**Hibernate & JPA**

**Limitations of JDBC technology :**

***1. A java developer has been forced to be aware of structured query language.***

***2. A java developer has been forced to write a boilerplate code repeatedly.***

***3. A java developer must be aware of database and table properties. A java developer must have an idea about the table on which the CRUD operations are going to be performed.***

***4. Using JDBC ,a developer cannot map a java object directly with the records in the table.***

**ORM tool :**

* ORM stands for **object relational mapping**.
* It is a programming technique that acts as a bridge between any **object-oriented code (e.g., in Python, Java, C#)** **and a relational database (e.g., PostgreSQL, MySQL, SQLite).**
* Its primary purpose is to **automatically translate data between incompatible type systems**.
* In simple terms, it lets you interact with your database using the objects and methods of your preferred programming language**, instead of writing raw SQL queries.**
* ORM tools are used to overcome the limitations of JDBC technology.
* Using ORM tools a java developer can *directly map a java object to the record in a table,*
* A java developer need not worry about structure query language, using ORM tool a java developer can normalize boilerplate code.
* It simplifies the coding by reducing boilerplate code.
* **ORM tools internally use JDBC technology itself.** ORM tools provide an abstraction layer between the java application and JDBC in order to connect java application with the database application.

**Some popular ORM tools available in the market**

**1. Hibernate**

**2. Eclipse Link**

**3. Top link**

**4. My Batis/ I Batis**

**5.Open JPA**

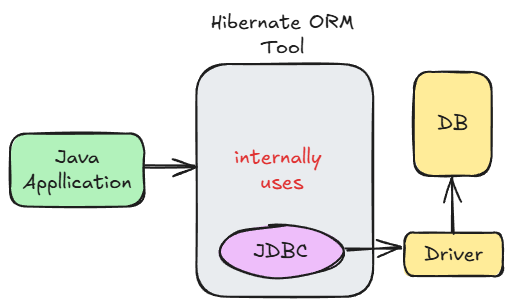
* Before Hibernate, developers used JDBC (Java Database Connectivity) to interact with databases. This involved writing a lot of boilerplate code for:

1. Opening/closing database connections.
2. Creating and executing SQL statements.
3. Manually mapping Java object properties to database table columns and vice versa.
4. Handling transactions and exceptions.

* Also, ***Using JDBC , a developer cannot map a java object directly with the records in the table.* (impedance mismatch issue).**
* **To overcome this, we can make use of Hibernate.**

**Hibernate**

* *Hibernate ORM tool is widely used ORM tool because it is* ***open source and rich in features.***
* Hibernate is an open-source **Java-based ORM tool/ framework** that provides a bridge between the **object-oriented model** and the **relational database**.
* It is used as an alternative for **JDBC**
* It is common for all the DBMS!!!
* It helps programmer to map
* ***java classes to database tables***
* ***datatypes of java with datatypes of SQL***
* ***States(variables) with columns***
* It has **built in mechanism** to perform most of the operations related to Loading class, connection and statement creation, etc.
* It has a **better performance,** while performing CRUD operations
* It will ***automatically create the table!!***
* It will ***automatically handle checked exceptions!!!***
* We can write **HQL (new version: JPQL)** queries, which are *independent of any DBMS!!*

****

**Limitations of ORM tools for database connectivity:**

Since all the ORM tools are incompatible to each other, it is not feasible for java developers to migrate from one ORM tool to another ORM tool. This incompatibility among multiple ORM tools has been overcome by introducing JPA.

**JPA :**

* It stands for **Java Persistence API.**
* JPA is a set of specifications or abstract functionality using which we can perform operation on the database.
* Its is like a blueprint specifying the functionalities to deal with database and how Java objects can be mapped to relational database tables (ORM).
* The abstract functionality in the JPA will be implemented by all the ORM tools which overcomes the incompatibility issues among all the ORM tools.

A diagram of a diagram

AI-generated content may be incorrect.

* In our case JPA is implemented by **hibernate ORM tool** which is called as **Hibernate JPA**
* ORM tools maps:
* Java class as table in the database.
* Properties/Variables as columns in the table.
* Java objects as records in the tables.

**Requirements to create Hibernate Maven Project**

* **Hibernate-core dependency:**

to add the external hibernate framework in our project. It will also add JPA internally.

* **PostgresSQL Database Driver dependency:**

to add external Driver software to interact with PostgreSQL Database.

* **persistance.xml:**
* It is a configuration file to store DB Configuration information ***(URL, user, password,etc)****,* so that the hibernate can locate the database and read the data!
* In one **persistence.xml** file, we can create **multiple persistence units,** each specifying different databases**(make sure to use class tag and exclude-unlisted-classes tag)**
* **Project Lombok(optional):**

It will automatically generates boilerplate codes for components like **constructors, getters, setters, toString(),equals(), hascode() etc.** with the help of some annotations.

**Step 1:** Create a simple maven project.

**Step 2:** **Visit** [**https://mvnrepository.com/**](https://mvnrepository.com/)

and add below dependencies in **pom.xml** (inside dependencies tag)

**a.** [**Hibernate Core**](https://mvnrepository.com/artifact/org.hibernate/hibernate-core)

**b.** [**PostgreSQL JDBC Driver**](https://mvnrepository.com/artifact/org.postgresql/postgresql)

**Pom.xml ( Project object model ) :**

It is a file which contains information about the external dependencies in the project.

For maven dependencies visit, maven repository.

**Step 3: Create a new folder in src/main/resources and name it as META-INF**

Inside **META-INF,** create one **persistence.xml** file

**Steps to create src/main/resources folder :**

*Step 1: select the project folder and press ctrl+N and search for source folder*

*Step 2: select the source folder present under java folder and give the name as src/main/resources and then click on finish.*

**Step 4:** Go to **github.com/ArpitKharche123**, go to Repositories and take code from **persistence.xml file present in Configuration-Files.**

**Step 5:** Paste the code in your own persistence.xml file, change **database name and password accordingly.**(Create a new database for hibernate)

**Step 6: Create one normal java class with main method and include this code:**

EntityManagerFactory entityManagerFactory=

Persistence.*createEntityManagerFactory*("arpit"); **//persistence-unit name**

System.***out***.println(entityManagerFactory);

**If it is executing successfully, then out configurations are correct!!!**

**Object to Table Mapping**

**A diagram of a student

AI-generated content may be incorrect.**

* We will use **@Entity** Annotation to map java class with Table.
* Class name and table name should be same.
* If we create only class with **@Entity** , hibernate will automatically create the table for it
* Whenever we create **an object** for entity class by passing some values and calling built in methods in hibernate, data will get stored in the mapped table.

**Rules to Create Entity Class**

* **We can use @Entity on a public non-abstract(concrete) class only!**
* **Entity Class should have one Primary key(  To uniquely identify each row/record in a database table.). We can make a variable/column as a primary key by using @Id annotation on top of the variable declaration. (required during fetching,updation and deletion)**
* **Entity class must have public no-arg constructor or public default constructor. (required during fetching,updation and deletion)**
* **For every variable of an Entity class, there should be public getter and setter methods**
* **Entity class should implement Serializable interface(optional).**

e.g. refer demo/Student.java

**Layers in Hibernate JPA :**

**DAO : DAO stands for Data Access Object.**

This layer is responsible to hold the main logic or the programs to perform CRUD operations on the database.

**e.g. EmployeeDAO**

**DTO : DTO stands for Data Transfer Object.**

This layer is responsible to hold the **entity classes** which are used to hold the application data. DTO layer acts as an interface between DAO layer and database application.

**e.g. EmployeeDTO**

**Prerequisite to perform CRUD**

* Create an Entity class with all the necessary properties, getters, setters, no-arg constructors and annotations.
* We have to use 3 built in interfaces to perform CRUD.
* **EntityManagerFactory:**
* An **EntityManagerFactory** is an application-managed factory that provides **EntityManager** instances.
* It is an **interface** present in **Jakarta.persistence package**, used to establish the connection with **a particular database.**
* It is configured based on the settings in the **persistence.xml** file .
* It acts as a factory for producing EntityManagers. The application typically has one **EntityManagerFactory** for a given "***persistence unit***" (a specific database configuration).
* In this ***persistence unit***, database info**(url,user,password,etc)** is present. **EMF** object will be created if all the info is correct!
* As we cannot directly create an object for **EMF interface,** we will use one helper class named **Persistance**, inside which we have one **factory method,** which will create an instance for **EntityManagerFactory**
* ***public static EntityManagerFactory createEntityManagerFactory(String persistenceUnitName)***
* ***EntityManagerFactory*** *is one for* ***one database****, as well as in* ***one persistence unit,*** *configuration of* ***one database*** *can be given.*
* **EntityManager:**
* It is **an interface** present in **Jakarta.persistence package.**
* It is the primary interface used by application to interact with the persistence context.
* The **persistence context** is a **set of managed entity instances (like a temporary memory)** within which the lifecycle of **entities/objects** is managed.
* EntityManager is used to create, read, update, and delete entities with the help of some **built in methods**.
* We can create multiple **EntityManager instances**  by using same **EntityManagerFactory instance** to perform actions on a particular database.**(to prevent data inconsistency due to lack of thread safety)**
* **EntityTransaction:**
* It is an **interface** present in **Jakarta.persistence package** used to manage transactions on one **EntityManager.**
* It is like an **on/off** switch for group of operations that can either pass or succeed as a unit.
* It is generally used when we want to deal with **DML** operations. (not required for others like **DDL, DQL etc).**
* It consists of non-static methods like **begin(), commit(), isActive(), rollback() etc** using which we can manage **one transaction** and perform multiple operations as a single unit**(either all will succeed,or none).**
* **Insert:**

**[refer CRUD/Employee and CRUD/InsertEmployee classes]**

* **public void persist(Object entity):** This method is used to save the specified entity inside the mapped table.
* **E.g.:**

**Employee e2=new Employee("Michael","m@gmail.com","o+");**

**em.persist(e2);**

* **Fetch:**

**[refer CRUD/Employee and CRUD/FetchEmployee classes]**

* **public <T> T find(Class<T> entityClass, Object primaryKey):** It is used to fetch the record/entity based on given **primary key.**
* **e.g:**

**Employee emp = em.find(Employee.class, 2);**

* **Update:**

**[refer CRUD/Employee and CRUD/UpdateEmployee classes]**

* **public <T> T merge(T entity):** It is used to merge the given state of an entity into the **current persistence context.** i.e it will update the entity record in table.
* **E.g.:**Employee employee=em.find(Employee.class, 2);

//Null Check

if(employee!=null) {

employee.setEmail("tokita@gmail.com");

et.begin();

em.merge(employee);

et.commit();

System.*out*.println("Employee is updated successfully!!");

}else {

System.*err*.println("Employee does not exists!");

}

* **Delete:**

**[refer CRUD/Employee and CRUD/DeleteEmployee classes]**

* **public void remove(Object entity):** It is used to remove entity instance./
* It is used to delete the given employee from the table.
* **e.g.:**

Employee employee=em.find(Employee.class, 5);

//Null Check

if(employee!=null) {

et.begin();

em.remove(employee);

et.commit();

System.out.println("Employee is removed successfully!!");

}

else {

System.err.println("Employee does not exists!");

}

**Hibernate Entity Life Cycle**

A diagram of a process

AI-generated content may be incorrect.

**JPQL :**

* **JPQL (Java Persistence Query Language)** is a *platform-independent object-oriented query language* defined as part of the **Java Persistence API (JPA) specification.**
* It is used to implement custom queries to perform custom operations on the database.
* JPQL is similar to SQL but in case of JPQL **class name is used in the place of table** and **the properties are used in the place of columns**.
* JPQL is a query language which is understandable by all the ORM tools which eliminates the incompatibility issue present among multiple ORM tools.

**E.g:**

**SELECT Queries**

SELECT e FROM Employee e WHERE e.salary > 50000

**UPDATE Queries**

UPDATE Employee e SET e.salary = e.salary \* 1.1 WHERE e.department = 'IT'

**DELETE Queries**

DELETE FROM Employee e WHERE e.status = 'INACTIVE'

* **Fetching all the Records from the Table using JPQL**

**[refer CRUD/Employee and CRUD/SelectAllEmployees classes]**

* **public Query createQuery(String qlString):** This method is used to create an instance of Query interface

**Query query=em.createQuery("select e from Employee e");**

* **List getResultList():** This method is used to execute a SELECT query written inside **createQuery()**

**List<Employee> employees=query.getResultList();**

* **Query Parameters in JPQL**
* Query parameters in JPQL are placeholders that allow you to pass values into your queries dynamically.
* They make your queries more flexible, secure, and reusable by separating the query structure from the actual values being used.

**Types of Query Parameters**

JPQL supports two main types of parameters:

**1. Positional Parameters**

* **Denoted by ? followed by a number (e.g., ?1, ?2)**
* **Parameters are referenced by their position in the query**

**2. Named Parameters**

* **Denoted by  :  followed by a name (e.g., :firstName, :minSalary)**
* **Parameters are referenced by their descriptive names.**
* **Dynamic JPQL**
* We can make use of **placeholders(?)** to write a dynamic jpql
* It is recommended to avoid **sql injection** as well as **to achieve dynamic input from user.**

**e.g.: [refer DynamicCRUD]**

**e.email=?1, e.eid=:empId"**

* **? and :** acts as **placeholders**
* **1** is the parameter position or we can also give parameter name**(empId)**
* **Query setParameter(int position, Object value);**

*:- Bind an argument value to a positional parameter.*

**updateEmp.setParameter(1,email);**

**updateEmp.setParameter("empId", 1);**

**Composite Primary Key**

* Composite Key is a primary key that consists of two or more columns used to uniquely identify each row in a table.
* To create a composite key in Entity Class we have to follow some steps

**Step 1:** Create a class in **DTO** only to store composite keys/ variables, and use ***@Embeddable*** annotation with that class( overriding **equals()** method is recommended)

**[refer DTO/CompositeKeysOfProfile]**

**Step 2:** Create one entity class in **DTO** and include the Composite keys class in it using **@EmbeddedId** *(Composition/ Has-a)* and other non—primary columns.

**[refer DTO/Profile]**

**Step 3:** Create one Implementation class in **DAO** and try inserting/updating the records.

**[refer DAO/UserProfileInsert]**

**Annotations used in Hibernate JPA :**

**@Entity:** This annotation is a class level annotation which is used to mark a class as entity class, so that ORM tool can map that class as table inside the database.

**@Id:** This annotation is a property level annotation which is used to mark a property as ID so that that property will be mapped as a column with primary key constraint in the table.

**@Table(name=”xyz”):** This is a class level annotation which is used to modify table properties in the database.

**@Column(name=”xyz”):** This annotation is a property level annotation which is used to modify the column properties in the table

**@GeneratedValue:** This annotation is used to generate the values for ID property automatically.

***@CreationTimestamp*:** Automatically sets the value of the field with the current date/time when the entity is first saved (INSERTED) into the database.

It will **not change** on updates.

**@UpdateTimestamp:** Automatically updates the field with the **current date/time whenever the entity is UPDATED** in the database.

It will also set the time when the entity is first inserted.

**Example:[Refer DTO.Teacher and DAO.InsertTeacher]**

**Generation Types**

* In Hibernate , **@GeneratedValue** is used for **primary key generation strategies**.
* Hibernate provides multiple **generation types** (GenerationType) that decide how the primary key (id) is created, generated and managed in the database.

**Generation Types in Hibern****ate**

**1. AUTO**

* **Behavior**:
  + In MySQL/PostgreSQL → behaves like **IDENTITY** (auto-increment).
  + In Oracle → behaves like **SEQUENCE**.
* **Use Case**:
  + Best when you want Hibernate to decide automatically without worrying about the database.
* [**Refer DTO.Consumer and DAO.InsertConsumer**]
* **Note: Here, one sequence will also be created along with the entity table with columns, *last\_value,long\_cnt,is\_called with single record only, which will be updated when new data is inserted***

**e.g:**

***@GeneratedValue*(strategy = *GenerationType*.*AUTO*)**

**🡪** This will increment **Id and last\_value** in following way:

**Id=last\_value+1;**

**🡪 last\_value** will increase by 50 after a certain point

**🡪Id: 1,2,52,102,152**

**🡪last\_ value: 1,51 ,101,151**

* **To create a custom sequence**

***@GeneratedValue*(strategy = *GenerationType*.*AUTO*,generator = "con")**

***@SequenceGenerator*(name = "con",**

initialValue = **1*, //starting value***

allocationSize = **10*,// by how much to increase***

sequenceName = "**consumer\_sequece\_info**"**)*//name of sequence table***

**🡪** This will increment **Id and last\_value** in following way:

**Id=last\_value+1;**

**🡪 last\_value** will increase by **10** after a certain point

**🡪Id: 1,2,12,22…..**

**🡪last\_ value: 1,11,21,31…**

**2. IDENTITY**

* Auto increment by 1, starting from 1
* [**Refer DTO.Teacher and DAO.InsertTeacher**]
* **Note: Here, one sequence will also be created along with the entity table with columns, *last\_value,long\_cnt,is\_called with single record only, which will be updated when new data is inserted***

**Example:**

***@GeneratedValue*(strategy = *GenerationType*.*IDENTITY*)**

**Here ,** the **last\_value and id** would be like:

**🡪Id: 1,2,3,4,5…..**

**🡪last\_ value: 1,2,3,4,5…..**

**3. SEQUENCE**

* Uses a database sequence object to generate unique values.
* **Behavior:**
  + Supported in databases like Oracle, PostgreSQL, H2.
  + Hibernate can fetch ids in advance **(better for batch inserts).**
* **Use Case:**
  + Large-scale applications where sequence objects are supported and performance is key.
* **Note: Here, one sequence will also be created along with the entity table with columns, *last\_value,long\_cnt,is\_called with single record only, which will be updated when new data is inserted***

**[refer DTO.Bike and DAO.InsertBike]**

**Example:**

***@GeneratedValue*(strategy = *GenerationType*.*SEQUENCE*)**

**🡪**It will act similar to **GenerationType.AUTO** default behaviour of automatically incrementing id value**(1,2,52….)**

🡪We can also customize the strategy with the help of ***@SequenceGenerator***

**//Custom strategy similar to GenerationType.IDENTITY**

***@GeneratedValue*(**strategy **= *GenerationType*.*SEQUENCE*,**generator = **"bike")**

***@SequenceGenerator*(**name **= "bike",**

initialValue **= 1, //starting value**

allocationSize = **1,// by how much to increase**

sequenceName = **"bike\_seq")**

**🡪** This will increment **Id and last\_value** in following way:

**Id=last\_value+1;**

**Here ,** the **last\_value and id** would be like:

**🡪Id: 1,2,3,4,5…..**

**🡪last\_ value: 1,2,3,4,5…..**

**4. TABLE**

* Uses a special database table to maintain and generate primary key values.
* **Behavior:**
  + Creates a table **(like hibernate\_sequences)** that stores the next available id.
  + Works across all databases.
* **Use Case:**
  + **When portability across databases is required but the DB doesn’t support sequences.**
* **Note: Here, *one sequence table* will also be created along with the entity table with columns, *sequence\_name , next\_val with single record only, which will be updated when new data is inserted.***

**[Refer DTO.Mobile and DAO.InsertMobile]**

**Example:**

***@GeneratedValue*(strategy = *GenerationType*.*TABLE*)**

**🡪** One extra table will be generated with name ***hibernate\_sequences*** *in tables section.*

**🡪**It will act similar to **GenerationType.AUTO** default behaviour of automatically incrementing id values**(1,2,52….)**

🡪We can also customize the strategy with the help of ***@TableGenerator***

**//Custom strategy similar to identity**

***@GeneratedValue***(strategy = *GenerationType*.***TABLE***,generator = "**mob**")

***@TableGenerator***(

name = "**mob**",

initialValue = 0,

allocationSize = 1,

table = "**mob\_seq**" //**sequence table name**

)

**🡪** Now, one extra table will be generated with name **mob\_seq** *in tables section.*

**📊 Comparison Table**

| **Strategy** | **Database Dependency** | **When to Use** |
| --- | --- | --- |
| **AUTO** | Let’s Hibernate decide | Multi-DB support |
| **IDENTITY** | Auto-increment | Simple inserts, small apps |
| **SEQUENCE** | Sequence-supported DBs (Oracle, PostgreSQL) | High-performance, batch inserts |
| **TABLE** | Works on all DBs but slower | DBs without sequences |

**Mappings in Hibernate JPA :**

* Hibernate Mapping is one of the key techniques using which we can map/establish connection between two or more Entity Classes/ Tables.
* We can establish connections/ mapping in two ways:
* **Uni-directional mapping**
  + - If the relationship is defined inside any one of the entities, then it is called Uni-directional mapping.
    - Only one entity knows about the relationship, the other entity has no reference back.
    - The mapping exists in **one direction only**.
* **Bi-directional mapping**
* If the relationship is defined inside both the entities, then it is called as Bi-directional mapping.
* Both entities know about each other.
* The mapping/relationship exists in **both directions**.
* **Dependent Entity Class:** An entity class which contains the object or collection of objects of another entity class as a property is called as dependent entity class.
* **Dependency Entity Class**: An entity class whose object or collection of objects has been used as a property in another class is called as dependency entity class.

1. **one to one unidirectional mapping** :

* If a record from table 1 belongs to exactly one record from table 2 then it is called as **one to one uni-directional mapping**.
* Two tables will be created for 2 entity classes with **foreign key column in dependent table(Car table)**.
* **Examples:**
* **One Car to One Engine entity mapping(One Car has a one engine)**
* **One Passport to one person.**

A diagram of a network

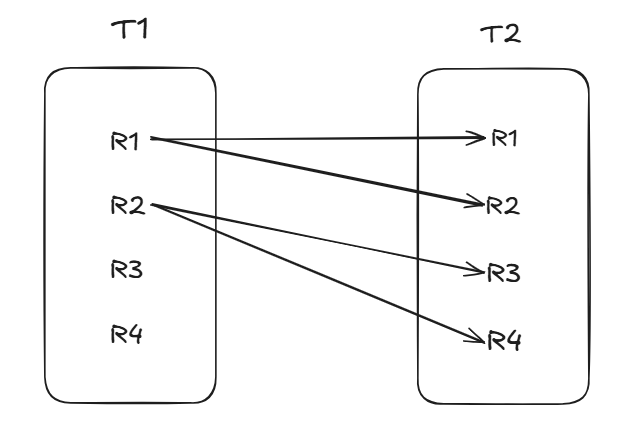
Description automatically generated

**[refer Mappings/DTO/Unidirectional and Mappings/DAO/Unidirectional]**

1. **one to many unidirectional mapping :**

* If a record from table 1 is related to or belongs to more than one records from table 2 then it is called as one-to-many unidirectional mapping.
* Three tables **(one showing the related data(primary key relations)** will be created for 2 entity classes.
* **Examples:**
* **One Bank has multiple accounts**
* **One Car has multiple tyres**
* **One Customers can have multiple orders**

**[refer Mappings/DTO/Unidirectional and Mappings/DAO/Unidirectional]**



1. **many to one unidirectional mapping :**

* If more than one records from table 1 belong to only one record from table 2 then it is called as many to one unidirectional mapping.
* Two tables will be created for 2 entity classes**(product\_id column will be added extra to reviews table)**
* **Examples:**
* **Many Reviews belongs to One Product**
* **Many Products belongs to One Order**

A diagram of a diagram

Description automatically generated

1. **many to many unidirectional mapping :**

* If more than one records from table 1 belong to multiple records from table 2 then it is called as many to many mapping.
* Three tables **(one showing the related data(primary key relations)(junction table)** will be created for 2 entity classes.
* **Examples:**
* **Many Subjects belongs to Many Students**
* **Many Users can have many mutual followers**

A diagram of a network

Description automatically generated

1. **one to one bi-directional mapping** :

* If a record from table 1 belongs to exactly one record from table 2 **and vice versa** then it is called as **one to one bi-directional mapping**.
* Two tables will be created for 2 entity classes with **2 foreign key columns in both classes/tables**.
* **Bi- directional association** can be achieved i.e ***using Car class, we can fetch Engine and using Engine class we can fetch car.***
* **Examples:**
* **One Car to One Engine entity mapping**
* **One Passport to one person.**

A diagram of a number of objects

AI-generated content may be incorrect.

* **To remove extra foreign key column from dependency class**

**Step 1:** Drop the existing tables with bidirectional association.

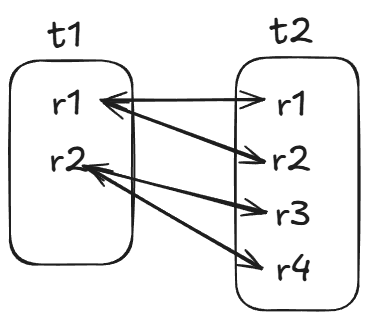
**Step 2:** Use **@JoinColumn(name=”my\_engine\_id”)** along with **@OneToOne** in Car2 class above **Engine engine**.

**Step 3:** Use **@OneToOne(mappedBy=”engine”)** in Engine class above **Car car;**

* ***This will create only one foreign key column In Car2 table with name* my\_engine\_id *, retaining the bi directional association.***
* ***This will help us to avoid redundant column creation, as bi directional association can also be achieved using one foreign key column .***

1. **one to many and many to one bi-directional mapping** :

* If a record from table 1 belongs to more than one records from table 2 **and vice versa**, then it is called as **one-to-many and many to one bi-directional mapping.**
* **Three tables** will be created for two entity classes, showing the **foreign key relation**.
* Also, one foreign key column will be created for **single entity class** in **multiple entity class table**(**e.g. bank\_id column in account table)**
* **Examples:**
* **One Bank has multiple accounts and vice versa**
* **One Car has multiple tyres and vice versa**
* **One Customers can have multiple orders and vice versa**



* **[refer Mappings.DTO.Bi.O2M2O.BankAccountDTO and Mappings.DAO.Bi.O2M2O.BankAccountDAO]**
* **To remove extra 3rd table (relationship table)**

**Step 1:** Delete 3rd table **(bank2\_account2 table)**

**Step 2:** Use:

***@OneToMany*(mappedBy = "bank")**

private List<Account2> accounts;

in **Bank2 class**

**Step 3:** Use:

***@ManyToOne***

***@JoinColumn***

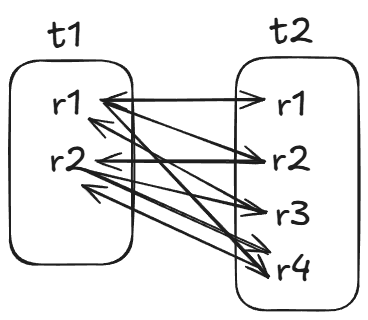
private Bank2 bank;

in **Account2 class**

* ***This will avoid the creation of unnecessary third table* (bank2\_account2 table), *retaining the bi directional association.***

1. **many to many bi-directional mapping :**

* If more than one records from table 1 belong to multiple records from table 2 then it is called as many to many mapping.
* Four tables **(two showing the related data(primary key relations)(junction table)** will be created for 2 entity classes.
* These **extra 2 relation tables** are redundant, so we have to avoid the creation of **one of these 2 tables**!! Using **@JoinTable** annotation.
* **[Refer Mappings.DTO.Bi.M2M.Students2Subjects2DTO and Mappings.DAO.Bi.M2M.Students2Subjects2DAO]**
* **Examples:**
* **Many Subjects belongs to Many Students**
* **Many Users can have many mutual followers**

****

**Fetch types:**

* Fetch types define the way of fetching **dependency objects** whenever **dependent object** is fetched from the database.
* When we work with **relationships(association)** between entities (like Student–Subject, Employee–Department, etc.), Hibernate has to decide:

*"Should I load related objects immediately from the database, or only when they are really needed?"*

**So, this is decided on the basis of fetch type.**

**There are two fetch types:**

**EAGER** → Load everything **immediately**.

**LAZY** → Load related things **only when asked for**

1. **Eager fetch type:**

* In eager fetch type whenever **a** **dependent object** is fetched from the database then all the related **dependency objects** also get fetched from the database automatically.
* Whenever we load the main object, Hibernate also fetches **all the related objects at the same time**.
* Think of it as **ordering a thali (meal plate)** → we will get **all dishes served immediately**, whether you eat them or not.

1. **Lazy fetch type:**

* In lazy fetch type whenever **a dependent object** is fetched from the database then the related **dependency objects** will not get fetched automatically.
* Hibernate will load the **main object only**, and related objects will be fetched **only when you call getter methods**.
* Think of it as **ordering separate dishes**→ Plates and glass will be given, we will get a particular food **only when you order it**.
* **Default fetch types:**
* One to One & Many to One: **Eager**
* One to Many & Many to Many: **Lazy**
* **Examples(Car-engine)**

1. ***@OneToOne***

private Engine engine;

(*here default fetch type is* ***eager****, so it will load car as well as engine as follows:)*

**Car car=em.find(Car.class, 101);**

**o/p:**

**select**

**c1\_0.id,**

**c1\_0.cost,**

**e1\_0.id,**

**e1\_0.cc,**

**e1\_0.type,**

**c1\_0.name**

**from**

**Car c1\_0**

**left join**

**Engine e1\_0**

**on e1\_0.id=c1\_0.engine\_id**

**where**

**c1\_0.id=?**

1. ***@OneToOne (fetch = FetchType.LAZY)***

private Engine engine;

(*here it will load objects only when they are accessed as follows:)*

**Car car=em.find(Car.class, 101);**

**o/p:**

**select**

**c1\_0.id,**

**c1\_0.cost,**

**c1\_0.engine\_id,**

**c1\_0.name**

**from**

**Car c1\_0**

**where**

**c1\_0.id=?**

**Cascading**

* It is a way using which we can strictly bound **dependency entity** objects to **dependent entity** *i.e* ***operation done on dependent entity will also get reflected on dependency entity.***
* **The standard cascade types (JPA)**

**CascadeType** is an enum.

**CascadeType.ALL** = (PERSIST, MERGE, REMOVE, REFRESH, DETACH).

* **PERSIST**:

🡪 em.persist(parent) also **persist** children.

🡪Use when child should be automatically created when parent is created.

* **MERGE**:

🡪em.merge(parent) also merge**(update)** children.

* **REMOVE**:

🡪em.remove(parent) also **remove** children (delete from DB).

* **ALL** — all of the above.

**Example:**

***@OneToOne*(cascade = {*CascadeType*.*PERSIST*,**

***CascadeType*.*REMOVE*,**

***CascadeType*.*MERGE*} )**

**private Engine engine;**

**Hibernate Caching**

**🔹 What is Caching in General?**

* **Cache** means storing frequently accessed data in temporary memory so that it can be retrieved faster.
* Instead of hitting the **database every time** (which is costly in terms of time and resources), we fetch from cache if the data is already present.

**🔹 What is Hibernate Caching?**

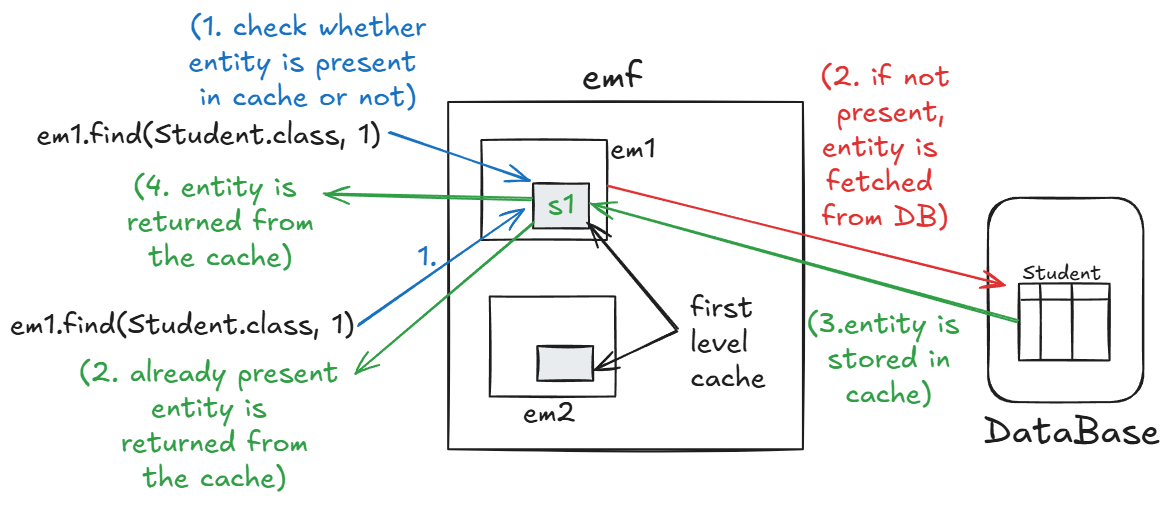
* Hibernate caching is a mechanism to **reduce the number of database queries** by storing frequently used objects in memory.
* When you fetch an entity in Hibernate:
* First, Hibernate checks the **cache**.
* If the data is found → returned from **cache (fast).**
* If not → Hibernate fetches from **DB and puts it into the cache for future use.**
* This improves:
* **Performance** (less DB calls).
* **Application scalability**.

**🔹 Types of Hibernate Cache**

Hibernate provides **2 levels of caching**:

**1. First-Level Cache**

* Enabled by **default** in Hibernate.
* **Working:**
* While **fetching records,** it will check ***whether the record is present in first level cache or not.***
* If not present, it will **go to the database, *executes a select query* and fetches the record.**
* The **fetched record/entity** will be stored in cache first, then it will be returned to the caller.
* If the **record/entity** is already present in the first level cache, it will be returned to the caller directly , ***without executing any database query,*** hence **improving the application performance.**



**Example:**

**[refer DAO.TestFirstLevelCache]**

* ***Note: first level cache is one for an entity manager***
* So, if we try to access same record using another **entity manager instance,** it have to follow all 4 steps to fetch the **entity,** as it is not present in cache, it has to ***execute query again.***

**A diagram of a diagram

AI-generated content may be incorrect.**

* So, to optimize such situations, we can make use of **second level cache.**

**2. Second-Level Cache** TODO code not done

* Not enabled by **default**, we need to configure it explicitly.
* There are many third-party vendors who provides some **dependencies/ APIs** to achieve **second level caching.**
* ***Note: second level cache is one for an entity manager factory, so it can be shared with all entity managers of one emf***

A diagram of a diagram

AI-generated content may be incorrect.

* **Working:**
* If the record is not present in **first level cache,** it will check the **second level cache.**
* If the record is not present in **second level cache,**  then it will ***execute the query and fetch the record.***
* Fetched record is then stored in **second level cache,** then stored in **first level cache,** then returned to the caller**.**
* If the record is present in **second level cache,** then it will store the record in **first level cache,** then return it to the caller, ***without executing the query***, hence optimizing the overall execution**.**

**Steps to implement second level cache:**

**Step 1**: Go to [mvnrepository.com](https://mvnrepository.com/artifact/org.hibernate/hibernate-ehcache) and select [Hibernate Ehcache Relocation](https://mvnrepository.com/artifact/org.hibernate/hibernate-ehcache)

**Step 2:** Select version similar/ nearby to the hibernate core version and paste it in pom.xml

**Step 3:** Go to **persistence.xml** and enable the second level cache, add the following:

* **<shared-cache-mode>ENABLE\_SELECTIVE</shared-cache-mode>**
* **<!-- Enable 2nd Level Cache -->**

**<property name=*"hibernate.cache.use\_second\_level\_cache"* value=*"true"*/>**

* **<!-- Cache Provider-->**

**<property name=*"hibernate.cache.region.factory\_class"* value=*"org.hibernate.cache.ehcache.EhCacheRegionFactory"*/>**

**Example: [refer DAO.TestSecondLevelCache]**