Lecture 02&03: Hibernate Introduction

What is Hibernate?

- **Definition**: Hibernate is an Object-Relational Mapping (ORM) framework for Java
- **Purpose**: To increase developer productivity by simplifying database operations
- **Core Concept**: Java is object-oriented programming, so we should work with objects rather than SQL statements directly

Problem Without Hibernate

When working without Hibernate, developers must:

- Write complex SQL queries manually
- Handle database connections explicitly
- Manage data type conversions between Java objects and database tables
- Write boilerplate code for CRUD operations

Example of a manual approach:

```
String sql = "INSERT INTO student (rollNo, name, age) VALUES (?, ?, ?)";
PreparedStatement pstmt = connection.prepareStatement(sql);
pstmt.setInt(1, student.getRollNo());
pstmt.setString(2, student.getName());
pstmt.setInt(3, student.getAge());

pstmt.executeUpdate();
```

ORM Mapping Concept

- **Object**: Java classes and their instances
- **Relational**: Database tables and rows
- **Mapping**: Automatic conversion between objects and database records

The ORM mapping shows how a Student class maps to a student table:

- Class fields → Table columns
- Object instances → Table rows
- Java data types → SQL data types

Lecture 04: Project Setup

Creating a New Hibernate Project

1. **IDE Setup**: Use IntelliJ IDEA or similar IDE
2. **Project Type**: Maven project for dependency management
3. **Java Version**: Use Java 8 or higher
4. **Build System**: Maven for managing dependencies

Adding Dependencies

Add the following dependencies to your `pom.xml`:

```xml
<dependencies>
   <!-- Hibernate Core -->
   <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>6.3.Final</version>
   </dependency>

   <!-- PostgreSQL Driver -->
   <dependency>
      <groupId>org.postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <version>42.7.3</version>
   </dependency>
</dependencies>
```

Key Dependencies Explained

- **hibernate-core**: Main Hibernate functionality
- **postgresql**: Database driver for PostgreSQL connectivity

Lecture 02&03: Hibernate Introduction

What is Hibernate?

- **Definition**: Hibernate is an Object-Relational Mapping (ORM) framework for Java
- **Purpose**: To increase developer productivity by simplifying database operations
- **Core Concept**: Java is object-oriented programming, so we should work with objects rather than SQL statements directly

Problem Without Hibernate

When working without Hibernate, developers must:

- Write complex SQL queries manually
- Handle database connections explicitly
- Manage data type conversions between Java objects and database tables
- Write boilerplate code for CRUD operations

Example of a manual approach:

```
String sql = "INSERT INTO student (rollNo, name, age) VALUES (?, ?, ?)";
PreparedStatement pstmt = connection.prepareStatement(sql);
pstmt.setInt(1, student.getRollNo());
pstmt.setString(2, student.getName());
pstmt.setInt(3, student.getAge());

pstmt.executeUpdate();
```

ORM Mapping Concept

- **Object**: Java classes and their instances
- **Relational**: Database tables and rows
- **Mapping**: Automatic conversion between objects and database records

The ORM mapping shows how a Student class maps to a student table:

- Class fields → Table columns
- Object instances → Table rows
- Java data types → SQL data types

Lecture 04: Project Setup

Creating a New Hibernate Project

1. **IDE Setup**: Use IntelliJ IDEA or similar IDE
2. **Project Type**: Maven project for dependency management
3. **Java Version**: Use Java 8 or higher
4. **Build System**: Maven for managing dependencies

Adding Dependencies

Add the following dependencies to your `pom.xml`:

```xml
<dependencies>
  <!-- Hibernate Core -->
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>6.3.Final</version>
  </dependency>

  <!-- PostgreSQL Driver -->
  <dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.7.3</version>
  </dependency>
</dependencies>
```

Key Dependencies Explained

- **hibernate-core**: Main Hibernate functionality
- **postgresql**: Database driver for PostgreSQL connectivity

Lecture 05: Failed Attempt to Save Data

Creating a Student POJO Class

```java
public class Student {
    private int rollNo;
    private String name;
    private int age;

    // Constructors
    public Student() {}

    // Getters and Setters
    public int getRollNo() { return rollNo; }
    public void setRollNo(int rollNo) { this.rollNo = rollNo; }

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

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

Initial Save Attempt (Failed)

```java
public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.setName("Navin");
        s1.setRollNo(101);
        s1.setAge(30);

        Configuration cfg = new Configuration();
        SessionFactory sf = cfg.buildSessionFactory();
        Session session = sf.openSession();

        session.save(s1);  // This will fail

        System.out.println(s1);
    }
}
```

Why This Failed

- No Hibernate configuration file
- Entity class not properly annotated

- No mapping between Java class and database table
- Missing transaction management

Lecture 06: Successful Attempt to Save Data

Important Version Note

Save() Method Deprecation:

- `save()` method was **removed in Hibernate 7.1.0**
- `save()` method was **deprecated in version 6.6.3**
- Use `persist()` method instead for newer versions

Creating Hibernate Configuration File

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

```
<!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 properties -->
        <property name="hibernate.connection.driver_class">org.postgresql.Driver</property>
        <property
name="hibernate.connection.url">jdbc:postgresql://localhost:5432/telusko</property>
        <property name="hibernate.connection.username">postgres</property>
        <property name="hibernate.connection.password">0000</property>

        <!-- Hibernate properties -->
        <property name="hibernate.hbm2ddl.auto">update</property>

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

Updated Student Entity with Annotations

```
import jakarta.persistence.Entity;
import jakarta.persistence.Id;

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

    // Constructors, getters, and setters remain the same
```

TELUSKO

```
}
```

Successful Save Implementation

```
public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.setName("Navin");
        s1.setRollNo(101);
        s1.setAge(30);

        Configuration cfg = new Configuration();
        cfg.addAnnotatedClass(com.telusko.Student.class);
        cfg.configure();

        SessionFactory sf = cfg.buildSessionFactory();
        Session session = sf.openSession();

        Transaction transaction = session.beginTransaction();

        session.save(s1);  // or use session.persist(s1) for newer versions

        transaction.commit();

        System.out.println(s1);
    }
}
```

Key Success Factors

1. **Configuration file**: Properly configured database connection
2. **Entity annotations**: `@Entity` and `@Id` annotations
3. **Class registration**: Adding annotated class to configuration
4. **Transaction management**: Beginning and committing transactions

Lecture 07: Show SQL Configuration

Adding SQL Visibility Properties

Update `hibernate.cfg.xml` to show generated SQL:

```
<property name="hibernate.hbm2ddl.auto">update</property>
<property name="hibernate.show_sql">true</property>

<property name="hibernate.format_sql">true</property>
```

Configuration Properties Explained

- **hibernate.hbm2ddl.auto**:
  - `update`: Creates table if it does not exists, updates if schema changes
  - `create`: Drops and creates table every time
  - `validate`: Only validates the schema
  - `none`: No automatic schema management
- **hibernate.show_sql**: Displays generated SQL in console
- **hibernate.format_sql**: Formats SQL for better readability

Additional Useful Properties

```
<property name="hibernate.dialect">org.hibernate.dialect.PostgreSQLDialect</property>
```

Lecture 08: Refactoring – Using Persist Method

Updated Code with Persist Method

```java
public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.setName("Gaurav");
        s1.setRollNo(105);
        s1.setAge(22);

        Configuration cfg = new Configuration();
        cfg.addAnnotatedClass(com.telusko.Student.class);
        cfg.configure();

        SessionFactory sf = cfg.buildSessionFactory();
        Session session = sf.openSession();

        Transaction transaction = session.beginTransaction();

        session.persist(s1);  // Using persist instead of save

        transaction.commit();

        session.close();
        sf.close();

        System.out.println(s1);
    }
}
```

Persist vs Save Method

- **persist()**: JPA standard method, preferred for new code
- **save()**: Hibernate-specific, deprecated in newer versions
- **Functionality**: Both methods save transient objects to the database.
- **Return value**: save() returns generated ID; persist() doesn't

Best Practices

1. Always use transactions for data modifications
2. Close session and session factory properly
3. Use persist() for better portability
4. Handle exceptions appropriately

Lecture 09: Fetching Data

Reading Data from Database

```
public static void main(String[] args) {
    Student s1 = new Student();
    s1.setName("Arya");
    s1.setRollNo(106);
    s1.setAge(21);

    Student s2 = null;

    SessionFactory sf = new Configuration()
            .addAnnotatedClass(com.telusko.Student.class)
            .configure()
            .buildSessionFactory();

    Session session = sf.openSession();

    s2 = session.get(Student.class, 102);  // Fetch student with rollNo 102

    session.close();
    sf.close();

    System.out.println(s2);
}
```

Get vs Load Methods

- **get()**: Returns null if object not found, eager loading
- **load()**: Throws exception if not found, lazy loading
- **Usage**: get() is more commonly used and safer

Lecture 10: Update and Delete Operations

Update Operation

```
public static void main(String[] args) {
    Student s1 = new Student();
    s1.setName("Harsh");
    s1.setRollNo(103);
    s1.setAge(25);

    SessionFactory sf = new Configuration()
            .addAnnotatedClass(com.telusko.Student.class)
            .configure()
            .buildSessionFactory();

    Session session = sf.openSession();

    session.merge(s1);  // Updates existing record or inserts new one

    session.close();
    sf.close();

    System.out.println(s1);
}
```

Delete Operation

```
public static void main(String[] args) {
    SessionFactory sf = new Configuration()
            .addAnnotatedClass(com.telusko.Student.class)
            .configure()
            .buildSessionFactory();

    Session session = sf.openSession();

    Student s1 = session.get(Student.class, 109);  // First fetch the object

    Transaction transaction = session.beginTransaction();

    session.remove(s1);  // Delete the object

    transaction.commit();

    session.close();
    sf.close();
}
```

Important Version Notes

- **get() method**: Deprecated in Hibernate 7.1.0 but **not removed**
- **remove()**: Use instead of delete() for JPA compliance
- **merge()**: Use instead of update() or saveOrUpdate()

Lecture 11: Changing Table and Column Names

Custom Table Name

```
@Entity
@Table(name="alien_table")
public class Alien {
    @Id
    private int aid;
    private String aname;
    private String tech;

    // getters and setters
}
```

Custom Column Names

```
@Entity
public class Alien {
    @Id
    private int aid;

    @Column(name="alien_name")
    private String aname;

    @Transient
    private String tech;

    // getters and setters
}
```

Annotation Explanations

- **@Entity**: Marks class as JPA entity
- **@Table(name="custom_name")**: Specifies custom table name
- **@Id**: Marks primary key field
- **@Transient**: Ignore the field in data base
- **@Column(name="custom_name")**: Specifies custom column name

Entity vs Table Differences

- **Entity**: Java class representation
- **Table**: Database table representation
- **Mapping**: Annotations bridge the gap between object and relational models
- **Flexibility**: Can have different names for class/field vs table/column

Lecture 12: Embeddables - Complex Data Types

Creating an Embeddable Class

```
@Embeddable
public class Laptop {
    private String brand;
    private String model;
    private int ram;

    // Constructors
    public Laptop() {}

    // Getters and Setters
    public String getBrand() { return brand; }
    public void setBrand(String brand) { this.brand = brand; }

    public String getModel() { return model; }
    public void setModel(String model) { this.model = model; }

    public int getRam() { return ram; }
    public void setRam(int ram) { this.ram = ram; }
}
```

Using Embeddable in Entity

```
@Entity
public class Alien {
    @Id
    private int aid;
    private String aname;
    private String tech;
    private Laptop laptop;  // Embedded object

    // Constructors, getters, and setters
}
```

Key Concepts

- **@Embeddable**: Marks a class as embeddable (value type)
- **Composition**: Alien "has-a" Laptop relationship
- **Table Structure**: Laptop fields become columns in Alien table
- **No Separate Table**: Embeddable objects don't get their own table

Benefits of Embeddables

1. **Code Organization**: Group related fields together
2. **Reusability**: Same embeddable can be used in multiple entities
3. **Type Safety**: Better than using primitive types for complex data
4. **Maintenance**: Changes to embeddable affect all using entities

Summary of Key Hibernate Concepts

Core Annotations

- `@Entity`: Marks a class as a JPA entity
- `@Id`: Designates the primary key
- `@Table`: Specifies table name and properties
- `@Column`: Specifies column properties
- `@Embeddable`: Marks a class as embeddable

Session Methods (Modern Approach)

- `persist()`: Save new entity
- `merge()`: Update existing or save new
- `get()`: Retrieve by primary key
- `remove()`: Delete entity

Configuration Essentials

1. Database connection properties
2. Hibernate-specific properties
3. Entity class registration
4. Transaction management

Best Practices

1. Always use transactions for modifications
2. Close resources properly
3. Use JPA standard methods when possible
4. Handle exceptions appropriately
5. Configure logging for development