https://bytemartdigital.in/

{code}

API

Hibernate By ByteMart

Copyright and Disclaimer

Hibernate by ByteMart

Copyright Notice

Disclaimer

This book is provided "as is" without any guarantees or warranties, express or implied. While every effort has been made to ensure accuracy and completeness, ByteMart makes no representations regarding the currentness, performance, or applicability of the information. The contents are intended for **educational purposes only**. As technologies such as Hibernate are constantly evolving, readers are advised to consult the official documentation for the most up-to-date information and adapt the examples accordingly.

Legal and Ethical Use Guidelines

- Reproduction, republication, or unauthorized distribution of this book, in part or in full, is **strictly prohibited**.
- Sharing this content on unauthorized platforms or using it for commercial purposes without written permission is **illegal**.
- Using code or content from this book for **unethical or unlawful activities** is not permitted.

⚠️ **Warning**

This book is the result of significant expertise and effort. Any act of intellectual property theft—including plagiarism, unauthorized redistribution, or content misuse—will result in **immediate legal action** under applicable copyright and IP laws.
ByteMart assumes no responsibility for any loss, damage, or consequences resulting from the use of the information or code provided in this publication.

By Accessing or Using This Book, You Agree To:

1. Respect the intellectual property rights of ByteMart.
2. Use the content responsibly and ethically.
3. Refrain from any activity that violates the copyright terms outlined in this document.

ByteMart reserves the right to pursue **strict legal action** against any unauthorized use, duplication, or distribution of this material.
Proceed responsibly.

Contact Information

For permissions, collaborations, or support inquiries:

- **Email:** contact@bytemartdigital.in
- **Website:** https://bytemartdigital.in

Hibernate

Table of Contents

- Transaction Concurrency Problem & Solutions
- Types of Transactions

16. Hibernate Connection Management
    - JDBC Transaction
    - JTA Transaction
    - CMT Transaction
17. Hibernate Architecture
    - Exploring SessionFactory
    - Object States
18. Introduction of Hibernate Cache
    - Different types of Hibernate Cache
    - Hibernate Cache Architecture

Chapter 1: Introduction to Hibernate

11 What is Hibernate?

Hibernate is an open-source Object-Relational Mapping (ORM) framework for Java. It simplifies the development of Java applications to interact with databases. By mapping Java classes to database tables, Hibernate takes care of the underlying SQL complexities and allows developers to work with higher-level objects.

12 Why Use Hibernate?

Advantages of Hibernate

1. **Simplicity**: Hibernate abstracts the database interactions, allowing developers to work with Java objects instead of SQL queries.
2. **Productivity**: It reduces the boilerplate code for database operations.
3. **Portability**: Hibernate supports multiple databases, making it easier to switch from one database to another.

4. **Performance**: It includes caching mechanisms that enhance application performance.
5. **Maintainability**: Changes in the database schema can be managed with minimal changes in the application code.
6. **Scalability**: It supports large-scale enterprise applications with high transaction volumes.

How Hibernate Works

Hibernate operates through a set of configuration files and Java classes. The configuration file specifies the database connection settings, and mapping files define how Java classes are mapped to database tables. Hibernate provides APIs to perform CRUD (Create, Read, Update, Delete) operations, transactions, and query execution.

13 Core Components of Hibernate

SessionFactory

The `SessionFactory` is a factory for `Session` objects. It holds the configuration details (like database connection information) and is responsible for initializing Hibernate.

Session

The `Session` object represents a single unit of work with the database. It is used to create, read, and delete persistent objects.

Transaction

The `Transaction` interface is used to manage transactions in Hibernate. It abstracts the underlying transaction management of the database.

Query

The `Query` interface allows the execution of HQL (Hibernate Query Language) or SQL queries against the database.

Criteria

The `Criteria` API is a more object-oriented way to create and execute queries.

Configuration

The `Configuration` class represents the entire configuration for Hibernate. It is used to configure the `SessionFactory`.

14 Key Features of Hibernate

1. **Automatic Table Creation**: Hibernate can automatically generate database tables based on Java classes.
2. **Hibernate Query Language (HQL)**: HQL is similar to SQL but works with Java objects instead of tables.
3. **Criteria Queries**: Allows the creation of queries using Java-based criteria.
4. **Caching**: Hibernate supports both first-level cache (Session cache) and second-level cache (SessionFactory cache).
5. **Lazy Loading**: Data is fetched on-demand, reducing the initial load time.
6. **Eager Loading**: Data is fetched immediately, useful for retrieving associated objects.
7. **Transaction Management**: Provides built-in support for managing transactions.
8. **Automatic Dirty Checking**: Changes in the state of an object are automatically detected and synchronized with the database.
9. **Association Mapping**: Supports various types of associations like one-to-one, one-to-many, and many-to-many relationships.
10. **Inheritance Mapping**: Provides different strategies for mapping inheritance hierarchies to database tables.

15 Setting Up Hibernate

Prerequisites

- Java Development Kit (JDK)
- Integrated Development Environment (IDE) like Eclipse or IntelliJ IDEA
- Database (e.g., MySQL, PostgreSQL)
- Maven or Gradle for dependency management

Step-by-Step Setup

1. Add Hibernate Dependencies

If using Maven, add the following dependencies to your `pom.xml`:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.4.32.Final</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>5.4.32.Final</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.23</version>
</dependency>
```

2. Create Configuration File (`hibernate.cfg.xml`)

```
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibe
rnate Configuration DTD 3.0//EN"
        "<http://hibernate.sourceforge.net/hibernate-config
uration-3.0.dtd>">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.dialect">org.hibernate.di
alect.MySQLDialect</property>
        <property name="hibernate.connection.driver_class">
com.mysql.cj.jdbc.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysq
l://localhost:3306/your_database</property>
        <property name="hibernate.connection.username">your
_username</property>
        <property name="hibernate.connection.password">your
_password</property>
```

```
        <property name="hibernate.hbm2ddl.auto">update</pro
perty>
        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.format_sql">true</propert
y>

        <!-- Mapping files -->
        <mapping resource="com/example/model/YourModel.hbm.
xml"/>
    </session-factory>
</hibernate-configuration>
```

3. Create a Model Class

```
package com.example.model;

public class Employee {
    private int id;
    private String name;
    private String department;
    private double salary;

    // Getters and setters
}
```

4. Create a Mapping File (`Employee.hbm.xml`)

```
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate
Mapping DTD 3.0//EN"
        "<http://hibernate.sourceforge.net/hibernate-mappin
g-3.0.dtd>">
<hibernate-mapping>
    <class name="com.example.model.Employee" table="EMPLOYE
E">
        <id name="id" column="ID">
            <generator class="native"/>
        </id>
```

```
        <property name="name" column="NAME"/>
        <property name="department" column="DEPARTMENT"/>
        <property name="salary" column="SALARY"/>
    </class>
</hibernate-mapping>
```

5. Write Hibernate Utility Class

```
package com.example.util;

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

public class HibernateUtil {
    private static final SessionFactory sessionFactory;

    static {
        try {
            sessionFactory = new Configuration().configure
().buildSessionFactory();
        } catch (Throwable ex) {
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

6. Create a Main Class to Test Hibernate

```
package com.example.main;

import com.example.model.Employee;
import com.example.util.HibernateUtil;
import org.hibernate.Session;
```

```
import org.hibernate.Transaction;

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

        try {
            transaction = session.beginTransaction();

            // Create an employee object
            Employee employee = new Employee();
            employee.setName("John Doe");
            employee.setDepartment("Engineering");
            employee.setSalary(75000);

            // Save the employee object
            session.save(employee);

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

16 Summary

Hibernate is a powerful ORM tool that simplifies Java application development by handling database interactions efficiently. It abstracts complex SQL queries into high-level Java objects, offers robust transaction management, supports various types of caching, and provides a rich query language (HQL). Setting up

Hibernate involves adding dependencies, configuring the `hibernate.cfg.xml` file, creating model classes, and writing mapping files.

By using Hibernate, developers can focus more on business logic rather than dealing with intricate database operations, leading to more maintainable and scalable applications.

Chapter 2: Difference between JDBC and Hibernate

21 Introduction

Java applications often need to interact with databases to store, retrieve, and manipulate data. The two primary methods for this interaction are JDBC (Java Database Connectivity) and Hibernate. Both have their advantages and specific use cases. This chapter will explore the key differences between JDBC and Hibernate.

22 Overview of JDBC

What is JDBC?

JDBC (Java Database Connectivity) is a Java-based API that enables Java applications to interact with databases. It provides methods to execute SQL queries and retrieve results. JDBC is a low-level API that requires detailed handling of SQL and database connections.

Key Features of JDBC

1. **Direct SQL Handling**: Developers write SQL queries directly.
2. **Database Connectivity**: Manages connections to the database using drivers.
3. **ResultSet Management**: Handles the retrieval and manipulation of query results.
4. **Transaction Management**: Manages transactions manually.
5. **Batch Processing**: Supports executing multiple queries in a single batch.

Example of JDBC Usage

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class JDBCExample {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/your_data
base";
        String username = "your_username";
        String password = "your_password";

        try (Connection connection = DriverManager.getConne
ction(url, username, password)) {
            String query = "SELECT * FROM employees";
            PreparedStatement statement = connection.prepar
eStatement(query);
            ResultSet resultSet = statement.executeQuery();

            while (resultSet.next()) {
                System.out.println("ID: " + resultSet.getIn
t("id"));
                System.out.println("Name: " + resultSet.get
String("name"));
                System.out.println("Department: " + resultS
et.getString("department"));
                System.out.println("Salary: " + resultSet.g
etDouble("salary"));
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

23 Overview of Hibernate

What is Hibernate?

Hibernate is an open-source ORM (Object-Relational Mapping) framework for Java. It simplifies database interactions by mapping Java classes to database tables. Hibernate abstracts the underlying SQL, allowing developers to work with high-level Java objects.

Key Features of Hibernate

1. **Object-Relational Mapping (ORM)**: Maps Java objects to database tables.
2. **Hibernate Query Language (HQL)**: A query language similar to SQL but works with Java objects.
3. **Automatic Table Creation**: Can generate database tables based on Java classes.
4. **Caching**: Supports first-level (session) and second-level (session factory) caching.
5. **Lazy and Eager Loading**: Controls the loading behavior of associated objects.
6. **Transaction Management**: Provides built-in transaction management.
7. **Automatic Dirty Checking**: Automatically synchronizes object state changes with the database.
8. **Association and Inheritance Mapping**: Supports various relationships and inheritance hierarchies.

Example of Hibernate Usage

```
package com.example;

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

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

```
        Transaction transaction = null;

        try {
            transaction = session.beginTransaction();

            Employee employee = new Employee();
            employee.setName("John Doe");
            employee.setDepartment("Engineering");
            employee.setSalary(75000);

            session.save(employee);

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

24 Key Differences Between JDBC and Hibernate

Feature/Aspect	JDBC	Hibernate
API Level	Low-level API for database interaction	High-level ORM framework
SQL Handling	Developers write and manage SQL directly	Hibernate abstracts SQL into HQL/Criteria queries
Database Independence	Tightly coupled with specific SQL dialects	Supports multiple databases with minimal changes
Object Mapping	Manual mapping between tables and Java objects	Automatic mapping between tables and Java objects
Transaction Management	Manual transaction handling	Built-in transaction management

Caching	No built-in caching	First-level and second-level caching
Lazy Loading	Not supported	Supported
Eager Loading	Not supported	Supported
Schema Generation	Manual	Can generate based on mapping files
Maintenance	More code to maintain due to SQL management	Less code, easier to maintain
Performance Tuning	Requires manual tuning	Built-in performance tuning mechanisms
Batch Processing	Supported	Supported
Tooling	Requires external tools for ORM features	Provides comprehensive ORM tooling
Learning Curve	Easier to learn for SQL-savvy developers	Steeper learning curve due to additional abstraction

25 When to Use JDBC

- Simple applications with straightforward database interactions.
- When you need fine-grained control over SQL and database operations.
- Applications where performance tuning at the SQL level is critical.
- Projects with short timelines that require immediate implementation.

26 When to Use Hibernate

- Complex applications with multiple relationships and entities.
- Applications requiring database independence and portability.
- When you want to leverage ORM features like caching, lazy loading, and automatic table creation.
- Projects where maintainability and scalability are crucial.

27 Summary

JDBC and Hibernate are both powerful tools for database interaction in Java applications, each with its own strengths and use cases. JDBC offers low-level control and is suitable for simple applications and scenarios requiring direct

SQL management. Hibernate, on the other hand, provides a high-level abstraction that simplifies database operations through ORM, making it ideal for complex and large-scale applications.

Understanding the differences between JDBC and Hibernate helps developers choose the right tool for their specific needs, balancing control, simplicity, performance, and maintainability.

Chapter 3: Hibernate Features

3.1 Introduction

Hibernate is a comprehensive Object-Relational Mapping (ORM) framework for Java that simplifies database interactions by mapping Java classes to database tables. It provides a plethora of features that enhance productivity, performance, and maintainability. This chapter explores the key features of Hibernate in detail.

3.2 Key Features of Hibernate

3.2.1 Object-Relational Mapping (ORM)

ORM is the core feature of Hibernate. It allows developers to map Java classes to database tables and vice versa. This abstraction hides the complexities of database interactions, enabling developers to work with Java objects instead of raw SQL queries.

3.2.2 Hibernate Query Language (HQL)

HQL is a powerful query language provided by Hibernate, similar to SQL but designed to work with Hibernate's entity model. HQL queries are written in terms of the object model and are automatically translated into SQL by Hibernate.

Example:

```
String hql = "FROM Employee E WHERE E.salary > :salary";
Query query = session.createQuery(hql);
```

```
query.setParameter("salary", 50000);
List<Employee> results = query.list();
```

3.2.3 Criteria API

The Criteria API offers a programmatic way to create queries. It is a more object-oriented alternative to HQL, providing a flexible and type-safe way to build queries.

Example:

```
CriteriaBuilder builder = session.getCriteriaBuilder();
CriteriaQuery<Employee> query = builder.createQuery(Employe
e.class);
Root<Employee> root = query.from(Employee.class);
query.select(root).where(builder.greaterThan(root.get("sala
ry"), 50000));
List<Employee> results = session.createQuery(query).getResu
ltList();
```


3.2.4 Automatic Table Creation

Hibernate can automatically generate database tables based on the mapping of Java classes. This feature simplifies schema management and ensures that the database schema is always in sync with the application code.

Example Configuration (`hibernate.cfg.xml`):

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

3.2.5 Caching

Hibernate supports both first-level (Session) and second-level (SessionFactory) caching to improve performance by reducing database access.

- **First-Level Cache**: A built-in cache associated with the Session object. It is enabled by default.
- **Second-Level Cache**: A configurable cache shared among sessions, which can be provided by third-party caching solutions like EHCache or

Infinispan.

Example Configuration for Second-Level Cache (`hibernate.cfg.xml`):

```
<property name="hibernate.cache.use_second_level_cache">tru
e</property>
<property name="hibernate.cache.region.factory_class">org.h
ibernate.cache.ehcache.EhCacheRegionFactory</property>
```

3.2.6 Lazy and Eager Loading

Hibernate provides mechanisms to control the loading behavior of associated entities.

- **Lazy Loading**: Associated entities are loaded on demand.
- **Eager Loading**: Associated entities are loaded immediately with the parent entity.

Example:

```
@Entity
public class Department {
    @OneToMany(fetch = FetchType.LAZY, mappedBy = "departme
nt")
    private Set<Employee> employees;
}
```

3.2.7 Transaction Management

Hibernate offers robust transaction management, abstracting the underlying transaction mechanisms of the database. It supports both programmatic and declarative transaction management.

Example:

```
Transaction transaction = session.beginTransaction();
try {
    // Perform database operations
    transaction.commit();
} catch (Exception e) {
    transaction.rollback();
```

```
    throw e;
}
```

3.28 Automatic Dirty Checking

Hibernate automatically detects changes made to persistent objects and synchronizes these changes with the database, reducing the need for explicit save or update calls.

Example:

```
session.beginTransaction();
Employee employee = session.get(Employee.class, 1);
employee.setSalary(60000);
session.getTransaction().commit();
```

3.29 Association and Inheritance Mapping

Hibernate supports various types of associations (one-to-one, one-to-many, many-to-many) and inheritance strategies (single table, table per class, joined tables).

- **One-to-Many Association**:

```
@Entity
public class Department {
    @OneToMany(mappedBy = "department")
    private Set<Employee> employees;
}

@Entity
public class Employee {
    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;
}
```

- **Inheritance Mapping**:

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Person {
    @Id
    private int id;
    private String name;
}

@Entity
public class Employee extends Person {
    private double salary;
}
```

3.2.10 HQL and Native SQL Queries

Hibernate allows the execution of both HQL and native SQL queries, providing flexibility for complex query requirements.

- **HQL Query**:

```
String hql = "FROM Employee E WHERE E.department.name = :de
ptName";
Query query = session.createQuery(hql);
query.setParameter("deptName", "Engineering");
List<Employee> results = query.list();
```

- **Native SQL Query**:

```
String sql = "SELECT * FROM EMPLOYEE WHERE DEPARTMENT_ID =
:deptId";
SQLQuery query = session.createSQLQuery(sql);
query.addEntity(Employee.class);
query.setParameter("deptId", 1);
List<Employee> results = query.list();
```

3.3 Summary

Hibernate offers a rich set of features that simplify database interactions, enhance performance, and improve maintainability. Key features like ORM, HQL, Criteria API, automatic table creation, caching, lazy and eager loading, transaction management, automatic dirty checking, association and inheritance mapping, and support for HQL and native SQL queries make Hibernate a powerful framework for enterprise-level applications. By leveraging these features, developers can build robust, scalable, and efficient applications with ease.

Chapter 4: First Example of Hibernate using `hbm.xml` File

4.1 Introduction

This chapter provides a step-by-step guide to creating a simple Hibernate application using an `hbm.xml` file for mapping Java classes to database tables. We'll create a basic example involving an `Employee` entity and demonstrate how to configure Hibernate, create mapping files, and perform CRUD operations.

4.2 Prerequisites

Before starting, ensure you have the following installed:

- Java Development Kit (JDK)
- Integrated Development Environment (IDE) like Eclipse or IntelliJ IDEA
- Maven or Gradle for dependency management
- A database (e.g., MySQL)

4.3 Step-by-Step Guide

Step 1: Set Up the Project

Create a new Maven project in your IDE. Add the required dependencies to your `pom.xml` file:

```
<dependencies>
    <dependency>
        <groupId>org.hibernate</groupId>
```

```
        <artifactId>hibernate-core</artifactId>
        <version>5.4.32.Final</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>5.4.32.Final</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.23</version>
    </dependency>
</dependencies>
```

Step 2: Create the Hibernate Configuration File

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

```
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibe
rnate Configuration DTD 3.0//EN"
        "<http://hibernate.sourceforge.net/hibernate-config
uration-3.0.dtd>">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.dialect">org.hibernate.di
alect.MySQLDialect</property>
        <property name="hibernate.connection.driver_class">
com.mysql.cj.jdbc.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysq
l://localhost:3306/your_database</property>
        <property name="hibernate.connection.username">your
_username</property>
        <property name="hibernate.connection.password">your
_password</property>
        <property name="hibernate.hbm2ddl.auto">update</pro
perty>
        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.format_sql">true</propert
```

```
y>

        <!-- Mapping files -->
        <mapping resource="com/example/model/Employee.hbm.x
ml"/>
    </session-factory>
</hibernate-configuration>
```

Step 3: Create the Model Class

Create a simple Java class representing the `Employee` entity. Save it in the `src/main/java/com/example/model` directory:

```
package com.example.model;

public class Employee {
    private int id;
    private String name;
    private String department;
    private double salary;

    // Getters and setters
    public int getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

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

    public String getDepartment() {
```

```
        return department;
    }

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

    public double getSalary() {
        return salary;
    }

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

Step 4: Create the Mapping File

Create an XML mapping file named `Employee.hbm.xml` in the `src/main/resources/com/example/model` directory:

```
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate
Mapping DTD 3.0//EN"
        "<http://hibernate.sourceforge.net/hibernate-mappin
g-3.0.dtd>">
<hibernate-mapping>
    <class name="com.example.model.Employee" table="EMPLOYE
E">
        <id name="id" column="ID">
            <generator class="native"/>
        </id>
        <property name="name" column="NAME"/>
        <property name="department" column="DEPARTMENT"/>
        <property name="salary" column="SALARY"/>
    </class>
</hibernate-mapping>
```

Step 5: Create Hibernate Utility Class

Create a utility class to initialize Hibernate and provide the `SessionFactory`. Save it in the `src/main/java/com/example/util` directory:

```
package com.example.util;

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

public class HibernateUtil {
    private static final SessionFactory sessionFactory;

    static {
        try {
            sessionFactory = new Configuration().configure
().buildSessionFactory();
        } catch (Throwable ex) {
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

Step 6: Create the Main Class

Create a main class to test Hibernate operations. Save it in the `src/main/java/com/example/main` directory:

```
package com.example.main;

import com.example.model.Employee;
import com.example.util.HibernateUtil;
import org.hibernate.Session;
import org.hibernate.Transaction;

public class HibernateExample {
    public static void main(String[] args) {
```

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

        try {
            transaction = session.beginTransaction();

            // Create an employee object
            Employee employee = new Employee();
            employee.setName("John Doe");
            employee.setDepartment("Engineering");
            employee.setSalary(75000);

            // Save the employee object
            session.save(employee);

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

44 Running the Application

To run the application, ensure your database server is running and that the database specified in the `hibernate.cfg.xml` file exists. Run the `HibernateExample` class, and you should see Hibernate SQL output in the console, indicating that the `Employee` object was successfully saved to the database.

45 Summary

This chapter demonstrated a simple example of using Hibernate with an `hbm.xml` mapping file. We covered setting up the project, creating configuration and

mapping files, and performing basic CRUD operations. By following these steps, you can create and run a Hibernate application that maps Java classes to database tables, leveraging Hibernate's powerful ORM capabilities.

Chapter 5: First Example of Hibernate using Annotations

5.1 Introduction

In this chapter, we will explore how to use annotations for mapping Java classes to database tables with Hibernate. Annotations provide a cleaner and more readable way to define the mapping compared to XML configuration files. We will create a simple example involving an `Employee` entity and demonstrate how to configure Hibernate, use annotations for mapping, and perform CRUD operations.

5.2 Prerequisites

Ensure you have the following installed:

- Java Development Kit (JDK)
- Integrated Development Environment (IDE) like Eclipse or IntelliJ IDEA
- Maven or Gradle for dependency management
- A database (e.g., MySQL)

5.3 Step-by-Step Guide

Step 1: Set Up the Project

Create a new Maven project in your IDE. Add the required dependencies to your `pom.xml` file:

```
<dependencies>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.4.32.Final</version>
    </dependency>
```

```
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>5.4.32.Final</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.23</version>
    </dependency>
</dependencies>
```

Step 2: Create the Hibernate Configuration File

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

```
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibe
rnate Configuration DTD 3.0//EN"
        "<http://hibernate.sourceforge.net/hibernate-config
uration-3.0.dtd>">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.dialect">org.hibernate.di
alect.MySQLDialect</property>
        <property name="hibernate.connection.driver_class">
com.mysql.cj.jdbc.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysq
l://localhost:3306/your_database</property>
        <property name="hibernate.connection.username">your
_username</property>
        <property name="hibernate.connection.password">your
_password</property>
        <property name="hibernate.hbm2ddl.auto">update</pro
perty>
        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.format_sql">true</propert
y>

        <!-- Annotated classes -->
```

```
        <mapping class="com.example.model.Employee"/>
    </session-factory>
</hibernate-configuration>
```

Step 3: Create the Model Class with Annotations

Create a Java class representing the `Employee` entity using annotations. Save it in the `src/main/java/com/example/model` directory:

```
package com.example.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "EMPLOYEE")
public class Employee {

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

    private String name;

    private String department;

    private double salary;

    // Getters and setters
    public int getId() {
        return id;
    }

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

```
    public String getName() {
        return name;
    }

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

    public String getDepartment() {
        return department;
    }

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

    public double getSalary() {
        return salary;
    }

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

Step 4: Create Hibernate Utility Class

Create a utility class to initialize Hibernate and provide the `SessionFactory`. Save it in the `src/main/java/com/example/util` directory:

```
package com.example.util;

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

public class HibernateUtil {
    private static final SessionFactory sessionFactory;
```

```
    static {
        try {
            sessionFactory = new Configuration().configure
().buildSessionFactory();
        } catch (Throwable ex) {
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

Step 5: Create the Main Class

Create a main class to test Hibernate operations. Save it in the `src/main/java/com/example/main` directory:

```
package com.example.main;

import com.example.model.Employee;
import com.example.util.HibernateUtil;
import org.hibernate.Session;
import org.hibernate.Transaction;

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

        try {
            transaction = session.beginTransaction();

            // Create an employee object
            Employee employee = new Employee();
            employee.setName("John Doe");
            employee.setDepartment("Engineering");
```

```
            employee.setSalary(75000);

            // Save the employee object
            session.save(employee);

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

54 Running the Application

To run the application, ensure your database server is running and that the database specified in the `hibernate.cfg.xml` file exists. Run the `HibernateExample` class, and you should see Hibernate SQL output in the console, indicating that the `Employee` object was successfully saved to the database.

55 Summary

This chapter demonstrated how to use annotations with Hibernate to map Java classes to database tables. We covered setting up the project, creating configuration files, defining the entity with annotations, and performing basic CRUD operations. Using annotations simplifies the configuration process and enhances the readability of the mapping, making it easier to maintain and manage.

Chapter 6: Hibernate Inheritance Mapping

Hibernate supports inheritance mapping, allowing you to map a class hierarchy to database tables. There are several strategies to achieve this, including:

- Table Per Class Mapping
- Table Per Concrete Class Mapping

In this chapter, we will discuss these two strategies in detail and provide examples for each.

6.1 Table Per Class Mapping

6.1.1 Introduction

In the Table Per Class strategy, each class in the inheritance hierarchy is mapped to its own database table. The table for a subclass includes columns for its own fields and the fields inherited from its superclass.

6.1.2 Example

Consider an inheritance hierarchy with a base class `Person` and two subclasses `Employee` and `Customer`.

Step 1: Create the Entity Classes

Person.java

```
package com.example.model;

import javax.persistence.*;

@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    private String name;

    // Getters and setters
    public int getId() {
        return id;
    }

    public void setId(int id) {
```

```
        this.id = id;
    }

    public String getName() {
        return name;
    }

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

Employee.java

```
package com.example.model;

import javax.persistence.Entity;

@Entity
public class Employee extends Person {
    private String department;
    private double salary;

    // Getters and setters
    public String getDepartment() {
        return department;
    }

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

    public double getSalary() {
        return salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
```

```
    }
}
```

Customer.java

```
package com.example.model;

import javax.persistence.Entity;

@Entity
public class Customer extends Person {
    private String loyaltyLevel;

    // Getters and setters
    public String getLoyaltyLevel() {
        return loyaltyLevel;
    }

    public void setLoyaltyLevel(String loyaltyLevel) {
        this.loyaltyLevel = loyaltyLevel;
    }
}
```

Step 2: Create Hibernate Configuration

Add the annotated classes to your `hibernate.cfg.xml`:

```
<mapping class="com.example.model.Person"/>
<mapping class="com.example.model.Employee"/>
<mapping class="com.example.model.Customer"/>
```

Step 3: Create Hibernate Utility Class

We will use the same `HibernateUtil` class created in previous examples.

Step 4: Main Class to Test the Mapping

HibernateInheritanceExample.java

```
package com.example.main;

import com.example.model.Customer;
import com.example.model.Employee;
import com.example.util.HibernateUtil;
import org.hibernate.Session;
import org.hibernate.Transaction;

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

        try {
            transaction = session.beginTransaction();

            // Create an employee object
            Employee employee = new Employee();
            employee.setName("John Doe");
            employee.setDepartment("Engineering");
            employee.setSalary(75000);

            // Create a customer object
            Customer customer = new Customer();
            customer.setName("Jane Doe");
            customer.setLoyaltyLevel("Gold");

            // Save the objects
            session.save(employee);
            session.save(customer);

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

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

6.1.3 Table Structure

- `Person` table contains columns:`id`, `name`
- `Employee` table contains columns: `id`, `name`, `department`, `salary`
- `Customer` table contains columns: `id`, `name`, `loyaltyLevel`

6.2 Table Per Concrete Class Mapping

6.2.1 Introduction

In the Table Per Concrete Class strategy, each concrete class in the inheritance hierarchy is mapped to its own table. These tables contain all fields of the class, including inherited fields.

6.2.2 Example

Using the same `Person`, `Employee`, and `Customer` classes as before, we will now map them using the Table Per Concrete Class strategy.

Step 1: Modify the Entity Classes

Update the `Person` class to use `InheritanceType.TABLE_PER_CLASS`:

Person.java

```
package com.example.model;

import javax.persistence.*;

@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
```

```
    private String name;

    // Getters and setters
    public int getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

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

Employee.java and **Customer.java** remain the same as in the previous example.

Step 2: Update Hibernate Configuration

The `hibernate.cfg.xml` configuration remains the same as in the previous example.

Step 3: Main Class to Test the Mapping

The `HibernateInheritanceExample.java` class remains the same as in the previous example.

623 Table Structure

- `Employee` table contains columns: `id`, `name`, `department`, `salary`
- `Customer` table contains columns: `id`, `name`, `loyaltyLevel`

The `Person` class is not directly mapped to a table in this strategy.

63 Summary

This chapter covered two inheritance mapping strategies in Hibernate:

- **Table Per Class Mapping**: Each class in the hierarchy has its own table.
- **Table Per Concrete Class Mapping**: Each concrete class has its own table, and inherited fields are included in each table.

We provided examples to illustrate both strategies. By using these strategies, you can effectively map complex class hierarchies to relational database tables, leveraging Hibernate's powerful ORM capabilities to manage inheritance in your applications.

Chapter 7: Association Mapping

Hibernate supports various types of associations between entities. This chapter explores different association mappings:

- One-to-One Uni-directional Mapping
- One-to-One Bi-directional Mapping
- Many-to-Many Mapping

We will provide examples and code for each type of association.

7.1 One-to-One Uni-directional Mapping

7.1.1 Introduction

In a one-to-one uni-directional mapping, one entity has a reference to another entity, and the association is only navigable in one direction.

7.1.2 Example

Consider an example where each `Person` has one `Address`.

Step 1: Create the Entity Classes

Person.java

```
package com.example.model;

import javax.persistence.*;

@Entity
```

```
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;

    @OneToOne
    @JoinColumn(name = "address_id")
    private Address address;

    // Getters and setters
    public int getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

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

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }
}
```

Address.java

```
package com.example.model;

import javax.persistence.*;

@Entity
public class Address {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String street;
    private String city;

    // Getters and setters
    public int getId() {
        return id;
    }

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

    public String getStreet() {
        return street;
    }

    public void setStreet(String street) {
        this.street = street;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }
}
```

Step 2: Create Hibernate Configuration

Add the annotated classes to your `hibernate.cfg.xml`:

```
<mapping class="com.example.model.Person"/>
<mapping class="com.example.model.Address"/>
```

Step 3: Create Hibernate Utility Class

The `HibernateUtil` class remains the same as in previous examples.

Step 4: Main Class to Test the Mapping

HibernateOneToOneUniExample.java

```
package com.example.main;

import com.example.model.Address;
import com.example.model.Person;
import com.example.util.HibernateUtil;
import org.hibernate.Session;
import org.hibernate.Transaction;

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

        try {
            transaction = session.beginTransaction();

            // Create an address object
            Address address = new Address();
            address.setStreet("123 Main St");
            address.setCity("Springfield");

            // Create a person object
            Person person = new Person();
            person.setName("John Doe");
```

```
            person.setAddress(address);

            // Save the objects
            session.save(address);
            session.save(person);

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

713 Table Structure

- `Person` table contains columns: `id`, `name`, `address_id`
- `Address` table contains columns: `id`, `street`, `city`

72 One-to-One Bi-directional Mapping

721 Introduction

In a one-to-one bi-directional mapping, both entities have references to each other, and the association is navigable in both directions.

722 Example

Using the same `Person` and `Address` classes, we will now make the association bi-directional.

Step 1: Update the Entity Classes

Person.java

```
package com.example.model;

import javax.persistence.*;

@Entity
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;

    @OneToOne(mappedBy = "person")
    private Address address;

    // Getters and setters
    public int getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

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

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
```

```
    }
}
```

Address.java

```
package com.example.model;

import javax.persistence.*;

@Entity
public class Address {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String street;
    private String city;

    @OneToOne
    @JoinColumn(name = "person_id")
    private Person person;

    // Getters and setters
    public int getId() {
        return id;
    }

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

    public String getStreet() {
        return street;
    }

    public void setStreet(String street) {
        this.street = street;
    }
```

```
    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public Person getPerson() {
        return person;
    }

    public void setPerson(Person person) {
        this.person = person;
    }
}
```

Step 2: Update Hibernate Configuration

The `hibernate.cfg.xml` configuration remains the same.

Step 3: Main Class to Test the Mapping

HibernateOneToOneBiExample.java

```
package com.example.main;

import com.example.model.Address;
import com.example.model.Person;
import com.example.util.HibernateUtil;
import org.hibernate.Session;
import org.hibernate.Transaction;

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

```
        try {
            transaction = session.beginTransaction();

            // Create an address object
            Address address = new Address();
            address.setStreet("123 Main St");
            address.setCity("Springfield");

            // Create a person object
            Person person = new Person();
            person.setName("John Doe");
            person.setAddress(address);
            address.setPerson(person);

            // Save the objects
            session.save(address);
            session.save(person);

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

723 Table Structure

- `Person` table contains columns: `id`, `name`
- `Address` table contains columns: `id`, `street`, `city`, `person_id`

73 Many-to-Many Mapping

731 Introduction

In a many-to-many mapping, multiple instances of one entity are associated with multiple instances of another entity. This type of association typically requires a join table.

732 Example

Consider an example where `Student` and `Course` entities have a many-to-many relationship.

Step 1: Create the Entity Classes

Student.java

```
package com.example.model;

import javax.persistence.*;
import java.util.HashSet;
import java.util.Set;

@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;

    @ManyToMany
    @JoinTable(
        name = "student_course",
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "course_i
d")
    )
    private Set<Course> courses = new HashSet<>();

    // Getters and setters
    public int getId() {
        return id;
    }
```

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

    public String getName() {
        return name;
    }

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

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

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

Course.java

```
package com.example.model;

import javax.persistence.*;
import java.util.HashSet;
import java.util.Set;

@Entity
public class Course {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String title;

    @ManyToMany(mappedBy = "courses")
    private Set<Student> students = new HashSet<>();
```

```
    // Getters and setters
    public int getId() {
        return id;
    }

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

    public String getTitle() {
        return title;
    }

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

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

    public void setStudents(Set<Student> students) {
        this.students = students;


    }
}
```

Step 2: Create Hibernate Configuration

Add the annotated classes to your `hibernate.cfg.xml`:

```
<mapping class="com.example.model.Student"/>
<mapping class="com.example.model.Course"/>
```

Step 3: Main Class to Test the Mapping

HibernateManyToManyExample.java

```
package com.example.main;

import com.example.model.Course;
import com.example.model.Student;
import com.example.util.HibernateUtil;
import org.hibernate.Session;
import org.hibernate.Transaction;

import java.util.HashSet;
import java.util.Set;

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

        try {
            transaction = session.beginTransaction();

            // Create course objects
            Course course1 = new Course();
            course1.setTitle("Mathematics");
            Course course2 = new Course();
            course2.setTitle("Physics");

            // Create a student object
            Student student = new Student();
            student.setName("Alice");

            // Create the association
            Set<Course> courses = new HashSet<>();
            courses.add(course1);
            courses.add(course2);
            student.setCourses(courses);

            // Save the objects
            session.save(course1);
```

```
            session.save(course2);
            session.save(student);

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

7.3.3 Table Structure

- `Student` table contains columns: `id`, `name`
- `Course` table contains columns: `id`, `title`
- `student_course` join table contains columns: `student_id`, `course_id`

7.4 Summary

This chapter covered three types of association mappings in Hibernate:

- **One-to-One Uni-directional Mapping**: One entity has a reference to another, and the association is navigable in one direction.
- **One-to-One Bi-directional Mapping**: Both entities have references to each other, making the association navigable in both directions.
- **Many-to-Many Mapping**: Multiple instances of one entity are associated with multiple instances of another, requiring a join table.

We provided detailed examples and code for each type of mapping. These mappings enable you to define complex relationships between entities and leverage Hibernate's ORM capabilities to manage these associations efficiently.

Chapter 8: Version Mapping

Version mapping in Hibernate is a technique used to manage concurrent access to data in a database. It ensures data consistency by using a versioning mechanism that helps prevent the "lost update" problem, where multiple transactions simultaneously modify the same data and the last transaction overwrites the previous changes.

8.1 Introduction to Version Mapping

8.1.1 What is Version Mapping?

Version mapping involves adding a version property to an entity class. This property is automatically updated by Hibernate each time an entity instance is updated. When Hibernate performs an update, it checks the version property to ensure it matches the current version in the database. If the versions do not match, it indicates another transaction has updated the entity, and Hibernate will throw an exception to prevent overwriting changes.

8.1.2 Benefits of Version Mapping

- **Data Integrity**: Ensures that concurrent transactions do not overwrite each other's changes.
- **Optimistic Locking**: Provides a mechanism for optimistic locking, allowing multiple transactions to read data concurrently but preventing conflicts during updates.

8.2 Example of Version Mapping

Let's consider an example of a `Product` entity with version mapping.

8.2.1 Step-by-Step Implementation

Step 1: Create the Entity Class

Product.java

```
package com.example.model;

import javax.persistence.*;

```

```
@Entity
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;
    private double price;

    @Version
    private int version;

    // Getters and setters
    public int getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

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

    public double getPrice() {
        return price;
    }

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

    public int getVersion() {
        return version;
```

```
    }

    public void setVersion(int version) {
        this.version = version;
    }
}
```

Step 2: Create Hibernate Configuration

Add the annotated class to your `hibernate.cfg.xml` :

```
<mapping class="com.example.model.Product"/>
```

Step 3: Create Hibernate Utility Class

We will use the same `HibernateUtil` class created in previous examples.

Step 4: Main Class to Test Version Mapping

HibernateVersionMappingExample.java

```
package com.example.main;

import com.example.model.Product;
import com.example.util.HibernateUtil;
import org.hibernate.Session;
import org.hibernate.Transaction;

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

        try {
            transaction = session.beginTransaction();

            // Create a product
            Product product = new Product();
            product.setName("Laptop");
```

```
            product.setPrice(1500.00);

            // Save the product
            session.save(product);
            transaction.commit();

            // Retrieve and update the product
            transaction = session.beginTransaction();
            Product retrievedProduct = session.get(Product.
class, product.getId());
            retrievedProduct.setPrice(1400.00);
            session.update(retrievedProduct);
            transaction.commit();

            // Simulate concurrent update
            transaction = session.beginTransaction();
            Product concurrentProduct = session.get(Produc
t.class, product.getId());
            concurrentProduct.setPrice(1300.00);
            session.update(concurrentProduct);
            transaction.commit();

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

8.2.2 Explanation

- The `Product` entity has a `@Version` field, which Hibernate uses to manage versioning.

- When saving or updating a `Product`, Hibernate automatically checks the version field to ensure data consistency.
- If a concurrent transaction modifies the same entity, Hibernate will throw an `OptimisticLockException` to prevent the second update from overwriting the first.

83 Handling OptimisticLockException

When multiple transactions attempt to update the same entity concurrently, an `OptimisticLockException` is thrown. This exception should be handled appropriately to ensure that the application can recover gracefully.

HandlingOptimisticLockException.java

```
package com.example.main;

import com.example.model.Product;
import com.example.util.HibernateUtil;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.exception.LockAcquisitionException;

public class HandlingOptimisticLockException {
    public static void main(String[] args) {
        Session session1 = HibernateUtil.getSessionFactory
().openSession();
        Session session2 = HibernateUtil.getSessionFactory
().openSession();
        Transaction transaction1 = null;
        Transaction transaction2 = null;

        try {
            transaction1 = session1.beginTransaction();
            transaction2 = session2.beginTransaction();

            // Retrieve the same product in two different s
essions
            Product product1 = session1.get(Product.class,
1);
```

```
            Product product2 = session2.get(Product.class,
1);

            // Update the product in the first session
            product1.setPrice(1400.00);
            session1.update(product1);
            transaction1.commit();

            // Try to update the product in the second sess
ion
            product2.setPrice(1300.00);
            session2.update(product2);
            transaction2.commit();

        } catch (LockAcquisitionException e) {
            if (transaction2 != null) {
                transaction2.rollback();
            }
            System.out.println("Optimistic lock exception:
" + e.getMessage());
        } finally {
            session1.close();
            session2.close();
        }
    }
}
```

83.1 Explanation

- Two sessions are used to simulate concurrent updates.
- The product is retrieved in both sessions.
- The first session updates the product and commits the transaction.
- The second session attempts to update the same product and an `OptimisticLockException` is thrown, indicating that the product was modified by another transaction.

84 Summary

Version mapping is a crucial feature in Hibernate for managing concurrent access to entities. By using version properties and optimistic locking, Hibernate ensures data consistency and prevents the "lost update" problem. This chapter covered the basics of version mapping, provided a step-by-step implementation example, and demonstrated how to handle `OptimisticLockException`. Understanding version mapping helps in building robust applications that handle concurrent data access effectively.

Chapter 9: Timestamp Mapping

9.1 Introduction to Timestamp Mapping

Timestamp mapping in Hibernate allows you to automatically manage and update timestamp fields within your entities. These fields are useful for tracking the creation and last update times of database records, which can be critical for maintaining data integrity and auditing purposes.

9.2 Why Use Timestamp Mapping?

Timestamp fields help in:

- **Auditing**: Keeping a history of changes made to the data.
- **Concurrency Control**: Managing concurrent access to database records.
- **Data Integrity**: Ensuring that the most recent changes are always reflected in the data.

9.3 Mapping Timestamps in Hibernate

Hibernate provides two annotations for timestamp management:

- `@CreationTimestamp`: Automatically sets the timestamp when an entity is first saved.
- `@UpdateTimestamp`: Automatically updates the timestamp when an entity is updated.

Example Entity with Timestamp Mapping

Consider an entity `User` with `createdAt` and `updatedAt` timestamp fields.

```
import javax.persistence.*;
import java.time.LocalDateTime;

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

    private String username;

    @CreationTimestamp
    @Column(updatable = false)
    private LocalDateTime createdAt;

    @UpdateTimestamp
    private LocalDateTime updatedAt;

    // Getters and Setters
}
```

9.3.1 Creation Timestamp

The `@CreationTimestamp` annotation ensures that the `createdAt` field is set when the entity is persisted for the first time. This field is typically marked as `updatable = false` because its value should not change once set.

9.3.2 Update Timestamp

The `@UpdateTimestamp` annotation ensures that the `updatedAt` field is updated with the current timestamp whenever the entity is updated.

9.4 Database Configuration

Ensure your database table for `User` has the appropriate columns for storing timestamps:

```
CREATE TABLE users (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDAT
E CURRENT_TIMESTAMP
);
```

95 Example of Using Timestamp Mapping

95.1 Saving an Entity

When you save a new `User` entity, Hibernate will automatically set the `createdAt` and `updatedAt` fields.

```
public void createUser(Session session, String username) {
    Transaction transaction = session.beginTransaction();
    User user = new User();
    user.setUsername(username);
    session.save(user);
    transaction.commit();
}
```

95.2 Updating an Entity

When you update an existing `User` entity, Hibernate will automatically update the `updatedAt` field.

```
public void updateUser(Session session, Long userId, String
newUsername) {
    Transaction transaction = session.beginTransaction();
    User user = session.get(User.class, userId);
    if (user != null) {
        user.setUsername(newUsername);
        session.update(user);
    }
```

```
    transaction.commit();
}
```

96 Practical Use Case

Use Case: Auditing Changes

Consider an application where user profile changes need to be tracked. Using timestamp fields, the application can display when the profile was created and last updated.

```
public void printUserDetails(Session session, Long userId)
{
    User user = session.get(User.class, userId);
    if (user != null) {
        System.out.println("Username: " + user.getUsername
());
        System.out.println("Created At: " + user.getCreated
At());
        System.out.println("Updated At: " + user.getUpdated
At());
    }
}
```

Handling Concurrency with Versioning

In addition to timestamps, you can use a version field to manage concurrent updates.

```
import javax.persistence.*;

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

    private String username;
```

```
    @CreationTimestamp
    @Column(updatable = false)
    private LocalDateTime createdAt;

    @UpdateTimestamp
    private LocalDateTime updatedAt;

    @Version
    private int version;

    // Getters and Setters
}
```

Using the `@Version` annotation, Hibernate increments the version number with each update, helping to prevent concurrent update issues.

9.7 Conclusion

Timestamp mapping in Hibernate simplifies the management of creation and update timestamps within your entities. By leveraging `@CreationTimestamp` and `@UpdateTimestamp`, you can automate this process, ensuring accurate and consistent timestamp data for auditing and concurrency control.

In the next chapter, we will explore the Data Access Object (DAO) pattern and how it can be implemented in Hibernate to separate the data persistence logic from business logic, providing a cleaner and more modular codebase.

Chapter 10: Hibernate Example on DAO Pattern

10.1 Introduction to DAO Pattern

The Data Access Object (DAO) pattern is a structural pattern that abstracts the data persistence layer of an application. It provides an interface to perform CRUD (Create, Read, Update, Delete) operations without exposing the

underlying database details. This separation of concerns helps in managing and maintaining the codebase more effectively.

10.2 Why Use the DAO Pattern?

- **Separation of Concerns**: Separates the business logic from the data access logic.
- **Maintainability**: Easier to manage and update the data access code.
- **Testability**: Simplifies unit testing by mocking the DAO layer.
- **Reusability**: DAO classes can be reused across different parts of the application.

10.3 DAO Pattern Components

- **DAO Interface**: Defines the standard operations to be performed on a model object(s).
- **DAO Implementation**: Implements the DAO interface using Hibernate for data persistence.
- **Model Class**: Represents the entity to be persisted.
- **Service Layer**: Uses DAO classes to perform business logic operations.

10.4 Example Application

Let's build a simple application using Hibernate and the DAO pattern to manage `User` entities.

10.4.1 Setting Up the Project

Ensure you have the necessary dependencies in your `pom.xml` (for Maven) or `build.gradle` (for Gradle) file:

```
<!-- Maven pom.xml -->
<dependencies>
    <!-- Hibernate dependencies -->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.4.32.Final</version>
```

```
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>5.4.32.Final</version>
    </dependency>
    <!-- Other dependencies like JPA, MySQL Connector, etc.
-->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.23</version>
    </dependency>
</dependencies>
```

1042 Hibernate Configuration

Configure Hibernate in `hibernate.cfg.xml`:

```
<!-- hibernate.cfg.xml -->
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibe
rnate Configuration DTD 3.0//EN" "<http://hibernate.sourcef
orge.net/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:mysq
l://localhost:3306/your_database</property>
        <property name="hibernate.connection.username">your
_username</property>
        <property name="hibernate.connection.password">your
_password</property>

        <!-- JDBC connection pool settings -->
        <property name="hibernate.c3p0.min_size">5</propert
y>
        <property name="hibernate.c3p0.max_size">20</proper
```

```
ty>
        <property name="hibernate.c3p0.timeout">300</proper
ty>
        <property name="hibernate.c3p0.max_statements">50</

property>
        <property name="hibernate.c3p0.idle_test_period">30
00</property>

        <!-- Specify dialect -->
        <property name="hibernate.dialect">org.hibernate.di
alect.MySQL8Dialect</property>

        <!-- Enable Hibernate's automatic session context m
anagement -->
        <property name="hibernate.current_session_context_c
lass">thread</property>

        <!-- Echo all executed SQL to stdout -->
        <property name="hibernate.show_sql">true</property>

        <!-- Drop and re-create the database schema on star
tup -->
        <property name="hibernate.hbm2ddl.auto">update</pro
perty>


        <!-- Specify annotated classes -->
        <mapping class="com.example.model.User"/>
    </session-factory>
</hibernate-configuration>
```

1043 Creating the User Entity

Create a simple `User` entity with annotations for Hibernate.

```
import javax.persistence.*;
import java.time.LocalDateTime;

@Entity
@Table(name = "users")
```

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

    private String username;
    private String password;

    @CreationTimestamp
    @Column(updatable = false)
    private LocalDateTime createdAt;

    @UpdateTimestamp
    private LocalDateTime updatedAt;

    // Getters and setters
}
```

1044 Creating the DAO Interface

Define the DAO interface with the standard CRUD operations.

```
import java.util.List;

public interface UserDao {
    void save(User user);
    User findById(Long id);
    List<User> findAll();
    void update(User user);
    void delete(User user);
}
```

1045 Implementing the DAO Interface

Create the DAO implementation using Hibernate.

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

```
import org.hibernate.cfg.Configuration;
import java.util.List;

public class UserDaoImpl implements UserDao {

    private SessionFactory sessionFactory;

    public UserDaoImpl() {
        sessionFactory = new Configuration().configure().bu
ildSessionFactory();
    }

    @Override
    public void save(User user) {
        Session session = sessionFactory.openSession();
        Transaction transaction = session.beginTransaction
();
        session.save(user);
        transaction.commit();
        session.close();
    }

    @Override
    public User findById(Long id) {
        Session session = sessionFactory.openSession();
        User user = session.get(User.class, id);
        session.close();
        return user;
    }

    @Override
    public List<User> findAll() {
        Session session = sessionFactory.openSession();
        List<User> users = session.createQuery("from User",
User.class).list();
        session.close();
        return users;
    }
```

```
    @Override
    public void update(User user) {
        Session session = sessionFactory.openSession();
        Transaction transaction = session.beginTransaction
();
        session.update(user);
        transaction.commit();
        session.close();
    }

    @Override
    public void delete(User user) {
        Session session = sessionFactory.openSession();
        Transaction transaction = session.beginTransaction
();
        session.delete(user);
        transaction.commit();
        session.close();
    }
}
```

10.4.6 Using the DAO in a Service Layer

Create a service layer to use the `UserDao`.

```
import java.util.List;

public class UserService {

    private UserDao userDao;

    public UserService() {
        userDao = new UserDaoImpl();
    }

    public void createUser(String username, String passwor
d) {
        User user = new User();
```

```
        user.setUsername(username);
        user.setPassword(password);
        userDao.save(user);
    }

    public User getUserById(Long id) {
        return userDao.findById(id);
    }

    public List<User> getAllUsers() {
        return userDao.findAll();
    }

    public void updateUser(Long id, String username, String
password) {
        User user = userDao.findById(id);
        if (user != null) {
            user.setUsername(username);
            user.setPassword(password);
            userDao.update(user);
        }
    }

    public void deleteUser(Long id) {
        User user = userDao.findById(id);
        if (user != null) {
            userDao.delete(user);
        }
    }
}
```

10.4.7 Testing the DAO Pattern Implementation

Create a simple test class to verify the implementation.

```
public class Main {
    public static void main(String[] args) {
        UserService userService = new UserService();
```

```
        // Create users
        userService.createUser("john_doe", "password123");
        userService.createUser("jane_doe", "password456");

        // Fetch all users
        List<User> users = userService.getAllUsers();
        users.forEach(user -> System.out.println(user.getUs
ername()));

        // Update a user
        User user = userService.getUserById(1L);
        if (user != null) {
            userService.updateUser(user.getId(), "john_doe_
updated", "newpassword123");
        }

        // Fetch updated user
        User updatedUser = userService.getUserById(1L);
        if (updatedUser != null) {
            System.out.println("Updated username: " + updat
edUser.getUsername());
        }

        // Delete a user
        userService.deleteUser(1L);

        // Verify user deletion
        User deletedUser = userService.getUserById(1L);
        if (deletedUser == null) {
            System.out.println("User successfully delete
d");

        }
    }
}
```

105 Conclusion

The DAO pattern provides a clear and maintainable structure for managing data access in Hibernate applications. By separating the data access logic from the business logic, you can achieve a modular, reusable, and testable codebase. The example provided demonstrates how to implement and use the DAO pattern effectively with Hibernate, laying a solid foundation for building robust applications.

In the next chapter, we will explore the Hibernate Query Language (HQL) and its powerful features for querying the database in a more object-oriented way.

Chapter 11: Hibernate Query Language

Hibernate Query Language (HQL) is a powerful, object-oriented query language similar to SQL but works with Hibernate's data entities rather than directly with database tables. HQL supports features such as polymorphic queries, positional and named parameters, and can be used to perform CRUD operations. In addition to HQL, Hibernate also supports Criteria Queries (QBC), Native SQL Queries, and Named Queries for more complex querying requirements.

11.1 SQL | HQL | QBC | Native Query | Named Query

11.1.1 SQL (Structured Query Language)

SQL is the standard language for relational database management systems. In Hibernate, SQL can be used through native SQL queries when HQL does not provide the needed functionality.

11.1.2 HQL (Hibernate Query Language)

HQL is similar to SQL but operates on Hibernate's entity model rather than directly on database tables. It supports features like inheritance, polymorphism, and associations.

Example of HQL Query

```
// Fetch all users
String hql = "FROM User";
```

```
Query query = session.createQuery(hql);
List<User> users = query.list();
```

11.1.3 QBC (Query by Criteria)

QBC allows the construction of queries programmatically using criteria objects. It is type-safe and preferred for dynamic queries.

Example of QBC

```
CriteriaBuilder builder = session.getCriteriaBuilder();
CriteriaQuery<User> query = builder.createQuery(User.clas
s);
Root<User> root = query.from(User.class);
query.select(root);

Query<User> q = session.createQuery(query);
List<User> users = q.getResultList();
```

11.1.4 Native Query

Native SQL queries are used when you need to execute database-specific SQL queries that HQL cannot handle.

Example of Native Query

```
String sql = "SELECT * FROM users";
Query query = session.createNativeQuery(sql, User.class);
List<User> users = query.getResultList();
```

11.1.5 Named Query

Named queries are defined using annotations or XML and provide a way to predefine HQL or SQL queries with a name, which can then be reused throughout the application.

Example of Named Query

Define the named query in the `User` entity:

```
@Entity
@Table(name = "users")
@NamedQueries({
    @NamedQuery(name = "User.findAll", query = "FROM Use
r"),
    @NamedQuery(name = "User.findByUsername", query = "FROM
User u WHERE u.username = :username")
})
public class User {
    // Entity fields and methods
}
```

Using the named query:

```
Query query = session.createNamedQuery("User.findByUsernam
e");
query.setParameter("username", "john_doe");
User user = (User) query.getSingleResult();
```

11.2 Polymorphic Query

Polymorphic queries in HQL allow querying of entity classes and their subclasses. This is useful when dealing with inheritance hierarchies.

Example of Polymorphic Query

Assume `Employee` is a superclass and `Manager` is a subclass:

```
String hql = "FROM Employee";
Query query = session.createQuery(hql);
List<Employee> employees = query.list();
```

This query will fetch all instances of `Employee` and its subclasses.

11.3 Positional Parameters & Named Parameters

11.3.1 Positional Parameters

Positional parameters are denoted by `?` followed by the parameter index in the HQL query.

Example of Positional Parameters

```
String hql = "FROM User WHERE username = ?1";
Query query = session.createQuery(hql);
query.setParameter(1, "john_doe");
User user = (User) query.getSingleResult();
```

11.3.2 Named Parameters

Named parameters are denoted by `:` followed by the parameter name in the HQL query.

Example of Named Parameters

```
String hql = "FROM User WHERE username = :username";
Query query = session.createQuery(hql);
query.setParameter("username", "john_doe");
User user = (User) query.getSingleResult();
```

11.4 Practical Examples

11.4.1 Using HQL to Fetch All Users

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

String hql = "FROM User";
Query query = session.createQuery(hql);
List<User> users = query.list();

transaction.commit();
session.close();

for (User user : users) {
```

```
    System.out.println(user.getUsername());
}
```

1142 Using Criteria Query to Fetch Users by Username

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

CriteriaBuilder builder = session.getCriteriaBuilder();
CriteriaQuery<User> query = builder.createQuery(User.clas
s);
Root<User> root = query.from(User.class);
query.select(root).where(builder.equal(root.get("usernam
e"), "john_doe"));

Query<User> q = session.createQuery(query);
List<User> users = q.getResultList();

transaction.commit();
session.close();

for (User user : users) {
    System.out.println(user.getUsername());
}
```

1143 Using Named Query to Fetch User by Username

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

Query query = session.createNamedQuery("User.findByUsernam
e");
query.setParameter("username", "john_doe");
User user = (User) query.getSingleResult();

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

```
System.out.println(user.getUsername());
```

11.5 Conclusion

Hibernate Query Language (HQL) provides a powerful and flexible way to interact with your database through Hibernate's ORM capabilities. It supports a wide range of querying options, including polymorphic queries, positional and named parameters, and integrates seamlessly with SQL, Criteria API, Native Queries, and Named Queries. By leveraging these tools, developers can write more readable, maintainable, and efficient data access code.

In the next chapter, we will delve into handling primary keys in Hibernate, covering simple, custom, and composite primary keys.

Chapter 12: Simple Primary Key

12.1 Introduction to Primary Keys

In any relational database, the primary key is a critical concept. A primary key uniquely identifies each record in a table, ensuring data integrity and enabling efficient data retrieval. In Hibernate, defining a primary key is straightforward and essential for entity management.

12.2 Why Use Primary Keys?

Primary keys provide several benefits:

- **Uniqueness**: Ensures each record is unique.
- **Indexing**: Facilitates fast data retrieval.
- **Referential Integrity**: Supports relationships between tables.

12.3 Defining a Simple Primary Key in Hibernate

In Hibernate, a simple primary key can be defined using the `@Id` annotation. Typically, primary keys are either auto-generated or assigned manually.

12.3.1 Auto-Generated Primary Key

Hibernate supports several strategies for auto-generating primary keys:

- **Identity**: Uses the database's identity column.
- **Sequence**: Uses a database sequence.
- **Table**: Uses a table to simulate sequence behavior.
- **Auto**: Automatically selects the appropriate strategy based on the database dialect.

Example with Auto-Generated Primary Key

Consider an `Employee` entity with an auto-generated primary key.

```
import javax.persistence.*;

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

    private String name;
    private String department;

    // Getters and setters
}
```

In this example, the `id` field is the primary key, and its value is automatically generated using the database's identity column.

1232 Manually Assigned Primary Key

Sometimes, primary keys are assigned manually rather than generated by the database.

Example with Manually Assigned Primary Key

```
import javax.persistence.*;

@Entity
```

```
@Table(name = "employees")
public class Employee {
    @Id
    private Long id;

    private String name;
    private String department;

    // Getters and setters
}
```

In this case, you need to ensure that the `id` value is set manually before persisting the entity.

12.4 Practical Example

Let's walk through a complete example of defining and using a simple primary key in Hibernate.

12.4.1 Setting Up the Project

Ensure you have the necessary dependencies in your `pom.xml` (for Maven) or `build.gradle` (for Gradle) file:

```
<!-- Maven pom.xml -->
<dependencies>
    <!-- Hibernate dependencies -->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.4.32.Final</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>5.4.32.Final</version>
    </dependency>
    <!-- Other dependencies like JPA, MySQL Connector, etc.
-->
    <dependency>
```

```
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.23</version>
    </dependency>
</dependencies>
```

1242 Hibernate Configuration

Configure Hibernate in `hibernate.cfg.xml`:

```
<!-- hibernate.cfg.xml -->
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibe
rnate Configuration DTD 3.0//EN" "<http://hibernate.sourcef
orge.net/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:mysq
l://localhost:3306/your_database</property>
        <property name="hibernate.connection.username">your
_username</property>
        <property name="hibernate.connection.password">your
_password</property>

        <!-- JDBC connection pool settings -->
        <property name="hibernate.c3p0.min_size">5</propert
y>
        <property name="hibernate.c3p0.max_size">20</proper
ty>
        <property name="hibernate.c3p0.timeout">300</proper
ty>
        <property name="hibernate.c3p0.max_statements">50</

property>
        <property name="hibernate.c3p0.idle_test_period">30
00</property>

        <!-- Specify dialect -->
```

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

        <!-- Enable Hibernate's automatic session context management -->
        <property name="hibernate.current_session_context_class">thread</property>

        <!-- Echo all executed SQL to stdout -->
        <property name="hibernate.show_sql">true</property>

        <!-- Drop and re-create the database schema on startup -->
        <property name="hibernate.hbm2ddl.auto">update</property>

        <!-- Specify annotated classes -->
        <mapping class="com.example.model.Employee"/>
    </session-factory>
</hibernate-configuration>
```

1243 Creating the Employee Entity

Define the `Employee` entity with an auto-generated primary key.

```
import javax.persistence.*;

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

    private String name;
    private String department;
```

```
    // Getters and setters
}
```

1244 Performing CRUD Operations

Creating an Employee

```
public void createEmployee(Session session, String name, St
ring department) {
    Transaction transaction = session.beginTransaction();
    Employee employee = new Employee();
    employee.setName(name);
    employee.setDepartment(department);
    session.save(employee);
    transaction.commit();
}
```

Reading an Employee

```
public Employee getEmployeeById(Session session, Long id) {
    return session.get(Employee.class, id);
}
```

Updating an Employee

```
public void updateEmployee(Session session, Long id, String
name, String department) {
    Transaction transaction = session.beginTransaction();
    Employee employee = session.get(Employee.class, id);
    if (employee != null) {
        employee.setName(name);
        employee.setDepartment(department);
        session.update(employee);
    }
    transaction.commit();
}
```

Deleting an Employee

```
public void deleteEmployee(Session session, Long id) {
    Transaction transaction = session.beginTransaction();
    Employee employee = session.get(Employee.class, id);
    if (employee != null) {
        session.delete(employee);
    }
    transaction.commit();
}
```

1245 Testing the CRUD Operations

Create a simple test class to verify the CRUD operations.

```
public class Main {
    public static void main(String[] args) {
        SessionFactory sessionFactory = new Configuration
().configure().buildSessionFactory();
        Session session = sessionFactory.openSession();

        // Create employee
        createEmployee(session, "John Doe", "Engineering");

        // Fetch employee
        Employee employee = getEmployeeById(session, 1L);
        System.out.println("Employee: " + employee.getName
());

        // Update employee
        updateEmployee(session, 1L, "John Doe", "Marketin
g");


        // Fetch updated employee
        Employee updatedEmployee = getEmployeeById(session,
1L);
        System.out.println("Updated Employee: " + updatedEm
ployee.getDepartment());
```

```
        // Delete employee
        deleteEmployee(session, 1L);

        // Verify deletion
        Employee deletedEmployee = getEmployeeById(session,
1L);
        if (deletedEmployee == null) {
            System.out.println("Employee successfully delet
ed");
        }

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

12.5 Conclusion

Defining and using a simple primary key in Hibernate is straightforward and essential for ensuring the uniqueness and integrity of database records. By leveraging the `@Id` and `@GeneratedValue` annotations, developers can easily manage primary keys, enabling efficient CRUD operations and data management. The provided examples demonstrate how to implement and test these concepts effectively.

In the next chapter, we will explore custom primary keys and how to handle more complex primary key requirements in Hibernate.

Chapter 13: Custom Primary Key

13.1 Introduction to Custom Primary Keys

While auto-generated primary keys are common and convenient, some applications require custom primary key strategies. Custom primary keys allow for greater control and flexibility, enabling developers to define unique identification strategies that meet specific business requirements. In Hibernate,

custom primary keys can be implemented using various strategies, including custom sequences, UUIDs, and composite keys.

13.2 Defining a Custom Primary Key in Hibernate

13.2.1 Using Custom Sequences

A custom sequence is a database object that generates unique values in a specific order. This is useful when you need predictable and sequential primary keys.

Example of Custom Sequence

First, create a sequence in your database:

```
CREATE SEQUENCE custom_sequence START WITH 1 INCREMENT BY
1;
```

Then, define the entity in Hibernate:

```
import javax.persistence.*;

@Entity
@Table(name = "employees")
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, gen
erator = "custom_seq")
    @SequenceGenerator(name = "custom_seq", sequenceName =
"custom_sequence", allocationSize = 1)
    private Long id;

    private String name;
    private String department;

    // Getters and setters
}
```

13.2.2 Using UUIDs

UUIDs (Universally Unique Identifiers) are 128-bit values that can be used to uniquely identify records across distributed systems. They are particularly useful when dealing with distributed databases or ensuring global uniqueness.

Example of UUID

First, add a dependency for UUID generation if not already included:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.4.32.Final</version>
</dependency>
```

Then, define the entity:

```
import javax.persistence.*;
import java.util.UUID;

@Entity
@Table(name = "employees")
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(columnDefinition = "BINARY(16)")
    private UUID id;

    private String name;
    private String department;

    // Getters and setters
}
```

1323 Custom Identifier Generator

Hibernate allows the creation of custom identifier generators to define unique ID generation strategies.

Example of Custom Identifier Generator

First, create the custom identifier generator:

```
import org.hibernate.engine.spi.SharedSessionContractImplem
entor;
import org.hibernate.id.IdentifierGenerator;

import java.io.Serializable;
import java.util.UUID;

public class CustomUUIDGenerator implements IdentifierGener
ator {
    @Override
    public Serializable generate(SharedSessionContractImple
mentor session, Object object) {
        return UUID.randomUUID().toString();
    }
}
```

Then, define the entity to use this custom generator:

```
import org.hibernate.annotations.GenericGenerator;

import javax.persistence.*;

@Entity
@Table(name = "employees")
public class Employee {
    @Id
    @GeneratedValue(generator = "uuid")
    @GenericGenerator(name = "uuid", strategy = "com.exampl
e.CustomUUIDGenerator")
    private String id;

    private String name;
    private String department;

    // Getters and setters
}
```

133 Practical Example

Let's walk through a complete example of defining and using a custom primary key in Hibernate using a UUID strategy.

1331 Setting Up the Project

Ensure you have the necessary dependencies in your `pom.xml` (for Maven) or `build.gradle` (for Gradle) file:

```
<!-- Maven pom.xml -->
<dependencies>
    <!-- Hibernate dependencies -->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.4.32.Final</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>5.4.32.Final</version>
    </dependency>
    <!-- Other dependencies like JPA, MySQL Connector, etc.
-->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.23</version>
    </dependency>
</dependencies>
```

1332 Hibernate Configuration

Configure Hibernate in `hibernate.cfg.xml`:

```
<!-- hibernate.cfg.xml -->
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibe
rnate Configuration DTD 3.0//EN" "<http://hibernate.sourcef
orge.net/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:mysq
l://localhost:3306/your_database</property>
        <property name="hibernate.connection.username">your
_username</property>
        <property name="hibernate.connection.password">your
_password</property>

        <!-- JDBC connection pool settings -->
        <property name="hibernate.c3p0.min_size">5</propert
y>
        <property name="hibernate.c3p0.max_size">20</proper
ty>
        <property name="hibernate.c3p0.timeout">300</proper
ty>
        <property name="hibernate.c3p0.max_statements">50</

property>
        <property name="hibernate.c3p0.idle_test_period">30
00</property>

        <!-- Specify dialect -->
        <property name="hibernate.dialect">org.hibernate.di
alect.MySQL8Dialect</property>

        <!-- Enable Hibernate's automatic session context m
anagement -->
        <property name="hibernate.current_session_context_c
lass">thread</property>

        <!-- Echo all executed SQL to stdout -->
        <property name="hibernate.show_sql">true</property>

        <!-- Drop and re-create the database schema on star
tup -->
```

```
        <property name="hibernate.hbm2ddl.auto">update</pro
perty>

        <!-- Specify annotated classes -->
        <mapping class="com.example.model.Employee"/>
    </session-factory>
</hibernate-configuration>
```

1333 Creating the Employee Entity with UUID

Define the `Employee` entity with a UUID primary key.

```
import javax.persistence.*;
import java.util.UUID;

@Entity
@Table(name = "employees")
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(columnDefinition = "BINARY(16)")
    private UUID id;

    private String name;
    private String department;

    // Getters and setters
}
```

1334 Performing CRUD Operations

Creating an Employee

```
public void createEmployee(Session session, String name, St
ring department) {
    Transaction transaction = session.beginTransaction();
    Employee employee = new Employee();
    employee.setName(name);
```

```
    employee.setDepartment(department);
    session.save(employee);
    transaction.commit();
}
```

Reading an Employee

```
public Employee getEmployeeById(Session session, UUID id) {
    return session.get(Employee.class, id);
}
```

Updating an Employee

```
public void updateEmployee(Session session, UUID id, String
name, String department) {
    Transaction transaction = session.beginTransaction();
    Employee employee = session.get(Employee.class, id);
    if (employee != null) {
        employee.setName(name);
        employee.setDepartment(department);
        session.update(employee);
    }
    transaction.commit();
}
```

Deleting an Employee

```
public void deleteEmployee(Session session, UUID id) {
    Transaction transaction = session.beginTransaction();
    Employee employee = session.get(Employee.class, id);
    if (employee != null) {
        session.delete(employee);
    }
    transaction.commit();
}
```

1335 Testing the CRUD Operations

Create a simple test class to verify the CRUD operations.

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

import java.util.UUID;

public class Main {
    public static void main(String[] args) {
        SessionFactory sessionFactory = new Configuration
().configure().buildSessionFactory();
        Session session = sessionFactory.openSession();

        // Create employee
        createEmployee(session, "John Doe", "Engineering");

        // Fetch employee
        Employee employee = getEmployeeById(session, UUID.f
romString("your-uuid-string"));
        System.out.println("Employee: " + employee.getName
());

        // Update employee
        updateEmployee(session, UUID.fromString("your-uuid-
string"), "John Doe", "Marketing");

        // Fetch updated employee
        Employee updatedEmployee = getEmployeeById(session,
UUID.fromString("your-uuid-string"));
        System.out.println("Updated Employee: " + updatedEm
ployee.getDepartment());

        // Delete employee
        deleteEmployee(session, UUID.fromString("your-uuid-
string"));
```

```
        // Verify deletion
        Employee deletedEmployee = getEmployeeById(session,
UUID.fromString("your-uuid-string"));
        if (deletedEmployee == null) {
            System.out.println("Employee successfully delet
ed");
        }

        session.close();
        sessionFactory.close();
    }


    public static void createEmployee(Session session, Stri
ng name, String department) {
        Transaction transaction = session.beginTransaction
();
        Employee employee = new Employee();
        employee.setName(name);
        employee.setDepartment(department);
        session.save(employee);
        transaction.commit();
    }


    public static Employee getEmployeeById(Session session,
UUID id) {
        return session.get(Employee.class, id);
    }

    public static void updateEmployee(Session session, UUID
id, String name, String department) {
        Transaction transaction = session.beginTransaction
();
        Employee employee = session.get(Employee.class, i
d);
        if (employee != null

) {
```

```
            employee.setName(name);
            employee.setDepartment(department);
            session.update(employee);
        }
        transaction.commit();
    }

    public static void deleteEmployee(Session session, UUID
id) {
        Transaction transaction = session.beginTransaction
();
        Employee employee = session.get(Employee.class, i
d);
        if (employee != null) {
            session.delete(employee);
        }
        transaction.commit();
    }
}
```

13.4 Conclusion

Custom primary keys in Hibernate provide flexibility and control over how entities are uniquely identified. By using strategies such as custom sequences, UUIDs, and custom identifier generators, developers can tailor primary key generation to meet specific requirements. The provided examples demonstrate how to implement and test these custom primary key strategies effectively.

In the next chapter, we will explore composite primary keys, which are essential for complex key structures involving multiple columns.

Chapter 14: Composite Primary Key

14.1 Introduction to Composite Primary Keys

A composite primary key, also known as a compound key, consists of two or more columns that together uniquely identify a record in a table. This approach

is often used in cases where a single column cannot uniquely identify a record, especially in many-to-many relationships and complex business logic scenarios.

In Hibernate, composite primary keys can be implemented using either the `@IdClass` annotation or the `@EmbeddedId` annotation. Each approach has its use cases and benefits.

14.2 Using `@IdClass` Annotation

The `@IdClass` annotation is used to define a composite primary key class separately. This class must implement `Serializable` and override the `equals` and `hashCode` methods.

14.2.1 Defining the Composite Key Class

Create a class that represents the composite key:

```
import java.io.Serializable;
import java.util.Objects;


public class EmployeeId implements Serializable {
    private Long employeeId;
    private Long departmentId;

    // Default constructor
    public EmployeeId() {}

    // Parameterized constructor
    public EmployeeId(Long employeeId, Long departmentId) {
        this.employeeId = employeeId;
        this.departmentId = departmentId;
    }

    // Getters and setters
    public Long getEmployeeId() {
        return employeeId;
    }

    public void setEmployeeId(Long employeeId) {
```

```
        this.employeeId = employeeId;
    }

    public Long getDepartmentId() {
        return departmentId;
    }

    public void setDepartmentId(Long departmentId) {
        this.departmentId = departmentId;
    }

    // Override equals and hashCode
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return
false;
        EmployeeId that = (EmployeeId) o;
        return Objects.equals(employeeId, that.employeeId)
&& Objects.equals(departmentId, that.departmentId);
    }

    @Override
    public int hashCode() {
        return Objects.hash(employeeId, departmentId);
    }
}
```

14.2.2 Defining the Entity with `@IdClass`

Annotate the entity class with `@IdClass` and map the composite key fields:

```
import javax.persistence.*;

@Entity
@IdClass(EmployeeId.class)
@Table(name = "employee_department")
public class EmployeeDepartment {
```

```
    @Id
    private Long employeeId;

    @Id
    private Long departmentId;

    private String role;

    // Getters and setters
    public Long getEmployeeId() {
        return employeeId;
    }

    public void setEmployeeId(Long employeeId) {
        this.employeeId = employeeId;
    }

    public Long getDepartmentId() {
        return departmentId;
    }

    public void setDepartmentId(Long departmentId) {
        this.departmentId = departmentId;
    }

    public String getRole() {
        return role;
    }

    public void setRole(String role) {
        this.role = role;
    }
}
```

143 Using `@EmbeddedId` Annotation

The `@EmbeddedId` annotation is another way to define composite primary keys.

This approach involves embedding the composite key class directly into the

entity class.

14.3.1 Defining the Embedded Key Class

Create a class that represents the composite key:

```
import javax.persistence.Embeddable;
import java.io.Serializable;
import java.util.Objects;

@Embeddable
public class EmployeeDepartmentId implements Serializable {
    private Long employeeId;
    private Long departmentId;

    // Default constructor
    public EmployeeDepartmentId() {}

    // Parameterized constructor
    public EmployeeDepartmentId(Long employeeId, Long depar
tmentId) {
        this.employeeId = employeeId;
        this.departmentId = departmentId;
    }

    // Getters and setters
    public Long getEmployeeId() {
        return employeeId;
    }

    public void setEmployeeId(Long employeeId) {
        this.employeeId = employeeId;
    }

    public Long getDepartmentId() {
        return departmentId;
    }

    public void setDepartmentId(Long departmentId) {
```

```
        this.departmentId = departmentId;
    }

    // Override equals and hashCode
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return
false;
        EmployeeDepartmentId that = (EmployeeDepartmentId)
o;
        return Objects.equals(employeeId, that.employeeId)
&& Objects.equals(departmentId, that.departmentId);
    }

    @Override
    public int hashCode() {
        return Objects.hash(employeeId, departmentId);
    }
}
```

1432 Defining the Entity with `@EmbeddedId`

Annotate the entity class with `@EmbeddedId` and embed the composite key class:

```
import javax.persistence.*;

@Entity
@Table(name = "employee_department")
public class EmployeeDepartment {
    @EmbeddedId
    private EmployeeDepartmentId id;

    private String role;

    // Getters and setters
    public EmployeeDepartmentId getId() {
        return id;
```

```
    }

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

    public String getRole() {
        return role;
    }

    public void setRole(String role) {
        this.role = role;
    }
}
```

144 Practical Example

Let's walk through a complete example using `@EmbeddedId` to define and use a composite primary key in Hibernate.

144.1 Setting Up the Project

Ensure you have the necessary dependencies in your `pom.xml` (for Maven) or `build.gradle` (for Gradle) file:

```
<!-- Maven pom.xml -->
<dependencies>
    <!-- Hibernate dependencies -->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.4.32.Final</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>5.4.32.Final</version>
    </dependency>
    <!-- Other dependencies like JPA, MySQL Connector, etc.
```

```
-->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.23</version>
    </dependency>
</dependencies>
```

1442 Hibernate Configuration

Configure Hibernate in `hibernate.cfg.xml`:

```
<!-- hibernate.cfg.xml -->
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibe
rnate Configuration DTD 3.0//EN" "<http://hibernate.sourcef
orge.net/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:mysq
l://localhost:3306/your_database</property>
        <property name="hibernate.connection.username">your
_username</property>
        <property name="hibernate.connection.password">your
_password</property>

        <!-- JDBC connection pool settings -->
        <property name="hibernate.c3p0.min_size">5</propert
y>
        <property name="hibernate.c3p0.max_size">20</proper
ty>
        <property name="hibernate.c3p0.timeout">300</proper
ty>
        <property name="hibernate.c3p0.max_statements">50</

property>
        <property name="hibernate.c3p0.idle_test_period">30
00</property>
```

```
        <!-- Specify dialect -->
        <property name="hibernate.dialect">org.hibernate.di
alect.MySQL8Dialect</property>

        <!-- Enable Hibernate's automatic session context m
anagement -->
        <property name="hibernate.current_session_context_c
lass">thread</property>

        <!-- Echo all executed SQL to stdout -->
        <property name="hibernate.show_sql">true</property>

        <!-- Drop and re-create the database schema on star
tup -->
        <property name="hibernate.hbm2ddl.auto">update</pro
perty>

        <!-- Specify annotated classes -->
        <mapping class="com.example.model.EmployeeDepartmen
t"/>

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

1443 Creating the EmployeeDepartment Entity with EmbeddedId

Define the `EmployeeDepartment` entity with an embedded composite key.

```
import javax.persistence.*;

@Entity
@Table(name = "employee_department")
public class EmployeeDepartment {
    @EmbeddedId
    private EmployeeDepartmentId id;

    private String role;
```

```
    // Getters and setters
    public EmployeeDepartmentId getId() {
        return id;
    }

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

    public String getRole() {
        return role;
    }

    public void setRole(String role) {
        this.role = role;
    }
}
```

1444 Performing CRUD Operations

Creating an EmployeeDepartment

```
public void createEmployeeDepartment(Session session, Long
employeeId, Long departmentId, String role) {
    Transaction transaction = session.beginTransaction();
    EmployeeDepartmentId id = new EmployeeDepartmentId(empl
oyeeId, departmentId);
    EmployeeDepartment employeeDepartment = new EmployeeDep
artment();
    employeeDepartment.setId(id);
    employeeDepartment.setRole(role);
    session.save(employeeDepartment);
    transaction.commit();
}
```

Reading an EmployeeDepartment

```
public EmployeeDepartment getEmployeeDepartmentById(Session
session, Long employeeId, Long departmentId) {
    EmployeeDepartmentId id =

 new EmployeeDepartmentId(employeeId, departmentId);
    return session.get(EmployeeDepartment.class, id);
}
```

Updating an EmployeeDepartment

```
public void updateEmployeeDepartment(Session session, Long
employeeId, Long departmentId, String role) {
    Transaction transaction = session.beginTransaction();
    EmployeeDepartmentId id = new EmployeeDepartmentId(empl
oyeeId, departmentId);
    EmployeeDepartment employeeDepartment = session.get(Emp
loyeeDepartment.class, id);
    if (employeeDepartment != null) {
        employeeDepartment.setRole(role);
        session.update(employeeDepartment);
    }
    transaction.commit();
}
```

Deleting an EmployeeDepartment

```
public void deleteEmployeeDepartment(Session session, Long
employeeId, Long departmentId) {
    Transaction transaction = session.beginTransaction();
    EmployeeDepartmentId id = new EmployeeDepartmentId(empl
oyeeId, departmentId);
    EmployeeDepartment employeeDepartment = session.get(Emp
loyeeDepartment.class, id);
    if (employeeDepartment != null) {
        session.delete(employeeDepartment);
    }
```

```
    transaction.commit();
}
```

1445 Testing the CRUD Operations

Create a simple test class to verify the CRUD operations.

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

public class Main {
    public static void main(String[] args) {
        SessionFactory sessionFactory = new Configuration
().configure().buildSessionFactory();
        Session session = sessionFactory.openSession();

        // Create employee department
        createEmployeeDepartment(session, 1L, 1L, "Manage
r");



        // Fetch employee department
        EmployeeDepartment employeeDepartment = getEmployee
DepartmentById(session, 1L, 1L);
        System.out.println("Role: " + employeeDepartment.ge
tRole());

        // Update employee department
        updateEmployeeDepartment(session, 1L, 1L, "Directo
r");



        // Fetch updated employee department
        EmployeeDepartment updatedEmployeeDepartment = getE
mployeeDepartmentById(session, 1L, 1L);
        System.out.println("Updated Role: " + updatedEmploy
eeDepartment.getRole());

        // Delete employee department
```

```
        deleteEmployeeDepartment(session, 1L, 1L);

        // Verify deletion
        EmployeeDepartment deletedEmployeeDepartment = getE
mployeeDepartmentById(session, 1L, 1L);
        if (deletedEmployeeDepartment == null) {
            System.out.println("Employee department success
fully deleted");
        }

        session.close();
        sessionFactory.close();
    }

    public static void createEmployeeDepartment(Session ses
sion, Long employeeId, Long departmentId, String role) {
        Transaction transaction = session.beginTransaction
();
        EmployeeDepartmentId id = new EmployeeDepartmentId
(employeeId, departmentId);
        EmployeeDepartment employeeDepartment = new Employe
eDepartment();
        employeeDepartment.setId(id);
        employeeDepartment.setRole(role);
        session.save(employeeDepartment);
        transaction.commit();
    }

    public static EmployeeDepartment getEmployeeDepartmentB
yId(Session session, Long employeeId, Long departmentId) {
        EmployeeDepartmentId id = new EmployeeDepartmentId
(employeeId, departmentId);
        return session.get(EmployeeDepartment.class, id);
    }

    public static void updateEmployeeDepartment(Session ses
sion, Long employeeId, Long departmentId, String role) {
        Transaction transaction = session.beginTransaction
```

```
();
        EmployeeDepartmentId id = new EmployeeDepartmentId
(employeeId, departmentId);
        EmployeeDepartment employeeDepartment = session.get
(EmployeeDepartment.class, id);
        if (employeeDepartment != null) {
            employeeDepartment.setRole(role);
            session.update(employeeDepartment);
        }
        transaction.commit();
    }

    public static void deleteEmployeeDepartment(Session ses
sion, Long employeeId, Long departmentId) {
        Transaction transaction = session.beginTransaction
();
        EmployeeDepartmentId id = new EmployeeDepartmentId
(employeeId, departmentId);
        EmployeeDepartment employeeDepartment = session.get
(EmployeeDepartment.class, id);
        if (employeeDepartment != null) {
            session.delete(employeeDepartment);
        }
        transaction.commit();
    }
}
```

145 Conclusion

Composite primary keys in Hibernate provide a robust way to handle complex key structures involving multiple columns. By using annotations like `@IdClass` and `@EmbeddedId`, developers can easily define and manage composite keys, ensuring the uniqueness and integrity of their data. The provided examples demonstrate how to implement and test composite primary keys effectively.

In the next chapter, we will explore the intricacies of transaction management in Hibernate, including the ACID properties, concurrency issues, and various transaction management strategies.

Chapter 15: Introduction to Transaction Management

15.1 Introduction to Transaction Management

Transaction management is a fundamental concept in any database-related application. In Hibernate, transaction management ensures that a group of operations are executed in a secure and reliable manner, maintaining data consistency and integrity. Transactions in Hibernate can be managed programmatically or declaratively using various frameworks like JTA or Spring.

15.2 ACID Properties

ACID is an acronym that stands for Atomicity, Consistency, Isolation, and Durability. These properties ensure reliable processing of database transactions.

15.2.1 Atomicity

Atomicity ensures that a series of operations within a transaction are treated as a single unit. Either all operations succeed, or none of them do. This property prevents partial updates to the database, which could lead to data inconsistency.

15.2.2 Consistency

Consistency ensures that a transaction transforms the database from one valid state to another valid state. After a transaction completes, all data integrity constraints must be satisfied. This means that any given transaction will bring the database from one valid state to another, maintaining database rules.

15.2.3 Isolation

Isolation ensures that the operations within a transaction are isolated from the operations in other transactions. This prevents data anomalies due to concurrent transactions. The isolation level determines the degree to which the operations in one transaction are isolated from those in other transactions.

15.2.4 Durability

Durability ensures that once a transaction has been committed, it remains committed even in the case of a system failure. This means that the data

changes made by the transaction are permanently saved in the database.

153 Transaction Concurrency Problems and Solutions

Concurrency problems arise when multiple transactions are executed simultaneously, leading to potential conflicts and data anomalies. Common concurrency problems include dirty reads, non-repeatable reads, and phantom reads.

1531 Dirty Reads

Dirty Read occurs when a transaction reads uncommitted changes made by another transaction. This can lead to inconsistent data if the other transaction rolls back.

Solution

Use a higher isolation level like `READ_COMMITTED` or above to prevent dirty reads. Hibernate default is `READ_COMMITTED`.

1532 Non-Repeatable Reads

Non-Repeatable Read occurs when a transaction reads the same row twice and gets different data each time because another transaction has modified the row between the two reads.

Solution

Use the `REPEATABLE_READ` isolation level to ensure that once a row is read, it cannot be modified by other transactions until the first transaction completes.

1533 Phantom Reads

Phantom Read occurs when a transaction reads a set of rows that satisfy a condition and then another transaction inserts or deletes rows that satisfy the condition. When the first transaction re-reads the rows, it gets a different set.

Solution

Use the `SERIALIZABLE` isolation level to prevent phantom reads, which ensures complete isolation from other transactions.

15.4 Types of Transactions

In Hibernate, transactions can be managed in several ways, including programmatically using Hibernate API, JTA (Java Transaction API), and CMT (Container Managed Transactions).

15.4.1 Programmatic Transaction Management

Hibernate provides an API for managing transactions programmatically. This approach gives developers fine-grained control over transaction boundaries.

Example

```
Session session = sessionFactory.openSession();
Transaction transaction = null;
try {
    transaction = session.beginTransaction();
    // Perform database operations
    transaction.commit();
} catch (Exception e) {
    if (transaction != null) transaction.rollback();
    e.printStackTrace();
} finally {
    session.close();
}
```

15.4.2 JTA (Java Transaction API)

JTA is a standard API for managing transactions across multiple resources. JTA allows for distributed transactions, making it suitable for applications that interact with multiple databases or messaging systems.

Example

```
import javax.transaction.UserTransaction;
import javax.naming.InitialContext;
import javax.naming.NamingException;

UserTransaction userTransaction = (UserTransaction)new Init
ialContext().lookup("java:comp/UserTransaction");
```

```
try {
    userTransaction.begin();
    // Perform database operations
    userTransaction.commit();
} catch (Exception e) {
    userTransaction.rollback();
    e.printStackTrace();
}
```

1543 CMT (Container Managed Transactions)

CMT is used in enterprise applications where the application server manages the transaction boundaries. This approach simplifies transaction management by delegating it to the container (e.g., EJB, Spring).

Example with EJB

```
@Stateless
public class EmployeeService {
    @PersistenceContext
    private EntityManager entityManager;

    public void createEmployee(Employee employee) {
        entityManager.persist(employee);
    }

    public Employee findEmployee(Long id) {
        return entityManager.find(Employee.class, id);
    }
}
```

Example with Spring

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

```
    public void createEmployee(Employee employee) {
        employeeRepository.save(employee);
    }

    public Employee findEmployee(Long id) {
        return employeeRepository.findById(id).orElse(nul
l);
    }
}
```

15.5 Conclusion

Transaction management is crucial for maintaining data integrity and consistency in any application. Understanding and correctly implementing the ACID properties, addressing concurrency issues, and choosing the appropriate type of transaction management are essential skills for developers working with Hibernate. By leveraging programmatic transactions, JTA, and CMT, developers can ensure robust transaction handling in their applications.

In the next chapter, we will explore Hibernate connection management, including JDBC transactions, JTA transactions, and CMT transactions, and how they integrate with the Hibernate framework.

Chapter 16: Hibernate Connection Management

Hibernate is a powerful framework for mapping an object-oriented domain model to a relational database. Proper connection management is crucial for efficient database interactions. In this chapter, we will explore different types of transaction management supported by Hibernate, focusing on JDBC Transactions, JTA Transactions, and CMT Transactions.

16.1 JDBC Transaction

Explanation:
JDBC (Java Database Connectivity) transactions are the simplest form of transaction management. They are directly managed by the Java application using standard JDBC API calls. Hibernate provides support for JDBC transactions through its

`Session` interface, which encapsulates the JDBC connection and transaction management.

Example:

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

public class JDBCTransactionExample {

    public static void main(String[] args) {
        // Create a configuration object and configure Hibe
rnate
        Configuration configuration = new Configuration().c
onfigure();

        // Create a SessionFactory
        SessionFactory sessionFactory = configuration.build
SessionFactory();

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

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


        try {
            // Perform some database operations
            MyEntity entity = new MyEntity();
            entity.setName("Example Name");
            session.save(entity);

            // Commit the transaction
            transaction.commit();
        } catch (Exception e) {
            // Rollback the transaction in case of an error
```

```
            if (transaction != null) {
                transaction.rollback();
            }
            e.printStackTrace();
        } finally {
            // Close the session
            session.close();
        }

        // Close the SessionFactory
        sessionFactory.close();
    }
}
```

Details:

- **Session**: Represents a single-threaded unit of work. It is the main runtime interface between a Java application and Hibernate.
- **Transaction**: Represents a unit of work that is either entirely completed or entirely undone. It is associated with a `Session` and is used to control transaction boundaries.

162 JTA Transaction

Explanation:

Java Transaction API (JTA) transactions are used in distributed transaction environments, where transactions span multiple resources such as databases and message queues. Hibernate can participate in JTA transactions managed by a JTA-compliant transaction manager.

Example:

```
import javax.transaction.UserTransaction;
import javax.naming.InitialContext;

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

public class JTATransactionExample {
```

```
    public static void main(String[] args) {
        try {
            // Obtain the JTA UserTransaction
            InitialContext context = new InitialContext();
            UserTransaction userTransaction = (UserTransact
ion) context.lookup("java:comp/UserTransaction");

            // Create a configuration object and configure
Hibernate
            Configuration configuration = new Configuration
().configure();

            // Create a SessionFactory
            SessionFactory sessionFactory = configuration.b
uildSessionFactory();

            // Begin the JTA transaction
            userTransaction.begin();

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

            // Perform some database operations
            MyEntity entity = new MyEntity();
            entity.setName("Example Name");
            session.save(entity);

            // Commit the JTA transaction
            userTransaction.commit();

            // Close the session
            session.close();

            // Close the SessionFactory
            sessionFactory.close();
        } catch (Exception e) {
            // Handle exception and rollback JTA transactio
```

```
n
            e.printStackTrace();
            try {
                userTransaction.rollback();
            } catch (Exception rollbackException) {
                rollbackException.printStackTrace();
            }
        }
    }
}
```

Details:

- **UserTransaction**: The JTA interface for managing transactions programmatically. It provides methods to begin, commit, and rollback transactions.
- **InitialContext**: Used to perform JNDI (Java Naming and Directory Interface) lookups to obtain the `UserTransaction`.

163 CMT Transaction

Explanation:
Container-Managed Transactions (CMT) are managed by the EJB (Enterprise JavaBeans) container, relieving the application developer from explicitly managing transactions. The container handles the transaction lifecycle, ensuring that transactions are started and committed or rolled back as needed.

Example:

```
import javax.ejb.Stateless;
import javax.ejb.TransactionManagement;
import javax.ejb.TransactionManagementType;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
public class CMTTransactionExample {

    @PersistenceContext
```

```
    private EntityManager entityManager;

    public void saveEntity() {
        MyEntity entity = new MyEntity();
        entity.setName("Example Name");
        entityManager.persist(entity);
    }
}
```

Details:

- **Stateless**: An annotation indicating that the EJB is a stateless session bean.
- **TransactionManagement**: An annotation specifying the transaction management type. `TransactionManagementType.CONTAINER` indicates CMT.
- **EntityManager**: The JPA interface used for interacting with the persistence context.

Summary:

- **JDBC Transaction**: Simple, direct transaction management using JDBC API.
- **JTA Transaction**: Distributed transaction management using the JTA API.
- **CMT Transaction**: Container-managed transaction management for EJBs, relieving developers from explicit transaction management.

In the next chapter, we will delve into Hibernate Architecture, exploring `SessionFactory`, object states, and other architectural components.

Chapter 17: Hibernate Architecture

Understanding Hibernate's architecture is crucial for efficiently using this powerful ORM framework. In this chapter, we will explore two core components: `SessionFactory` and the various object states in Hibernate.

17.1 Exploring SessionFactory

Explanation:
The
`SessionFactory` is a foundational component in Hibernate's architecture. It is responsible for creating `Session` instances, which are the main point of

interaction between the application and the database. `SessionFactory` is a thread-safe, immutable object created once during application initialization and used to create multiple `Session` objects.

Key Characteristics:

- **Immutable**: Once configured, the `SessionFactory` cannot be changed.
- **Thread-safe**: It can be shared among multiple threads.
- **Heavyweight**: Due to its initialization overhead, it is advisable to create a single `SessionFactory` per database.

Example:

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

public class HibernateUtil {

    private static final SessionFactory sessionFactory;

    static {
        try {
            // Create the SessionFactory from hibernate.cf
g.xml
            sessionFactory = new Configuration().configure
().buildSessionFactory();
        } catch (Throwable ex) {
            // Log the exception as it might be swallowed
            System.err.println("Initial SessionFactory crea
tion failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

Details:

- **Configuration**: Configures Hibernate using the `hibernate.cfg.xml` file.
- **buildSessionFactory()**: Creates the `SessionFactory` instance.
- **getSessionFactory()**: Provides access to the singleton `SessionFactory`.

172 Object States

Hibernate manages the lifecycle of entity instances, which can be in one of three states: transient, persistent, or detached.

1. Transient State:
An object is in the transient state if it is newly created and not associated with any Hibernate `Session`. It has no representation in the database and is not managed by Hibernate.

```
MyEntity entity = new MyEntity();
entity.setName("Transient State");
```

2. Persistent State:
An object is in the persistent state if it is associated with an active Hibernate `Session` and is synchronized with the database. Hibernate manages the object's lifecycle and any changes made to it will be reflected in the database when the transaction is committed.

```
Session session = sessionFactory.openSession();
session.beginTransaction();
MyEntity entity = new MyEntity();
entity.setName("Persistent State");
session.save(entity);
session.getTransaction().commit();
session.close();
```

3. Detached State:
An object is in the detached state if it was once persistent but its `Session` has been closed. The object is no longer managed by Hibernate, and changes made to it will not be automatically synchronized with the database unless it is reattached to a new `Session`.

```
Session session = sessionFactory.openSession();
session.beginTransaction();
MyEntity entity = new MyEntity();
entity.setName("Detached State");
session.save(entity);
session.getTransaction().commit();
session.close();

// Detached state
entity.setName("Updated Detached State");

Session newSession = sessionFactory.openSession();
newSession.beginTransaction();
newSession.update(entity);
newSession.getTransaction().commit();
newSession.close();
```

Object Lifecycle Summary:

- **Transient**: Not associated with a session, no database representation.
- **Persistent**: Associated with a session, synchronized with the database.
- **Detached**: No longer associated with a session, changes not synchronized with the database.

Example Code Demonstrating Object States:

```
public class ObjectStateExample {

    public static void main(String[] args) {
        SessionFactory sessionFactory = HibernateUtil.getSe
ssionFactory();

        // Transient state
        MyEntity entity = new MyEntity();
        entity.setName("Transient State");

        // Persistent state
        Session session = sessionFactory.openSession();
```

```
            session.beginTransaction();
            session.save(entity);
            session.getTransaction().commit();
            session.close();

            // Detached state
            entity.setName("Updated Detached State");

            // Reattaching to a new session
            Session newSession = sessionFactory.openSession();
            newSession.beginTransaction();
            newSession.update(entity);
            newSession.getTransaction().commit();
            newSession.close();
        }
    }
```

Summary

In this chapter, we explored the `SessionFactory`, the backbone of Hibernate, responsible for creating `Session` instances. We also delved into the different object states: transient, persistent, and detached, each representing different phases in the lifecycle of an entity managed by Hibernate. Understanding these concepts is essential for effectively using Hibernate in your applications.

In the next chapter, we will introduce Hibernate caching mechanisms, exploring the different types and architecture of Hibernate cache.

Chapter 18: Introduction of Hibernate Cache

Caching is a crucial aspect of optimizing the performance of applications using Hibernate. It helps reduce the number of database hits by storing frequently accessed data in memory. In this chapter, we will explore the different types of Hibernate cache and the architecture of Hibernate caching.

18.1 Different Types of Hibernate Cache

Hibernate provides two levels of caching: the first-level cache and the second-level cache. Each serves a specific purpose in enhancing application

performance.

1. First-Level Cache (Session Cache):

- **Explanation:** The first-level cache is associated with the Hibernate `Session` object. It is enabled by default and caches objects within the scope of a session. When an object is requested, Hibernate first looks in the session cache before querying the database.
- **Scope:** Session-specific. Each session has its own first-level cache.
- **Lifecycle:** The cache is cleared when the session is closed.

Example:

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

// First query - hits the database and stores the object in
the first-level cache
MyEntity entity1 = session.get(MyEntity.class, 1);

// Second query - retrieves the object from the first-level
cache
MyEntity entity2 = session.get(MyEntity.class, 1);

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

2. Second-Level Cache:

- **Explanation:** The second-level cache is optional and provides a shared cache for all sessions. It caches objects across sessions and is configured at the `SessionFactory` level. It can significantly improve performance by reducing database access for frequently used data.
- **Scope:** Application-wide. Shared among multiple sessions.
- **Lifecycle:** The cache persists beyond the lifecycle of a single session.

Example Configuration:
To enable the second-level cache, you need to configure it in the Hibernate configuration file (
`hibernate.cfg.xml`) and specify a caching provider (e.g., Ehcache).

```
<!-- hibernate.cfg.xml -->
<hibernate-configuration>
    <session-factory>
        <!-- Other configurations -->

        <!-- Enable second-level cache -->
        <property name="hibernate.cache.use_second_level_cache">true</property>
        <property name="hibernate.cache.region.factory_class">org.hibernate.cache.ehcache.EhCacheRegionFactory</property>
        <property name="hibernate.cache.provider_class">net.sf.ehcache.hibernate.EhCacheProvider</property>

        <!-- Cache configuration for entities -->
        <mapping class="com.example.MyEntity">
            <cache usage="read-write" />
        </mapping>
    </session-factory>
</hibernate-configuration>
```

Example Usage:

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

// First query - hits the database and stores the object in the second-level cache
MyEntity entity1 = session.get(MyEntity.class, 1);
session.getTransaction().commit();
session.close();

Session anotherSession = sessionFactory.openSession();
anotherSession.beginTransaction();

// Second query - retrieves the object from the second-level cache
MyEntity entity2 = anotherSession.get(MyEntity.class, 1);
```

```
anotherSession.getTransaction().commit();
anotherSession.close();
```

182 Hibernate Cache Architecture

The architecture of Hibernate caching can be visualized as a layered structure where each layer provides different caching capabilities.

1. First-Level Cache (Session Cache):

- **Location:** Inside the `Session` object.
- **Behavior:** Each session maintains its own cache. Objects loaded during a session are stored in this cache and are cleared when the session is closed.

2. Second-Level Cache:

- **Location:** At the `SessionFactory` level, shared across all sessions.
- **Behavior:** Provides a global cache for the application. It can be configured to use various cache providers (e.g., Ehcache, Infinispan).

3. Query Cache:

- **Explanation:** An optional cache that stores the results of query executions. It works in conjunction with the second-level cache to cache the results of frequently executed queries.
- **Configuration:**

```
<property name="hibernate.cache.use_query_cache">true</property>
```

- **Example Usage:**

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

Query query = session.createQuery("from MyEntity where name = :name");
query.setParameter("name", "Example Name");
query.setCacheable(true); // Enable query cache
List<MyEntity> entities = query.list();
```

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

Summary

In this chapter, we explored the two main types of Hibernate cache: the first-level cache, which is session-specific and enabled by default, and the second-level cache, which is shared across sessions and needs to be explicitly configured. We also touched on the query cache, an optional cache for caching query results. Understanding and configuring these caching mechanisms can significantly enhance the performance of your Hibernate applications by reducing the load on the database.

In the next chapter, we will discuss Hibernate Inheritance Mapping, covering Table Per Class Mapping and Table Per Concrete Class Mapping techniques.