JPA & Hibernate ORM

Topics

1. JPA and Hibernate ORM Introduction
2. JPA annotations
3. Entity Manager
4. Fetching and Cascading Strategies
5. JPQL and Criteria API
6. Repository Pattern

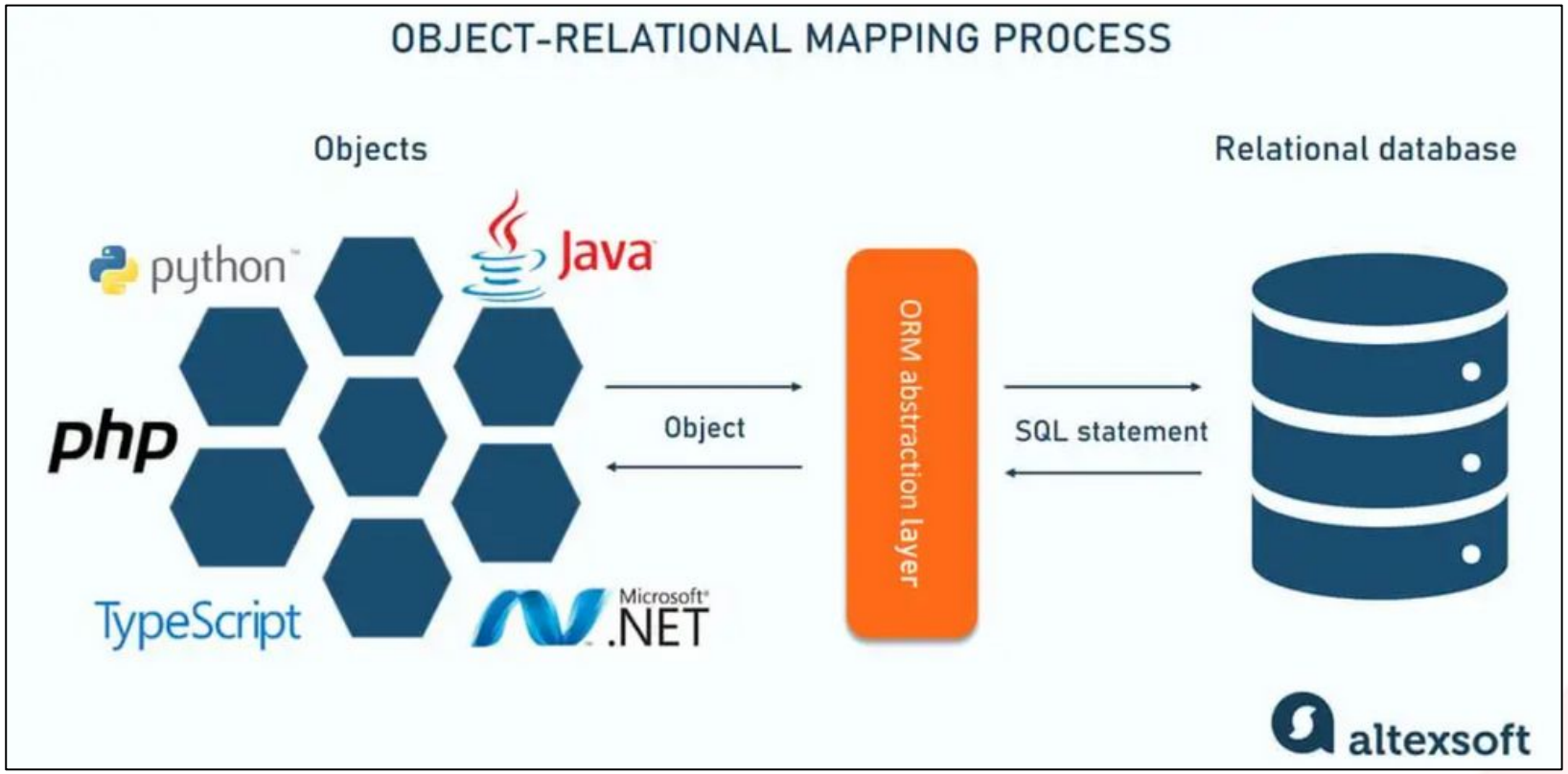
Jpa and Hibernate ORM Introduction

Objectives

1. Understand what ORM is.
2. Understand what JPA and Hibernate are and their relationship.
3. Understand the benefits of using JPA/Hibernate.

ORM

Object Relational Mapping (ORM) is a technique used in creating a "bridge" between object-oriented applications and relational databases.



Problem

In Java we work with **objects**, databases we work with **tables**.

So how do you take a Java object...

```
Product product = new Product(1L, “Macbook Pro M4”, “Laptop”);
```

...and save it into a database table?

```
INSERT INTO products (id, name, category) VALUES (1L, ‘Macbook Pro M4’, ‘Laptop’);
```

We could use JDBC and write SQL manually for everything — but it’s repetitive, error-prone, and a lot of work.

The JDBC way

```
Product product = new Product();
product.setProductName("Macbook Pro M4");
product.setCategory("Laptop");

String sql = "INSERT INTO products (id, name, category) VALUES (?, ?, ?)"

// 2. Prepare SQL statement
ps = conn.prepareStatement(sql);

// 3. Bind parameters
ps.setLong(1, product.getId());
ps.setString(2, product.getName());
ps.setString(3, product.getCategory());

// 4. Execute
ps.executeUpdate();
```

The ORM way

ORM automatically generates SQL and maps Java objects to database tables.

Example:

When you write:

```
Product product = new Product();
product.setProductName("Macbook Pro M4");
product.setCategory("Laptop");

em.persist(product);
em.getTransaction().commit();
```

Hibernate (an ORM) will generate SQL for you:

INSERT INTO products (id, name, category) VALUES (1L, 'Macbook Pro M4', 'Laptop');

➔ **ORM lets you work with the database using Java objects instead of SQL statements.**

Instead of writing this:

```sql
SELECT * FROM product WHERE id = 1;
```

You do this:

```java
Product p = entityManager.find(Product.class, 1);
```

And the result is already mapped to Product p, no more manual mapping of resultset.

JDBC mapping resultset

```
while (rs.next()) {

    int id = rs.getInt("id");
    String name = rs.getString("name");
    String category = rs.getString("category");

    Product product = new Product();
    student.setId(id);
    student.setName(name);
    student.setCategory(age);

}
```

What is JPA?

JPA stands for **Java Persistence API.**

Persistence = object state stored beyond program execution
API = the standardized set of interfaces and annotations

JPA (Java Persistence API) is a standard specification that defines how Java objects are mapped to and persisted in relational database tables.

JPA is a standard of doing Object Relation Mapping (ORM) in java.

Hibernate

Hibernate is an ORM framework that implements the JPA specification.

When you call the find method of EntityManager (interface from the JPA package):

```
Student student = em.find(Student.class, 1L);
```

Hibernate is the one who actually:

- opens the database connection
- generates the SELECT statement
- maps the results to the Student object
- closes the connection

Reasons to Use JPA

1. **Developer Productivity**
   - Automatic SQL generation
   - automatic mapping/binding
   - APIs for CRUD operations
   - reduces boilerplate code.
2. **ORM and Database Independent**
   - JPA allows you to switch between any JPA-compliant ORM providers, such as Hibernate, EclipseLink, or DataNucleus, without changing application code.
   - It also allows you to change databases simply by updating the SQL driver.
   - Implementations are abstracted by JPA APIs (EntityManager, Criteria API) and JPQL.
3. **Performance**
   - Caching(Reduced database access), Fetching and Cascading Strategies

Summary

ORM
idea or technique

JPA
standards

Hibernate
the tool that follows those standards and does the implementation

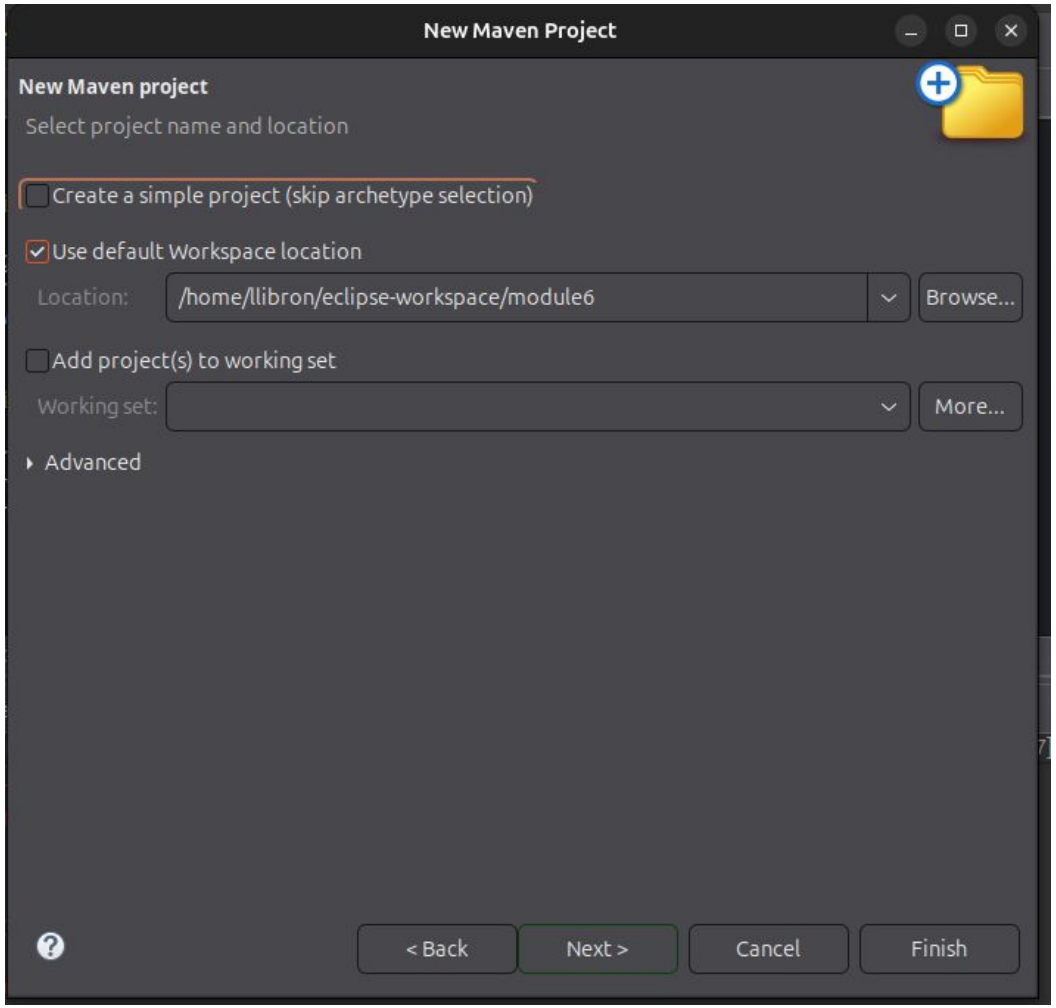
Hibernate Setup and Database Connection

Prerequisite

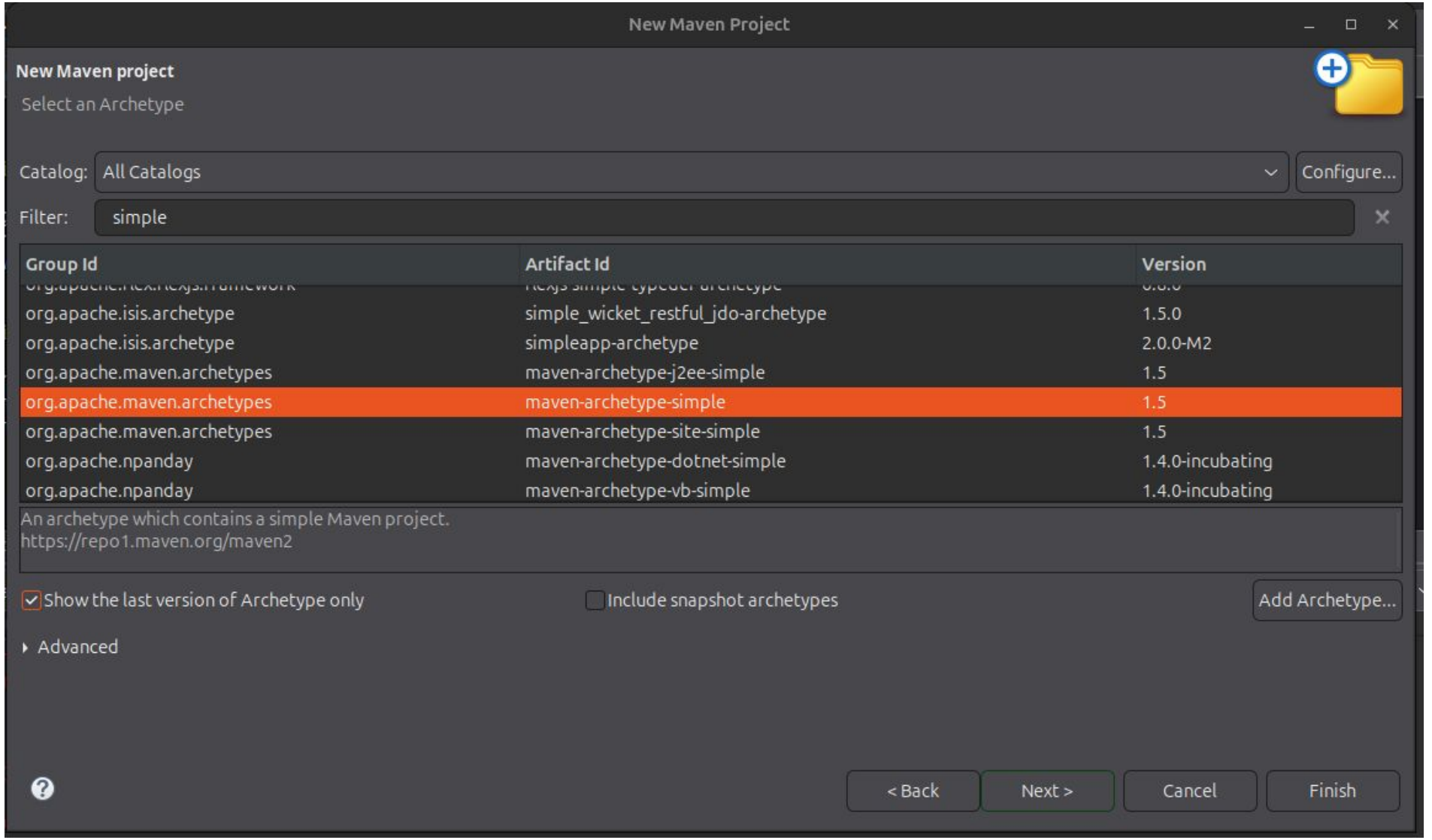
1. **Maven** is already installed. You can verify by opening a terminal and running “mvn -version”.

2. **Database Server** is running.
    - note the **url, username, and password** of your database

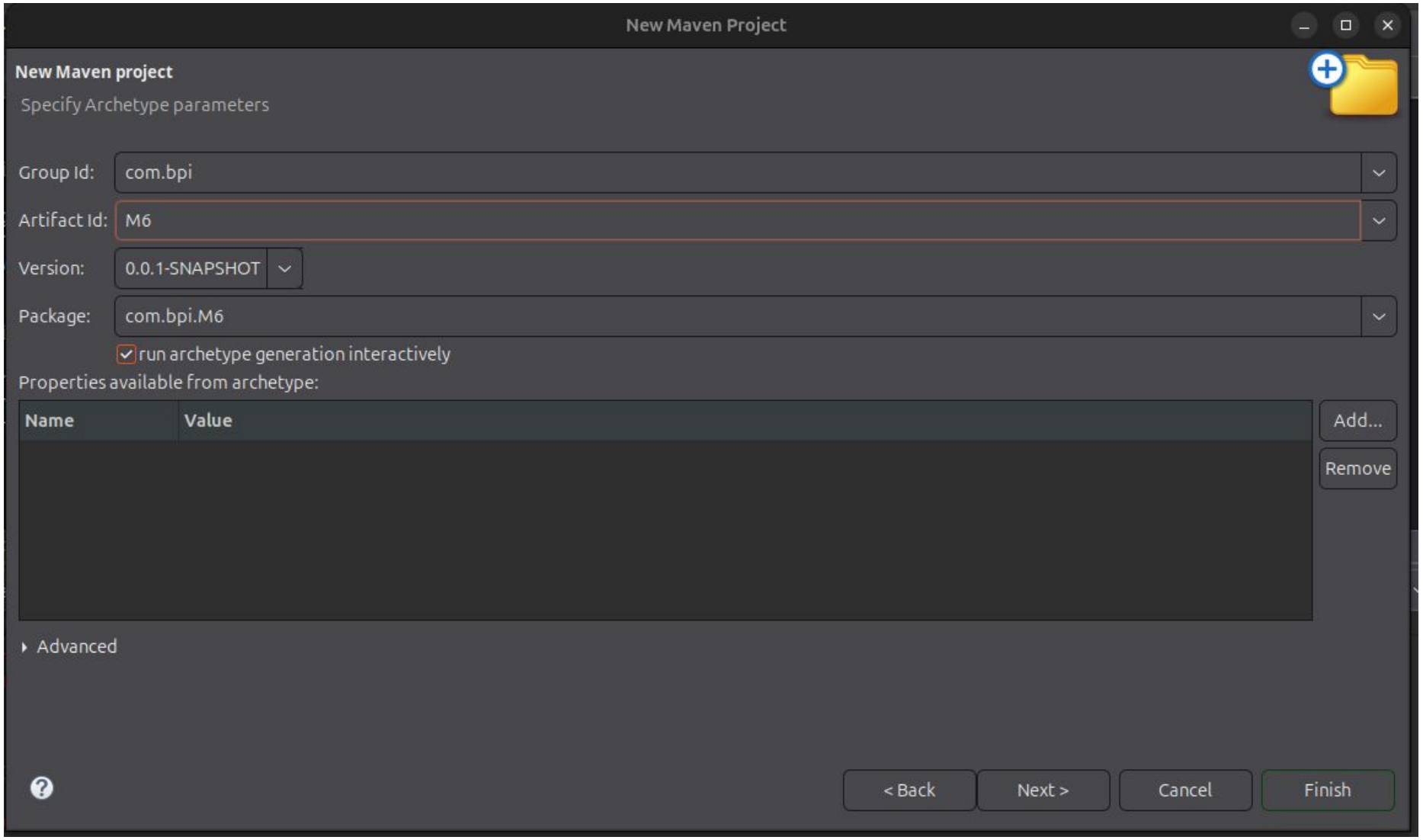
Create Maven Project



Create Maven Project



Create Maven Project



Create Maven Project

After Maven downloads the project, it will prompt you to check the project details. Type 'Y' and press Enter to confirm.

```
Downloaded from central: https://repo.maven.apache.org/maven2/archetype-catalog.xml (18 MB at 23 MB/s)
[INFO] Archetype repository not defined. Using the one from [org.apache.maven.archetypes:maven-archetype-simple:1.5] found in catalog remote
[INFO] Using property: groupId = com.bpi
[INFO] Using property: artifactId = M6
[INFO] Using property: version = 0.0.1-SNAPSHOT
[INFO] Using property: package = com.bpi.M6
Confirm properties configuration:
groupId: com.bpi
artifactId: M6
version: 0.0.1-SNAPSHOT
package: com.bpi.M6
 Y:
```

Create Maven Project

```
[INFO] Using property: version = 0.0.1-SNAPSHOT
[INFO] Using property: package = com.bpi.M6
Confirm properties configuration:
groupId: com.bpi
artifactId: M6
version: 0.0.1-SNAPSHOT
package: com.bpi.M6
 Y: Y
[INFO] ----------------------------------------------------------------------------
[INFO] Using following parameters for creating project from Archetype: maven-archetype-simple:1.5
[INFO] ----------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.bpi
[INFO] Parameter: artifactId, Value: M6
[INFO] Parameter: version, Value: 0.0.1-SNAPSHOT
[INFO] Parameter: package, Value: com.bpi.M6
[INFO] Parameter: packageInPathFormat, Value: com/bpi/M6
[INFO] Parameter: package, Value: com.bpi.M6
[INFO] Parameter: groupId, Value: com.bpi
[INFO] Parameter: artifactId, Value: M6
[INFO] Parameter: version, Value: 0.0.1-SNAPSHOT
[WARNING] Don't override file /home/llibron/eclipse-workspace/M6/src/main/java/com/bpi/M6
[WARNING] Don't override file /home/llibron/eclipse-workspace/M6/src/test/java/com/bpi/M6
[WARNING] Don't override file /home/llibron/eclipse-workspace/M6/src/site
[WARNING] CP Don't override file /home/llibron/eclipse-workspace/M6/.mvn
[INFO] Project created from Archetype in dir: /home/llibron/eclipse-workspace/M6
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  02:57 min
[INFO] Finished at: 2026-01-27T14:46:38+08:00
[INFO] ------------------------------------------------------------------------
```

Setting up - pom.xml

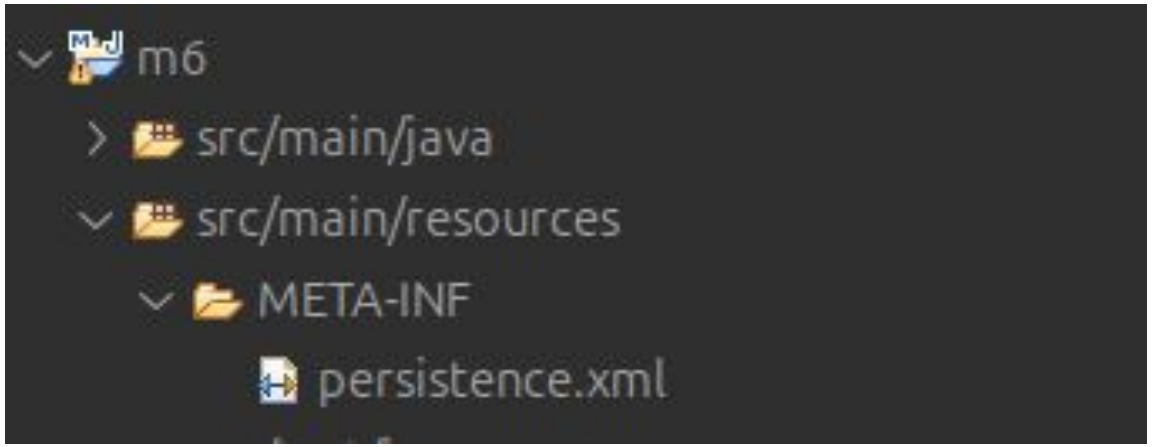
1. **Add Dependencies in pom.xml**
   - Include the database driver and Hibernate in the dependencies

```
<dependencies>

        <!-- https://mvnrepository.com/artifact/org.hibernate.orm/hibernate-core -->
        <dependency>
                <groupId>org.hibernate.orm</groupId>
                <artifactId>hibernate-core</artifactId>
                <version>7.1.11.Final</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.postgresql/postgresql -->
        <dependency>
                <groupId>org.postgresql</groupId>
                <artifactId>postgresql</artifactId>
                <version>42.7.8</version>
        </dependency>

</dependencies>
```

2. run "**maven clean install**"

Setting up - persistence.xml

2. **Add resources/META-INF/persistence.xml in src/main**
   - create a folder named META-INF in resource directory
   - create a file inside META-INF named persistence.xml
   - **reference**: https://github.com/enzolibronexist/module6/blob/setup/src/main/resources/META-INF/persistence.xml



Setting up - persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
                        https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
        version="3.0">

        //persistence unit tells JPA how to connect to the database, which entities to manage, and which settings to use.
        <persistence-unit name="default"
                transaction-type="RESOURCE_LOCAL">
                <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

                <properties>
                        <!-- JDBC connection -->
                        <property name="jakarta.persistence.jdbc.driver" value="org.postgresql.Driver" />
                        <property name="jakarta.persistence.jdbc.url" value="jdbc:postgresql://localhost:5432/sandboxdb" />
                        <property name="jakarta.persistence.jdbc.user" value="user" />
                        <property name="jakarta.persistence.jdbc.password" value="password" />

                        <!-- Hibernate dialect -->
                        <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect" />
                        <property name="hibernate.hbm2ddl.auto" value="none" />
                        <property name="hibernate.show_sql" value="true" />
                </properties>
        </persistence-unit>
</persistence>
```

Create a Utility class for Entity Manager

```java
public class EntityManagerUtil {

    private static EntityManagerUtil instance;

    private final EntityManagerFactory emf;

    private EntityManagerUtil() {
        this.emf = Persistence.createEntityManagerFactory("default");
    }

    public static synchronized EntityManagerUtil getInstance() {
        if (instance == null) {
            instance = new EntityManagerUtil();
        }
        return instance;
    }

    public EntityManager createEntityManager() {
        return emf.createEntityManager();
    }

    public boolean isOpen(EntityManager em) {
        return em != null && em.isOpen();
    }

    public void closeEntityManager(EntityManager em) {
        if (isOpen(em)) {
            em.close();
        }
    }

    public void shutdownFactory() {
        if (emf.isOpen()) {
            emf.close();
        }
    }
}
```

In the main application

```
public class App {

        public static void main(String[] args) {
          testConnection();
        }

        static void testConnection() {
          EntityManager em = EntityManagerUtil.getInstance().createEntityManager();

          try {
               if(em.isOpen()) {
                    System.out.println("entity manager open, ready to create transaction");
               }

          } finally {
               EntityManagerUtil.getInstance().closeEntityManager(em);
               EntityManagerUtil.getInstance().shutdownFactory();
          }
       }
}
```

Run the application

Run the application and check if it works. You should see the following in the console log:

```
Jan 24, 2026 8:59:02 PM org.hibernate.jpa.internal.util.LogHelper logPersistenceUnitInformation
INFO: HHH008540: Processing PersistenceUnitInfo [name: default]
Jan 24, 2026 8:59:02 PM org.hibernate.Version logVersion
INFO: HHH000001: Hibernate ORM core version 7.2.0.Final
Jan 24, 2026 8:59:02 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProvider configure
WARN: HHH10001002: Using built-in connection pool (not intended for production use)
Jan 24, 2026 8:59:02 PM org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator logConnectionInfo
INFO: HHH10001005: Database info:
      Database JDBC URL [jdbc:postgresql://localhost:5432/sandboxdb]
      Database driver: PostgreSQL JDBC Driver
      Database dialect: PostgreSQLDialect
      Database version: 18.0
      Default catalog/schema: sandboxdb/public
      Autocommit mode: false
      Isolation level: READ_COMMITTED
      JDBC fetch size: none
      Pool: DriverManagerConnectionProvider
      Minimum pool size: 1
      Maximum pool size: 20
Jan 24, 2026 8:59:02 PM org.hibernate.engine.transaction.jta.platform.internal.JtaPlatformInitiator initiateService
INFO: HHH000489: No JTA platform available (set 'hibernate.transaction.jta.platform' to enable JTA platform integration)
entity manager open, ready to create transaction
```

M6_Activity1 - Project Setup

Objective:

1. Create a Maven project.
2. Add the Hibernate ORM and PostgreSQL driver dependencies to the pom.xml file.
3. Configure the persistence.xml file.
4. Create a utility class responsible for managing the EntityManager.
5. Use the utility class to test creating a entity manager.

M6_Activity1 - Output

```
Jan 24, 2026 8:59:02 PM org.hibernate.jpa.internal.util.LogHelper logPersistenceUnitInformation
INFO: HHH008540: Processing PersistenceUnitInfo [name: default]
Jan 24, 2026 8:59:02 PM org.hibernate.Version logVersion
INFO: HHH000001: Hibernate ORM core version 7.2.0.Final
Jan 24, 2026 8:59:02 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProvider configure
WARN: HHH10001002: Using built-in connection pool (not intended for production use)
Jan 24, 2026 8:59:02 PM org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator logConnectionInfo
INFO: HHH10001005: Database info:
      Database JDBC URL [jdbc:postgresql://localhost:5432/sandboxdb]
      Database driver: PostgreSQL JDBC Driver
      Database dialect: PostgreSQLDialect
      Database version: 18.0
      Default catalog/schema: sandboxdb/public
      Autocommit mode: false
      Isolation level: READ_COMMITTED
      JDBC fetch size: none
      Pool: DriverManagerConnectionProvider
      Minimum pool size: 1
      Maximum pool size: 20
Jan 24, 2026 8:59:02 PM org.hibernate.engine.transaction.jta.platform.internal.JtaPlatformInitiator initiateService
INFO: HHH000489: No JTA platform available (set 'hibernate.transaction.jta.platform' to enable JTA platform integration)
entity manager open, ready to create transaction
```

JPA Annotations

Objectives

1. **Understand what Entities are**
2. **Learn to use JPA Annotations to:**
    - Create Entities
    - Map entity fields to table columns
    - Model entity relationships

Java Annotations

Java annotations are labels you attach to code (classes, methods, fields, etc.) to provide extra information to the compiler or frameworks/libraries (Hibernate).

They do not change the logic of the code.

Instead, they tell the compiler, tools and frameworks(like Hibernate) how to treat your code.

By using annotations, you can significantly reduce boilerplate code—they replace repetitive configuration and setup (like XML or manual wiring) with simple, declarative metadata, making the code cleaner, shorter, and easier to maintain.

Java Annotations Sample

```java
//Class-level annotation
@SuppressWarnings("all") // suppress compiler warnings for the whole class
public class Calculator {

    // Field-level annotation
    @Deprecated // marks the field as outdated
    private int lastResult;
    private String name = "Simple Calculator";

    // Method-level annotations
    @Override // indicates we are overriding a method from Object
    public String toString() {
        return name + " (lastResult=" + lastResult + ")";
    }
    @Deprecated // mark method as outdated
    public int add(int a, int b) {
        lastResult = a + b;
        return lastResult;
    }
    @SuppressWarnings("unused") // suppress warnings in this method
    public int multiply(int a, int b) {
        lastResult = a * b;
        return lastResult;
    }
}
```

JPA Annotations

JPA annotations are used to map Java classes to database tables and define relationships, constraints, and behaviors for entity classes.

What are Entities in JPA?

- Entities in JPA are java objects representing data that can be persisted in the database.
- An entity class represents a table. Every instance of an entity class represents a row in the table.

Entity Representation:
Entity class → Database table
Entity fields inside the entity class → Table column
Entity object (instance of entity class) → A row in the table

Entity class

```java
@Entity
@Table(name = "students")
public class Student {

    @Id
    private Long id;

    private String name;
    private int age;
    private String email;

}
```

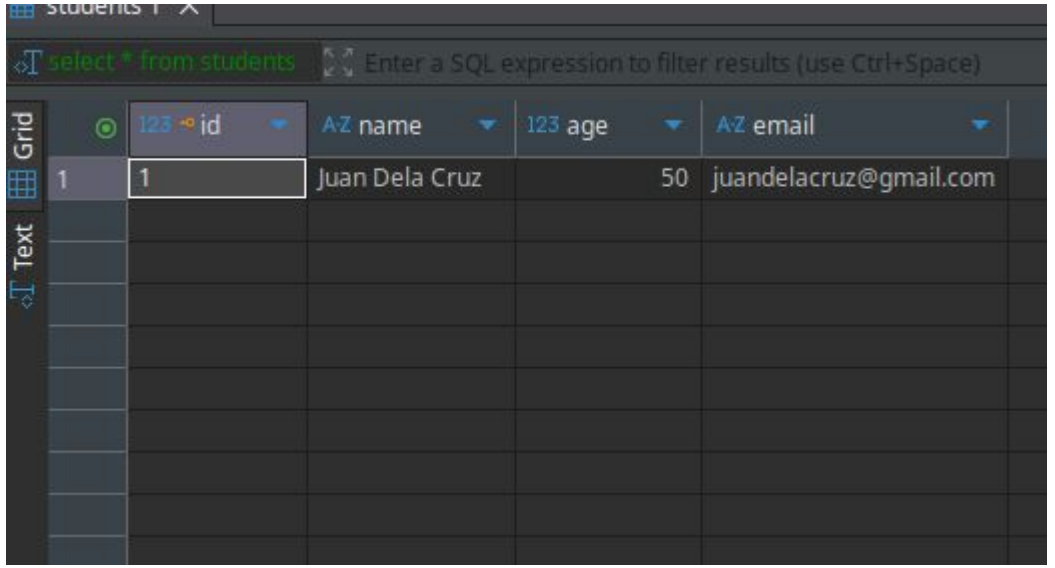
For example, Student class is an **entity class** because it represents the students table in the database

students	
PK	id
	name
	age
	email

Entity object

```
Student newStudent = new Student();
newStudent.setName("Juan Dela Cruz");
newStudent.setAge(50);
newStudent.setEmail("juandelacruz@gmail.com");
```

The object newStudent, instantiated from the Student entity class, is an **entity object** and represents a row in the database when it is persisted.



@Entity

- Marks a class as an entity class, making objects instantiated from this class eligible for persistence.

Example:

```
@Entity
public class Student {}
```

- By default, hibernate uses the class name as the table and entity name. In our example, hibernate will translate it as “Student”

What if the table in the database have a different name, let say “students”?

@Table

- In most cases, the name of the table in the database and the name of the entity won't be the same
- In these cases, we can specify the table name using this annotation

for example:

```
@Entity
@Table(name = "students")
public class Student {}
```

@Id

- **@Id** is a JPA annotation used to mark the **primary key** of an entity.

- **Every JPA entity class must have exactly one primary key because JPA uses it to:**
  - Identify each entity instance
  - Track changes
  - Manage caching
  - Perform updates and deletion correctly

- An error will occur if the entity does not have a field annotated with @Id.

```
Caused by: org.hibernate.AnnotationException: Entity 'com.bpi.module6.model.Product' has no
identifier (every '@Entity' class must declare or inherit at least one '@Id' or '@EmbeddedId'
property)
```

@Id

- Example:

```
@Entity
@Table(name = "students")
public class Student {

    @Id
    private Long id;

}
```

- What if the primary key in the students table has a different name? For example, “student_id”

  We can **explicitly** map the entity field detail to the table column detail by using **@Column**

@Column

@Column is a JPA annotation used to explicitly specify the mapping of a entity field to a database column.

By default, Hibernate maps entity fields to columns with the same name, but @Column lets you:

- Specify a different column name
- Define constraints like nullable or unique
- Specify the length, precision, and scale
- Specify SQL type with columnDefinition

@Column

- For example our columns look like this:

```sql
CREATE TABLE students (
     student_id SERIAL PRIMARY KEY NOT NULL,
     student_name VARCHAR(50) NOT NULL,
     student_age INT,
     email VARCHAR(100) UNIQUE
);
```

- We will explicitly map the details using @Column attributes

```java
@Id
@Column(name = "student_id")
private Long id;

@Column(name = "student_name", nullable = false, length = 50, columnDefinition = "VARCHAR(50)")
private String name;

@Column(name = "age", columnDefinition = "INT")
private int age;

@Column(name = "email", unique = true, length = 100, columnDefinition = "VARCHAR(100)")
private String email;
```

@Column

Attribute	Purpose
name	Maps the field to a specific column name (product_name in this case).
nullable	Whether the column can accept NULL values (false means NOT NULL).
length	Maximum length of String fields.
unique	Whether the column should have a UNIQUE constraint.
precision	Total number of digits for numeric fields.
scale	Number of digits after the decimal point for numeric fields.
columnDefinition	Custom SQL column type (overrides default mapping).

Implicit vs Explicit Mapping

Implicit mapping means you do not specify mapping details, and Hibernate uses default conventions.

For example:

```java
@Entity
public class Student {

    @Id
    private Long id;
    private String name;
    private int age;
    private String email;

}
```

What Hibernate assumes (by default):

- Table name → **Student**
- Column name → **id, name, age, email**
- Column type: inferred from Java type
  - String → **VARCHAR(255)**
  - int → **INTEGER**
  - Long → **BIGINT**

Implicit

Pros of implicit mapping

- Less code
- Faster to write
- Good for simple projects or prototype

Cons of implicit mapping

- Less Control
- Defaults vary between providers (Hibernate, EclipseLink)
- Unpredictable in naming strategy, unless naming strategy is configured in persistence.xml

Explicit

Explicit mapping means you specify how entity classes and fields map to database tables and columns using annotations.

For example:

```java
@Entity
@Table(name = "students")
public class Student {

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

	@Column(name = "name", nullable = false, length = 50, columnDefinition = "VARCHAR(50)")
	private String name;

	@Column(name = "age", columnDefinition = "INT")
	private int age;

	@Column(name = "email", unique = true, length = 100, columnDefinition = "VARCHAR(100)")
	private String email;

	//getters and setters
}
```

Explicit

Pros of explicit mapping

- Predictable behavior
- Database-friendly naming
- Safer for production

Cons of implicit mapping

- More annotations
- More boilerplate/code

@GeneratedValue

@GeneratedValue is a annotation used on primary key fields of an entity **to indicate that the value of the primary key should be generated automatically by the persistence provider (like Hibernate) or the database.**

Without it, you would need to manually assign a unique ID to each entity.

For Example:

```
@Id
@Column(name = "id")
@GeneratedValue
private Long id;
```

Here, you don’t have to set id manually when creating a new Product. When the entity is persisted, JPA automatically generates it.

@GeneratedValue GenerationType

GenerationType / Method	How it works	Database Example / Notes	When to use
AUTO	JPA chooses the strategy based on database dialect	MySQL → IDENTITY, PostgreSQL → SEQUENCE	Database portability if you don't care about exact strategy
IDENTITY	Database auto-increments the value	MySQL AUTO_INCREMENT, SQL Server IDENTITY, POSTGRES SERIAL	Simple numeric PKs; may affect batch inserts
SEQUENCE	Uses a database sequence	Oracle, PostgreSQL	Control over sequences; supports prefetching for performance
TABLE	Uses a special table to generate unique IDs	Any DB, slower than SEQUENCE	For DBs that don't support sequences or identity columns
UUID	Generates a 128-bit globally unique identifier	Can use Hibernate UUIDGenerator or Java UUID.randomUUID()	When you need unique IDs across systems, or you want non-guessable IDs

Explicitly Set GenerationType

- For example:

```
@Id
@Column(name = "id")
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
```

or

```
@Id
@Column(name = "id")
@GeneratedValue(strategy = GenerationType.UUID)
private Long id;
```

Add entity in Persistence.xml

```
<persistence-unit name="default" transaction-type="RESOURCE_LOCAL">

    <class>com.bpi.module6.model.Student</class>

</persistence-unit>
```

M6_Activity2 - JPA Core Annotations

1. Create a Student entity class, mapping the students table you created in Module 5 Exercise #3

    a. Explicitly map the database table name using @Table and also explicitly map the column details using @Column.
    (ex. @Column(**columnDefinition="VARCHAR(100)"**, **unique="true"**))

2. Use @GeneratedValue to automatically generate primary key values.

    a. Explicitly set the generation type (ex. strategy = GenerationType.IDENTITY)

3. Create a transaction using EntityManager and persist a student entity.

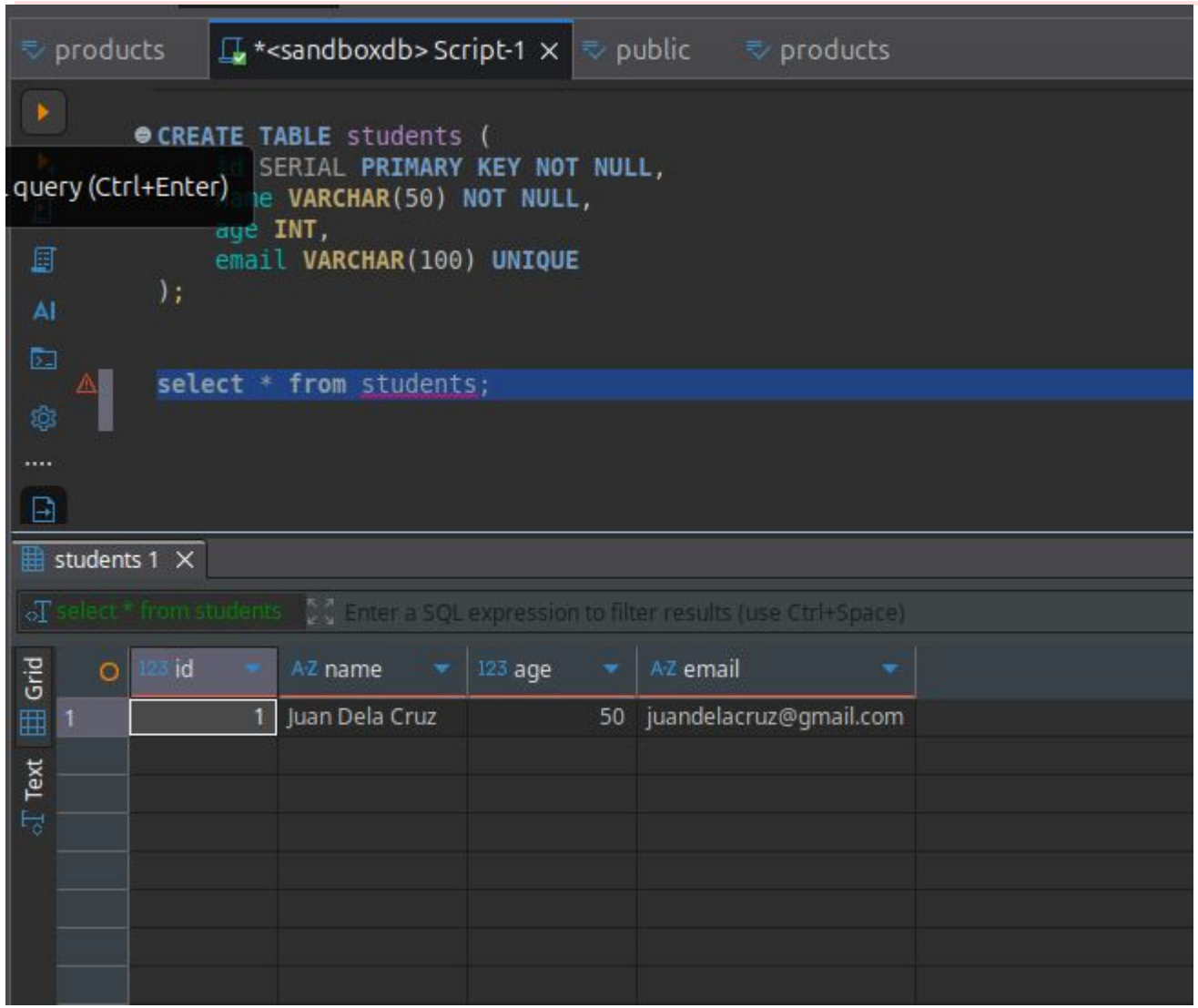
Create a transaction and persist a student entity

```
public class App {

	public static void main(String[] args) {
		EntityManager em = EntityManagerUtil.getInstance().createEntityManager();

		try {
			runM6Activity2(em);
		} finally {
			EntityManagerUtil.getInstance().closeEntityManager(em);
			EntityManagerUtil.getInstance().shutdownFactory();
		}
	}

	static void runM6Activity2(EntityManager em) {

		try {
			em.getTransaction().begin();

			Student newStudent = new Student();
			newStudent.setName("Juan Dela Cruz");
			newStudent.setAge(50);
			newStudent.setEmail("juandelacruz@gmail.com");

			em.persist(newStudent);
			em.getTransaction().commit();
		}

	}
}
```

M6_Activity2 – Expected Output (Console)

```
INFO: HHH000001: Hibernate ORM core version 7.2.0.Final
Jan 23, 2026 5:01:30 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProvider configure
WARN: HHH10001002: Using built-in connection pool (not intended for production use)
Jan 23, 2026 5:01:30 PM org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator logConnectionInfo
INFO: HHH10001005: Database info:
        Database JDBC URL [jdbc:postgresql://localhost:5432/sandboxdb]
        Database driver: PostgreSQL JDBC Driver
        Database dialect: PostgreSQLDialect
        Database version: 18.0
        Default catalog/schema: sandboxdb/public
        Autocommit mode: false
        Isolation level: READ_COMMITTED
        JDBC fetch size: none
        Pool: DriverManagerConnectionProvider
        Minimum pool size: 1
        Maximum pool size: 20
Jan 23, 2026 5:01:31 PM org.hibernate.engine.transaction.jta.platform.internal.JtaPlatformInitiator initiateService
INFO: HHH000489: No JTA platform available (set 'hibernate.transaction.jta.platform' to enable JTA platform integration)
Hibernate:
    insert
    into
        students
        (age, email, name)
    values
        (?, ?, ?)
```

M6_Activity2 - Expected Output (Database)




```
sandboxdb=# SELECT * FROM students;
 id |      name      | age |         email
----+----------------+-----+-----------------------
  1 | Juan Dela Cruz |  50 | juandelacruz@gmail.com
(1 row)
```

Relationship Annotations

Relationship Annotations

JPA lets you describe how entity classes relate to each other — similar to how database tables have relationships.

The main relationship types are:

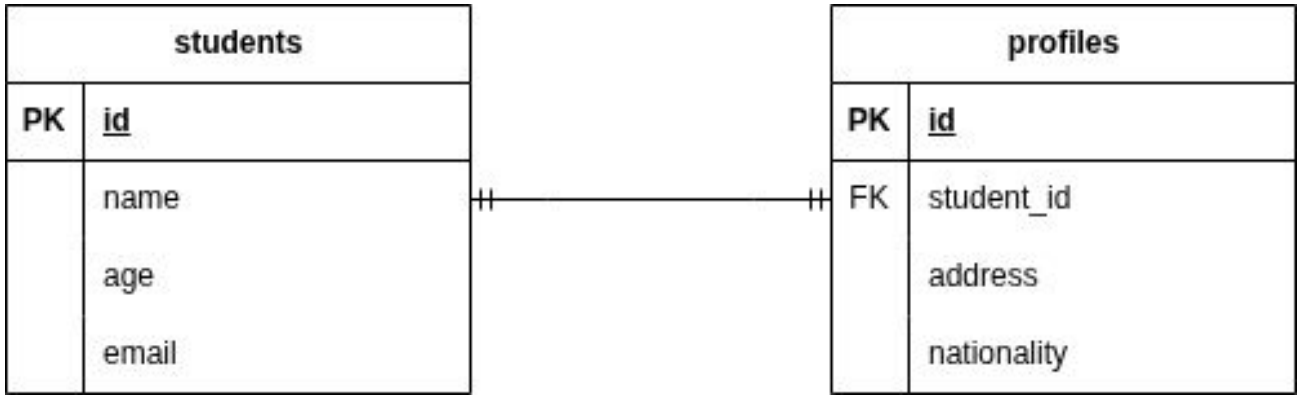
- @OneToOne
- @OneToMany
- @ManyToOne
- @ManyToMany

@OneToOne

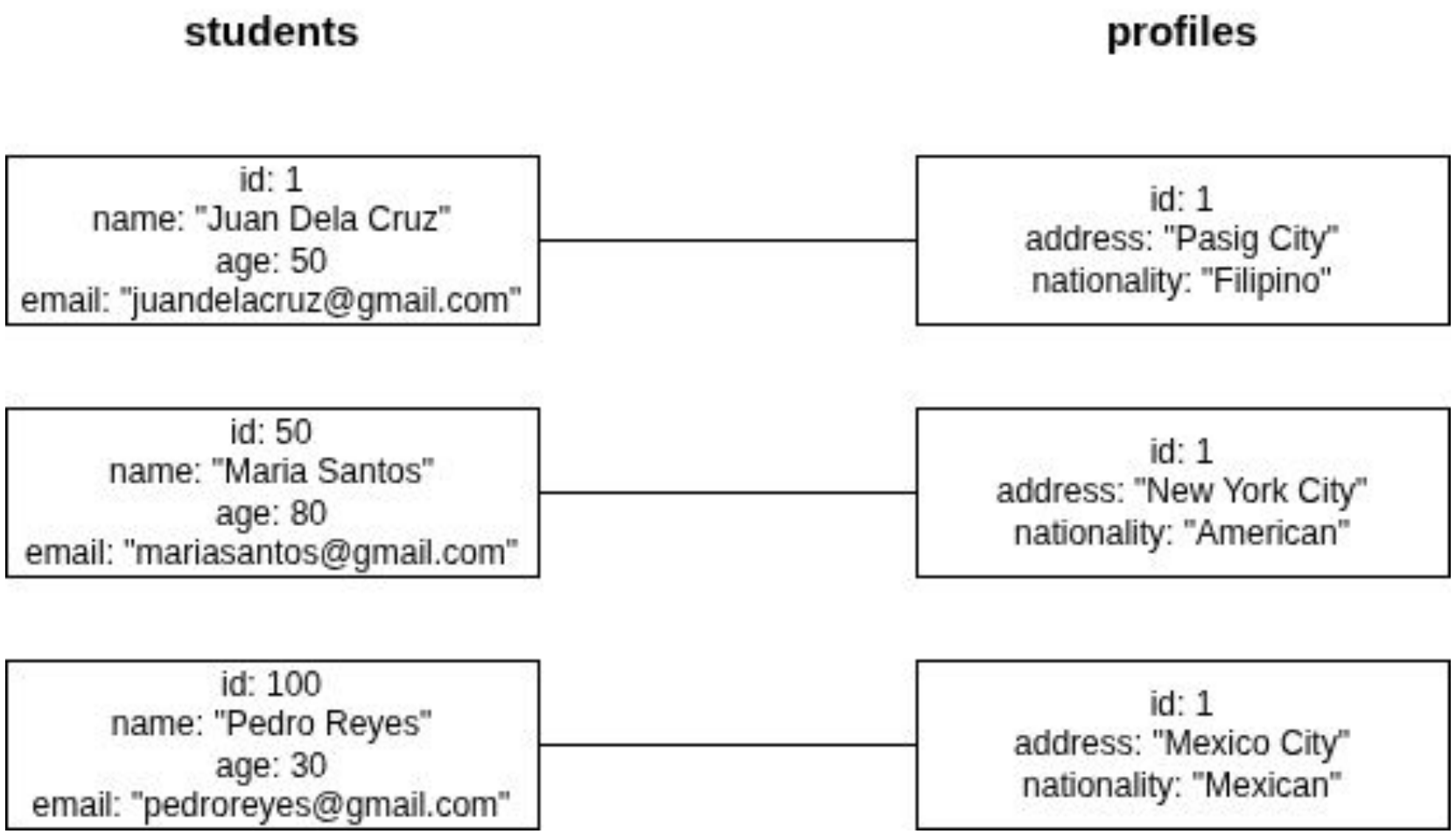
An entity instance may be associated with at most one instance of another entity.

For example:

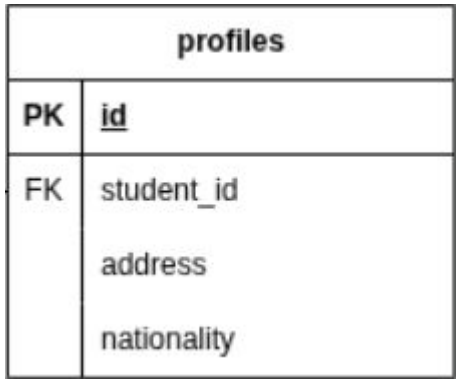
Each Student entity instance is associated with at most one Profile entity instance.



@OneToOne


students
profiles
id: 1
name: "Juan Dela Cruz"
age: 50
email: "juandelacruz@gmail.com"
id: 1
address: "Pasig City"
nationality: "Filipino"
id: 50
name: "Maria Santos"
age: 80
email: "mariasantos@gmail.com"
id: 1
address: "New York City"
nationality: "American"
id: 100
name: "Pedro Reyes"
age: 30
email: "pedroreyes@gmail.com"
id: 1
address: "Mexico City"
nationality: "Mexican"

@OneToOne Implementation




```java
@Entity
@Table(name = "profiles")
public class Profile {

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

    @Column(name = "address", columnDefinition = "VARCHAR(500)", length = 500)
    private String address;

    @Column(name = "nationality", columnDefinition = "VARCHAR(50)", length = 50)
    private String nationality;

    @OneToOne
    @JoinColumn(name = "student_id")
    private Student student;

    //setters and getters

}
```

@JoinColumn

```
@OneToOne
@JoinColumn(name = "student_id")
private Student student;
```

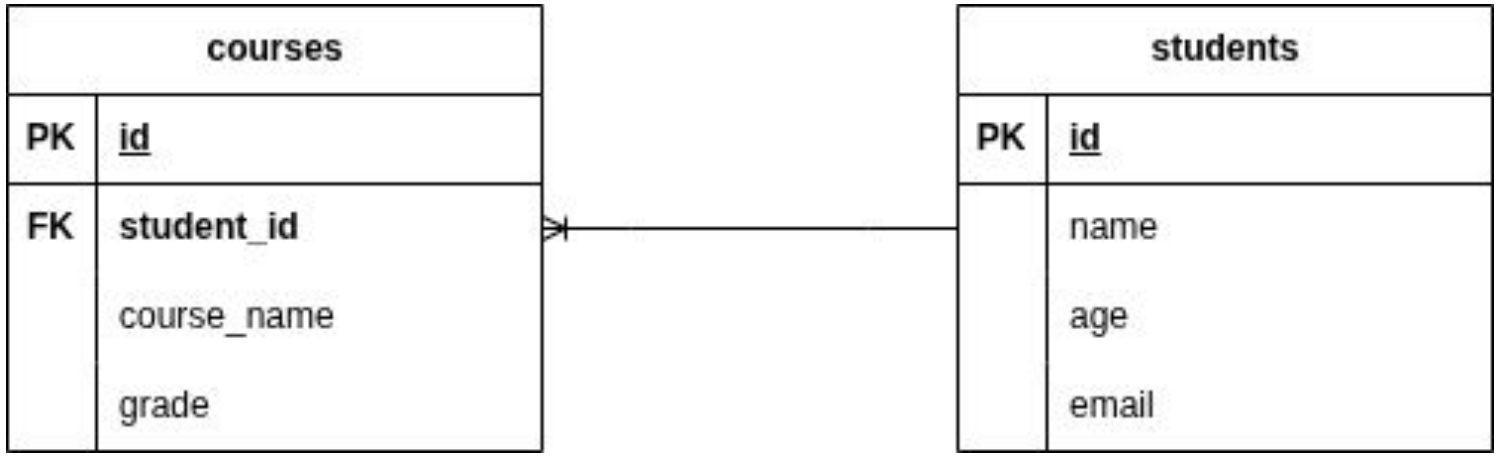
@JoinColumn is an annotation that indicates that a given column in the owner entity refers to a primary key in the reference entity.

@ManyToOne

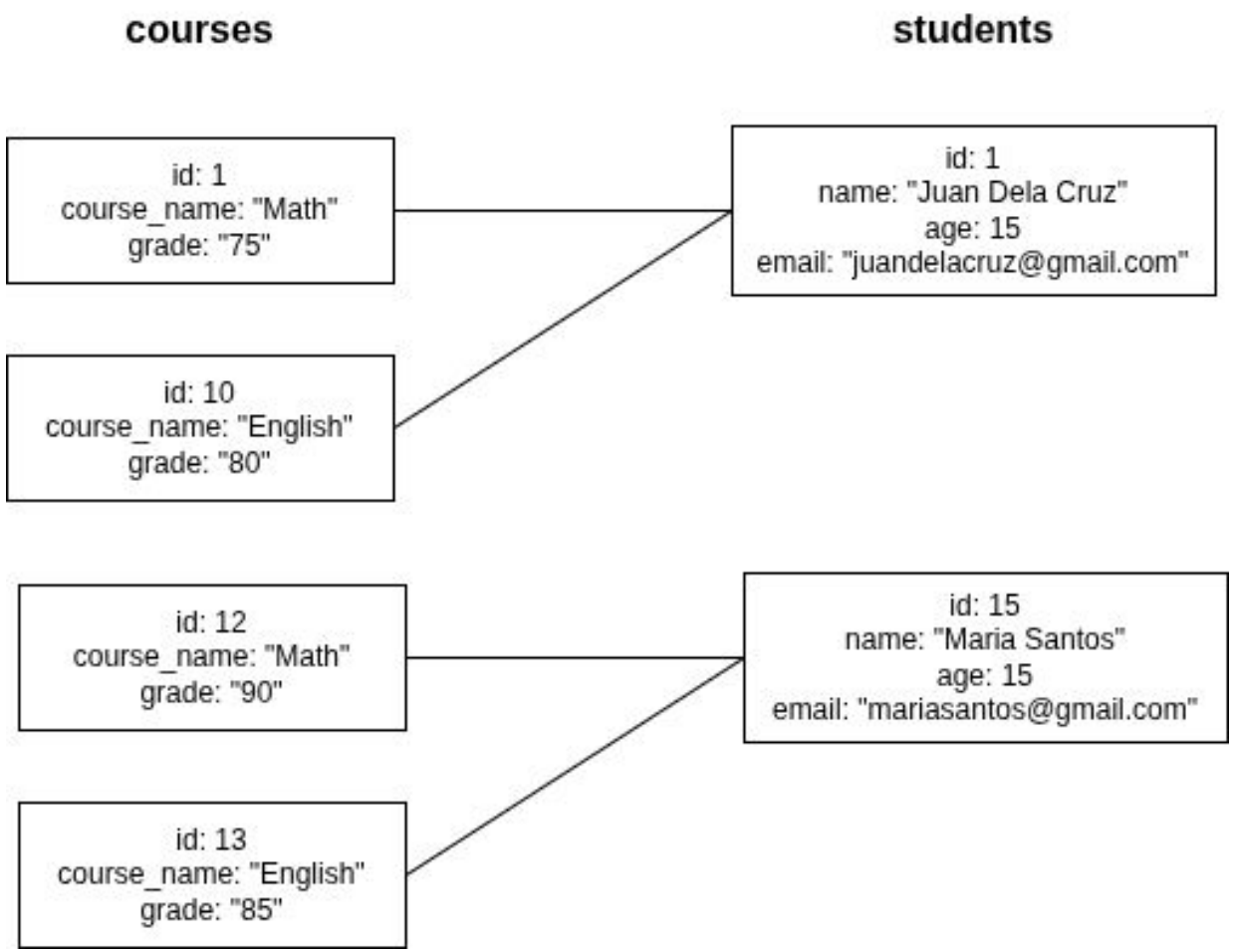
Many instances of one entity can be associated with a single instance of another entity.

For example:

Many instances of courses belongs to one instance of students.



@ManyToOne


courses
students
id: 1
course_name: "Math"
grade: "75"
id: 10
course_name: "English"
grade: "80"
id: 12
course_name: "Math"
grade: "90"
id: 13
course_name: "English"
grade: "85"
id: 1
name: "Juan Dela Cruz"
age: 15
email: "juandelacruz@gmail.com"
id: 15
name: "Maria Santos"
age: 15
email: "mariasantos@gmail.com"

@ManyToOne

courses	
PK	id
FK	student_id
	course_name
	grade

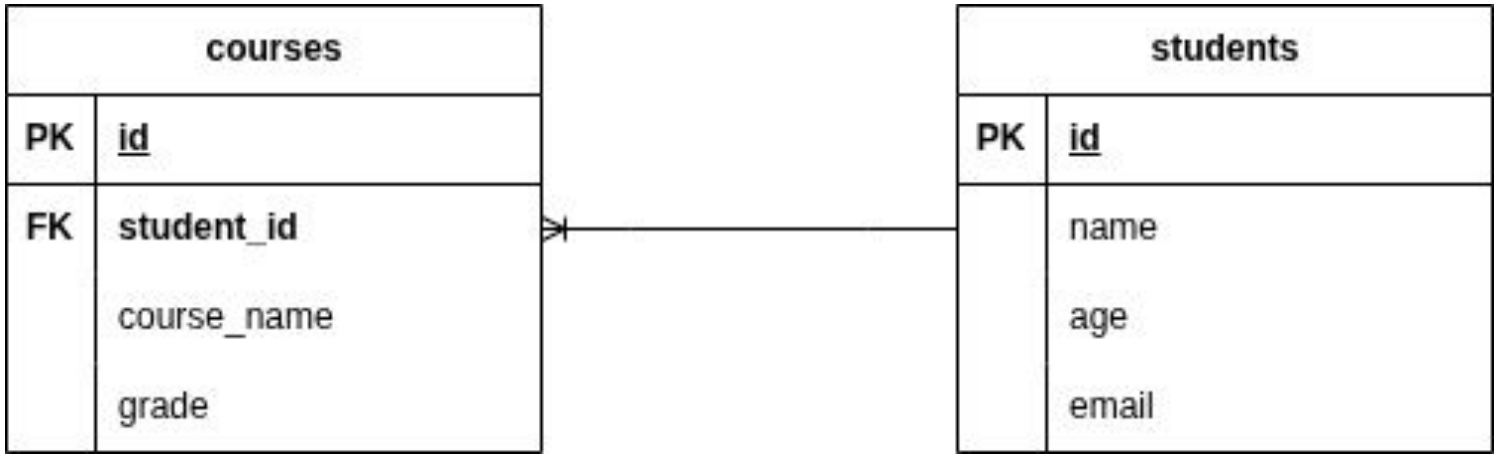
```java
@Entity
@Table(name = "courses")
public class Course {

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

    @Column(name = "course_name", columnDefinition = "VARCHAR(50)", length = 50)
    private String courseName;

    @Column(name = "grade", columnDefinition = "VARCHAR(2)", length = 2)
    private String grade;

    @ManyToOne
    @JoinColumn(name = "student_id")
    private Student student;

    //getters and setters
}
```

@OneToMany

One instance of an entity is associated to many instances of another entity.

For example:

One Student instance can be associated to many Course instances.



@OneToMany

So why use @OneToMany if there is no foreign key in the students table?

One of the use of using @OneToMany would be to point out that the foreign key is in the Course entity.

Another is to tell JPA that this side of the relationship is the **inverse** side by defining the **mappedBy** attribute in @OneToMany annotation, so JPA doesn't create another foreign key.

The **owning** side would be the Course which has the foreign key.

@OneToMany

```java
@Entity
@Table(name = "students")
public class Student {

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

    @Column(name = "name", nullable = false, length = 50, columnDefinition = "VARCHAR(50)")
    private String name;

    @Column(name = "age", columnDefinition = "INT")
    private int age;

    @Column(name = "email", unique = true, length = 100, columnDefinition = "VARCHAR(100)")
    private String email;

    @OneToMany(mappedBy = "student")
    private List<Course> courses;

    //getters and setters
}
```

@OneToMany Bidirectional

Another use is to create a **Bidirectional mapping** so both entity classes are aware of each other and can navigate the relationship from either side. For our example, you can access all the courses in the student, and vice versa.

```
    static void runBidirectional(EntityManager em) {

        em.getTransaction().begin();

        Student student = em.find(Student.class, 1L);

        student.getCourses().forEach(course -> System.out.print(course.getCourseName()));

        em.getTransaction().commit();
    }
```

Owning Side and Inverse Side

Owning side:

- Controls the foreign key
- Responsible for database updates
- Uses @JoinColumn

Inverse side:

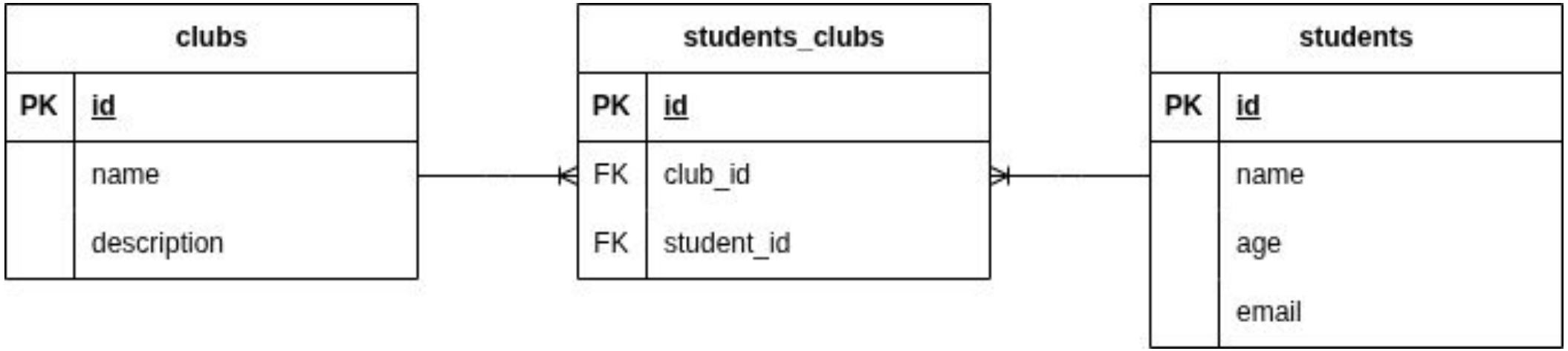
- Mirrors the relationship
- Uses mappedBy attribute
  - @OneToMany(mappedBy = “student”)

@ManyToMany

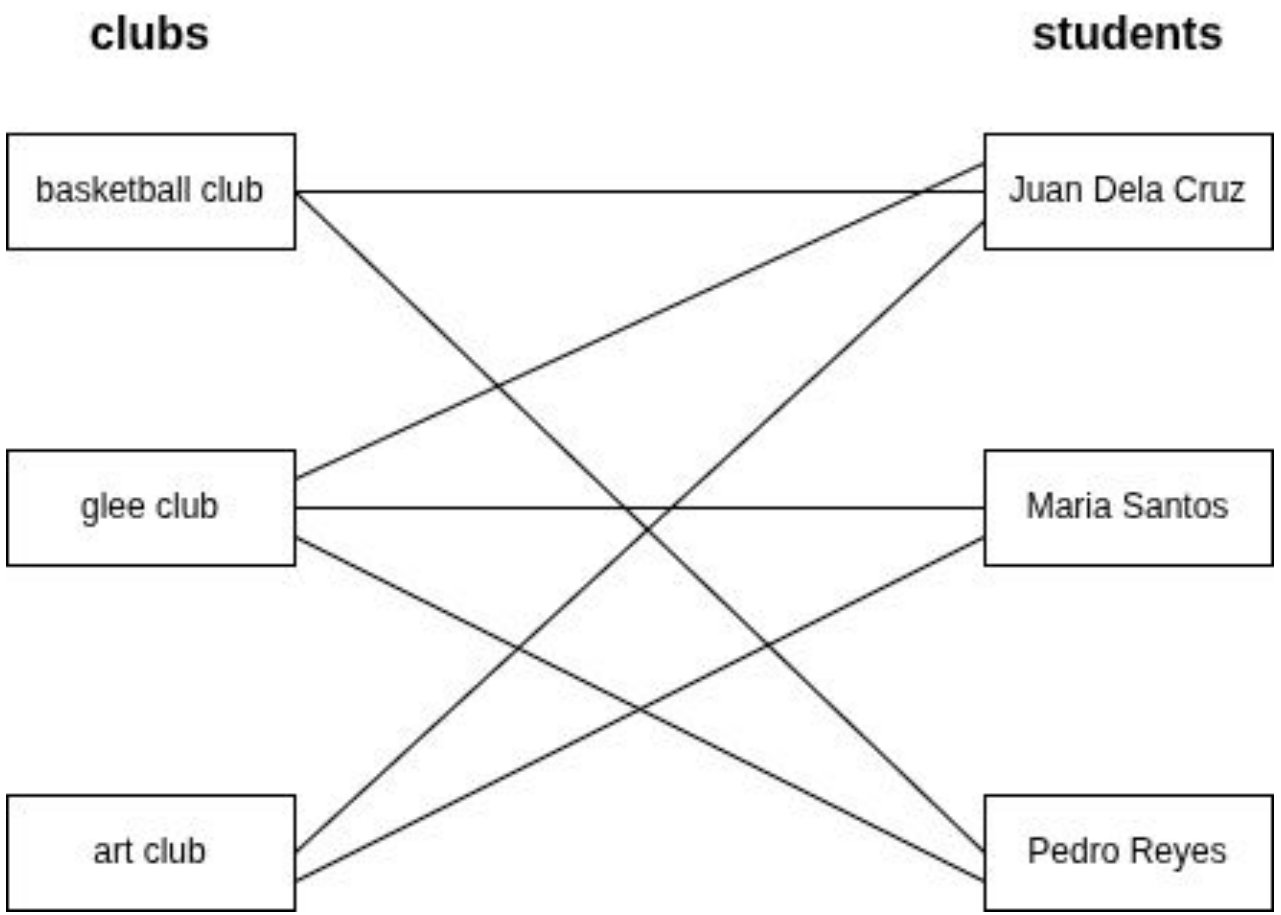
Multiple instances of one entity are associated with multiple instances of another entity.

For example, Many instances of Student can be associated to many instances of Club

@ManyToMany


clubs
PK
id
name
description
students_clubs
PK
id
FK
club_id
FK
student_id
students
PK
id
name
age
email

@ManyToMany


clubs
students
basketball club
glee club
art club
Juan Dela Cruz
Maria Santos
Pedro Reyes

@ManyToMany – Owning Side

clubs	
PK	id
	name
	description

```
@Entity
@Table(name = "clubs")
public class Club {

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

     @Column(name = "name"
          , unique = true
          , nullable = false
          , columnDefinition = "VARCHAR(50)"
          , length = 50)
     private String name;

     @Column(name = "description", columnDefinition = "VARCHAR(500)", length = 500)
     private String description;

     @ManyToMany
     @JoinTable(name = "students_clubs"
          , joinColumns = @JoinColumn(name = "club_id")
          , inverseJoinColumns = @JoinColumn(name = "student_id"))
     private List<Student> students;

     //getters and setters
}
```

@ManyToMany – Inverse side

students	
PK	id
	name
	age
	email

```java
@Entity
@Table(name = "students")
public class Student {

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

    @Column(name = "name", nullable = false, length = 50, columnDefinition = "VARCHAR(50)")
    private String name;

    @Column(name = "age", columnDefinition = "INT")
    private int age;

    @Column(name = "email", unique = true, length = 100, columnDefinition = "VARCHAR(100)")
    private String email;

    @OneToMany(mappedBy = "student")
    private List<Course> courses;

    @ManyToMany(mappedBy = "students")
    private List<Club> clubs;

    //getters and setters
}
```

@ManyToMany

```
static void persistManyToMany(EntityManager em) {

    em.getTransaction().begin();

    Student student = em.find(Student.class, 1L);

    Student student2 = em.find(Student.class, 2L);

    Club basketballClub = new Club();
    basketballClub.setName("Basketball Club");
    basketballClub.setDescription("For basketball enthusiast");
    List<Student> basketballClubstudentList = new ArrayList();
    basketballClubstudentList.add(student);
    basketballClub.setStudents(basketballClubstudentList);
    em.persist(basketballClub);

    Club gleeClub = new Club();
    gleeClub.setName("Glee Club");
    gleeClub.setDescription("For students that wants to sing");
    List<Student> gleeClubstudentList = new ArrayList();
    gleeClubstudentList.add(student);
    gleeClubstudentList.add(student2);
    gleeClub.setStudents(gleeClubstudentList);
    em.persist(gleeClub);

    Club artClub = new Club();
```

@ManyToMany – output

```
sandboxdb=# select * from students_clubs;
 club_id | student_id
---------+------------
       1 |          1
       2 |          1
       2 |          2
       3 |          1
       3 |          2
(5 rows)
```

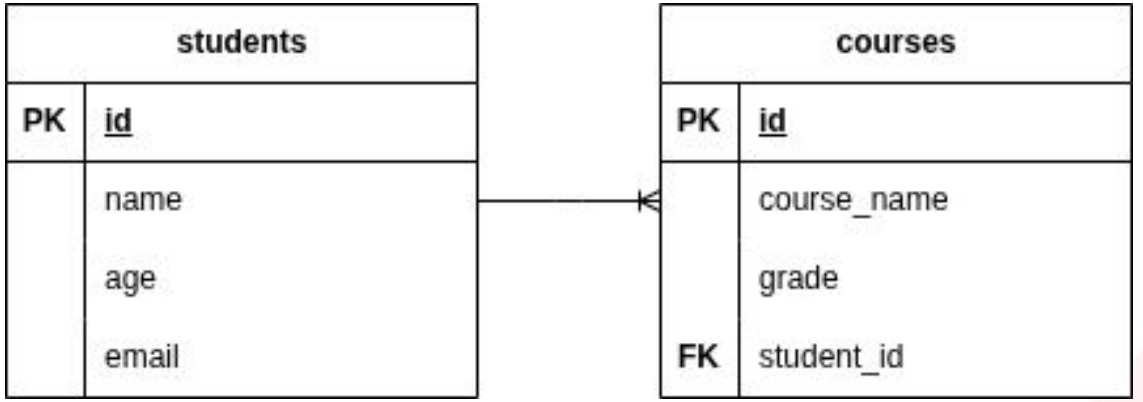
M6_Activity3 - Relationship Annotations

Objective:

Extend the existing JPA application by modeling a one-to-many relationship using @OneToMany and @ManyToOne. This activity emphasizes relationship mapping, foreign key mapping, and bidirectional mapping.

Scenario:

A Student can have many courses. This represents a one-to-many / many-to-one relationship.



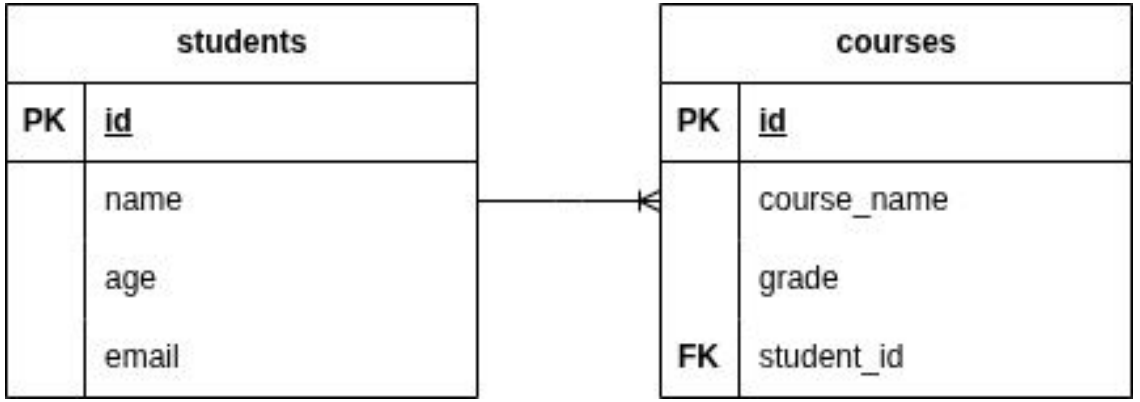
M6_Activity3 - Relationship Annotations

Instructions:

1. Create a new entity class for courses table
    - Add student field and add @ManyToOne
    - add @JoinColumn to map the foreign key column name

2. Update student entity class
    - Add a courses field and use @OneToMany
    - since this is the inverse side add mappedBy attribute

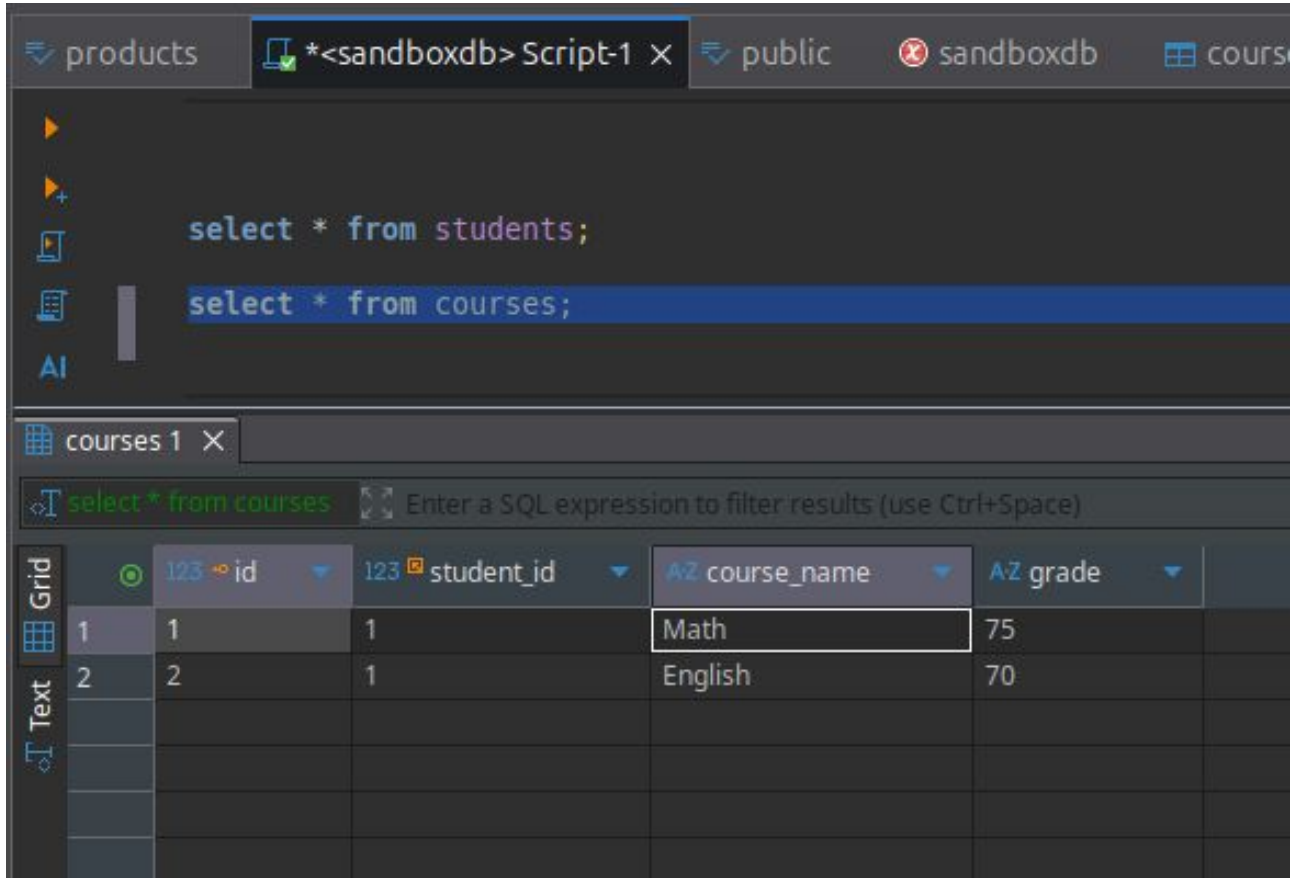
3. Create a transaction
    - Begin a transaction
    - Assign courses to a student



M6_Activity3 - Transaction

```
static void persistOneToMany(EntityManager em) {
    em.getTransaction().begin();

    Student student1 = em.find(Student.class, 1L);

    Course newCourse = new Course();
    newCourse.setCourseName("Math");
    newCourse.setGrade("80");
    newCourse.setStudent(student1);

    em.persist(newCourse);

    em.getTransaction().commit();
}
```

M6_Activity3 – Expected Output (Database)



M6_Activity3 – Expected Output (Console)

```
Hibernate:
    select
        s1_0.id,
        s1_0.age,
        s1_0.email,
        s1_0.name
    from
        students s1_0
    where
        s1_0.id=?
Hibernate:
    insert
    into
        courses
        (course_name, grade, student_id)
    values
        (?, ?, ?)
Hibernate:
    insert
    into
        courses
        (course_name, grade, student_id)
    values
        (?, ?, ?)
```

Entity Manager

Entity Manager

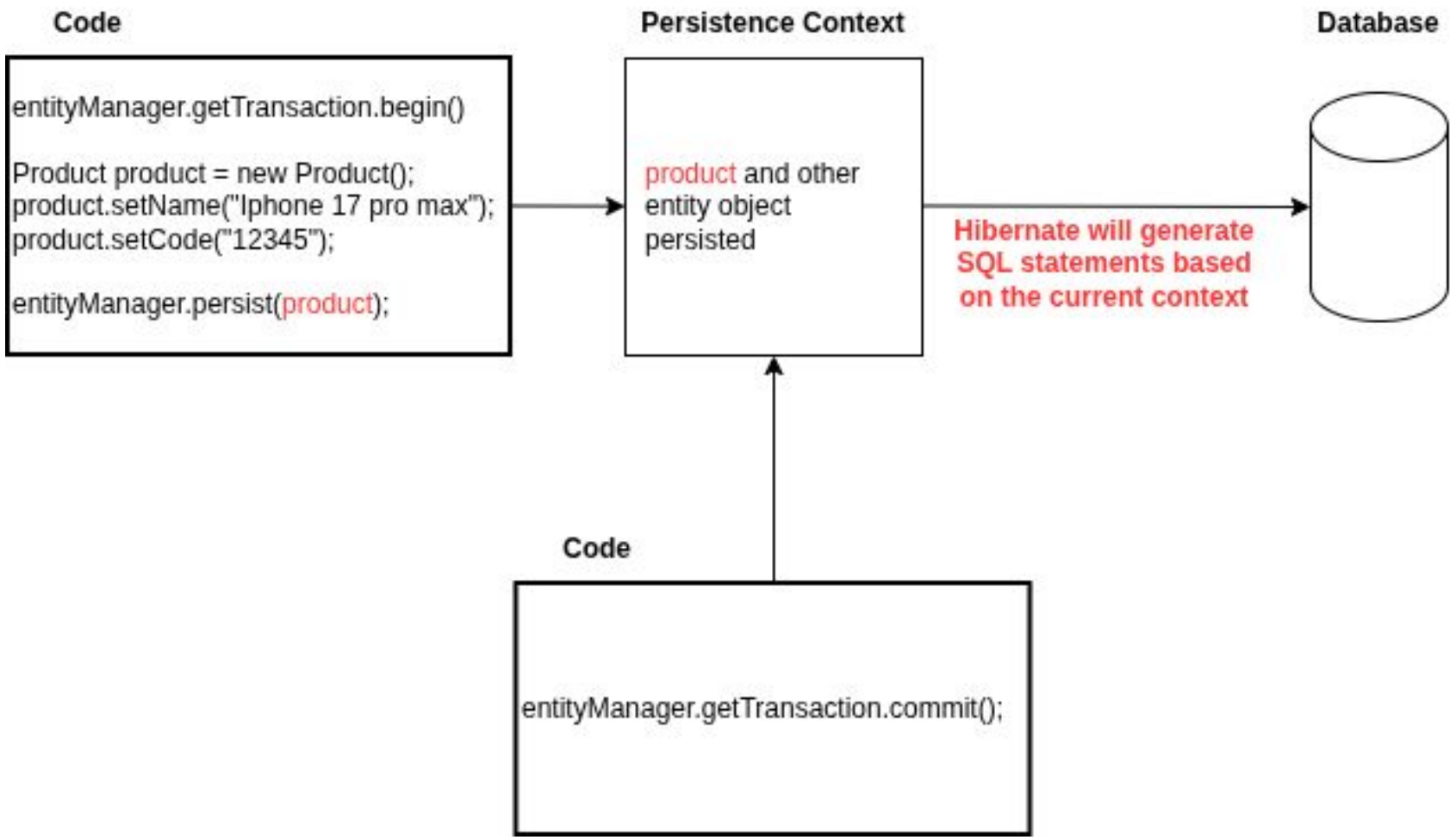
EntityManager manages entity objects and controls how they are stored, retrieved, updated, and deleted in the database.

How does the Entity Manager work?

When you call persist(), find(), or merge(), EntityManager keeps track of your objects in something called the **persistence context.**

In the persistence context, Hibernate automatically detects entity changes and synchronizes them with the database on commit.

Persistence Context



Entity Lifecycle

1. **Transient –** normal Java object
2. **Managed –** attached to Persistence Context
3. **Detached –** was managed, but no longer tracked
4. **Removed –** scheduled for deletion

Entity Manager Operations

Persist

Persist - Adds a new entity to the persistence context and marks it for insertion into the database on flush/commit.

For Example: **void persist(Object entity);**

```
static void persistSample(EntityManager em) {
    em.getTransaction().begin();

    Student newStudent = new Student(); // transient

    newStudent.setName("Pedro Reyes");
    newStudent.setAge(23);
    newStudent.setEmail("pedroreyes@gmail.com");

    em.persist(newStudent); // managed

    em.getTransaction().commit(); //hibernate generates insert sql, newStudent will be saved to db

    System.out.println("is newStudent inside the persistence context: " + em.contains(newStudent));
}
```

Find

Find - Fetch an entity from the database using its primary key and put it inside the persistence context.

- For Example: **<T> T find(Class<T> entityClass, Object primaryKey);**

```
static void findSample(EntityManager em) {

     em.getTransaction().begin();

     //Hibernate retrieves the Student entity via SELECT and places it into the persistence context.
     Student studentWithId1 = em.find(Student.class, 1L); //managed entity

     if (studentWithId1 != null) {
          System.out.println("is newStudent inside the persistence context: " + em.contains(studentWithId1));
     }

     em.getTransaction().commit();
}
```

Returns null if the entity doesn't exist.

Merge

Merge - Copy the state of a detached entity into the current persistence context and return a managed instance.

A common use case for merge() is when you want to update a detached entity—that is, an entity that was previously persisted but is no longer associated with the current persistence context.

Merge

```
static void mergeSample(EntityManager em) {

    em.getTransaction().begin();

    //Hibernate retrieves the Student entity via SELECT and places it into the persistence context.
    Student studentWithId1 = em.find(Student.class, 1L); //managed entity

    em.detach(studentWithId1); //detach studentWithId1

    System.out.println("is studentWithId1 inside the persistence context: " + em.contains(studentWithId1));// false

    // for example, studentWithId1 has changes while in detached
    studentWithId1.setAge(22);

    // in order for this changes to reflect in database
    // we need to re attach the object to the persistence context using merge
    // merge() returns a managed instance with the same persistent state
    // as the given entity instance, but a distinct Java object identity.
    studentWithId1 = em.merge(studentWithId1);

    em.getTransaction().commit();
}
```

Remove

Remove - Marks an entity for deletion in the database.

- For Example: **void remove(Object entity);**

```
static void removeSample(EntityManager em) {
      em.getTransaction().begin();
      Student student = em.find(Student.class, 8L); // managed
      em.remove(student);
      em.getTransaction().commit();
      System.out.println("is student inside the persistence context: " + em.contains(student)); // false
      System.out.println(student.toString()); //detached
}
```

Only works on managed entities. For detached entities, use merge first.

Refresh

Refresh - Synchronizes the state of the entity with the current database state.

void refresh(Object entity);

- Discards all in-memory changes of the entity
- Reloads the entity's state from the database
- Keeps the entity managed

Useful if another transaction updated the entity or if some field values are generated by the database.

Refresh

Use Case:

- The created_at value is set by the database, and we want to retrieve it after the entity is saved.

```
CREATE TABLE public.students (
    id SERIAL PRIMARY KEY NOT NULL,
    name VARCHAR(50) NOT NULL,
    age INT,
    email VARCHAR(100) UNIQUE
    created_at TIMESTAMP NOT NULL DEFAULT now();
);
```

Refresh

```
static void refreshSample(EntityManager em) {

        em.getTransaction().begin();

        Student newStudent = new Student();

        newStudent.setName("John Mark Santos");
        newStudent.setAge(21);
        newStudent.setEmail("johnmarksantos@gmail.com");

        em.persist(newStudent); // managed

        em.getTransaction().commit(); // commit insert to database

        System.out.println("BEFORE REFRESH: " + newStudent.toString()); //createdAt=null

        em.refresh(newStudent); //generates select statement

        System.out.println("AFTER REFRESH: " + newStudent.toString()); //createdAt=2026-01-26T15:24:36.123455Z

    }
```

Detach

Detach - Detaches an entity from the persistence context

After detaching:

- The entity is no longer managed
- Changes to the entity won't be persisted on flush/commit
- Useful when you want to stop JPA from tracking an entity

void detach(Object entity);

Detach

```
    static void detachSample(EntityManager em) {

        em.getTransaction().begin();

        Student student = em.find(Student.class, 1L); //managed

        em.detach(student); // detached

        student.setAge(100);

        em.getTransaction().commit(); // no update statement generated because student is detached

    }
```

Clear

Clear - Detaches all managed entities from the persistence context.

```
void clear();
```

Clear

```
static void clearSample(EntityManager em) {

    em.getTransaction().begin();

    Student student = em.find(Student.class, 1L); //managed
    Student student1 = em.find(Student.class, 9L); // managed

    System.out.println("BEFORE clear()");
    System.out.println("is student inside the persistence context: " + em.contains(student)); //true
    System.out.println("is student1 inside the persistence context: " + em.contains(student)); //true

    em.clear(); // detaches all managed entity

    System.out.println("AFTER clear()");
    System.out.println("is student inside the persistence context: " + em.contains(student)); //false
    System.out.println("is student1 inside the persistence context: " + em.contains(student)); //false
    em.getTransaction().commit();

}
```

Contains

Contains - Determine if the given object is a managed entity instance belonging to the current persistence context.

boolean contains(Object entity);

- For Example:

```
System.out.println("is newStudent student the persistence context: " + em.contains(student));
```

Commit vs Flush

Flush()

- Forces Hibernate to execute SQL statements immediately for all pending changes in the persistence context.
- Entities remain managed in memory.
- Transaction is still open, so you can rollback if needed.

Commit()

- Automatically calls flush() to synchronize the persistence context.
- Commits the transaction, making changes permanent in the database.
- After commit, you cannot rollback.

Rollback

- rollback() undoes all changes in the current transaction.
- After rollback, all entities in that transaction are detached.
- You must check tx.isActive() before rollback to avoid exceptions.
- Persistence context is cleared for rolled-back entities.

```java
static void rollbackSample(EntityManager em) {

        EntityTransaction tx = em.getTransaction();

        try {
            tx.begin();

            // first transaction
            Student student = em.find(Student.class, 1L);
            student.setAge(30);

            em.flush();

            //second transaction - mock error by saving data that already exist
            Student newStudent = new Student();
            newStudent.setName("Juan Dela Cruz");
            newStudent.setAge(20);
            newStudent.setEmail("juandelacruz@gmail.com");

            em.persist(newStudent);

            tx.commit();
        } catch (Exception e) {
            System.out.println("Exception occurred: " + e.getMessage());

            if (tx.isActive()) { // always check if transaction is active before calling rollback;
                tx.rollback();
                System.out.println("Transaction rolled back.");
            }
        }
```

Rollback – output

```
Exception occurred: could not execute statement [ERROR: duplicate key value violates unique constraint
"students_email_key"
 Detail: Key (email)=(juandelacruz@gmail.com) already exists.] [insert into students (age,email,name) values (?,?,?)]
Transaction rolled back.
```

M6_Activity4 - Entity Manager

Objective:

At the end of this activity, you will be able to use JPA EntityManager operations to perform CRUD actions and manage entity lifecycle states within a persistence context.

```java
static void m6Activity4Solution(EntityManager em) {

    em.getTransaction().begin();

    // 1. create Student object, assign values
    // 2. attach transient student object to persistence context
    // 3. call flush()
    // 4. detach the managed newStudent from the persistence context

    // 5. print "is newStudent inside the persistence context: " + call contains()
    System.out.println("is newStudent inside the persistence context: "  + em.contains(newStudent));

    // 6. reattach the detached newStudent
    // 7. update newStudent, change some values like age or email
    // 8. call flush()
    // 9. print "is newStudent inside the persistence context: " + call contains()
    // 11. mark managed newStudent for deletion
    // 12. call flush()
    // 13. print "is newStudent inside the persistence context: " + call contains()

    em.getTransaction().commit();
}
```

M6_Activity4 – Output (Console)

```
Hibernate:
    insert
    into
        students
        (age, email, name)
    values
        (?, ?, ?)
is newStudent inside the persistence context: false
Hibernate:
    select
        s1_0.id,
        s1_0.age,
        s1_0.created_at,
        s1_0.email,
        s1_0.name
    from
        students s1_0
    where
        s1_0.id=?
```

```
Hibernate:
    update
        students
    set
        age=?,
        email=?,
        name=?
    where
        id=?
is newStudent inside the persistence context: true
Hibernate:
    delete
    from
        students
    where
        id=?
is newStudent inside the persistence context: false
```

Fetching and Cascading Strategies

Objectives

1. Understand what fetching and cascading strategies are in JPA
2. Learn the different Fetching and Cascade Types

Fetching Strategies

Fetching Strategy decides when JPA/Hibernate loads related entities from the database.

There are two strategies:

1. **EAGER** – Load related entity immediately with the parent entity.
2. **LAZY** – Load related entity only when accessed.

FetchType.LAZY

To set fetch type LAZY:

```
@Entity
@Table(name = "products")
public class Product {


    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "category_id")
    private Category category;

}
```

Now, when retrieving products, categories are not loaded until getCategories() is called.

FetchType.EAGER

To set fetch type EAGER:

```
@Entity
@Table(name = "products")
public class Product {


    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "category_id")
    private Category category;

}
```

Now, when retrieving products, categories are loaded eagerly.

When to Use Each FetchType

- **EAGER**: Use for small, always-needed relationships.
- **LAZY**: Use for large collections or optional relationships to improve performance.

What is Cascading in JPA?

- Cascading defines what happens to related entities when you perform operations on the parent entity.

- Operations include: **PERSIST**, **MERGE**, **REMOVE**, **REFRESH**, **DETACH**.

Cascade Types

Cascade Type	Meaning	Example
PERSIST	Save child when parent is saved	`entityManager.persist(parent)`
MERGE	Update child when parent is updated	`entityManager.merge(parent)`
REMOVE	Delete child when parent is deleted	`entityManager.remove(parent)`
REFRESH	Refresh child when parent is refreshed	`entityManager.refresh(parent)`
DETACH	Detach child when parent is detached	`entityManager.detach(parent)`

CascadeType.ALL

CascadeType.ALL = all operations (PERSIST, MERGE, REMOVE, REFRESH, DETACH) apply to orders.

```
@OneToMany(cascade = CascadeType.ALL)
private List<Order> orders;
```

Java Persistence Query Language (JPQL)

Objectives

1. Understand the Purpose of JPQL
2. Write Basic JPQL queries
3. Query Entity Relationships
4. Control and Shape Query Results

What is JPQL?

JPQL (Java Persistence Query Language) is a query language used with JPA to retrieve and manipulate data.

- Similar to SQL in syntax
- Operates on entities and their fields
- Independent of the underlying database

JPQL vs SQL

JPQL

- Queries entities
- Uses entity fields
- Database-independent

SQL

- Queries tables
- Uses column names
- Database specific

For example:

```
SELECT u FROM User u;     //JPQL

SELECT * FROM users;      //SQL
```

Basic JPQL Structure

```
SELECT e FROM EntityName e;
```

- EntityName → Java entity class name
- e → alias used in the query

```
SELECT e FROM Employee e;
```

- Retrieves all Employee entities
- Equivalent to: SELECT * FROM employee
- retrieves all columns

EntityManager createQuery();

```
TypedQuery<T> createQuery(String qlString, Class<T> resultClass);
```

```
    static void selectAllStudent(EntityManager em) {

        em.getTransaction().begin();

        String jpql = "Select s FROM Student s";
        TypedQuery<Student> query = em.createQuery(jpql, Student.class);
        List<Student> students = query.getResultList();

        //print student names
        students.forEach(student -> System.out.println(student.getName()));

        em.getTransaction().commit();

    }
```

WHERE Clause and Named Parameters

Where Clause:

```
String jpql = "Select s FROM Student s WHERE s.age >= 21";
TypedQuery<Student> query = em.createQuery(jpql, Student.class);
List<Student> students = query.getResultList();
```

Named Parameters:

```
String jpql = "Select s FROM Student s WHERE s.name = :name";
TypedQuery<Student> query = em.createQuery(jpql, Student.class);
query.setParameter("name", name);
List<Student> students = query.getResultList();
```

Positional Parameters

```
static void selectStudentByNamePositionalParamaters(EntityManager em, String name, int age) {

    em.getTransaction().begin();

    String jpql = "Select s FROM Student s WHERE s.name = ?1 AND age = ?2";
    TypedQuery<Student> query = em.createQuery(jpql, Student.class);
    query.setParameter(1, name);
    query.setParameter(2, age);
    List<Student> students = query.getResultList();

    //print student names
    students.forEach(student -> System.out.println(student.getName()));

    em.getTransaction().commit();

}
```

Positional parameters are placeholders in a JPQL query that are identified by numbers (?1, ?2, etc.), and you bind values to them by position, not by name.

Selecting Specific Fields

```
static void selectAllStudentEmails(EntityManager em) {

    em.getTransaction().begin();

    String jpql = "Select s.email FROM Student s";
    TypedQuery<String> query = em.createQuery(jpql, String.class);
    List<String> studentEmails = query.getResultList();

    //print student emails
    studentEmails.forEach(email -> System.out.println(email));

    em.getTransaction().commit();

}
```

- Queries all student emails and stores them in a list.

```
Hibernate:
    select
        s1_0.email
    from
        students s1_0
mariasantos@gmail.com
johnmarksantos@gmail.com
juandelacruz@gmail.com
janedoe@gmail.com
```

JOIN in JPQL

- JOIN is used to fetch related entities via their associations.
- Works with @OneToOne, @OneToMany, @ManyToOne, @ManyToMany.
- JPQL Join Types
  - JOIN → INNER JOIN by default
  - LEFT JOIN → LEFT OUTER JOIN
  - FETCH JOIN → loads associated entities eagerly in the same query

INNER JOIN

```
static List<Student> selectAllStudentsAndCoursesInnerJoin(EntityManager em) {

    List<Student> students = em.createQuery("SELECT s FROM Student s JOIN s.courses", Student.class)
                .getResultList();

    return students;
}
```

INNER JOIN (Multiple)

```
static List<Student> selectAllStudentsAndCoursesInnerJoinMultiple(EntityManager em) {

     List<Student> students = em
          .createQuery("SELECT s FROM Student s JOIN s.courses JOIN s.profile JOIN s.clubs", Student.class)
          .getResultList();

     return students;
}
```

INNER JOIN (Multiple)

```
Hibernate:
    select
        s1_0.id,
        s1_0.age,
        s1_0.created_at,
        s1_0.email,
        s1_0.name
    from
        students s1_0
    join
        courses c1_0
            on
s1_0.id=c1_0.student_id
    join
        profiles p1_0
            on
s1_0.id=p1_0.student_id
    join
        students_clubs c2_0
            on
s1_0.id=c2_0.student_id
```

LEFT JOIN

```
static List<Student> selectAllStudentsAndCoursesLeftJoin(EntityManager em) {

    List<Student> students = em.createQuery("SELECT s FROM Student s LEFT JOIN s.courses", Student.class)
            .getResultList();

    return students;
}
```

RIGHT JOIN

```
static List<Course> selectAllStudentsAndCoursesRightJoin(EntityManager em) {

    List<Course> courses = em.createQuery("SELECT c FROM Course c LEFT JOIN FETCH c.student", Course.class)
            .getResultList();

    return courses;
}
```

Aggregate Functions

```
SELECT COUNT(e) FROM Employee e
```

Supported functions:

- COUNT
- SUM
- AVG
- MIN
- MAX

Aggregate Functions

```
static Long countStudents(EntityManager em) {

     Long totalStudents = em.createQuery(
               "SELECT COUNT(s) FROM Student s", Long.class
          ).getSingleResult();
     return totalStudents;
}

static Long countStudentsWithCourses(EntityManager em) {

     Long studentsWithCourses = em.createQuery(
               "SELECT COUNT(DISTINCT s) FROM Student s JOIN s.courses c", Long.class
          ).getSingleResult();
     return studentsWithCourses;
}
```

M6_Activity5 - JPQL

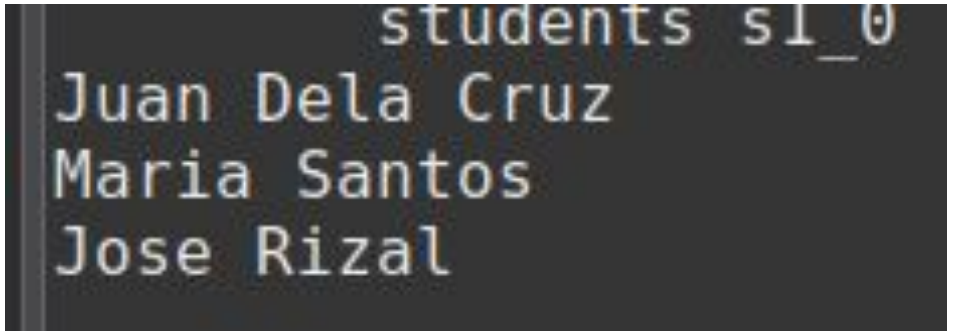
Objective:

Implement the following operations using JPQL and EntityManager:

//void printAllStudentNames(EntityManager em)
//Long countCoursesByStudentId(EntityManager em, Long id)
//Long countStudentsByAgeGreaterThan(EntityManager em, int age)

M6_Activity5 - JPQL (Output)

void printAllStudentNames()



M6_Activity5 - JPQL (Output)

void printAllStudentNames()

```
Hibernate:
    select
        count(c1_0.id)
    from
        courses c1_0
    join
        students s1_0
            on s1_0.id=c1_0.student_id
    where
        s1_0.id=?
2
```

M6_Activity5 - JPQL (Output)

Long countStudentsByAgeGreaterThan(EntityManager em, int age)

```
        (course_name, grade, student_id)
    values
        (?, ?, ?)
Hibernate:
    select
        count(s1_0.id)
    from
        students s1_0
    where
        s1_0.age>?
3
```

Criteria API

What is Criteria API?

Criteria API is a type-safe, programmatic way to build JPA queries.

- Queries are built using Java code, not strings
- Can be used for dynamic queries
- Reduces risk of syntax errors at runtime

Why Use Criteria API?

- **Type-safe**: compiler checks field names
- **Dynamic queries**: build queries programmatically
- **Safe refactoring**: IDE detects renamed fields
- **Works well with complex queries**

Basic Components

- **CriteriaBuilder → factory for building criteria queries**
- **CriteriaQuery → represents the query**
- **Root → represents the entity in the FROM clause**
- **Predicate → represents WHERE conditions**
- **TypedQuery → executes the query**

Simple SELECT Query

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

TypedQuery<Employee> query = entityManager.createQuery(cq);
List<Employee> result = query.getResultList();
```

- Selects all Employee entities
- Equivalent to **SELECT e FROM Employee e**

WHERE Clause

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

Predicate condition = cb.greaterThan(root.get("salary"), new BigDecimal("50000"));
cq.select(root).where(condition);

List<Employee> result = entityManager.createQuery(cq).getResultList();
```

- Filters employees with salary > 50,000

Multiple Conditions (AND / OR)

```
Predicate p1 = cb.greaterThan(root.get("salary"), new BigDecimal("50000"));
Predicate p2 = cb.equal(root.get("department"), "IT");


cq.select(root).where(cb.and(p1, p2));
```

- AND, OR, NOT are available through CriteriaBuilder

ORDER BY Clause

```
cq.select(root).orderBy(cb.asc(root.get("name"))); // ascending
cq.select(root).orderBy(cb.desc(root.get("salary"))); // descending
```

- Works like ORDER BY in SQL/JPQL

JOIN

```
Root<Order> orderRoot = cq.from(Order.class);
Join<Order, Customer> customerJoin = orderRoot.join("customer");
cq.select(orderRoot).where(cb.equal(customerJoin.get("name"), "Alice"));
```

- Works like ORDER BY in SQL/JPQL

Aggregate Functions

```
cq.select(cb.count(root));
```

- Supports COUNT, SUM, AVG, MAX, MIN

Repository Pattern

Objectives

- Understand the purpose and benefits of the Repository Pattern
- Learn and Implement repository pattern

What is Repository Pattern?

- Design pattern to abstract database access implementation code
  - Provides CRUD operations without exposing JPA/Hibernate details

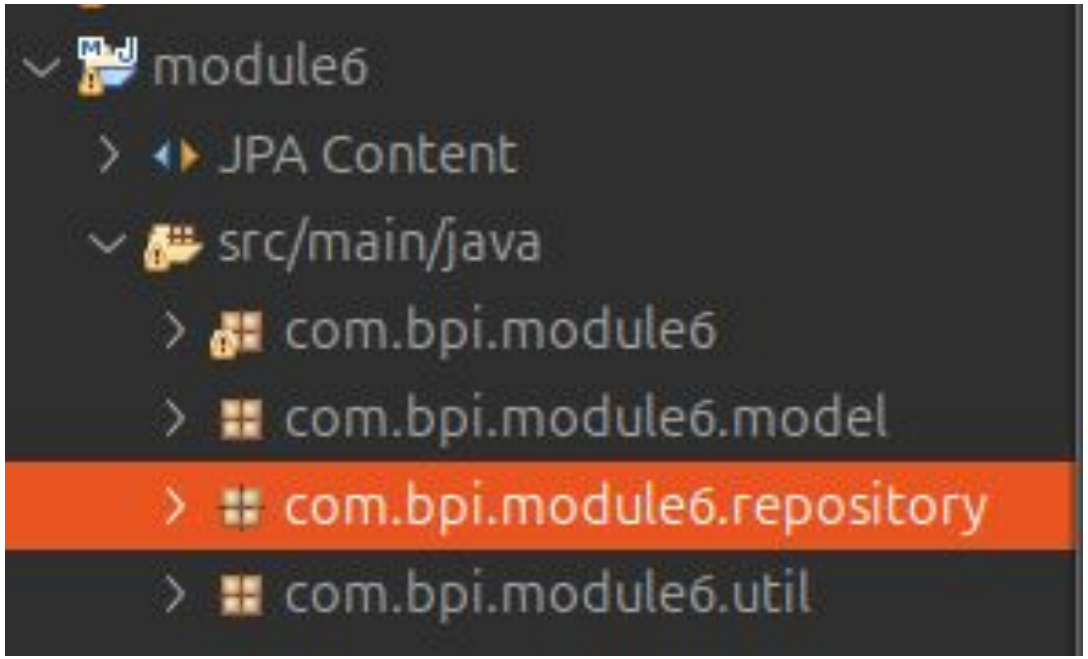
- Purpose is to separate business logic from data access logic

Why Use Repository Pattern?

- **Cleaner code**: avoids data access code in service layer
- **Reusability**: common CRUD methods are centralized
- **Testability**: easy to mock repositories in unit tests
- **Maintainability**: easier to switch database or ORM provider, easy to change since logic is in a single class.

Implementation

1. **Create repository package -** com.bpi.module6.repository



Implementation

2. Create repository interface

```java
import java.util.List;

public interface Repository<T, ID> {

    T save(T entity);

    void delete(T entity);

    void deleteById(ID id);

    T findById(ID id);

    List<T> findAll();

}
```

Implementation

3. Create Implementation class of Repository Interface for each entity

```java
public class StudentRepositoryImpl implements Repository<Student, Long> {

    private final EntityManager em;

    public StudentRepositoryImpl(EntityManager em) {
        this.em = em;
    }

    @Override
    public Student save(Student student) {
        if(student.getId() == null) {
            em.persist(student);
        } else {
            student = em.merge(student);
        }
        return student;
    }

    @Override
    public void delete(Student student) {
        em.remove(em.contains(student) ? student : em.merge(student));
    }

    @Override
    public void deleteById(Long id) {
        Student student = findById(id);

        if (student != null) {
            delete(student);
        }
    }
```

Repository Pattern Usage

Service Layer:

```
public interface StudentService {

     Student enrollStudent(Student student);

}
```

```java
public class StudentServiceImpl implements StudentService {

    private final EntityManager em;

    private final StudentRepositoryImpl studentRepository;

    public StudentServiceImpl(EntityManager em) {
        this.em = em;
        this.studentRepository = new StudentRepositoryImpl(em);
    }

    @Override
    public Student enrollStudent(Student student) {
        EntityTransaction tx = em.getTransaction();

        try {

            tx.begin();
            Student savedStudent = studentRepository.save(student);
            tx.commit();

            return savedStudent;
        } catch (Exception e) {

            if (tx.isActive()) {
                tx.rollback();
            }

            throw e;
        }
    }
```

Group Project Requirements

Update Library System Application

1. Convert to Maven Project
   a. import hibernate and PostgreSQL Driver
   b. configure persistence.xml
   c. configure jdbc connection
2. Create Entities and map relationships using JPA annotations
3. Practice Repository Pattern
   a. Create Repository Layer
   b. Create model package where you put your entity classes