Day 23- Facilitation Guide

Index

For (1.5 hrs) ILT

During the previous session, we explored Hibernate Configuration,SessionFactory, Session and Hibernate states and get() and load method() method.

We covered a variety of important concepts:

Hibernate State: In Hibernate, objects can be in one of three states: Transient, Persistent, or Detached.
Transient: Newly created objects not associated with a Hibernate Session.
Persistent: Objects associated with a Session, with changes tracked.
Detached: Objects that were once persistent but are no longer associated with a Session.

Hibernate Configuration: Hibernate configuration involves specifying database connection details, mapping classes to database tables, and setting up other properties in XML files or through Java annotations.
It's essential for Hibernate to know how to interact with the database.

Session Factory: The Session Factory is a central factory class in Hibernate.It's responsible for creating Session objects, which are used to interact with the database.
It should be created once during the application's lifecycle, as it's thread-safe and relatively expensive to create.

Session: A Session is a runtime interface between a Java application and the database.It represents a unit of work, allowing you to perform database operations and manage transactions.
Sessions should be opened and closed as needed during an application's execution.

get() and load() Methods: get() and load() are methods in Hibernate used to retrieve objects from the database.

get() returns null if the object is not found in the database.
load() throws an exception if the object is not found.
The choice between these methods depends on how you want to handle non-existent objects and exceptions.

In this session, we will delve into more annotations like Embeddable annotation,Hibernate Mappings.

I. Embeddable annotation

The @Embeddable and @Embedded annotations in Hibernate are used to map an object’s properties to columns in a database table. These annotations are used in combination to allow the properties of one class to be included as a value type in another class and then be persisted in the database as part of the containing class.

Overview

- The @Embeddable annotation is used to mark a class as being embeddable, meaning its properties can be included in another class as a value type. The class marked with @Embeddable is called the embeddable class.
- The @Embedded annotation is used to mark a field in a class as being an embeddable object, and it is used in the class that contains the embeddable object.
- By using these annotations, Hibernate can automatically persist the properties of the embeddable class within the containing class to the database table, without the need to create a separate table for the embeddable class.
- Using @Embeddable and @Embedded annotations in Hibernate allows for better data modeling, code reusability, normalization, and better performance. The annotations also allow encapsulation of the business logic within the embeddable class.

Benefits

The main benefits of using the @Embeddable and @Embedded annotations in Hibernate are:

1. **Code Reusability:** You can reuse the embeddable class in multiple entities, avoiding duplication of code.
2. **Normalization:** It helps in normalizing the database by reducing the number of tables, which in turn improves the performance.
3. **Data Integrity:** It ensures data integrity by maintaining the relationship between the embeddable and the containing class.
4. **Simplicity:** It simplifies the development process by reducing the number of classes and tables required to map the data.
5. **Better Data Modelling:** It allows for better data modeling by allowing you to encapsulate the properties of an object within another object, making the data structure more intuitive and easy to understand.
6. **Ease of maintenance:** it makes the maintenance of the codebase more manageable and easy.
7. **Flexibility:** The embeddable classes can be used in multiple entities, and it also supports complex data modeling with the use of @Embedded and @AttributeOverrides.
8. **Readability:** The use of @Embeddable and @Embedded annotations makes the code more readable and self-explanatory.
9. **Performance:** it improves the performance of the application by reducing the number of joins required to fetch the data.
10. **Business logic:** It allows encapsulation of the business logic within the embeddable class, making it more manageable and easy to understand.

Let's utilize a dummy database to illustrate the entire example:

Since we've already established the "employee" table within the assignment, we can employ it for this particular example.

Embeddable Address Class:

```
package assignments;
import javax.persistence.Embeddable;
@Embeddable
public class Address {

            private String street;
            private String city;
            private String state;
            private String zip;
                      public Address(String street, String city, String state, String zip) {
                                super();
                                this.street = street;
                                this.city = city;
                                this.state = state;
                                this.zip = zip;
                      }
                      @Override
public String toString() {
          return "Employee [empId=" + empId + ", empName=" + empName + ", salary=" +
salary + ", address=" + address + "]";
}

                      }

}
```

Entity Class:

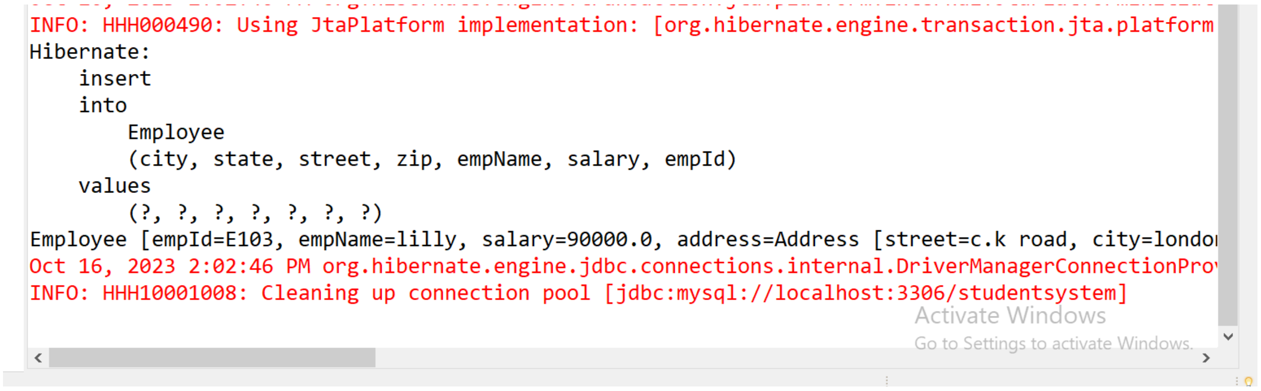
```
package assignments;
import javax.persistence.Column;
import javax.persistence.Embedded;
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
public class Employee {
@Id
private String empId;
@Column(length = 20)
private String empName;
private double salary;
@Embedded
private Address address;
public Employee() {
        super();
        // TODO Auto-generated constructor stub
}
public Employee(String empId, String empName, double salary, Address address) {
        super();
        this.empId = empId;
        this.empName = empName;
        this.salary = salary;
        this.address = address;
}
public Address() {
super();
}

@Override
public String toString() {
        return "Employee [empId=" + empId + ", empName=" + empName + ", salary=" +
salary + "]";
}
}
```

Note: You can also use the @AttributeOverrides to customize the column name mapping.

```
@Embedded
@AttributeOverrides({
        @AttributeOverride(name = "street", column = @column(name = "home_street"))
        })
        private Address address;
```

Main Class:

```
package assignments;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import assignments.*;
public class EmployeeOperation {
        public static void main(String[] args) {
                // Obtain a Hibernate SessionFactory
                SessionFactory factory = HibernateUtil.getSessionFactory();

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

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

                // create Address object
                Address address1 = new Address("c.k road", "london", "UK", "E1 7HT");

                // create Employee Object
                Employee emp1 = new Employee("E103", "lilly", 90000.0, address1);

                // Save the student to the database
                session.save(emp1);

                // Commit the transaction
                transaction.commit();

                // retrieve the data
                Employee employee = session.get(Employee.class, "E103");

                // display the data
                System.out.println(employee);

                // Close the Session
                session.close();

                // Close the Session Factory
                factory.close();
        }
}
```
Output:



```
INFO: HHH000490: Using JtaPlatform implementation: [org.hibernate.engine.transaction.jta.platform
Hibernate:
    insert
    into
        Employee
        (city, state, street, zip, empName, salary, empId)
    values
        (?, ?, ?, ?, ?, ?, ?)
Employee [empId=E103, empName=lilly, salary=90000.0, address=Address [street=c.k road, city=londo
Oct 16, 2023 2:02:46 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionPro
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/studentsystem]
```

In this example, the Address class is marked as @Embeddable, which means it can be included as a value type in another class. The Employee class has an Address field which is marked with the @Embedded annotation. This tells Hibernate that the Address object is an embeddable object and its properties should be mapped to columns in the same table as the Employee class.

When the Employee object is saved to the database, the properties of the embedded Address object will also be saved to the same table, with the column names being prefixed with the name of the field in the Employee class (in this case, “address_”).

Question: How does @Embeddable differ from @Entity in Hibernate, and when should you use each one?

II. Hibernate Association

In Hibernate, associations represent the relationships between entities (Java classes that correspond to database tables). These associations define how entities are related to each other and how they can be navigated in a database. Associations are a fundamental aspect of Hibernate, as they enable the creation of complex object-oriented data models that are persisted in a relational database. Hibernate supports several types of associations, including:

One-to-One (1:1) Association: In a one-to-one association, one entity is associated with exactly one instance of another entity. This is similar to a primary key-foreign key relationship in the database.

Example: An Employee entity associated with a ParkingSpace entity.

One-to-Many (1:N) Association: In a one-to-many association, one entity is associated with multiple instances of another entity. This is often used when one entity contains a collection of related entities.

Example: A Department entity associated with multiple Employee entities.

Many-to-One (N:1) Association: In a many-to-one association, multiple instances of one entity are associated with a single instance of another entity. This is the reverse of a one-to-many association.

Example: Multiple Employee entities associated with a single Manager entity.

Many-to-Many (N:N) Association: In a many-to-many association, multiple instances of one entity are associated with multiple instances of another entity. This is achieved using an intermediate table in the database.

Example: A Student entity associated with multiple Course entities, and vice versa.

These associations define how data in the database is related, and Hibernate uses these associations to generate the appropriate SQL statements for querying and persisting data. Associations can be established using annotations, XML configuration, or a combination of both. They also play a crucial role in shaping the object graph of your application, allowing you to navigate between related entities seamlessly.

You can find more day-to-day life examples if you observe. In database management systems one-to-one mapping is of two types-

1. One-to-one unidirectional
2. One-to-one bidirectional

Unidirectional Mapping:

- One-way Relationship: Unidirectional mapping defines a one-way relationship between two entities. In this setup, one entity knows about the other entity, but the reverse is not true.
- Simpler: Unidirectional mapping is simpler to set up, and it involves less configuration and code.

Bidirectional Mapping:

- Two-way Relationship: Bidirectional mapping establishes a two-way relationship between two entities. In this setup, each entity knows about the other and can navigate the relationship in both directions.
- Complex Configuration: Bidirectional mapping requires more configuration and is generally more complex. However, it allows for more efficient querying and better data consistency.

Question: What are the key differences between one-to-one, one-to-many, and many-to-many associations in Hibernate?

1.One-to-One (1:1) Association:

One to one represents that a single entity is associated with a single instance of the other entity. An instance of a source entity can be at most mapped to one instance of the target entity. We have a lot of examples around us that demonstrate this one-to-one mapping.

- One person has one passport, a passport is associated with a single person.
- Leopards have unique spots, a pattern of spots is associated with a single leopard.
- We have one college ID, a college ID is uniquely associated with a person.

To create a one-to-one mapping using annotations, you can use the @OneToOne annotation on one side of the relationship and the @JoinColumn annotation to specify the foreign key column.

Since we've already established the "employee" table within the assignment, we can employ it for this particular example of one to one Unidirectional mapping.

Employee Entity Class:

```
package assignments;

import javax.persistence.Column;
import javax.persistence.Embedded;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;
```

```
@Entity
public class Employee {
@Id
private String empId;

@Column(length = 20)
private String empName;

private double salary;

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

public Employee() {
	super();
	// TODO Auto-generated constructor stub
}

public Employee(String empId, String empName, double salary, Address address) {
	super();
	this.empId = empId;
	this.empName = empName;
	this.salary = salary;
	this.address = address;
}

@Override
public String toString() {
	return "Employee [empId=" + empId + ", empName=" + empName + ", salary=" +
salary + ", address=" + address + "]";
}

}
```

Address Entity Class:

```
package assignments;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Address {
	@Id
	private int addressId;
	@Column(length = 50)
	private String street;
	@Column(length = 10)
	private String city;
	@Column(length = 10)
	private String state;
	@Column(length = 10)
	private String zip;
	public Address(int addressId, String street, String city, String state, String zip) {
		super();
		this.addressId = addressId;
		this.street = street;
		this.city = city;
		this.state = state;
		this.zip = zip;
	}
	@Override
	public String toString() {
		return "Address [street=" + street + ", city=" + city + ", state=" + state + ",
zip=" + zip + "]";
	}
	public Address() {
		super();
		// TODO Auto-generated constructor stub
	}
}
```

HibernateUtil class:

```
package assignments;

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

import com.sms.Student;
```

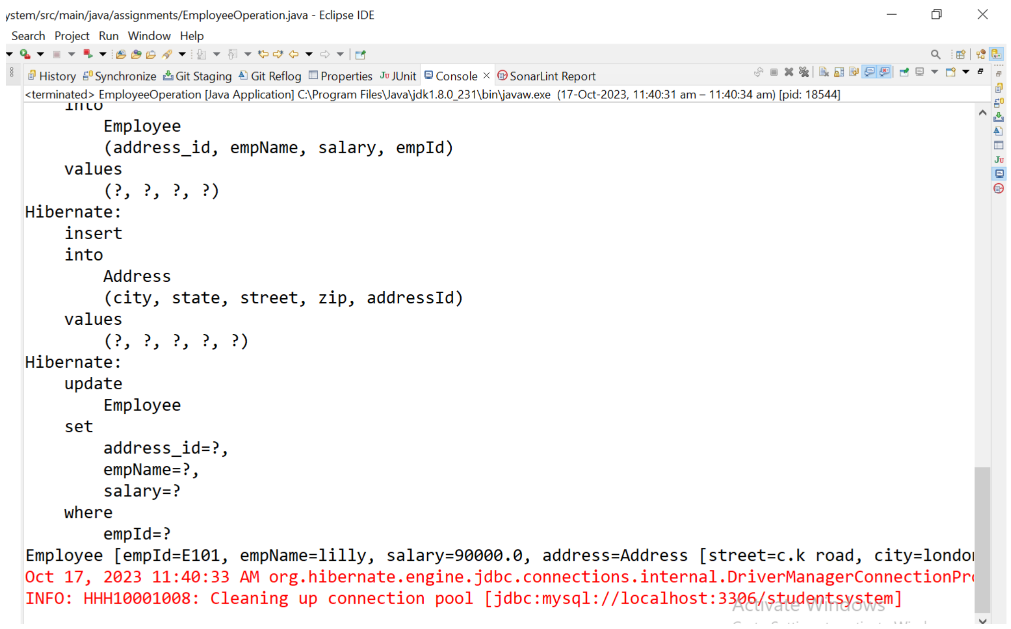
```
public class HibernateUtil {
	private static final SessionFactory sessionFactory = buildSessionFactory();

	private static SessionFactory buildSessionFactory() {
		try {
			return new
Configuration().configure("hibernate.cfg.xml").addAnnotatedClass(Student.class)

.addAnnotatedClass(Employee.class).addAnnotatedClass(Address.class).buildSession
Factory();
		} catch (Throwable ex) {
			throw new ExceptionInInitializerError(ex);
		}
	}

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

Main Class:

```
package assignments;

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

import assignments.*;

public class EmployeeOperation {

	public static void main(String[] args) {
		// Obtain a Hibernate SessionFactory
		SessionFactory factory = HibernateUtil.getSessionFactory();
		// Open a new session
		Session session = factory.openSession();

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

```
                // create Address object

                Address address1 = new Address(1,"c.k road", "london", "UK", "E1 7HT");
                // create Employee Object
                Employee emp1 = new Employee("E101", "lilly", 90000.0, address1);

                // Save the student to the database
                session.save(emp1);

                // Save the address to the database
                session.save(address1);
                // Commit the transaction
                transaction.commit();

                // retrieve the data
                Employee employee = session.get(Employee.class, "E101");

                // display the data
                System.out.println(employee);
                // Close the Session
                session.close();

                // Close the Session Factory
                factory.close();
        }

}
```

Output:


ystem/src/main/java/assignments/EmployeeOperation.java - Eclipse IDE
Search Project Run Window Help
History Synchronize Git Staging Git Reflog Properties JUnit Console SonarLint Report

```
<terminated> EmployeeOperation [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (17-Oct-2023, 11:40:31 am – 11:40:34 am) [pid: 18544]
    into
        Employee
        (address_id, empName, salary, empId)
    values
        (?, ?, ?, ?)
Hibernate:
    insert
    into
        Address
        (city, state, street, zip, addressId)
    values
        (?, ?, ?, ?, ?)
Hibernate:
    update
        Employee
    set
        address_id=?,
        empName=?,
        salary=?
    where
        empId=?
Employee [empId=E101, empName=lilly, salary=90000.0, address=Address [street=c.k road, city=londo
Oct 17, 2023 11:40:33 AM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionPr
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/studentsystem]
```

2.One-to-Many (1:N) Association and Many-to-One (N:1) Association:

One-to-Many (1:N) Association:

- In a one-to-many association, one entity is associated with multiple instances of another entity.
- It's a unidirectional relationship where one entity (the "one" side) has a collection of related entities on the "many" side.
- Each related entity on the "many" side has a reference to a single parent entity on the "one" side.
- This is typically used when modeling relationships where multiple entities are related to a single entity, like a student having many enrollments.

Many-to-One (N:1) Association:

- In a many-to-one association, multiple instances of one entity (the "many" side) are associated with a single instance of another entity (the "one" side).
- It's the reverse of a one-to-many relationship.
- Each related entity on the "many" side has a reference to a single parent entity on the "one" side.
- This is typically used when modeling relationships where multiple entities are related to a single entity, like many enrollment Ids are mapped with one Student id.

Let’s have a complete example of One-to-many and Many-to-one Bidirectional mapping example where we will demonstrate One Student maps to Many Enrolment ids (One-to-Many) and Many Enrolment ids map to one Student (Many-to-One):

Student Entity Class:

```
package com.sms;
import java.time.LocalDate;
import java.util.List;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.OneToMany;
@Entity
public class Student {
        @Id
        @Column(name = "StudentId", length = 10)
        private String studentId;
        @Column(name = "FirstName", length = 50)
        private String firstName;
        @Column(name = "LastName", length = 25)
        private String lastName;
        @Column(name = "DateOfBirth")
        private LocalDate dateOfBirth;
        @Column(name = "Gender", length = 25)
        private String gender;
        @Column(name = "Email", length = 30)
        private String email;
        @Column(name = "Phone", length = 25)
        private String phone;
        @OneToMany(mappedBy = "studentId")
        private List<Enrollment> enrollments;

        public List<Enrollment> getEnrollments() {
                return enrollments;
        }
        public void setEnrollments(List<Enrollment> enrollments) {
                this.enrollments = enrollments;
        }
        //Setter And Getter
```

```
public String getStudentId() {
        return studentId;
}
public void setStudentId(String studentId) {
        this.studentId = studentId;
}
public String getFirstName() {
        return firstName;
}
public void setFirstName(String firstName) {
        this.firstName = firstName;
}
public String getLastName() {
        return lastName;
}
public void setLastName(String lastName) {
        this.lastName = lastName;
}
public LocalDate getDateOfBirth() {
        return dateOfBirth;
}
public void setDateOfBirth(LocalDate dateOfBirth) {
        this.dateOfBirth = dateOfBirth;
}
public String getGender() {
        return gender;
}
public void setGender(String gender) {
        this.gender = gender;
}
public String getEmail() {
        return email;
}
public void setEmail(String email) {
        this.email = email;
}
public String getPhone() {
        return phone;
}
public void setPhone(String phone) {
		this.phone = phone;
	}
	//Default Constructor
	public Student() {
		super();
		// TODO Auto-generated constructor stub
	}

	//All argument Constructor
	public Student(String studentId, String firstName, String lastName, LocalDate
dateOfBirth, String gender,
			String email, String phone) {
		super();
		this.studentId = studentId;
		this.firstName = firstName;
		this.lastName = lastName;
		this.dateOfBirth = dateOfBirth;
		this.gender = gender;
		this.email = email;
		this.phone = phone;
	}

	//ToString method
	@Override
	public String toString() {
		return "Student [studentId=" + studentId + ", firstName=" + firstName + ",
lastName=" + lastName
				+ ", dateOfBirth=" + dateOfBirth + ", gender=" + gender + ",
email=" + email + ", phone=" + phone
				+ ", enrollments=" + enrollments + "]";
	}
}
```

Enrollment Entity Class:

```
package com.sms;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
@Entity
public class Enrollment {
@Id
private String EnrollmentID;
@ManyToOne
@JoinColumn(name="StudentId")
private Student studentId;
@Column(length = 10)
private String courseID;
@Column(length = 10)
private String InstructorID;
public Enrollment() {
	super();
	// TODO Auto-generated constructor stub
}
public Enrollment(String enrollmentID, Student studentId, String courseID, String
instructorID) {
	super();
	EnrollmentID = enrollmentID;
	this.studentId = studentId;
	this.courseID = courseID;
	InstructorID = instructorID;
}
public String getEnrollmentID() {
	return EnrollmentID;
}
public void setEnrollmentID(String enrollmentID) {
	EnrollmentID = enrollmentID;
}
public Student getStudentId() {
	return studentId;
}
public void setStudentId(Student studentId) {
	this.studentId = studentId;
}
public String getCourseID() {
	return courseID;
}
public void setCourseID(String courseID) {
        this.courseID = courseID;
}
public String getInstructorID() {
        return InstructorID;
}
public void setInstructorID(String instructorID) {
        InstructorID = instructorID;
}
@Override
public String toString() {
        return "Enrollment [EnrollmentID=" + EnrollmentID + ", studentId=" + studentId +
", courseID=" + courseID
                        + ", InstructorID=" + InstructorID + "]";
}
}
```

HibernateUtil Class:
package com.sms;

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

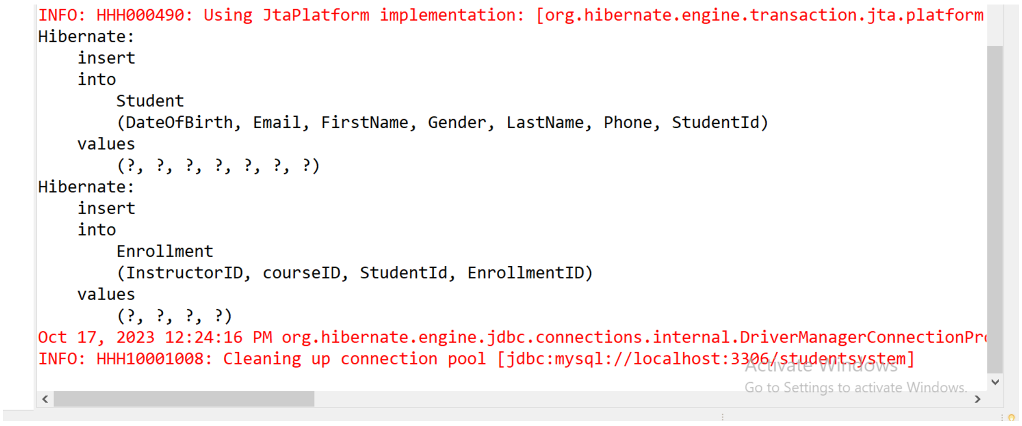
```
public class HibernateUtil {
        private static final SessionFactory sessionFactory = buildSessionFactory();

        private static SessionFactory buildSessionFactory() {
                try {
                        return new Configuration().configure("hibernate.cfg.xml")
                                        .addAnnotatedClass(Student.class)
                                        .addAnnotatedClass(Enrollment.class)
                                        .buildSessionFactory();
                } catch (Throwable ex) {
                        throw new ExceptionInInitializerError(ex);
                }
        }

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

Main class:

```
package com.sms;
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
public class JoiningOperaions {
	public static void main(String[] args) {
		// Obtain a Hibernate SessionFactory
		SessionFactory factory = HibernateUtil.getSessionFactory();
		// Create a new Student
		LocalDate date1 = LocalDate.of(1988, 1, 13);
		Student student1 = new Student("S112", "Oliver", "Henry", date1, "M",
"oliver@gmail.com", "6742906745");
		// Create Enrollment
		Enrollment enrollment1 = new Enrollment("E1001", student1, "C101",
"I101");
		// set EnrollmentID to student
		List<Enrollment> enrollments = new ArrayList<Enrollment>();
		enrollments.add(enrollment1);
		student1.setEnrollments(enrollments);
		// Open a new session
		Session session = factory.openSession();
		// Begin a transaction
		Transaction transaction = session.beginTransaction();
		// Save the student to the database
		session.save(student1);
		// Save the enrollement to the database
		session.save(enrollment1);
		// Commit the transaction
		transaction.commit();
		// Close the Session
		session.close();
		// Close the Session Factory
		factory.close();
	}
```

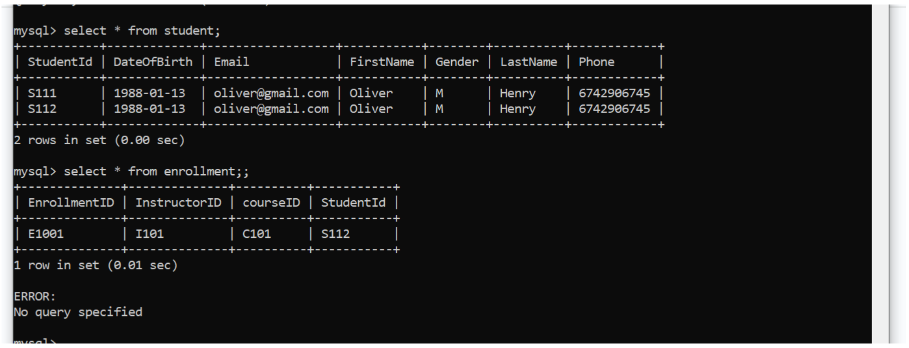
}

Output:



```
INFO: HHH000490: Using JtaPlatform implementation: [org.hibernate.engine.transaction.jta.platform
Hibernate: 
    insert 
    into
        Student
        (DateOfBirth, Email, FirstName, Gender, LastName, Phone, StudentId) 
    values
        (?, ?, ?, ?, ?, ?, ?)
Hibernate: 
    insert 
    into
        Enrollment
        (InstructorID, courseID, StudentId, EnrollmentID) 
    values
        (?, ?, ?, ?)
Oct 17, 2023 12:24:16 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionPr
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/studentsystem]
```

Database:



```
mysql> select * from student;
+-----------+-------------+------------------+-----------+--------+----------+------------+
| StudentId | DateOfBirth | Email            | FirstName | Gender | LastName | Phone      |
+-----------+-------------+------------------+-----------+--------+----------+------------+
| S111      | 1988-01-13  | oliver@gmail.com | Oliver    | M      | Henry    | 6742906745 |
| S112      | 1988-01-13  | oliver@gmail.com | Oliver    | M      | Henry    | 6742906745 |
+-----------+-------------+------------------+-----------+--------+----------+------------+
2 rows in set (0.00 sec)

mysql> select * from enrollment;;
+--------------+--------------+----------+-----------+
| EnrollmentID | InstructorID | courseID | StudentId |
+--------------+--------------+----------+-----------+
| E1001        | I101         | C101     | S112      |
+--------------+--------------+----------+-----------+
1 row in set (0.01 sec)

ERROR:
No query specified

mysql>
```

Exercise:

Use ChatGPT to explore Many To Many Association:

Put the below problem statement in the message box and see what ChatGPT says.

I am currently studying Hibernate associations and I am eager to delve into the concept of Many-to-Many associations. Could you please provide me with some sample code and direct me to a tutorial to facilitate my learning process?