Hibernate Essentials - Complete Step-by-Step Guide

Table of Contents

1. Basic Setup
2. Entity Mapping
3. CRUD Operations
4. Basic Relationships
5. Querying with HQL
6. Transaction Management

---

1. Basic Setup

What is Hibernate?

Definition: Hibernate is an ORM (Object-Relational Mapping) framework that maps Java objects to database tables automatically. Instead of writing SQL queries manually, you work with Java objects and Hibernate handles the database operations.

Analogy: Think of Hibernate as a translator between your Java code (objects) and your database (tables). You speak Java, the database speaks SQL, and Hibernate translates between them.

Step 1.1: Add Dependencies (Maven)

Add these to your `pom.xml`:

```xml
<dependencies>
    <!-- Hibernate Core -->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>6.2.7.Final</version>
    </dependency>

    <!-- MySQL Driver (or your database driver) -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.33</version>
    </dependency>
</dependencies>
```

Step 1.2: Configure Hibernate

Create `hibernate.cfg.xml` in `src/main/resources`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- Database Connection Settings -->
        <property
name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property
name="hibernate.connection.url">jdbc:mysql://localhost:3306/mydb</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">password</property>

        <!-- SQL Dialect -->
        <property
name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

        <!-- Show SQL in console -->
        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.format_sql">true</property>

        <!-- Auto create/update tables -->
        <property name="hibernate.hbm2ddl.auto">update</property>

        <!-- Map your entity classes here -->
        <mapping class="com.example.Student"/>
    </session-factory>
</hibernate-configuration>
```

Key Properties Explained:

- `hibernate.dialect`: Tells Hibernate which database you're using (MySQL, PostgreSQL, etc.)
- `hibernate.show_sql`: Shows generated SQL queries in console (great for learning!)
- `hibernate.hbm2ddl.auto`:
  - `create` - drops and creates tables every time
  - `update` - updates tables if schema changes
  - `validate` - just validates, doesn't change anything
  - `create-drop` - creates on start, drops on exit

Step 1.3: Understanding SessionFactory and Session

SessionFactory:

- **Definition**: A factory that creates Session objects. It's heavyweight and usually created once per application.
- **Analogy**: Like a factory building that manufactures cars (sessions). You build the factory once, then it produces many cars.

Session:

- **Definition**: Represents a single conversation with the database. It's lightweight and created per database operation.
- **Analogy**: Like a phone call to the database. You open it, do your work, then close it.

Creating SessionFactory:

```
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {
    private static SessionFactory sessionFactory;

    static {
        try {
            // Create SessionFactory from hibernate.cfg.xml
            sessionFactory = new Configuration()
                    .configure("hibernate.cfg.xml")
                    .buildSessionFactory();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

    public static void shutdown() {
        if (sessionFactory != null) {
            sessionFactory.close();
        }
    }
}
```

2. Entity Mapping

What is an Entity?

Definition: An entity is a Java class that represents a table in your database. Each instance of the class represents a row in that table.

Analogy: If your database table is a spreadsheet, then:

- The entity class is the template for each row
- Each object is one filled-in row
- Each field in the class is a column

Step 2.1: Creating Your First Entity

```
import jakarta.persistence.*;

@Entity
@Table(name = "students")  // Optional: specify table name
public class Student {

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

    @Column(name = "student_name", nullable = false, length = 100)
    private String name;

    @Column(name = "email", unique = true)
    private String email;

    private int age;  // Without @Column, column name will be "age"

    // Constructors
    public Student() {
        // Hibernate requires a no-arg constructor
    }

    public Student(String name, String email, int age) {
        this.name = name;
        this.email = email;
        this.age = age;
    }

    // Getters and Setters
    public Long getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "Student{id=" + id + ", name='" + name +
               "', email='" + email + "', age=" + age + "}";
    }
}
```

Key Annotations Explained

@Entity

- Marks this class as a Hibernate entity (database table)
- Required on every entity class

@Table

- Optional: Specifies the table name in the database
- If omitted, table name will be the class name (lowercase)

@Id

- Marks the primary key field
- Every entity must have exactly one @Id

@GeneratedValue

- Tells Hibernate how to generate primary key values
- Strategies:
  - `IDENTITY`: Database auto-increment (most common for MySQL)
  - `SEQUENCE`: Uses database sequences (PostgreSQL, Oracle)
  - `AUTO`: Hibernate picks the best strategy
  - `TABLE`: Uses a separate table to generate keys

@Column

- Optional: Configures the database column
- Attributes:
  - `name`: Column name in database
  - `nullable`: Can it be null? (default: true)
  - `unique`: Must values be unique? (default: false)
  - `length`: Max length for strings (default: 255)

Common Column Types

```
@Entity
public class Example {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;           // VARCHAR(255)
    private int age;                // INTEGER
    private double salary;          // DOUBLE
    private boolean isActive;       // BOOLEAN or BIT

    @Column(length = 1000)
    private String description;     // VARCHAR(1000)

    @Temporal(TemporalType.DATE)
    private Date birthDate;         // DATE (only date, no time)

    @Temporal(TemporalType.TIMESTAMP)
    private Date createdAt;         // TIMESTAMP (date + time)

    @Lob
    private String longText;        // TEXT or CLOB (large text)

    @Transient
    private String tempData;        // NOT stored in database
}
```

@Temporal: Used for Date fields to specify precision **@Lob**: Large Object - for very long text or binary data **@Transient**: Field is ignored by Hibernate (not saved to database)

---

3. CRUD Operations

What is CRUD?

Definition: Create, Read, Update, Delete - the four basic operations you can perform on database data.

Step 3.1: CREATE - Saving Data

persist() vs save():

- `persist()`: Saves object and keeps it managed (recommended)
- `save()`: Saves and returns the generated ID (legacy method)

```
import org.hibernate.Session;
import org.hibernate.Transaction;

public class CreateExample {
    public static void main(String[] args) {
        // Get SessionFactory
        SessionFactory factory = HibernateUtil.getSessionFactory();

        // Open a session
        Session session = factory.openSession();

        // Begin transaction
        Transaction transaction = session.beginTransaction();

        try {
            // Create a new student
            Student student = new Student("John Doe", "john@example.com", 20);

            // Save to database
            session.persist(student);

            // Commit transaction
            transaction.commit();

            System.out.println("Student saved with ID: " + student.getId());

        } catch (Exception e) {
            if (transaction != null) {
                transaction.rollback();
            }
            e.printStackTrace();
        } finally {
            session.close();
        }
    }
}
```

Step 3.2: READ - Retrieving Data

get() vs load():

- `get()`: Returns object or null if not found (use this)
- `load()`: Returns proxy, throws exception if not found (advanced)

```
public class ReadExample {
    public static void main(String[] args) {
        Session session = HibernateUtil.getSessionFactory().openSession();

        try {
            // Retrieve by ID
            Student student = session.get(Student.class, 1L);

            if (student != null) {
                System.out.println("Found: " + student);
            } else {
                System.out.println("Student not found");
            }

        } finally {
            session.close();
        }
    }
}
```

Step 3.3: UPDATE - Modifying Data

Method 1: Automatic Update (Managed Entity)

```
public class UpdateExample1 {
    public static void main(String[] args) {
        Session session = HibernateUtil.getSessionFactory().openSession();
        Transaction transaction = session.beginTransaction();

        try {
            // Retrieve student
            Student student = session.get(Student.class, 1L);

            // Modify it
            student.setAge(21);
            student.setEmail("newemail@example.com");

            // No need to call update() - Hibernate detects changes!

            transaction.commit();
            System.out.println("Student updated successfully");

        } catch (Exception e) {
            transaction.rollback();
            e.printStackTrace();
        } finally {
            session.close();
        }
    }
}
```

Method 2: Using merge() (Detached Entity)

```
public class UpdateExample2 {
    public static void main(String[] args) {
        // Create a student object (not from database)
        Student student = new Student();
        student.setId(1L);  // Set existing ID
        student.setName("Updated Name");
        student.setEmail("updated@example.com");
        student.setAge(22);

        Session session = HibernateUtil.getSessionFactory().openSession();
        Transaction transaction = session.beginTransaction();

        try {
            // Merge updates the database
            session.merge(student);

            transaction.commit();
            System.out.println("Student merged successfully");

        } catch (Exception e) {
            transaction.rollback();
            e.printStackTrace();
        } finally {
            session.close();
        }
    }
}
```

Step 3.4: DELETE - Removing Data

```
public class DeleteExample {
    public static void main(String[] args) {
        Session session = HibernateUtil.getSessionFactory().openSession();
        Transaction transaction = session.beginTransaction();

        try {
            // First, retrieve the student
            Student student = session.get(Student.class, 1L);

            if (student != null) {
                // Delete it
                session.remove(student);
                transaction.commit();
                System.out.println("Student deleted successfully");
            } else {
                System.out.println("Student not found");
            }

        } catch (Exception e) {
            transaction.rollback();
            e.printStackTrace();
        } finally {
            session.close();
        }
    }
}
```

Quick Reference:

```
// CREATE
session.persist(object);

// READ
Object obj = session.get(ClassName.class, id);

// UPDATE (managed entity)
obj.setSomething(newValue);  // Auto-detected

// UPDATE (detached entity)
session.merge(object);

// DELETE
session.remove(object);
```

---

4. Basic Relationships

Understanding Relationships

Analogy: Think of relationships like real-world connections:

- **One-to-One**: A person has one passport
- **One-to-Many**: A mother has many children
- **Many-to-One**: Many children have one mother
- **Many-to-Many**: Students enroll in many courses, courses have many students

Step 4.1: One-to-Many and Many-to-One

Scenario: One Author writes many Books

Author Entity (One Side):

```
@Entity
@Table(name = "authors")
public class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @OneToMany(mappedBy = "author", cascade = CascadeType.ALL, fetch =
FetchType.LAZY)
    private List<Book> books = new ArrayList<>();

    // Constructors
    public Author() {}
    public Author(String name) {
        this.name = name;
    }

    // Convenience method to add book
    public void addBook(Book book) {
        books.add(book);
        book.setAuthor(this);  // Set both sides of relationship
    }

    // Getters and Setters
    public Long getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public List<Book> getBooks() {
        return books;
    }

    public void setBooks(List<Book> books) {
        this.books = books;
    }
}
```

Book Entity (Many Side):

```
@Entity
@Table(name = "books")
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;
    private double price;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "author_id")  // Foreign key column name
    private Author author;
    // Constructors
    public Book() {}

    public Book(String title, double price) {
        this.title = title;
        this.price = price;
    }

    // Getters and Setters
    public Long getId() {
        return id;
    }

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

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public Author getAuthor() {
        return author;
    }

    public void setAuthor(Author author) {
        this.author = author;
    }
}
```

Using the Relationship:

```
public class RelationshipExample {
    public static void main(String[] args) {
        Session session = HibernateUtil.getSessionFactory().openSession();
        Transaction transaction = session.beginTransaction();

        try {
            // Create author
```

```java
            Author author = new Author("J.K. Rowling");

            // Create books
            Book book1 = new Book("Harry Potter 1", 29.99);
            Book book2 = new Book("Harry Potter 2", 31.99);

            // Establish relationship
            author.addBook(book1);
            author.addBook(book2);

            // Save author (books will be saved automatically due to cascade)
            session.persist(author);

            transaction.commit();
            System.out.println("Author and books saved!");

        } catch (Exception e) {
            transaction.rollback();
            e.printStackTrace();
        } finally {
            session.close();
        }
    }
}
```

Key Relationship Annotations Explained

@OneToMany

- Used on the "one" side (Author has many Books)
- `mappedBy`: Indicates the field in the other entity that owns the relationship
- Tells Hibernate: "Don't create a join table, the other side has the foreign key"

@ManyToOne

- Used on the "many" side (Many Books have one Author)
- This side owns the relationship (has the foreign key)
- `@JoinColumn`: Specifies the foreign key column name

@JoinColumn

- Specifies the foreign key column
- `name`: The column name in the database

Cascade Types

Definition: Cascade operations propagate from parent to child entities.

```java
@OneToMany(cascade = CascadeType.ALL)
private List<Book> books;
```

Common Cascade Types:

- `CascadeType.ALL`: All operations cascade (save, update, delete, etc.)
- `CascadeType.PERSIST`: Only save operations cascade
- `CascadeType.MERGE`: Only merge operations cascade
- `CascadeType.REMOVE`: Only delete operations cascade
- `CascadeType.REFRESH`: Only refresh operations cascade

Example:

```java
// With CascadeType.ALL
Author author = new Author("John");
author.addBook(new Book("Book 1", 20.0));
session.persist(author);  // Book is also saved automatically!

// Without cascade
Author author = new Author("John");
Book book = new Book("Book 1", 20.0);
author.addBook(book);
session.persist(author);  // Only author saved
session.persist(book);    // Must save book separately
```

Fetch Types

Definition: Determines when related entities are loaded from the database.

FetchType.LAZY (Default for @OneToMany, @ManyToMany):

- Related entities are loaded only when accessed
- Better performance, loads data on demand
- **Analogy**: Like opening a book only when you want to read it

```java
@OneToMany(fetch = FetchType.LAZY)
private List<Book> books;

// Usage
Author author = session.get(Author.class, 1L);  // Author loaded
// Books NOT loaded yet from database
System.out.println(author.getBooks().size());   // NOW books are loaded
```

FetchType.EAGER:

- Related entities are loaded immediately with parent
- Can cause performance issues with large datasets
- **Analogy**: Like carrying all your books wherever you go

```java
```

```
@ManyToOne(fetch = FetchType.EAGER)
private Author author;

// Usage
Book book = session.get(Book.class, 1L);  // Both Book AND Author loaded
immediately
```

Best Practice: Use LAZY by default, only use EAGER when you know you'll always need the related data.

Step 4.2: Many-to-Many Relationship

Scenario: Students enroll in many Courses, Courses have many Students

Student Entity:

```
@Entity
@Table(name = "students")
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    @JoinTable(
        name = "student_course",  // Join table name
        joinColumns = @JoinColumn(name = "student_id"),  // FK to Student
        inverseJoinColumns = @JoinColumn(name = "course_id")  // FK to Course
    )
    private List<Course> courses = new ArrayList<>();

    // Constructors
    public Student() {}

    public Student(String name) {
        this.name = name;
    }

    // Convenience methods
    public void enrollInCourse(Course course) {
        courses.add(course);
        course.getStudents().add(this);
    }

    public void dropCourse(Course course) {
        courses.remove(course);
        course.getStudents().remove(this);
    }

    // Getters and Setters
```

```java
    public Long getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public List<Course> getCourses() {
        return courses;
    }

    public void setCourses(List<Course> courses) {
        this.courses = courses;
    }
}
```

Course Entity:

```java
@Entity
@Table(name = "courses")
public class Course {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;
    private int credits;

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

    // Constructors
    public Course() {}

    public Course(String title, int credits) {
        this.title = title;
        this.credits = credits;
    }

    // Getters and Setters
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public int getCredits() {
        return credits;
    }

    public void setCredits(int credits) {
        this.credits = credits;
    }

    public List<Student> getStudents() {
        return students;
    }

    public void setStudents(List<Student> students) {
        this.students = students;
    }
}
```

Using Many-to-Many:

```
public class ManyToManyExample {
    public static void main(String[] args) {
        Session session = HibernateUtil.getSessionFactory().openSession();
        Transaction transaction = session.beginTransaction();

        try {
            // Create courses
            Course java = new Course("Java Programming", 3);
            Course database = new Course("Database Systems", 4);

            // Create students
            Student alice = new Student("Alice");
            Student bob = new Student("Bob");

            // Enroll students in courses
            alice.enrollInCourse(java);
            alice.enrollInCourse(database);
            bob.enrollInCourse(java);

            // Save everything
            session.persist(java);
            session.persist(database);
            session.persist(alice);
            session.persist(bob);

            transaction.commit();
            System.out.println("Students and courses saved!");

        } catch (Exception e) {
            transaction.rollback();
            e.printStackTrace();
        } finally {
            session.close();
        }
    }
}
```

@JoinTable Explained:

- Creates an intermediate join table for many-to-many relationships
- `name`: Name of the join table
- `joinColumns`: Foreign key to the owning entity (Student)
- `inverseJoinColumns`: Foreign key to the other entity (Course)

Database Structure:

```
students table: id, name
courses table: id, title, credits
student_course table: student_id, course_id
```

---

5. Querying with HQL

What is HQL?

Definition: Hibernate Query Language - similar to SQL but works with Java objects instead of tables.

Analogy: SQL talks to tables, HQL talks to your Java classes. Instead of `SELECT * FROM students`, you write `FROM Student`.

Step 5.1: Basic HQL Queries

Simple SELECT:

```
public class HQLExample1 {
    public static void main(String[] args) {
        Session session = HibernateUtil.getSessionFactory().openSession();

        try {
            // HQL query - notice "Student" is the class name, not table name
            String hql = "FROM Student";
            List<Student> students = session.createQuery(hql,
Student.class).list();

            for (Student student : students) {
                System.out.println(student);
            }

        } finally {
            session.close();
        }
    }
}
```

SELECT with WHERE clause:

```
public class HQLExample2 {
    public static void main(String[] args) {
        Session session = HibernateUtil.getSessionFactory().openSession();

        try {
            // Using parameters (ALWAYS use parameters, never string
concatenation!)
            String hql = "FROM Student WHERE age > :minAge";

            List<Student> students = session.createQuery(hql, Student.class)
                    .setParameter("minAge", 18)
                    .list();

            students.forEach(System.out::println);

        } finally {
            session.close();
        }
    }
}
```

Step 5.2: HQL with Conditions

Multiple Conditions:

```
String hql = "FROM Student WHERE age > :minAge AND name LIKE :namePattern";

List<Student> students = session.createQuery(hql, Student.class)
        .setParameter("minAge", 18)
        .setParameter("namePattern", "John%")
        .list();
```

ORDER BY:

```
String hql = "FROM Student ORDER BY age DESC, name ASC";
List<Student> students = session.createQuery(hql, Student.class).list();
```

Pagination:

```
String hql = "FROM Student";

List<Student> students = session.createQuery(hql, Student.class)
        .setFirstResult(0)      // Start from record 0
        .setMaxResults(10)      // Get 10 records
        .list();                // Like LIMIT 0, 10 in SQL
```

Step 5.3: HQL with Joins

Fetching Related Entities:

```
// Without JOIN FETCH - causes N+1 problem
String hql1 = "FROM Author";
List<Author> authors = session.createQuery(hql1, Author.class).list();
// If you access author.getBooks(), it queries database for EACH author

// With JOIN FETCH - loads everything in one query
String hql2 = "FROM Author a JOIN FETCH a.books";
List<Author> authors2 = session.createQuery(hql2, Author.class).list();
// Books are already loaded, no additional queries needed!
```

Filtering Related Entities:

```
String hql = "FROM Author a JOIN a.books b WHERE b.price > :minPrice";

List<Author> authors = session.createQuery(hql, Author.class)
        .setParameter("minPrice", 30.0)
        .list();
```

Step 5.4: Aggregate Functions

COUNT:

```
String hql = "SELECT COUNT(s) FROM Student s";
Long count = session.createQuery(hql, Long.class).uniqueResult();
System.out.println("Total students: " + count);
```

AVG, SUM, MAX, MIN:

```java
// Average age
String hql1 = "SELECT AVG(s.age) FROM Student s";
Double avgAge = session.createQuery(hql1, Double.class).uniqueResult();

// Sum of all book prices
String hql2 = "SELECT SUM(b.price) FROM Book b";
Double totalPrice = session.createQuery(hql2, Double.class).uniqueResult();

// Maximum age
String hql3 = "SELECT MAX(s.age) FROM Student s";
Integer maxAge = session.createQuery(hql3, Integer.class).uniqueResult();
```

GROUP BY:

```java
String hql = "SELECT a.name, COUNT(b) FROM Author a JOIN a.books b GROUP BY a.name";

List<Object[]> results = session.createQuery(hql, Object[].class).list();

for (Object[] row : results) {
    String authorName = (String) row[0];
    Long bookCount = (Long) row[1];
    System.out.println(authorName + " has written " + bookCount + " books");
}
```

Step 5.5: UPDATE and DELETE with HQL

UPDATE:

```java
public class HQLUpdateExample {
    public static void main(String[] args) {
        Session session = HibernateUtil.getSessionFactory().openSession();
        Transaction transaction = session.beginTransaction();

        try {
            String hql = "UPDATE Student SET age = :newAge WHERE name = :name";

            int updatedCount = session.createMutationQuery(hql)
                    .setParameter("newAge", 21)
                    .setParameter("name", "John Doe")
                    .executeUpdate();

            transaction.commit();
            System.out.println("Updated " + updatedCount + " records");
        } catch (Exception e) {
            transaction.rollback();
            e.printStackTrace();
        } finally {
            session.close();
        }
    }
}
```

DELETE:

```
String hql = "DELETE FROM Student WHERE age < :minAge";

int deletedCount = session.createMutationQuery(hql)
        .setParameter("minAge", 18)
        .executeUpdate();

System.out.println("Deleted " + deletedCount + " records");
```

HQL Quick Reference

```
// SELECT all
"FROM Student"

// SELECT with condition
"FROM Student WHERE age > :age"

// SELECT specific fields
"SELECT s.name, s.email FROM Student s"

// JOIN
"FROM Author a JOIN a.books b"

// JOIN FETCH (load related entities)
"FROM Author a JOIN FETCH a.books"

// ORDER BY
"FROM Student ORDER BY age DESC"

// COUNT
"SELECT COUNT(s) FROM Student s"

// GROUP BY
"SELECT s.age, COUNT(s) FROM Student s GROUP BY s.age"

// UPDATE
"UPDATE Student SET age = :age WHERE id = :id"

// DELETE
```

```
"DELETE FROM Student WHERE age < :age"

// LIKE operator
"FROM Student WHERE name LIKE :pattern"  // Use with "John%" or "%Doe"

// IN operator
"FROM Student WHERE age IN :ages"  // Use with List or array

// BETWEEN
"FROM Student WHERE age BETWEEN :min AND :max"

// IS NULL / IS NOT NULL
"FROM Student WHERE email IS NOT NULL"
```

Parameter Binding:

```
// Named parameters (recommended)
query.setParameter("name", "John");

// Multiple parameters
query.setParameter("minAge", 18).setParameter("maxAge", 25);

// List parameter (for IN clause)
List<Integer> ages = Arrays.asList(18, 19, 20);
query.setParameter("ages", ages);
```

---

6. Transaction Management

What is a Transaction?

Definition: A transaction is a unit of work that either completely succeeds or completely fails. It ensures data consistency.

Analogy: Think of a bank transfer. Money must be deducted from one account AND added to another. Both operations must succeed, or neither should happen. That's a transaction.

The ACID Properties

Transactions follow ACID principles:

- **Atomicity**: All operations succeed, or all fail (no partial updates)
- **Consistency**: Database moves from one valid state to another
- **Isolation**: Concurrent transactions don't interfere with each other
- **Durability**: Once committed, changes are permanent

Step 6.1: Basic Transaction Pattern

Standard Transaction Template:

```java
public class TransactionExample {
    public static void saveStudent(Student student) {
        Session session = HibernateUtil.getSessionFactory().openSession();
        Transaction transaction = null;

        try {
            // 1. Begin transaction
            transaction = session.beginTransaction();

            // 2. Perform database operations
            session.persist(student);

            // 3. Commit transaction (save changes)
            transaction.commit();

        } catch (Exception e) {
            // 4. Rollback on error (undo changes)
            if (transaction != null) {
                transaction.rollback();
            }
            e.printStackTrace();
            throw e;  // Re-throw if needed

        } finally {
            // 5. Always close session
            session.close();
        }
    }
}
```

Step 6.2: Why Transactions Matter

Without Transaction (DON'T DO THIS):

```java
// BAD - No transaction!
Session session = HibernateUtil.getSessionFactory().openSession();
Student student = new Student("John", "john@email.com", 20);
session.persist(student);  // Might not save!
session.close();
```

With Transaction (CORRECT):

```java
// GOOD - Wrapped in transaction
Session session = HibernateUtil.getSessionFactory().openSession();
Transaction tx = session.beginTransaction();
try {
    Student student = new Student("John", "john@email.com", 20);
    session.persist(student);
    tx.commit();  // Changes are saved
} catch (Exception e) {
    tx.rollback();  // Changes are undone
} finally {
    session.close();
}
```

Step 6.3: Transaction Rollback Example

Scenario: Transfer books from one author to another. If any step fails, nothing should change.

```
public class TransactionRollbackExample {
    public static void transferBooks(Long fromAuthorId, Long toAuthorId) {
        Session session = HibernateUtil.getSessionFactory().openSession();
        Transaction transaction = session.beginTransaction();

        try {
            // Get both authors
            Author fromAuthor = session.get(Author.class, fromAuthorId);
            Author toAuthor = session.get(Author.class, toAuthorId);

            if (fromAuthor == null || toAuthor == null) {
                throw new RuntimeException("Author not found!");
            }

            // Transfer all books
            List<Book> books = new ArrayList<>(fromAuthor.getBooks());
            for (Book book : books) {
                fromAuthor.getBooks().remove(book);
                toAuthor.addBook(book);
            }

            // If this line throws an exception, everything above is rolled back
            // someRiskyOperation();

            // Commit - all changes are saved
            transaction.commit();
            System.out.println("Books transferred successfully!");

        } catch (Exception e) {
            // Rollback - all changes are undone
            transaction.rollback();
            System.out.println("Transfer failed! Rolling back changes.");
            e.printStackTrace();

        } finally {
            session.close();
        }
    }
}
```

Step 6.4: Multiple Operations in One Transaction

```
public class MultipleOperationsExample {
    public static void performMultipleOperations() {
        Session session = HibernateUtil.getSessionFactory().openSession();
        Transaction transaction = session.beginTransaction();

        try {
            // Operation 1: Create new student
            Student student = new Student("Alice", "alice@email.com", 19);
            session.persist(student);

            // Operation 2: Update existing student
            Student existingStudent = session.get(Student.class, 1L);
            existingStudent.setAge(21);

            // Operation 3: Delete a student
            Student studentToDelete = session.get(Student.class, 5L);
            session.remove(studentToDelete);

            // Operation 4: Create new course
            Course course = new Course("Spring Boot", 4);
            session.persist(course);

            // All operations succeed together or fail together
            transaction.commit();
            System.out.println("All operations completed successfully!");

        } catch (Exception e) {
            // If ANY operation fails, ALL are rolled back
            transaction.rollback();
            System.out.println("Error occurred! All changes rolled back.");
            e.printStackTrace();

        } finally {
            session.close();
        }
    }
}
```

Step 6.5: Transaction Best Practices

1. Keep Transactions Short

```
// BAD - Transaction open too long
transaction.begin();
Student student = session.get(Student.class, 1L);
// ... lots of business logic ...
// ... maybe some user input ...
// ... network calls ...
```

```java
student.setAge(21);
transaction.commit();  // Transaction was open for too long!

// GOOD - Transaction only for database operations
Student student = session.get(Student.class, 1L);
// ... business logic outside transaction ...
int newAge = calculateAge();  // Complex calculation
transaction.begin();
student.setAge(newAge);
transaction.commit();  // Quick transaction
```

2. Always Use try-catch-finally

```java
// GOOD pattern
Session session = null;
Transaction transaction = null;
try {
    session = HibernateUtil.getSessionFactory().openSession();
    transaction = session.beginTransaction();

    // Your code here

    transaction.commit();
} catch (Exception e) {
    if (transaction != null) {
        transaction.rollback();
    }
    e.printStackTrace();
} finally {
    if (session != null) {
        session.close();
    }
}
```

3. Don't Catch and Ignore Exceptions

```java
// BAD - Silent failure
try {
    transaction.commit();
} catch (Exception e) {
    // Empty catch block - error is hidden!
}

// GOOD - Handle or propagate errors
try {
    transaction.commit();
} catch (Exception e) {
    transaction.rollback();
    e.printStackTrace();
```

```
        throw new RuntimeException("Failed to save data", e);
    }
```

4. One Transaction Per Business Operation

```
// Separate transactions for different operations
public void saveStudent(Student student) {
    // Transaction 1
    Session session = openSession();
    Transaction tx = session.beginTransaction();
    try {
        session.persist(student);
        tx.commit();
    } catch (Exception e) {
        tx.rollback();
    } finally {
        session.close();
    }
}

public void updateStudent(Student student) {
    // Transaction 2
    Session session = openSession();
    Transaction tx = session.beginTransaction();
    try {
        session.merge(student);
        tx.commit();
    } catch (Exception e) {
        tx.rollback();
    } finally {
        session.close();
    }
}
```

Complete Working Examples

Example 1: Simple CRUD Application

```
public class StudentManager {
    private static SessionFactory factory = HibernateUtil.getSessionFactory();

    // CREATE
    public static void createStudent(String name, String email, int age) {
        Session session = factory.openSession();
        Transaction tx = session.beginTransaction();

        try {
            Student student = new Student(name, email, age);
            session.persist(student);
            tx.commit();
            System.out.println("Student created with ID: " + student.getId());
        } catch (Exception e) {
            tx.rollback();
            e.printStackTrace();
        } finally {
            session.close();
        }
    }

    // READ
    public static Student getStudent(Long id) {
        Session session = factory.openSession();
        try {
            return session.get(Student.class, id);
        } finally {
            session.close();
        }
    }

    // READ ALL
    public static List<Student> getAllStudents() {
        Session session = factory.openSession();
        try {
            return session.createQuery("FROM Student", Student.class).list();
        } finally {
            session.close();
        }
    }

    // UPDATE
    public static void updateStudent(Long id, String newEmail) {
        Session session = factory.openSession();
        Transaction tx = session.beginTransaction();

        try {
            Student student = session.get(Student.class, id);
            if (student != null) {
                student.setEmail(newEmail);
                tx.commit();
                System.out.println("Student updated successfully");
            } else {
                System.out.println("Student not found");
            }
        } catch (Exception e) {
            tx.rollback();
            e.printStackTrace();
        } finally {
            session.close();
        }
    }

    // DELETE
    public static void deleteStudent(Long id) {
        Session session = factory.openSession();
        Transaction tx = session.beginTransaction();

        try {
            Student student = session.get(Student.class, id);
            if (student != null) {
                session.remove(student);
                tx.commit();
                System.out.println("Student deleted successfully");
            } else {
                System.out.println("Student not found");
            }
        } catch (Exception e) {
            tx.rollback();
            e.printStackTrace();
        } finally {
            session.close();
        }
    }

    // SEARCH
    public static List<Student> searchByName(String namePattern) {
        Session session = factory.openSession();
        try {
            String hql = "FROM Student WHERE name LIKE :pattern";
            return session.createQuery(hql, Student.class)
                    .setParameter("pattern", "%" + namePattern + "%")
                    .list();
        } finally {
            session.close();
        }
    }

    // Main method to test
    public static void main(String[] args) {
        // Create
        createStudent("John Doe", "john@email.com", 20);
        createStudent("Jane Smith", "jane@email.com", 22);

        // Read
        Student student = getStudent(1L);
        System.out.println("Found: " + student);

        // Read All
        System.out.println("\nAll students:");
        getAllStudents().forEach(System.out::println);

        // Update
        updateStudent(1L, "newemail@example.com");

        // Search
        System.out.println("\nSearch results:");
        searchByName("John").forEach(System.out::println);
        // Delete
        deleteStudent(2L);

        // Cleanup
        HibernateUtil.shutdown();
    }
}
```

Example 2: Working with Relationships

```
public class LibraryManager {
    private static SessionFactory factory = HibernateUtil.getSessionFactory();

    // Create author with books
    public static void createAuthorWithBooks() {
        Session session = factory.openSession();
        Transaction tx = session.beginTransaction();

        try {
            // Create author
            Author author = new Author("George Orwell");

            // Create books
            Book book1 = new Book("1984", 15.99);
            Book book2 = new Book("Animal Farm", 12.99);

            // Establish relationships
            author.addBook(book1);
            author.addBook(book2);

            // Save (cascade will save books too)
            session.persist(author);

            tx.commit();
            System.out.println("Author and books saved!");

        } catch (Exception e) {
            tx.rollback();
            e.printStackTrace();
        } finally {
            session.close();
        }
    }

    // Get author with all books
    public static void displayAuthorWithBooks(Long authorId) {
        Session session = factory.openSession();

        try {
            // Use JOIN FETCH to load books in single query
            String hql = "FROM Author a LEFT JOIN FETCH a.books WHERE a.id = :id";
            Author author = session.createQuery(hql, Author.class)
                    .setParameter("id", authorId)
                    .uniqueResult();

            if (author != null) {
                System.out.println("Author: " + author.getName());
                System.out.println("Books:");
                for (Book book : author.getBooks()) {
                    System.out.println("  - " + book.getTitle() + " ($" + book.getPrice() + ")");
                }
            }

        } finally {
            session.close();
        }
    }

    // Find all authors with books above certain price
    public static void findAuthorsWithExpensiveBooks(double minPrice) {
        Session session = factory.openSession();

        try {
            String hql = "SELECT DISTINCT a FROM Author a JOIN a.books b WHERE b.price > :price";
            List<Author> authors = session.createQuery(hql, Author.class)
                    .setParameter("price", minPrice)
                    .list();

            System.out.println("Authors with books over $" + minPrice + ":");
            authors.forEach(a -> System.out.println("  - " + a.getName()));

        } finally {
            session.close();
        }
    }

    public static void main(String[] args) {
        createAuthorWithBooks();
        displayAuthorWithBooks(1L);
        findAuthorsWithExpensiveBooks(14.00);

        HibernateUtil.shutdown();
    }
}
```

Example 3: Many-to-Many Enrollment System

```
public class EnrollmentManager {
    private static SessionFactory factory = HibernateUtil.getSessionFactory();
    // Setup: Create students and courses
    public static void setupData() {
        Session session = factory.openSession();
        Transaction tx = session.beginTransaction();

        try {
            // Create courses
            Course java = new Course("Java Programming", 3);
            Course database = new Course("Database Systems", 4);
            Course web = new Course("Web Development", 3);

            // Create students
            Student alice = new Student("Alice", "alice@email.com", 20);
            Student bob = new Student("Bob", "bob@email.com", 21);
            Student charlie = new Student("Charlie", "charlie@email.com", 19);

            // Enroll students
            alice.enrollInCourse(java);
            alice.enrollInCourse(database);
            bob.enrollInCourse(java);
            bob.enrollInCourse(web);
            charlie.enrollInCourse(database);
            charlie.enrollInCourse(web);

            // Save all
            session.persist(java);
            session.persist(database);
            session.persist(web);
            session.persist(alice);
            session.persist(bob);
            session.persist(charlie);

            tx.commit();
            System.out.println("Data setup complete!");

        } catch (Exception e) {
            tx.rollback();
            e.printStackTrace();
        } finally {
            session.close();
        }
    }

    // Display all students in a course
    public static void displayCourseEnrollment(Long courseId) {
        Session session = factory.openSession();

        try {
            String hql = "FROM Course c LEFT JOIN FETCH c.students WHERE c.id = :id";
            Course course = session.createQuery(hql, Course.class)
                    .setParameter("id", courseId)
                    .uniqueResult();
            if (course != null) {
                System.out.println("\nCourse: " + course.getTitle());
                System.out.println("Enrolled students:");
                for (Student student : course.getStudents()) {
                    System.out.println("  - " + student.getName());
                }
            }

        } finally {
            session.close();
        }
    }

    // Display all courses for a student
    public static void displayStudentCourses(Long studentId) {
        Session session = factory.openSession();

        try {
            String hql = "FROM Student s LEFT JOIN FETCH s.courses WHERE s.id =
:id";
            Student student = session.createQuery(hql, Student.class)
                    .setParameter("id", studentId)
                    .uniqueResult();

            if (student != null) {
                System.out.println("\nStudent: " + student.getName());
                System.out.println("Enrolled courses:");
                for (Course course : student.getCourses()) {
                    System.out.println("  - " + course.getTitle() + " (" +
course.getCredits() + " credits)");
                }
            }

        } finally {
            session.close();
        }
    }

    // Find students enrolled in multiple courses
    public static void findBusyStudents(int minCourses) {
        Session session = factory.openSession();

        try {
            String hql = "SELECT s FROM Student s WHERE SIZE(s.courses) >= :min";
            List<Student> students = session.createQuery(hql, Student.class)
                    .setParameter("min", minCourses)
                    .list();

            System.out.println("\nStudents taking " + minCourses + "+ courses:");
            for (Student student : students) {
                System.out.println("  - " + student.getName() + " (" +
student.getCourses().size() + " courses)");
            }
        } finally {
            session.close();
        }
    }

    public static void main(String[] args) {
        setupData();
        displayCourseEnrollment(1L);
        displayStudentCourses(1L);
        findBusyStudents(2);

        HibernateUtil.shutdown();
    }
}
```

Common Mistakes to Avoid

1. Forgetting to Close Session

```
// BAD
Session session = factory.openSession();
session.persist(student);
// Session never closed - memory leak!

// GOOD
Session session = factory.openSession();
try {
    session.persist(student);
} finally {
    session.close();  // Always close
}
```

2. Not Using Transactions

```
// BAD
Session session = factory.openSession();
session.persist(student);  // Might not save!
session.close();

// GOOD
Session session = factory.openSession();
Transaction tx = session.beginTransaction();
try {
    session.persist(student);
    tx.commit();
} catch (Exception e) {
    tx.rollback();
} finally {
    session.close();
}
```

3. LazyInitializationException

```
// BAD
Session session = factory.openSession();
Author author = session.get(Author.class, 1L);
session.close();
// Later...
author.getBooks().size();  // ERROR! Session is closed!

// GOOD - Option 1: Access within session
Session session = factory.openSession();
Author author = session.get(Author.class, 1L);
author.getBooks().size();  // Access before closing
session.close();

// GOOD - Option 2: Use JOIN FETCH
String hql = "FROM Author a JOIN FETCH a.books WHERE a.id = :id";
Author author = session.createQuery(hql, Author.class)
        .setParameter("id", 1L)
        .uniqueResult();
session.close();
author.getBooks().size();  // OK! Books already loaded
```

4. N+1 Query Problem

```
// BAD - Causes N+1 queries
String hql = "FROM Author";
List<Author> authors = session.createQuery(hql, Author.class).list();
// 1 query to get authors
for (Author author : authors) {
    author.getBooks().size();  // N queries (one per author)!
}

// GOOD - Single query
String hql = "FROM Author a LEFT JOIN FETCH a.books";
List<Author> authors = session.createQuery(hql, Author.class)
        .setResultTransformer(Transformers.distinctRootEntity())  // Remove
duplicates
        .list();
// Only 1 query total!
```

5. Not Setting Both Sides of Bidirectional Relationship

```java
// BAD
Author author = new Author("John");
Book book = new Book("Title", 20.0);
author.getBooks().add(book);  // Only one side set!
session.persist(author);
// book.author is null!

// GOOD - Use convenience method
author.addBook(book);  // Sets both sides
session.persist(author);
```

Document Information

Author: Mahesh Mekala

Contact:

- Email: [mmahesh23022@gmail.com]
- GitHub: [github.com/mmahesh7]
- LinkedIn: [linkedin.com/in/mmahesh7]

Happy Learning!