Databases.

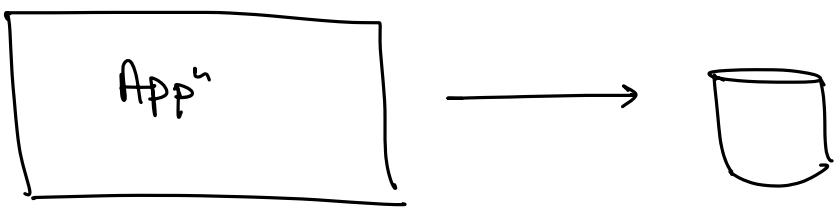
- Introduction to JPA
- Repository Pattern.
- UUID's.

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

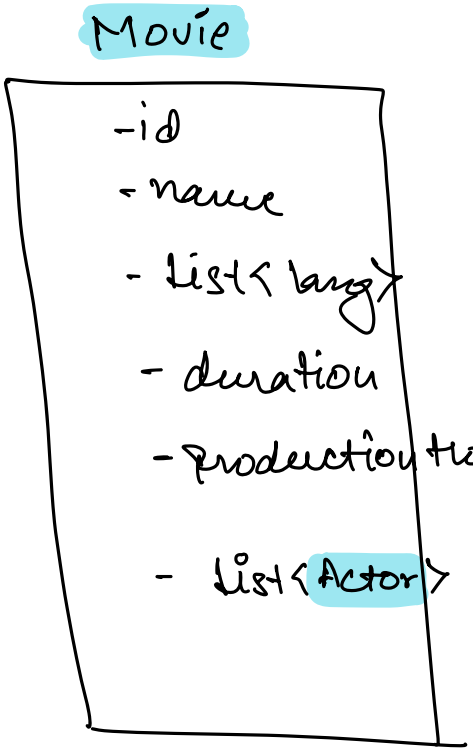
→ Schema ⇒ Creating tables & relation b/w the tables

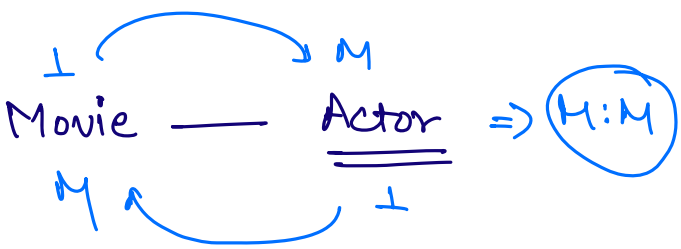
→ All CRUD operations.

```
createNewProduct(Product p){
    Database db = new Database(url, username,
                               password);

    db.connect();

    Query q = insert into products
              value (- - - - - -)

    db.execute(q);
}
```

#

Class Diagram	≡	Schema Design





⇒ ORM. ⇒ Object Relational Mapping

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.

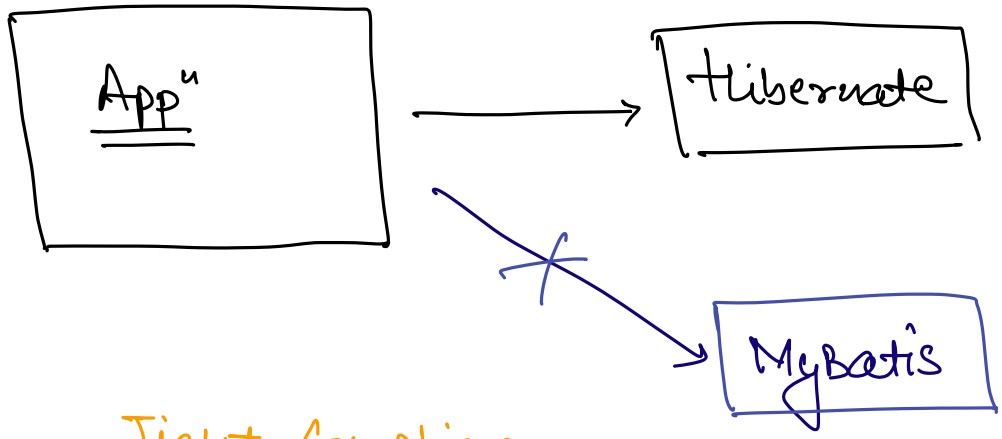
```
productRepository.findById(10)
```

↓

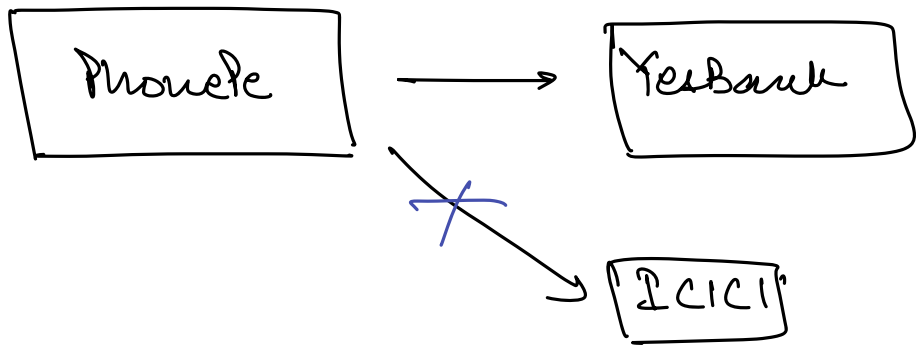
```
select * from
products where id = 10;
```

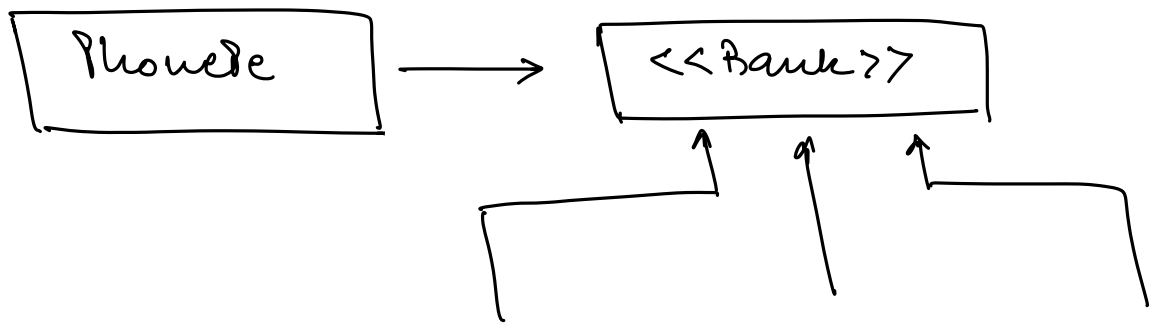
ORM's.

1) Hibernate
2) MyBatis
3) JOOQ



Tight Coupling

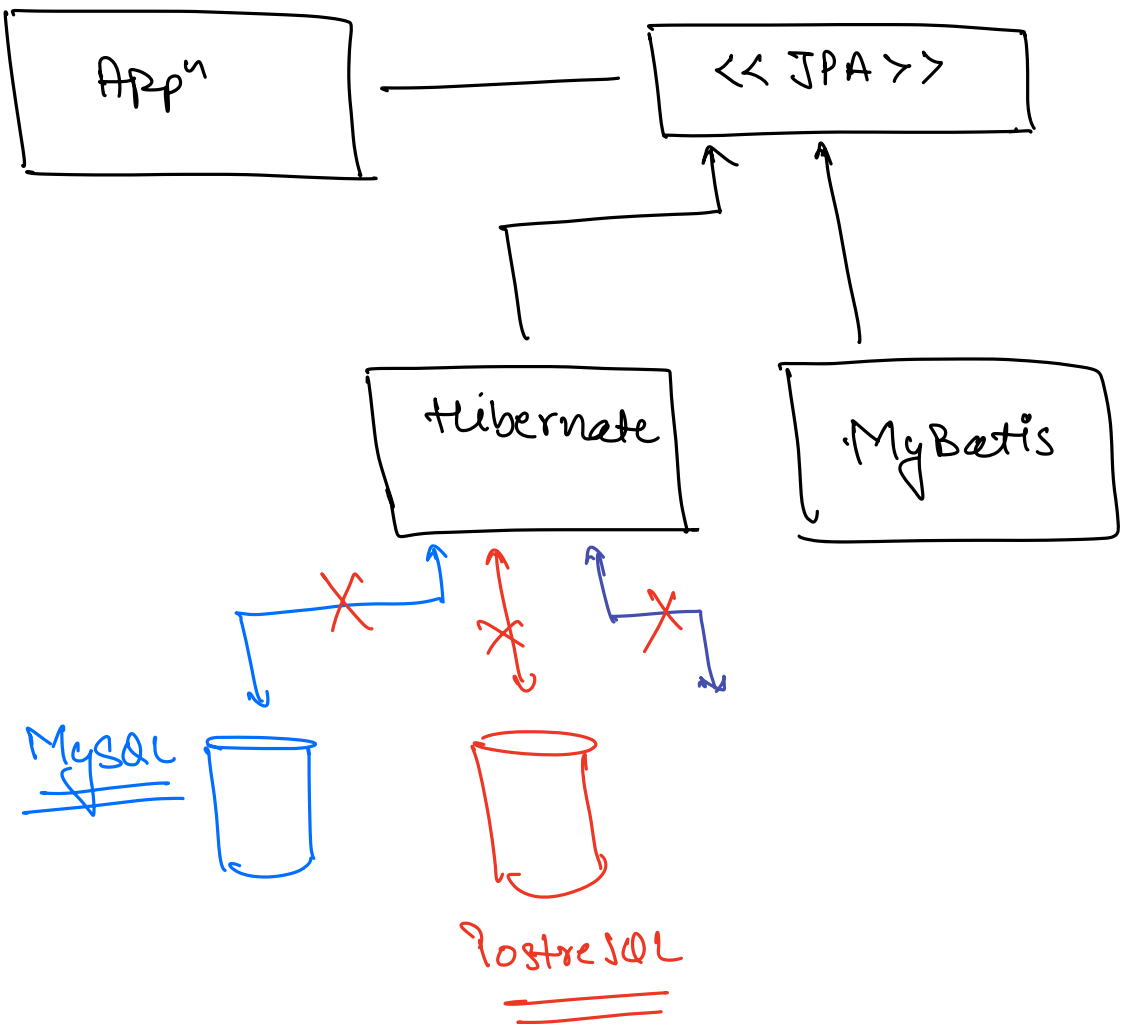


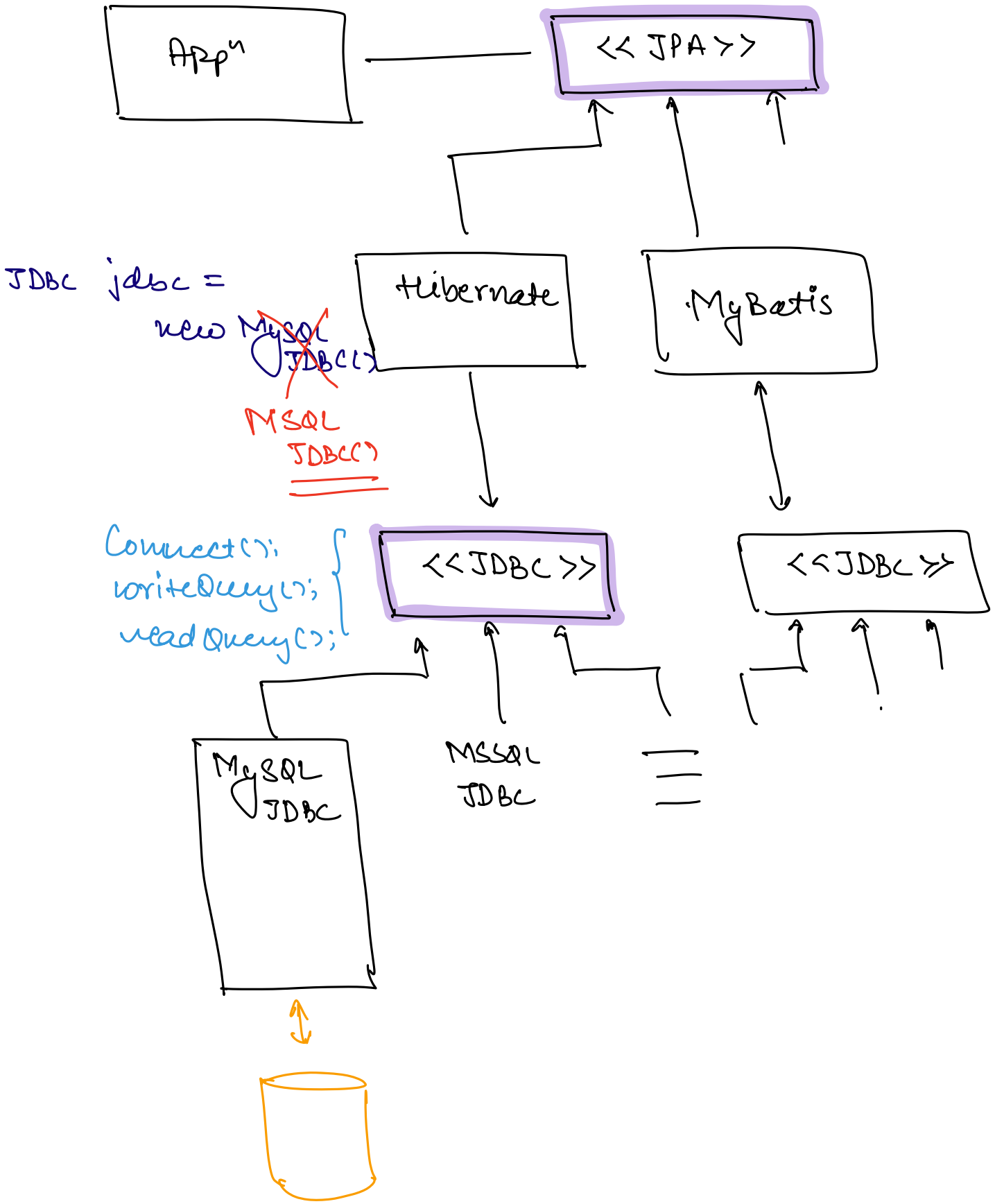


⇒ Program to interface, not to implementation.

JPA ⇒ JAVA PERSISTENCE API

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




App^n
<< JPA >>
Hibernate
MyBatis
JDBC jdbc =
new MySQL JDBC();
MSQL JDBC()
Connect();
writeQuery();
readQuery();
<<JDBC>>
<<JDBC>>
MySQL JDBC
MSSQL JDBC

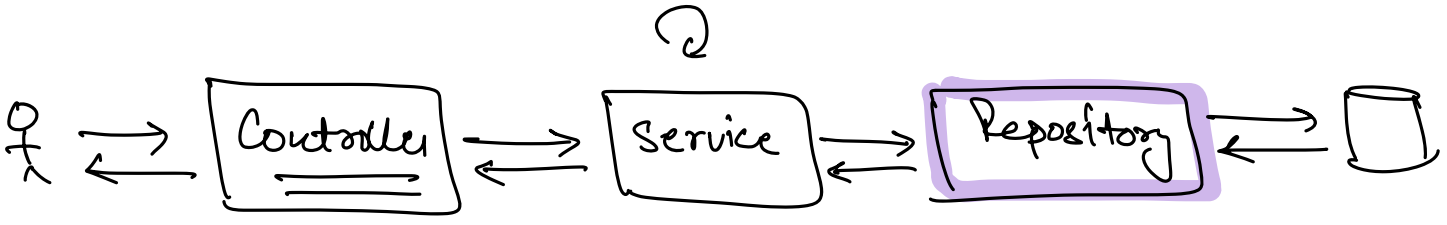
Repository Pattern.

Database

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

Service.

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



→ SRP ✗ Tight Coupling

```
ProductService {

    MySQLDb db = ————————;

    Save() {

        db.execute(————);

    }

}
```

```
ProductRepository {
    Database db = ———— ;

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

ProductService {
    ProductRepository pr ;

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

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

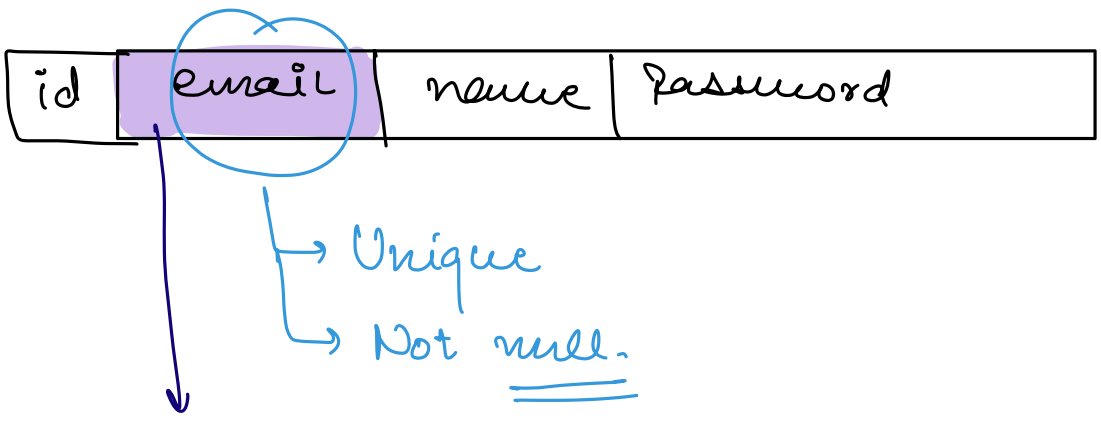
Everything in Computer Science is a trade off.

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
↳ String comparisons are costly.

③ By default index is created on primary key.

- More space.
- Write operations become costly.

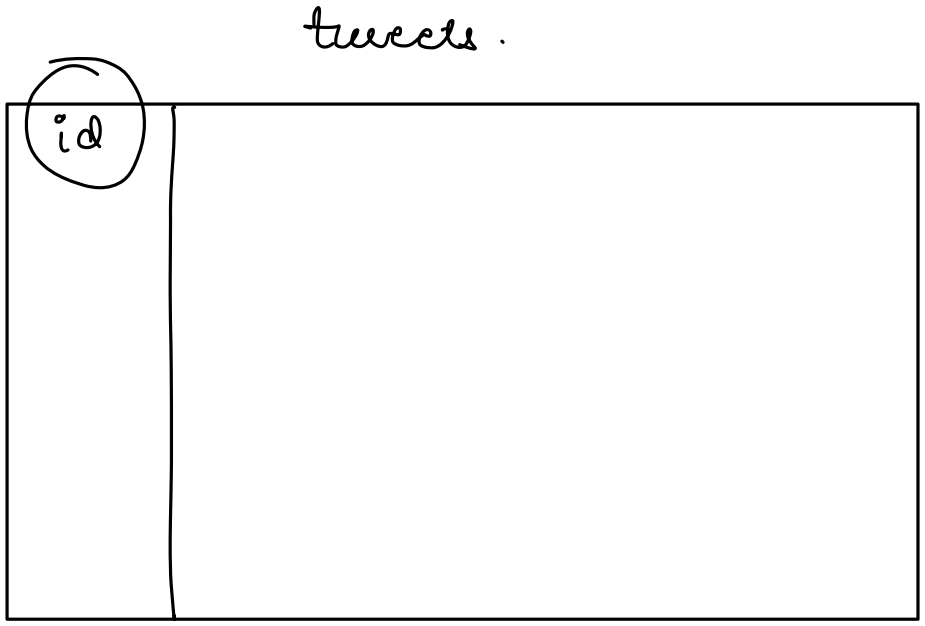
⇒ An extra id column as a PK

1) int
   - 4B ⇒ 32 Bits.
   - $2^{32} \approx 2 \times 10^9$
   - ⇒ 2 Billion

2) BigInt | long ⇒ Most frequently used id.
   - 8B
   - 64 bits
   - $\approx 10^{18}$

⇒ Auto increment.

twitter.com/tweets/id



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

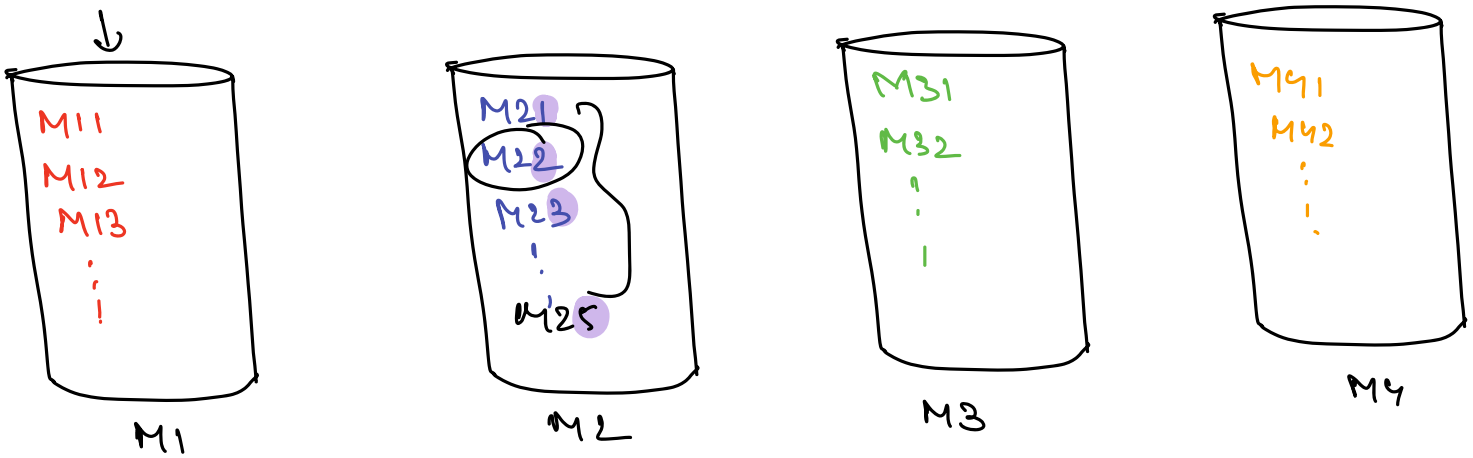
amazon.com/products/id

```
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 across multiple machines the auto increment won't work.

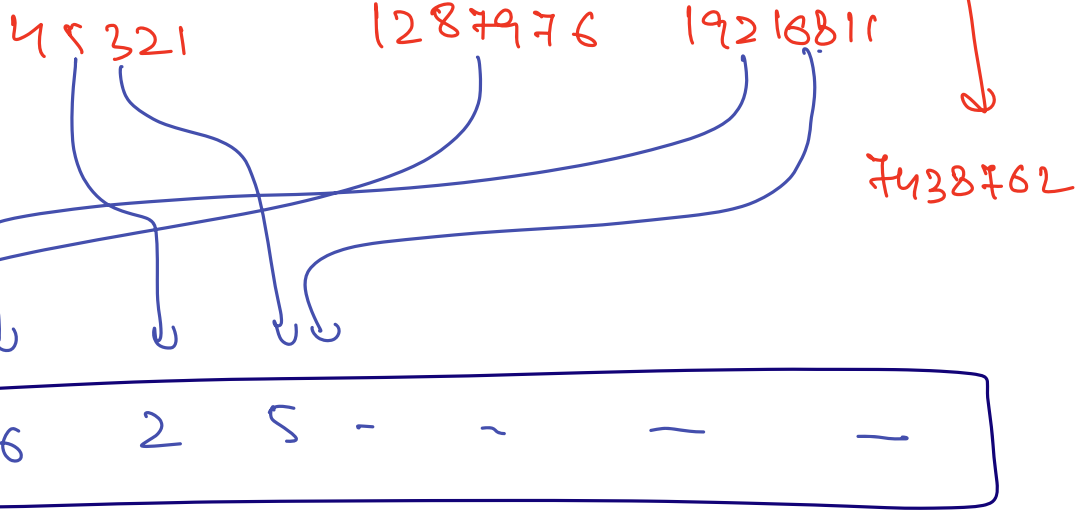


Id ⇒ machine_id + auto inc

⇒ Instead of auto increment, we need some randomness.

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

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



Completely Random

→ 128 bit Number.

→ $2^{128} = 4 \times 10^{36}$

10101111001100110 - - - -

128 bits.

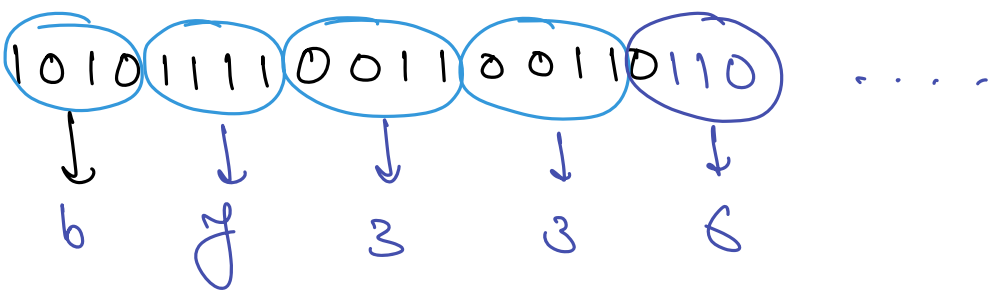
=> Hexadecimal. (16 base)

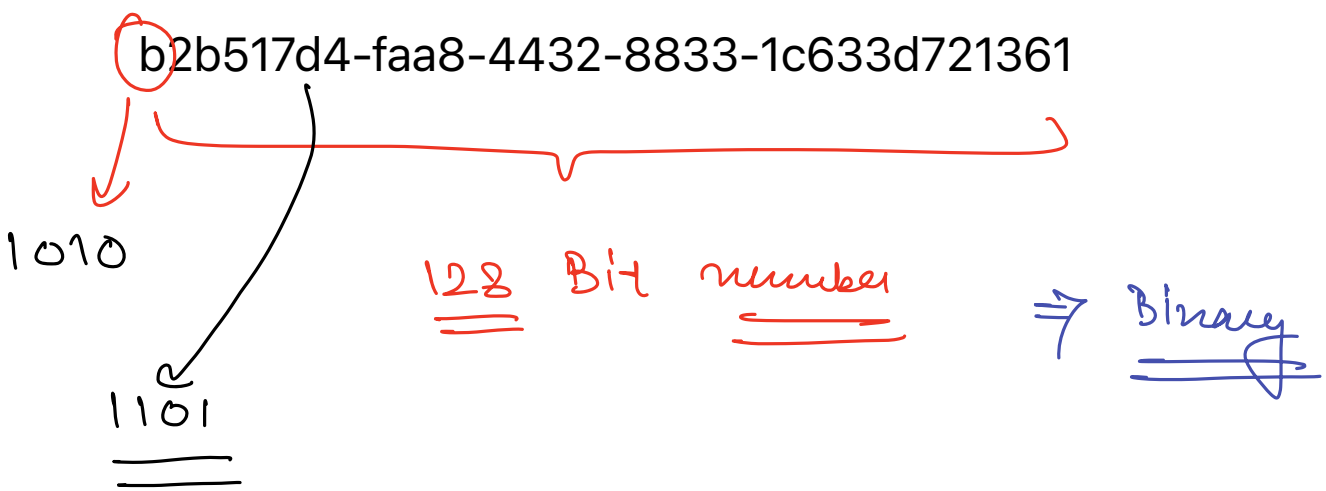
xxxx

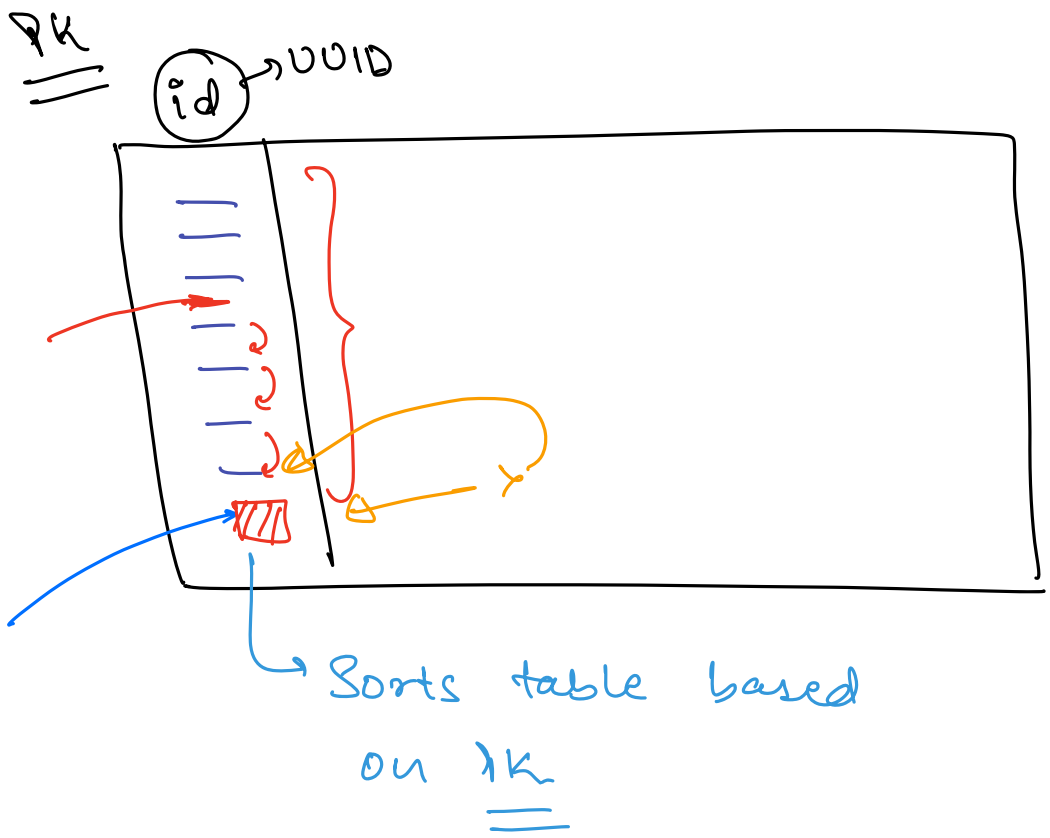
0000
0001
:
:
:
1111

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

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







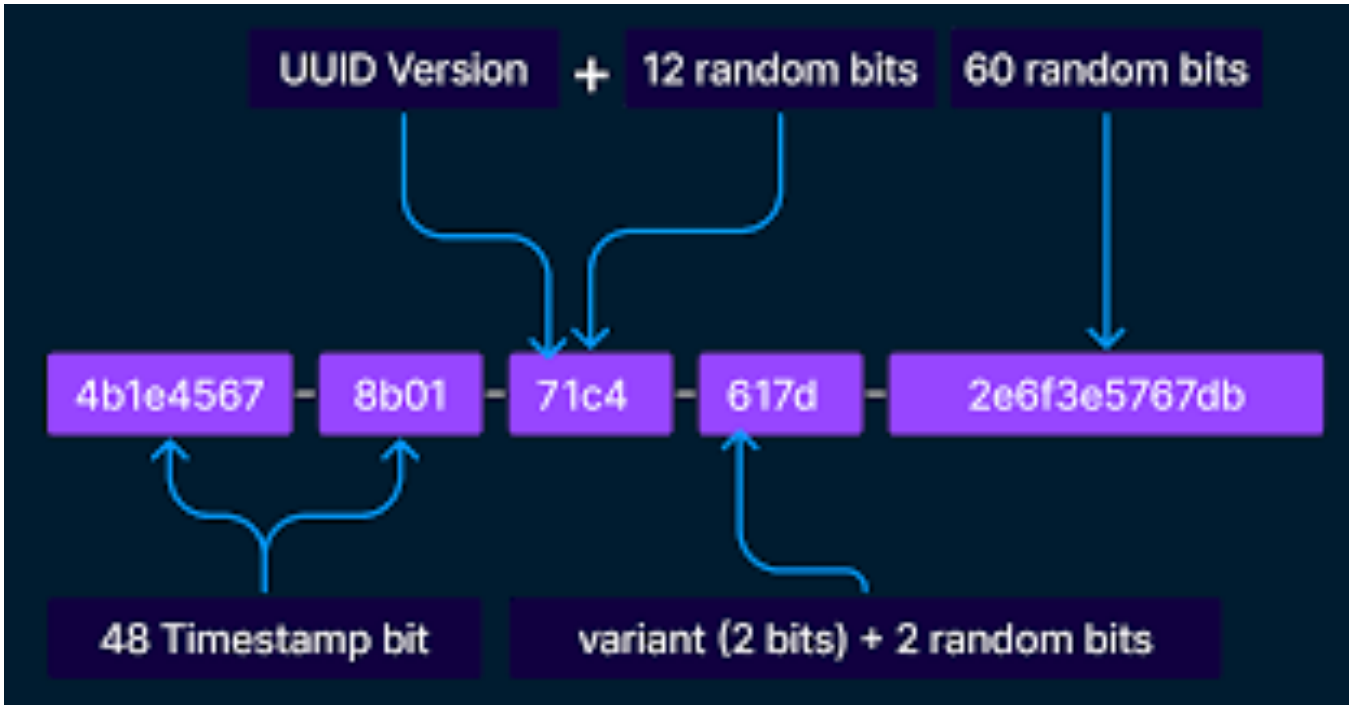
⇒ Somehow, along with randomness if UUID can maintain that the new uuid will be greater than the previous uuid. ⇒ ✓

⇒ timestamp

⇒ 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