **performance optimization, auditing, multi-tenancy, and real-world complexity**.

1. **E-Commerce Platform (Hibernate + Spring Boot)**
   - Features: Products, users, orders, shopping cart, order history.
   - Concepts: Caching strategies, batch processing, second-level cache, transaction management.
2. **Banking System with Auditing**
   - Features: Manage accounts, transactions, and generate audit history.
   - Concepts: Hibernate Envers for auditing, optimistic & pessimistic locking, multi-level associations.
3. **Multi-Tenant SaaS Application**
   - Features: Supports multiple companies (tenants) with separate data access.
   - Concepts: Multi-tenancy support (Database-per-tenant or Schema-per-tenant), Hibernate filters, Spring integration.
4. **Social Media Platform (Mini Version)**
   - Features: Users, posts, comments, likes.
   - Concepts: Complex entity relationships, fetch strategies to avoid N+1 select, query cache, event listeners.