# Previously we used to do like this

MySQL client DB

Request

Response Response

# Now we will use java client instead of MySQL client

Now db(chip) we have but we can’t use directly so we have driver(different for different db) which will be used from java code now different db version can have different driver but we are going to use driver directly so our code will be totally dependent on the driver and if in future we want to use different database then our code will be totally different that we don’t want

We want our code to be independent of any driver, We do not want our code to be directly dependent on a particular driver for this jdbc api is created in java

java client

jdbc driver DB

(low level code)

now Instead of using the driver directly you're going to use database connector API. Now you can move to any database your code will not get changed, because you are not directly dependent on driver, you are dependent on the standard specification.

Driver has the low level code to connect you application to the database

Let us see other problems now you have started using jdbc now let us see why jdbc is not used in industry that much I mean we never code on jdbc but it always gets used the reason is-

Jdbc is created for sql databases okay so you have sql database in which tables are in the metrics format(row column)

|  |  |  |
| --- | --- | --- |
| id | name | age |
| 1 | sohan | 22 |
| 2 | rohan | 23 |
| 3 | mohan | 24 |

# Now in java side we are not representing in this way we represent like

So you can see there is definitely mismatch here we have object oriented side, here we have metrics format. That means if you're going to write this details which you have in this object if you're going to insert in table, and let us suppose that row doesn’t exist and you're going to insert so you have to convert from object to row format and vice versa so at both side we will do conversion n number of times and this is a big big headache that’s why jdbc is not much used

So what we required here is ORM(object relational mapping) that will do conversion both side automatically

Now you have created java application and ORM library you are using is hibernate( that is vendor based implementation) so now in future if you want to use toplinks instead of hibernate so then lot of code changes will be there why because you are using the api which is not standard

Sun Microsystems actually created a standard at a later time, and that is very popular which is Jpa that is only used for sql based database

🡪JPA is a library and hibernate is implementation,JPA contains lot of interfaces

JPA

Hibernate TopLinks

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

Now, your code will not be dependent on vendor based API your code will be using Jpa standard. So, in the future, if you want to move to another vendor like toplink you can move because your code is not directly dependent on particular implementation

conversion

conversion

so what kind of operation you are able to do obviously CRUD operation

C🡪Create or insert the row

R🡪 Read/fetch the row

U🡪change the row

D🡪 delete the row

So the API we are going to use is EntityManager based API which is there in JPA so there we have CRUD operations method

C🡪persist(obj)

R🡪 find(id)

U🡪merge(obj)

D🡪 remove(row)

So using EntityManager object we are going to work on the Entity(ex-Product, Employee Eetc)

Now each entity manager is associated with persistent context(one to one relationship)

em1 pc1

em2 pc2

One entity manager cannot use persistent context of another entity manager, what a persistent context is persistent contexts are the set of entity object which you have ,pc is like cache So you can see, this is your application.

Pc will contain the object of entities

🡪 it will only contain the unique entity object

For example you have product with id 1 so you can’t have another product object with id 1

# Pc(first level cache)

# DB

em

give

To whatever you are going to insert or going to fetch from the database , For example, you are doing insert operation, the before of our row is inserted, the object will be kept in the persistent context. Similarly, when you're fetching our object which is equivalent to row details First It will also be kept in the persistent context.

Now our entity object is stored in the pc that is cache memory now you have Samsung with id 1 and now you are saying fetch me product details with id 1 your disc will not be hit ,your database is slow it will give you from the pc , if you will ask 20 times it will give you from pc ,disc will not be hit if it is not in the pc first it will be fetched from disc then will be kept in the pc then you can get from the pc so first time it will be kept in the pc and rest 19 times you can get from the pc itself your disc will not be hit discs are slow databases are not that fast your cache are first similarly for insertion object will first stored in the pc then it will inserted in the database

Now we already know pc is associated with the entity manager so if you em doesn’t exist pc will not exist it will be destroyed

Then you will create a new entity manager, you will get a new persistent context

Now whenever you object is created ,new object is created like this

Product product=new Product(1,”Samsung”,10000);

Jpa and hibernate know nothing about it at this time our object is in new state or transist state now you are doing

em.persist(obj) or em.merge(obj) so then this object will kept in the persistent context now hibernate knows about it, now our object is in managed state previously it was in new state

Now At a later time, let us suppose the object be removed from the Persistent context or your entity manager gets close so pc will be destroyed now again object will be again in the detached state again hibernate knows nothing about it but the difference between the new state and the detached state is that in detached state your object will have Atleast the primary key, ID or whatever, which is representing the row in that table. But the new state your object might contain or might not contain ID. And it might not represent any of the row on the table , but your deteched objects will surely be representing one of the row of the table

# Now do some practical

## Now I will add [Hibernate Core Relocation](https://mvnrepository.com/artifact/org.hibernate/hibernate-core) dependency in pom.xml file but i'm not adding JPA dependency because hibernate has transitive dependency on the JPA, so if I am adding hibernate by default we will get JPA dependency

## Now for MySQL we will add MySQL driver dependency [MySQL Connector/J](https://mvnrepository.com/artifact/mysql/mysql-connector-java) dependency

Note—Entity class represents the table in the database

Now suppose we have entity class Employee

Class Employee{

}

This Entity (Employee) class representing the table in the database

Now hibernate will not consider Employee class as the Entity class so we will mention @Entity on top of the class so now hibernate will know Employee is a Entity class

By default table name will be same as Entity class name for here table name will be Employee

But if we want different or custom table name then for example we want table name employees then we can mention @Table(name=”employees”) followed by @Entity

Now you can’t be having entity class without primary key in jpa because it represents the table in the database so its object will represents the row in the database so we will mention @Id annotation on the field for let us say private int id because id is unique now we want id to be generated automatically by hibernate I’ll not write the code to generate for that we will mention

For ex🡪

@Entity

Class Employee{

@GeneratedValue

@Id

private Long id;

private String name;

private int age;

}

Now we have to mention the hibernate mysql configuration for that we will be having resources directory parallel to java inside it we will create META-INF directory and inside it we will be having persistence.xml file here we will provide configuration for MySQL , you will be having different configuration for other SQL based database

Now in the persistence.xml file we have persistence-unit

persistence-unit represents all the entity class in you application that are managed by EntityManager instance in your application

in the persistence-unit we have name attribute for example name=”empms”

<property name="hibernate.hbm2ddl.auto" value="create"/>

So now hibernate will automatically create the table so we have more than one strategy but here we have mentioned create strategy using value=”create”

So now what will happen ,

It is good for development so now when your application will start table will be created with the rows etc application stops table will be still there your application starts previous table will get deleted and data inside it will be lost a new table will be created every time you restart the application new table will be created and this is good for development because this will be near about guarantee that the entity and entity structure or entity definition and table structure are in sync that’s why this strategy is good for development

Now do some coding

We will first create EntityManagerFactory

EntityManagerFactory entityManagerFactory=Persistence.createEntityManagerFactory("empms");

Here empms is persistence-unit

So basically EntityManagerFactory is a container or a factory that is set of entity manager object other thing which a factory does is it helps in creating entity manager object so instead of writing new EntityManager you are going to use a method in factory which will give you the entity manager object or the new object

🡪EntityManager basically holding the connection to the database and there are actually limit to the number of connection, you can have, at one time.

**em1 em2 em3**

**em4**

So now we have entityManagerFactory object so now instead of mentioning new EntityManager we use entityManagerFactory. createEntityManager() which is going to give me the EntityManager

🡺createEntityManager is actually interface there we have CRUD methods

🡺createEntityManager will actually give you the object of its implementation

### Transaction Management

The transaction concept is that you should move from one stable state to another stable state and if you cannot move because of any failure, etc, so please go back to the previous state. And if all the operations are successfully completed the then move to the new state and if there is any failure, then roll back to the previous state it and if there is no problem so then successfully go back to the next step.

Only after all the operations are successfully completed changes will be stored in the database, else it will not be stored

Now we need to get the transaction object for that we will do

EntityTransaction transaction=entityManager.getTransaction();

Now we have to begin the transaction for that we will write

transaction.begin();

And after everything is completed successfully we will write

transaction.commit();

and if there is any failure let us suppose exception or any failure commit() will not get executed so you will be having roll back by default that means you will roll back to the previous state everything got successfully completed commit() will be executed

🡪In between transaction.begin() and transaction.commit() we will do our work

To close the entityManagerFactory and entityManager we will write

entityManager.close();

entityManagerFactory.close();

when table will be created in database the one more table is created named as hibernate\_sequence it keeps track of the id generated so that it can create new id ,it knows previous generated id

* We don’t need to do transaction management for the read operation beacuse we are not causing the database
* merge method is used for both adding and updating the object

@Column(nullable = false,unique = true) is used for make the field of the object not null so that we can’t add the object with field null

#### JpaRepository

JpaRepository is particularly a JPA specific extension for Repository. It has full API CrudRepository and PagingAndSortingRepository. So, basically, Jpa Repository contains the APIs for basic CRUD operations, the APIS for pagination, and the APIs for sorting.

for example if we have DepartmentRepository then syntax will be

public interface DepartmentRepository extends JpaRepository<T, ID> {}

Where:

T: Domain type that repository manages (Generally the Entity/Model class name)

ID: Type of the id of the entity that repository manages (Generally the wrapper class of your @Id that is created inside the Entity/Model class)

Some of the most important methods that are available inside the JpaRepository are given below

Method 1: save(): adding one entity

Syntax:

<S extends T> S save(S entity)

Method 2: saveAll(): Saves all given entities.

Syntax:

<S extends T> List<S> saveAll(Iterable<S> entities)

Method 3: getById():

Syntax:

Optional<T> getById(ID id)

Method 4: findAll():

Syntax:

Iterable<T> findAll()

Method 5: for deleting there are many methods

Syntax:

deleteAll(),deleteById(T id), delete(Entity entity) 🡪 all have void return type

many more methods are also there

[learn more](https://docs.spring.io/spring-data/jpa/docs/current/reference/html/)

#### JpaRepository Custom Finder Method/Derived Query Methods

* all the methods we will write inside interface and spring will automatically give the implementation

suppose we want to find the User by name then method name will be like this

public List<User> findByName(String name);

here ‘find’ is introducer, ‘ByName’ is criteria, name is property that is member of object

similary by city🡪

public List<User> findByCity(String city);

🡪by name and password

public List<User> findByNameAndPassword(String name,String password);

public List<User> findByNameStartingWith(String prefix);

and so on…

[learn more](https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#reference)

Executing JPQL and Native Queries with Spring Data JPA | @Query @Param Annotations

🡪we can fire our query(jpql and native sql query) using @Query annotation

suppose we are creating getAllUser method and using jpql then we will do like like this

we will create getAllUser() inside interface and mention jpql query using @Query

@Query(“From User”)

List<User> getAllUser();

🡪spring will automatically execute the query and we will get result

similarly

@Query(“From User where name=:nameArg”)

List<User> getUserByName(@Param(nameArg) String name);

🡪now if we are using native query then

@Query(value = "select \* from User" ,nativeQuery = true)

List<User> getUsers();

Spring Data JPA Mappings

It means creating the relation between the entities

1. One to One mapping
2. One to Many Mapping
3. Many to Many mapping

1.One to One mapping

suppose we have two entity Student and Laptop and one student has only one laptop then this is one to one mapping

Laptop

Student

similarly

Cart

customer

now do the implementation

suppose we have two class Student and Laptop and inside the Student class we have Laptop class as a field and because here two entities have one to one relationship then we mention @OneToOne on top of Laptop field then Laptop table id will be the foreign key of Student table

currently there is uni-directional mapping because inside Student class we have Laptop field so if we have Student object then we can directly get the Laptop object but inside the Laptop class there is no such field of Student class so we can’t get Student object if we have Laptop object

to make bi- directional we will also declare the Student class field inside Laptop class with @OneToOne , so now both the tables contain the foreign key(id) of each other

now we can make bidirectional in a better way, now we don’t need to create the foreign key from both end , using only one foreign key we can establish bidirectional mapping

suppose if inside Laptop table we want foreign key(id) of Student table then inside Student class top of the Laptop field we will mention @OneToOne(mappedBy = "student") , and inside Student table there will not be any foreign key of Laptop Table

now suppose if we want to change the name of foreign key let’s say student\_id then inside Laptop class Top of the Student field we will mention @JoinColumn(name = "student\_id") followed by @OneToOne

now we will do CRUD operation

so we will use setter and getter for to set the all the fields of Student object and Laptop object so for storing the student object we will create StudentRepository similarly for Laptop but we can save the laptop in another way also

inside Student class on the top of Laptop field we will mention

@OneToOne(mappedBy = "student",cascade = CascadeType.ALL)

CascadeType.ALL means suppose if we add Student object then Laptop object will automatically get added ,same for Read,update,delete

2. One to many mapping

Suppose we have two entity Student and Address and one student has multiple address then this is one to many mapping and from other side it’s Many to One mapping

address

student

address

address

here inside Student class we will mention List<Address> field and top of that we will mention @OneToMany so there will another table will be created that will contain foreign key of address table and student table , till now it’s unidirectional

to make bidirectional on the both class we will mention the field of each class and to avoid this extra creation of table we inside address class we can keep foreign key of student table and mention @ManyToOne and @JoinColumn(name = "student\_id") inside Address class and @OneToMany(mappedBy = "student",cascade = CascadeType.ALL) inside Student class

now we will do CRUD operation and get the result

3.Many to Many Mapping

product

Category

electronics Iphone 14 pro max

Samsung s12

mobile phones Tv

another example

User

Role

ADMIN Guchhu

Rohan

NORMAL Scooby

now suppose we create two class Category and Product and both contains List of each other object now if inside Product class on the top of List<Category> field we write @ManyToMany then it’ll be uni-directional , to make it bi-directional we will mention @ManyToMany inside both the class

now since we are using List on the both side so two extra table will be created to manage these two List from the both side

to avoid creation of two extra table and creating only one extra table let’s say inside Category class on the top of the List< Product> we mention @ManyToMany(mappedBy = "categories",cascade = CascadeType.ALL)

So now only one extra table will be created for managing the categories that will contain the primary field of both the class

Now we will do CRUD operation

So now we will create the repository for both the class Product and Category

Now we will add the products inside category and vice versa but if don’t want add in the vice versa way then change the mapping by properties from mappedBy = "categories" to mappedBy = " products ",

If we are taking foreign key of some table suppose categories is using products primary key as foreign key then we should save categories with products to the categoryRepo