Databases.

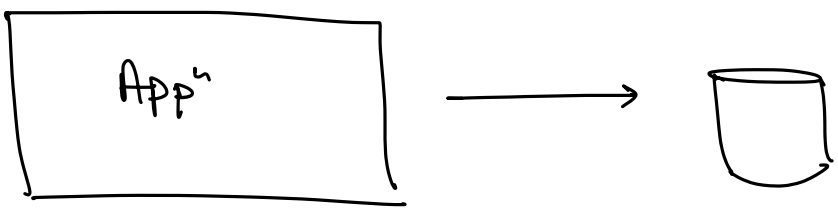
- Introduction to JPA
- Repository Pattern.
- UUID's.

ProductService.

⇒ As of now we've built just a proxy service to fetch data from FakeStore.

⇒ Now we are going to build our own service using our own Database.

#



→ Connection Connect to db--> give url, give port#, etc.

Define cardinality: for many to many -> you have to create a mapping table

→ Schema ⇒ Creating tables & relation b/w the tables

→ All CRUD operations.

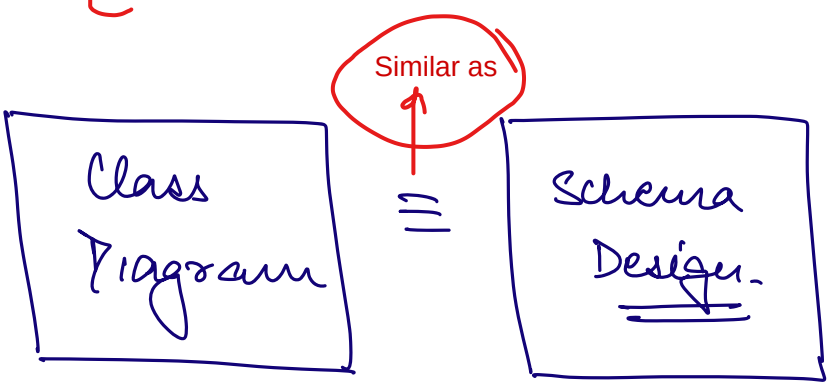
```
createNewProduct(Product p){
    Database db = new Database(url, username, password);
    db.connect();
    Query q = insert into products value(-----)
    db.execute(q);
}
```

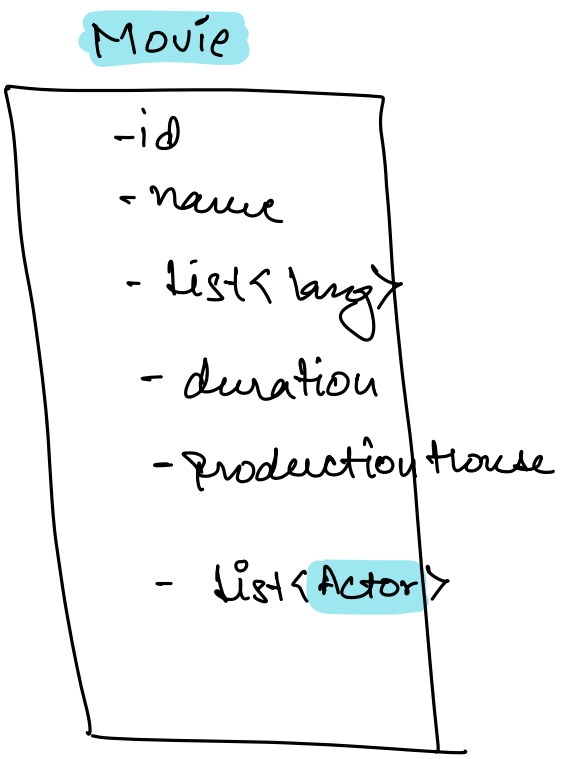
Create a db

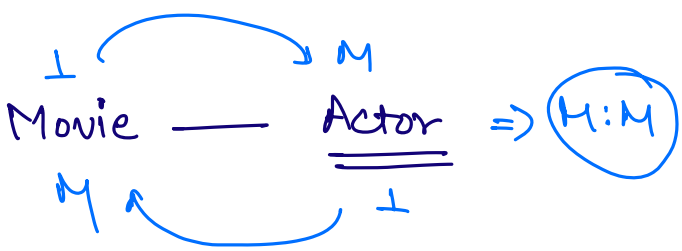
Inserting values

executing the queries

All these processes takes a lot of time. Even CRUD operation will take time and it is not wise to do such processes manually







ORM converts classes into tables keeping the relationship in the table.

Apply the rules whatever you have defined in the class diagram into tables of the dbs.

You just have to define some methods in ORM libraries. Eg: Remove Boilerplate code(writing select * queries again and again--> this is called redundant or boilerplate code)

⇒ ORM. ⇒ Object Relational Mapping

ORMs are the libraries which creates the mapping between your object(classes) and relation(tables)

Provides us an easy way to work with Databases based on the Models that are there in our Codebase.

1. Automatically creates corresponding tables for Models
2. Automatically perform CRUD operations.

```
productRepository.findById(10)
```

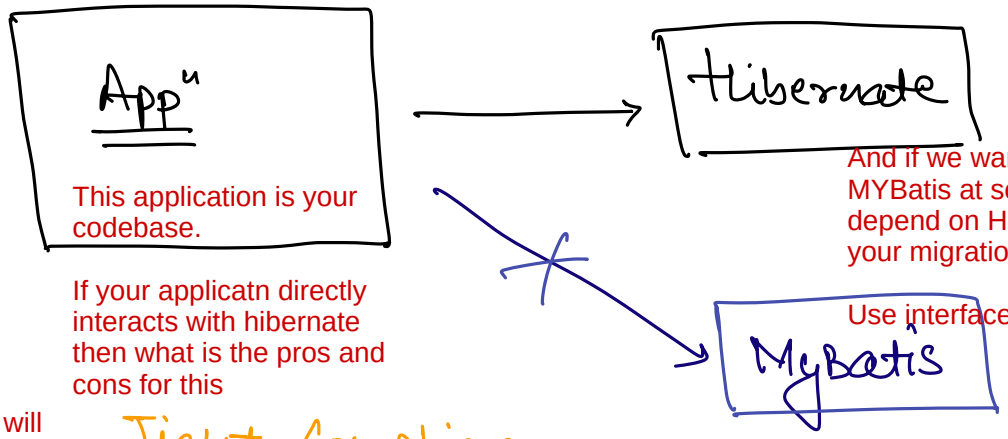
these libraries are provided by ORM and is an automatic way of performing CRUD operations

↓

```
select * from
products where id = 10;
```

ORM's.

1) Hibernate
2) MyBatis
3) JOOQ



This application is your codebase.

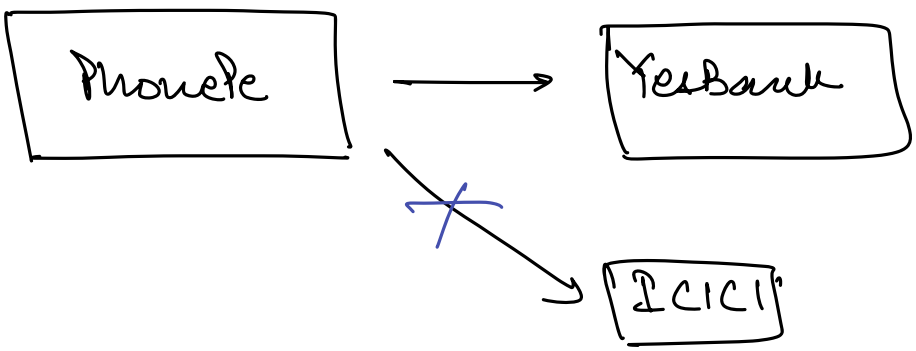
If your applicatn directly interacts with hibernate then what is the pros and cons for this

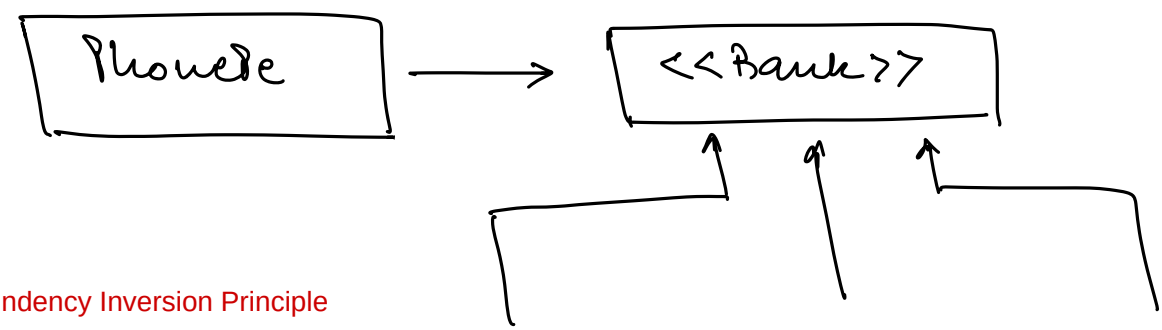
Our code will become tightly coupled

Tight Coupling

And if we want to migrate your code to MYBatis at some of time. But if you are depend on Hibernate or 3rd party --> your migration is not easy.

Use interface





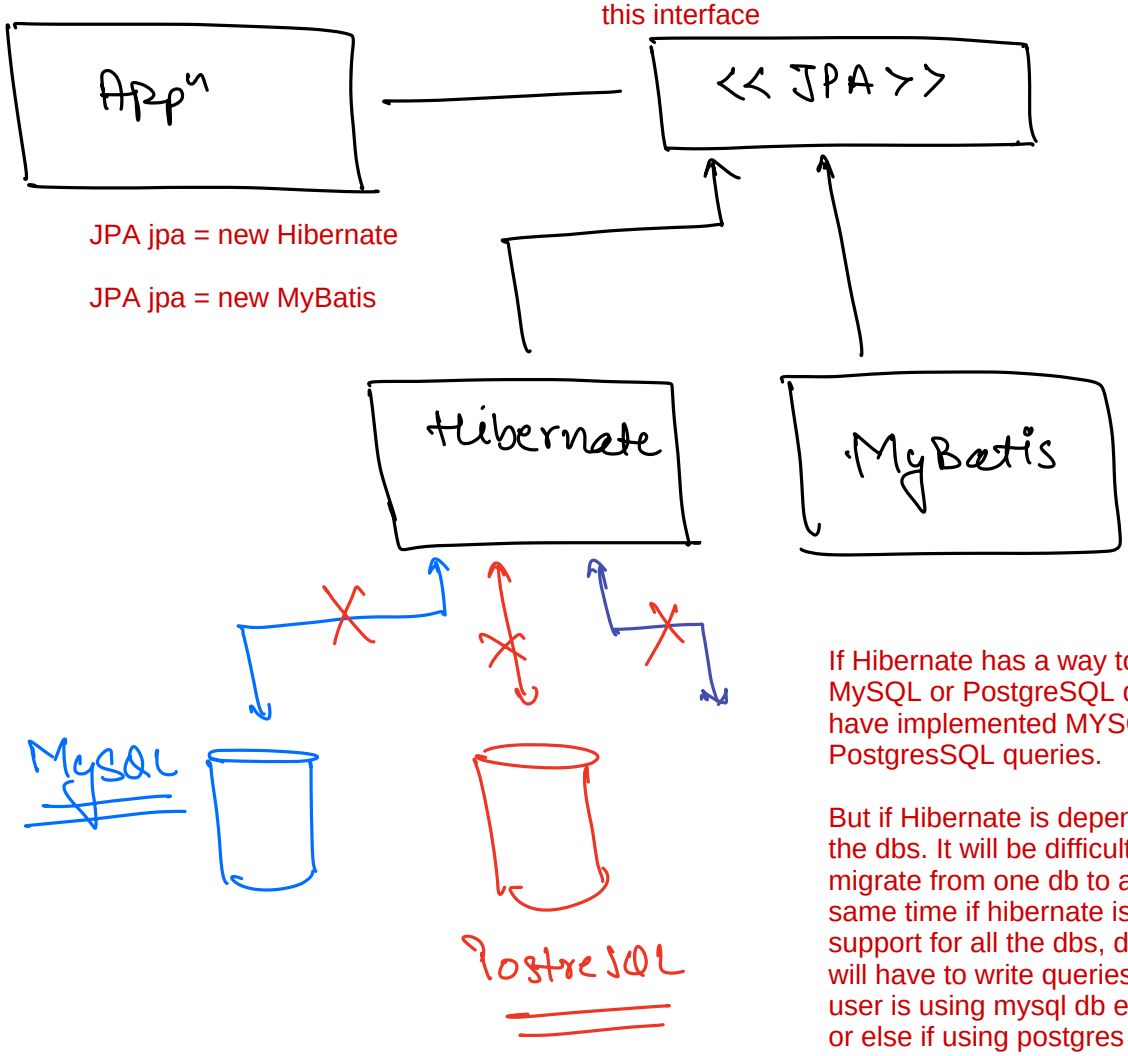
Dependency Inversion Principle

If your appl wantst o talk t hibernate -> get an interface in between -> This interface is called JPA interface.
Java Persistance API

=> Program to interface, not to implementation.

JPA ⇒ JAVA PERSISTENCE API

By default Java has an interface that can be implemented by different ORM Libraries.



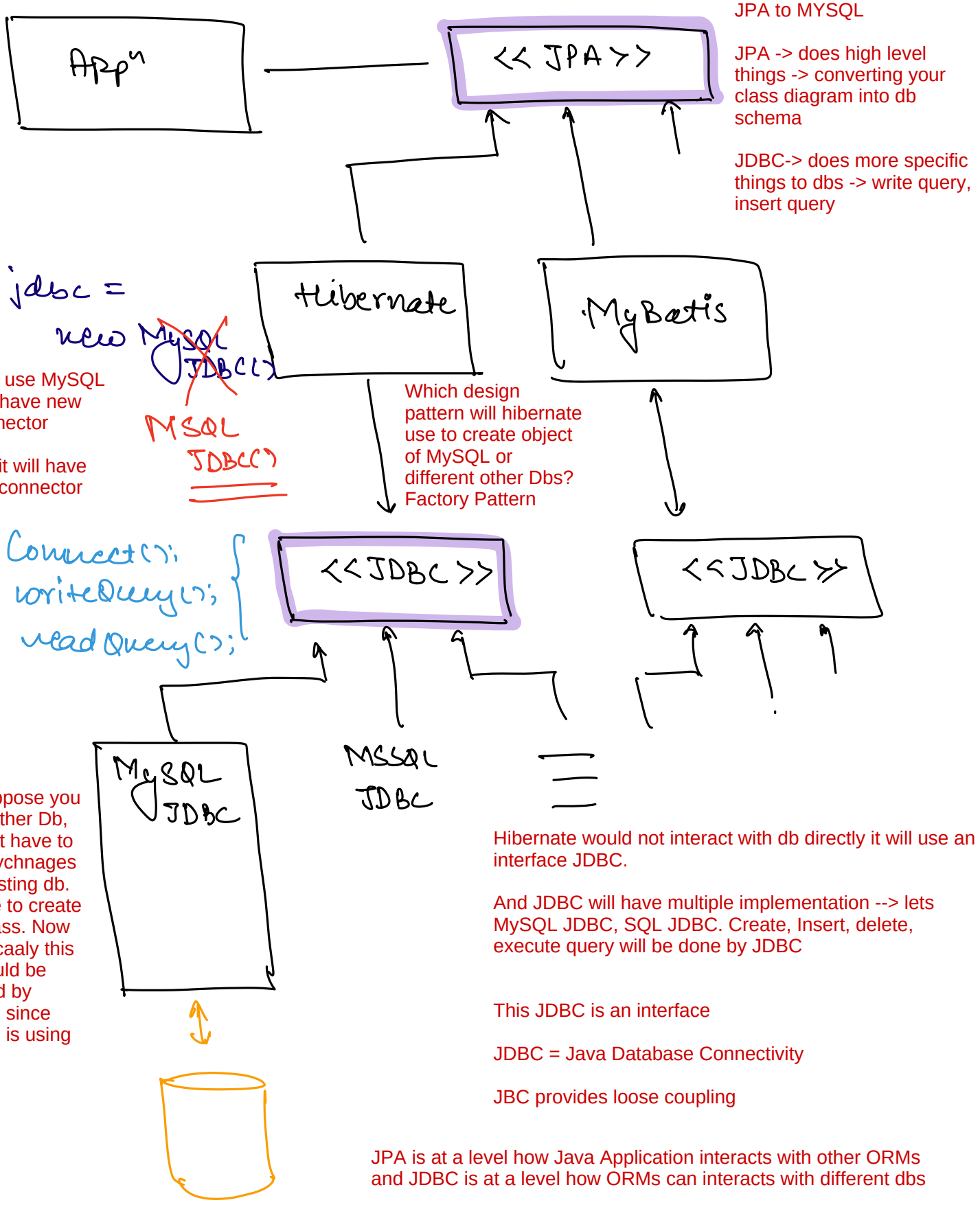
If Hibernate has a way to connect to MySQL or PostgreSQL queries and it may have implemented MYSQL and PostgresSQL queries.

But if Hibernate is depending on one of the dbs. It will be difficult for Hibernate to migrate from one db to another. Or at the same time if hibernate is providing support for all the dbs, don't you think it will have to write queries for my sql--> if user is using mysql db execute this query or else if using postgres db execute this query.

Hibernate would not interact with db directly it will use a JDBC.

And JDBC will have multiple implementation --> lets MySQL JDBC, SQL JDBC.

This JDBC is an interface



Down the hierarchy things are becoming specific from JPA to MYSQL

JPA -> does high level things -> converting your class diagram into db schema

JDBC-> does more specific things to dbs -> write query, insert query

If we want to use MySQL JDBC, it will have new MYSQL connector

For MSSQL it will have new MSSQl connector

Which design pattern will hibernate use to create object of MySQL or different other Dbs? Factory Pattern

If lets suppose you have another Db, you donot have to make anychnages in the existing db. You have to create a new class. Now automaticaaly this class would be supported by hibernate since hibernate is using JDBC

Hibernate would not interact with db directly it will use an interface JDBC.

And JDBC will have multiple implementation --> lets MySQL JDBC, SQL JDBC. Create, Insert, delete, execute query will be done by JDBC

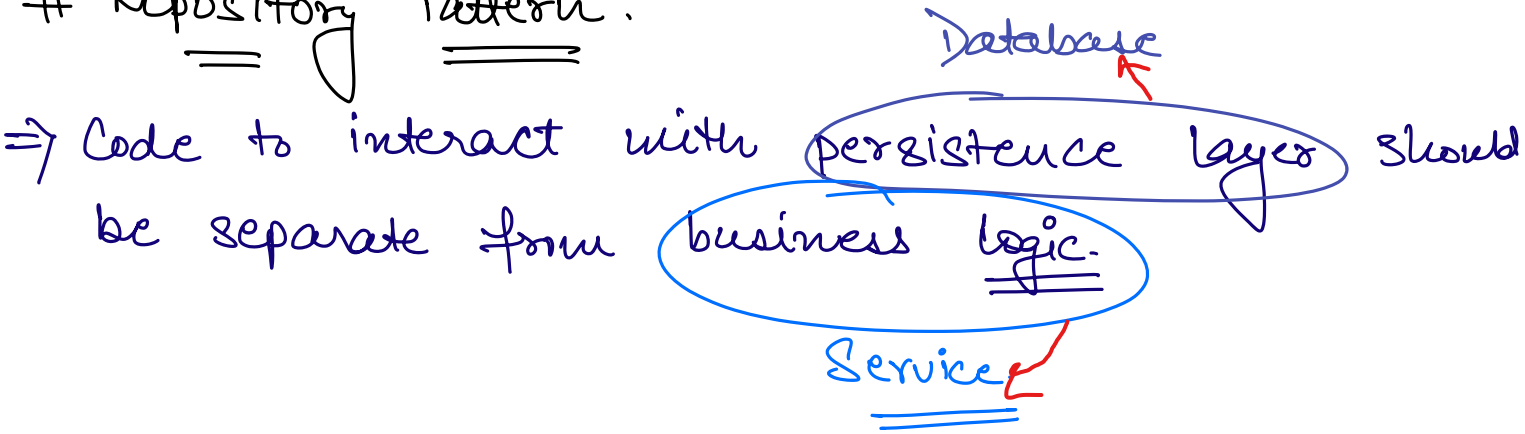
This JDBC is an interface

JDBC = Java Database Connectivity

JBC provides loose coupling

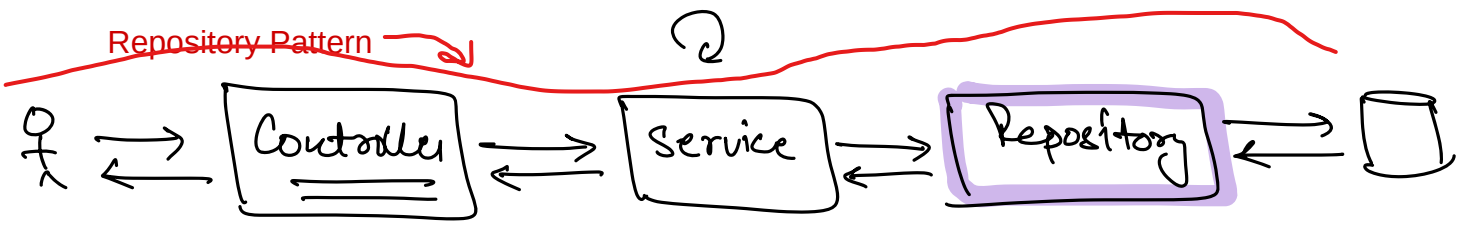
JPA is at a level how Java Application interacts with other ORMs and JDBC is at a level how ORMs can interacts with different dbs

Repository Pattern.



⇒ Code to interact with persistence layer should be separate from business logic.

⇒ Code to interact with Database should it be present in service layer



Single Responsibility Principle → SRP ✗ Tight Coupling

```
ProductService {

    MySQLDb db = ————————;

    Save() {

        db.execute(————);

    }
}
```

Also this MySQL db is tightly coupled with Product Service

Why we are using this repository layer in between. If we remove it, then we have issues like tight coupling, open close principle, extensibility to add new feature, maintainbility to maintain new code

If we directly talk to db through Service layer and donot have a repository layer.

Now lets say you want to save a product , we use d.execute query. Now lets say tomorrow you want to migrate from MySQL to some other thing or want to support bulk add, As of now you are adding products one after the other.
Now you want to overwrite this method save to add multiple products at once. Doing this ProductService will violate SRP (Single Responsibility principle). Coz SRP states every class, every method code should have single reason to change. But here ProductService class will have multiple reasons to change. If you want to add some business logic-> you need code update. If you want to change any db logic -> again you need code update

```
ProductRepository {
    Database db = ---------- ;

    save() {
        ==
    }
    read() {
        ==
    }
    update() {
        ==
    }
}

ProductService {
    ProductRepository pr ;

    ProductService (ProductRepository pr) {
        this.pr = pr;
    }

    Product p = pr.findById(10);
}
```

Hence we use Product repository. this will talk to db. It will have save, read, update. And this will only be responsible for talking to db

And ProductService will be using your Product Repository

We will be injecting this Product repository in Product Service
Inside Product Repository, accept the object of Product Service

If lets suppose product repository is giving me a method findById--> I have to just call this method.

In Product Service I have to just call the method using right reference.

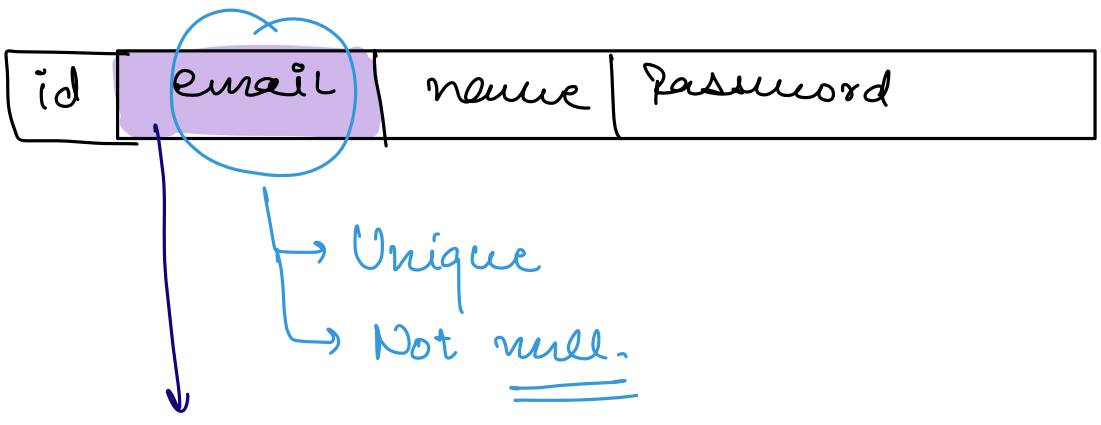
Everything in Computer Science is a trade off.

UUID's. (Universally Unique Id).

=> Every table in the database should have a PK.

=> PK is required to uniquely identify a record in database.

users



(1) Email is a user attribute & it can change.

(2) String attr

Joins on string are costly

↳ String comparisons are costly.

③ By default index is created on primary key.

↳ More space.

↳ Write operations become costly.

Index is created on primary key and ~~this index table~~ will be stored on the hardisk. If you creatr string as index it will take a lot of space

Plus due to index your write operation become costly. Coz if you delete or update the data, you have to do in the index table as well

⇒ An extra id column as a PK

Hence an extra id column is added

1) int

↳ 4B ⇒ 32 Bits.

$2^{32} \approx 2 \times 10^9$

⇒ 2Billion

2) BigInt | long ⇒ Most frequently used id.

↳ 8B

↳ 64 bits

↳ $\approx 10^{18}$

⇒ Auto increment.

Long can be autoincremented but But there is a problem with int and long is these are predictable specially in user table and it will be slow when increment on db

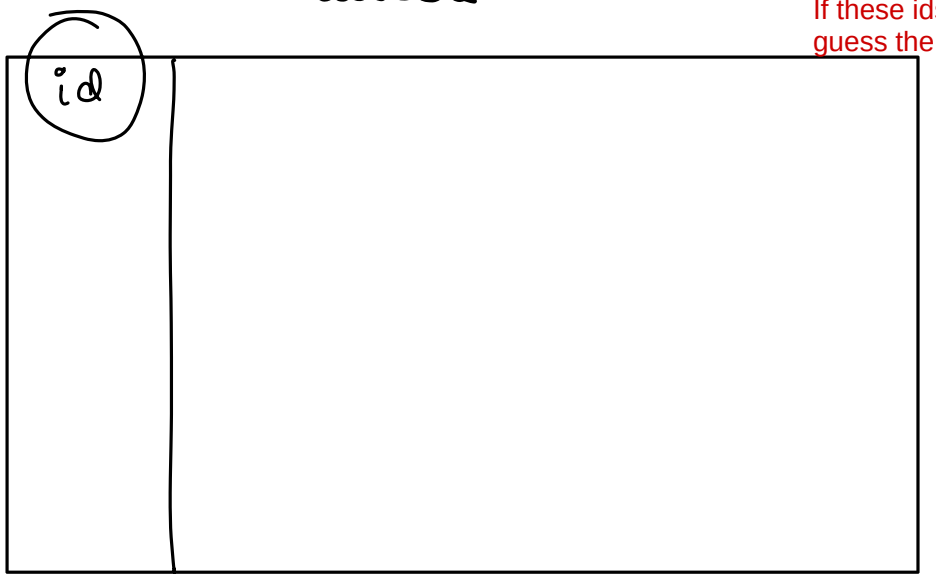
twitter.com | tweets | id

API -> tweets

Now lets suppose Google employee wants this twitter API to analyze some data so twitter will be charging hell lot of money from Google. These type of APIs are not free APIs.

If these ids are predictable they can guess the ids.

tweets.



⇒ If tweeter is maintaining auto increment id's then it will be very easy for anyone to predict tweet id's & get tweets.

amazon.com | products | id

List<Product> products

while (true) {

products.add(restTemplate.get(⟵));

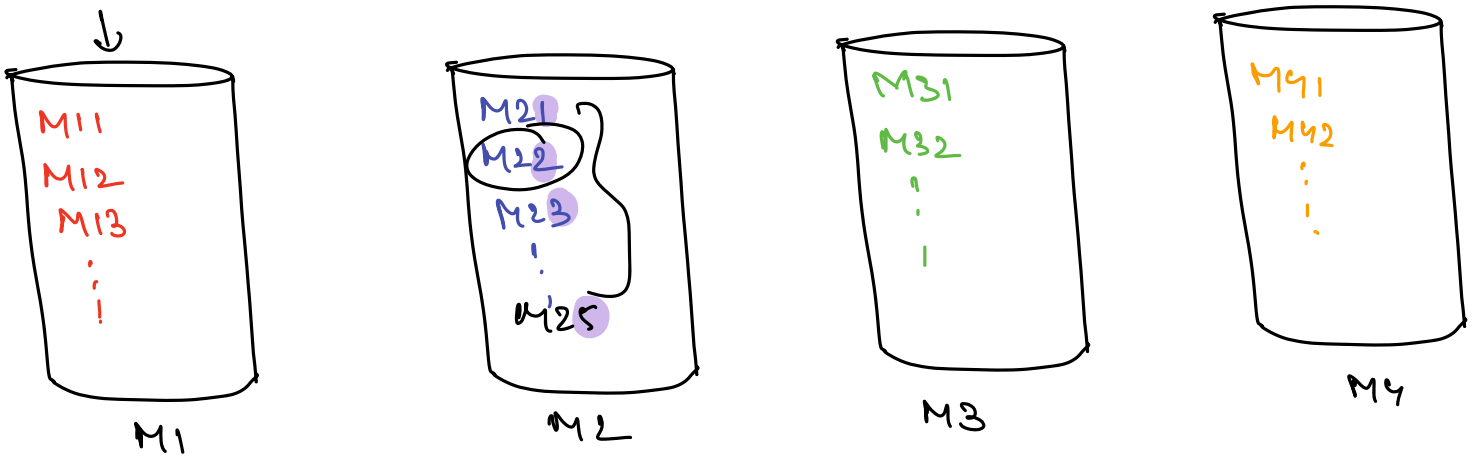
id++;

}

Scaler.com | users | (1) X

(1) If there are public APIs to fetch data the auto increment ids won't good.

(2) If the database is distributed (Sharded or distributed) across multiple machines the auto increment won't work.



Id ⇒ machine_id + auto inc

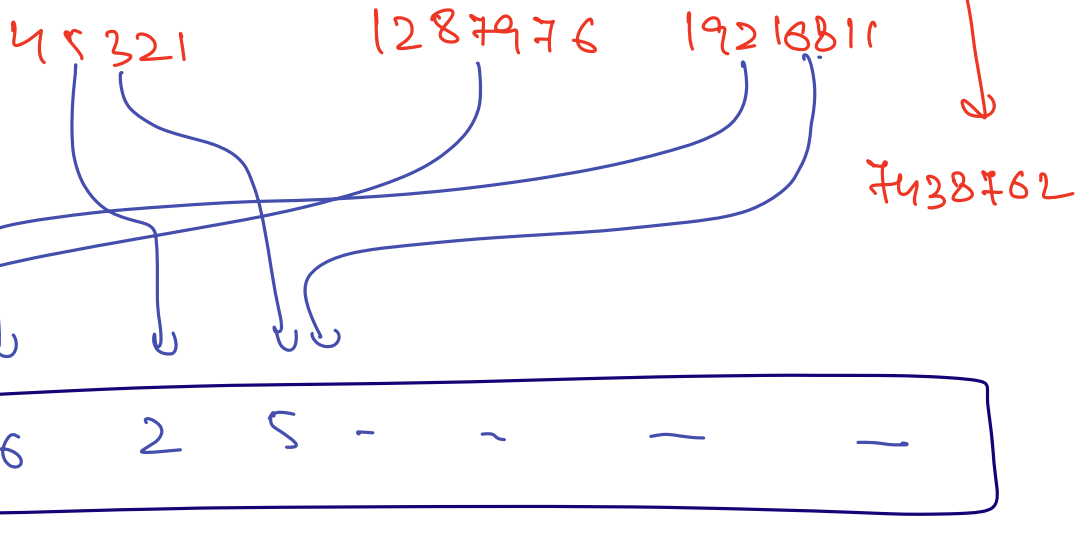
If Id is a combination of machine_id and autoincrement feature --> still it is not good

⇒ Instead of auto increment, we need some randomness.

We need to have randomness -> here the role of UUIDs comes into picture

CommitId = fun(m/c id + timestamp + i/p + user_id + ...)

UUID. => fun(machine-id , user id , ip , timestamp +---)



Completely Random

128 bit Number.

$2^{128} = 4 \times 10^{36}$

range of UUID

UUID is a 128 bit number stored in hexadecimal format

10101111001100110 - - - -

128 bits.

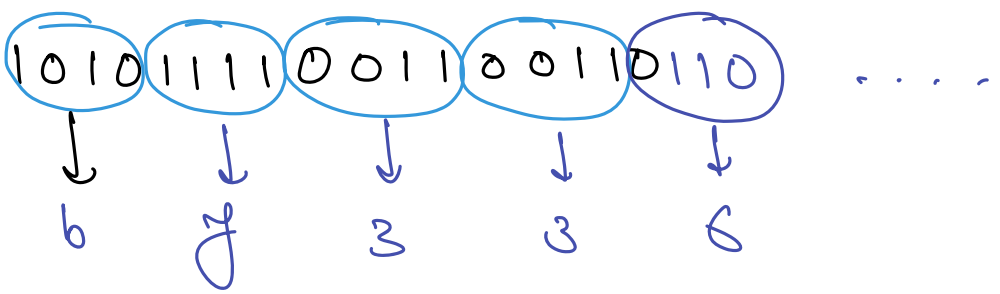
=> Hexadecimal. (16 base)

x x x x

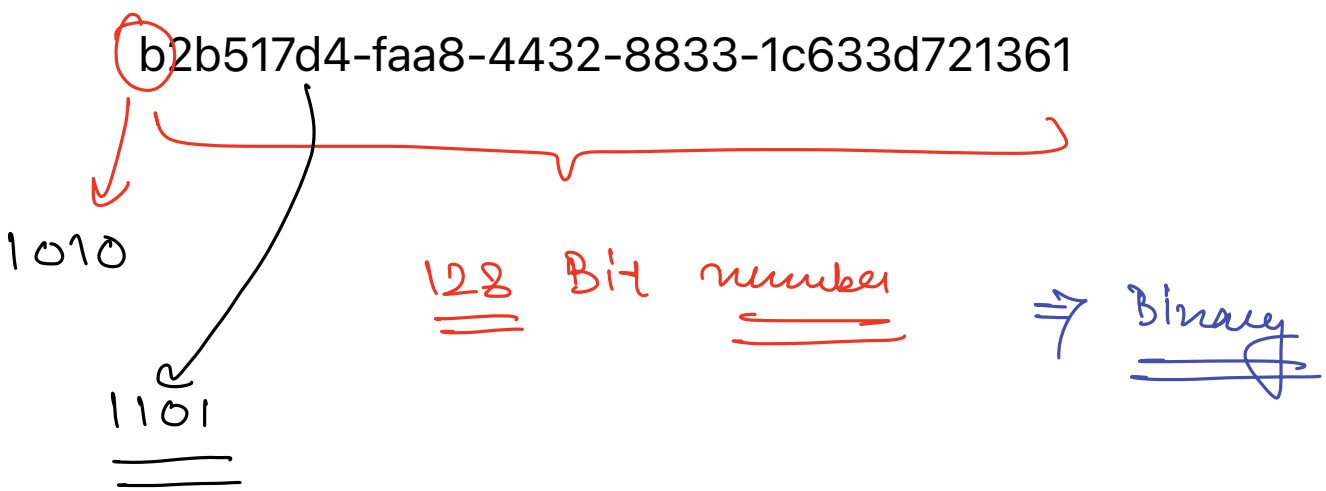
0000
0001
!
!
!
1111

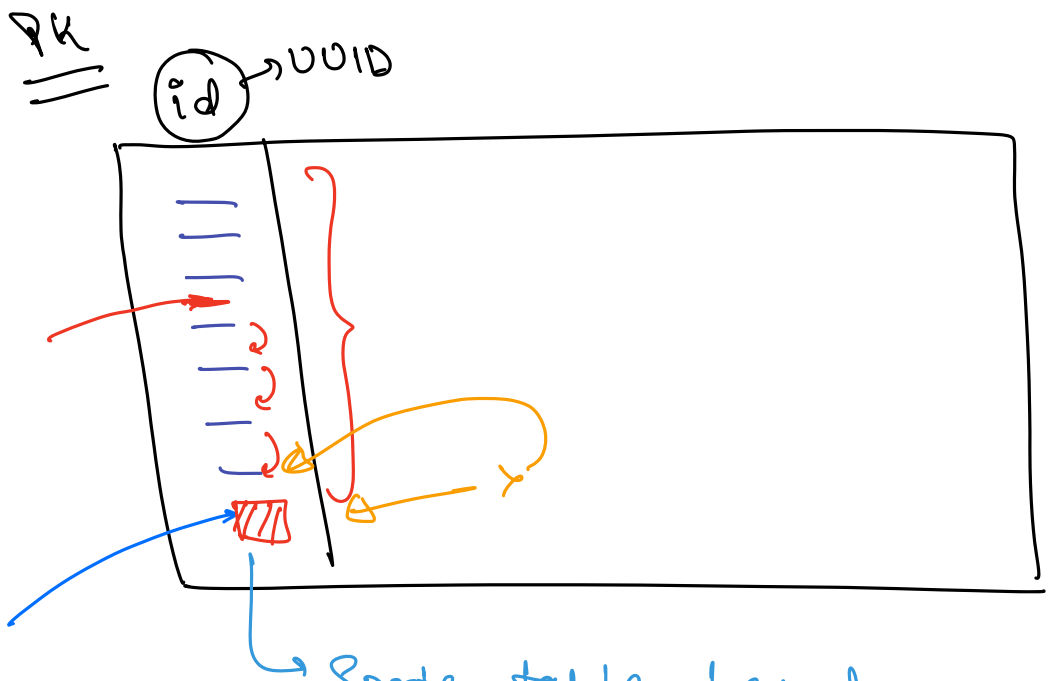
0000 → 0
0001 → 1
0010 → 2
0011 → 3
0100 → 4
0101 → 5
0110 → 6
0111 → 7
1000 → 8
1001 → 9
1010 → a
1011 → b
1100 → c
1101 → d
1110 → e
1111 → f

$\frac{128}{4} \rightarrow (32)$



Sample UUID --> 128/4 = 32 --> it will have 32 digits





Now if UUID is a PK, and since in PK the data is sorted.

If we add a new UUID, it will insert somewhere in between the data based on its value and the below data needs to be shifted.

This is not a good approach. So we make sure that UUID newly created is greater than the last UUID in the table.

In this way we can maintain randomness of UUID as well solve the PK issue

⇒ Somehow, along with randomness if UUID can maintain that the new uuid will be greater than the previous uuid. ⇒ ✓

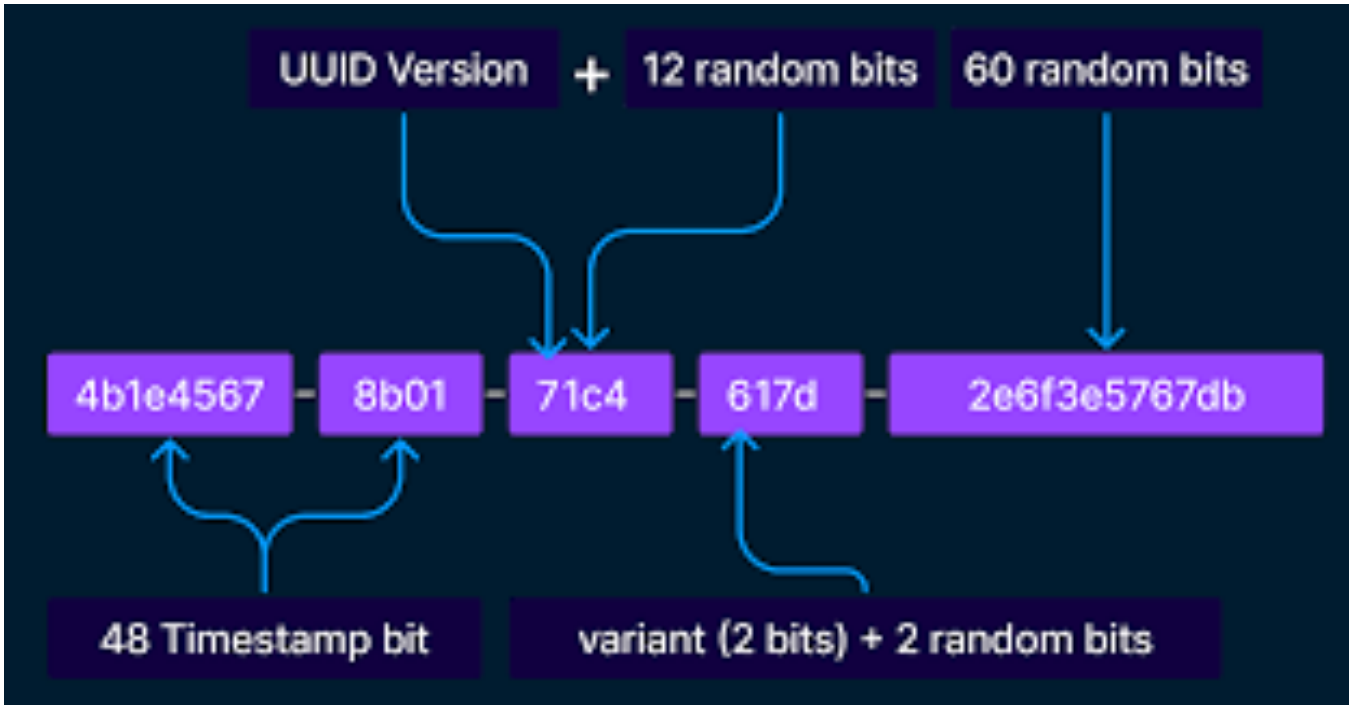
⇒ timestamp

Hence the latest version of UUID which is V7 uses timestamp to maintain the incrementally feature

⇒ UUID V7 ⇒ uses timestamp

No. of milliseconds passed since 01/01/1970

⇒ UUID


UUID Version
+
12 random bits
60 random bits
4b1e4567
8b01
71c4
617d
2e6f3e5767db
48 Timestamp bit
variant (2 bits) + 2 random bits