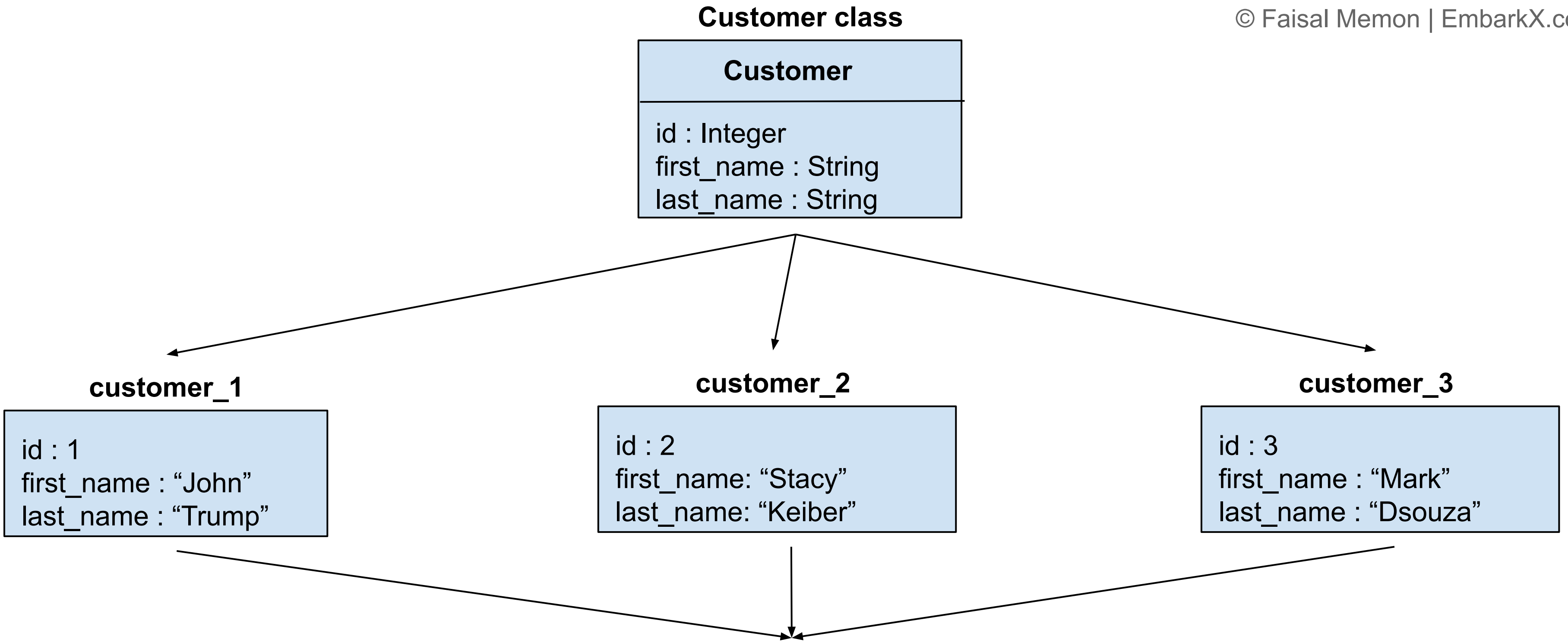
Mastering JPA and Hibernate

Faisal Memon (EmbarkX)

What is ORM?

Faisal Memon (EmbarkX)



Customer in database

id	first_name	last_name
1	John	Trump
2	Stacy	Keiber
3	Mark	Dsouza

ORM

→ Whenever there is a class, that class can be automatically converted to a table with its attributes being converted to columns

→ So now the developer does not have to write queries for table creation, it's created automatically

→ Whenever an object is created, its data can be saved in the database as row in table, this is automatically handled by ORM

ORM

→ ORM as a concept makes developers lives easier and lets developers focus on application logic rather than SQL queries

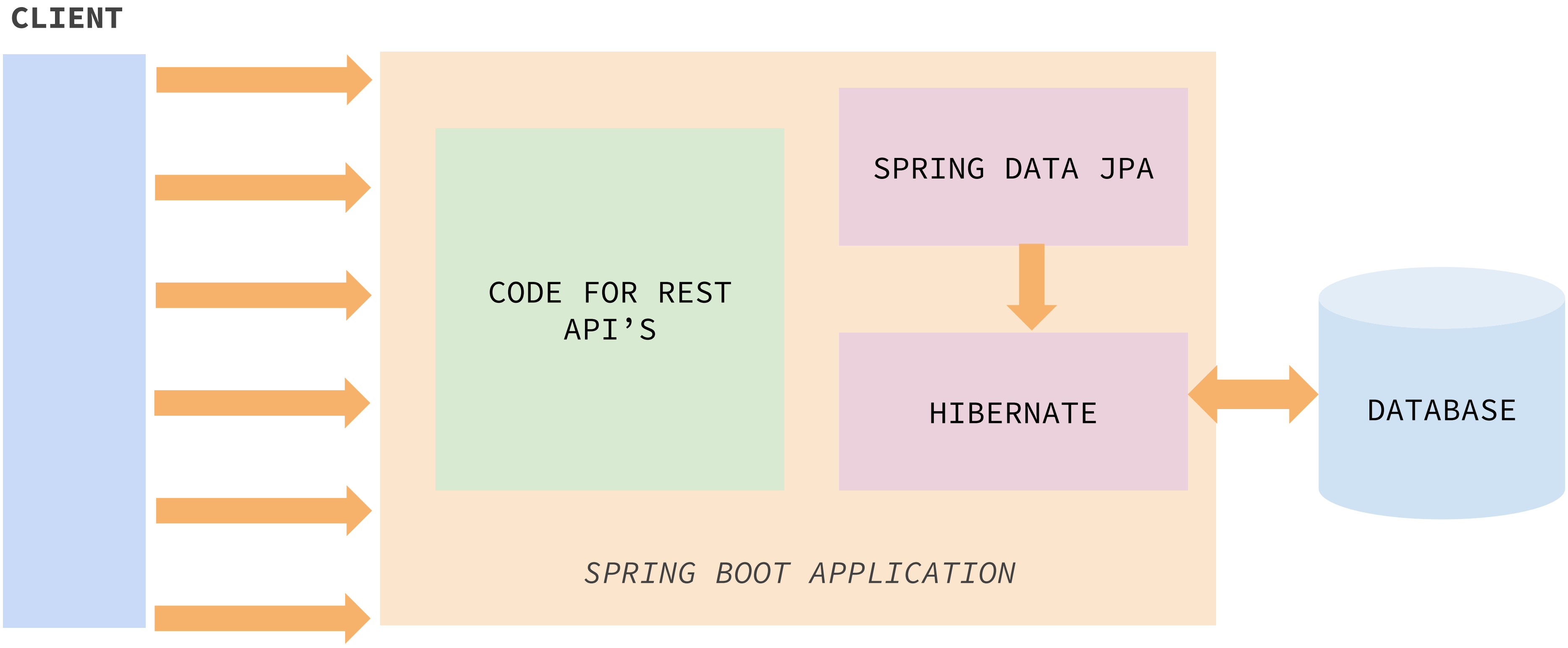
→ Because of ORM developers don't need to learn how to write SQL queries since the translation from application to SQL is handled by ORM itself

→ It's a powerful technique in programming which also minimizes mistakes since developers are not writing queries on their own

What is Hibernate

Faisal Memon (EmbarkX)

Hibernate is one of the **most popular ORM** tools in Java.


CLIENT
CODE FOR REST API'S
SPRING DATA JPA
HIBERNATE
SPRING BOOT APPLICATION
DATABASE

H2 Database

Faisal Memon (EmbarkX)

In-memory database is a database which uses memory to store data as opposed to the disk space

H2 is a **lightweight** database that runs in memory

H2 Database

→ *It's Fast*

→ *No need to install anything*

→ *Perfect for Development*

Hibernate Architecture

Faisal Memon (EmbarkX)

Core Hibernate Components

- *SessionFactory*
- *Session*
- *Transaction*

SessionFactory

→ *It reads the settings from your configuration file (hibernate.cfg.xml)*

→ *It knows which classes (like User) are linked to which tables in the database.*

→ *It creates a blueprint for how Hibernate will talk to the database.*

→ *You create it once when your app starts, and reuse it.*

Session

→ *You open a session when you want to talk to the database.*

→ *Once you're done, you close the session.*

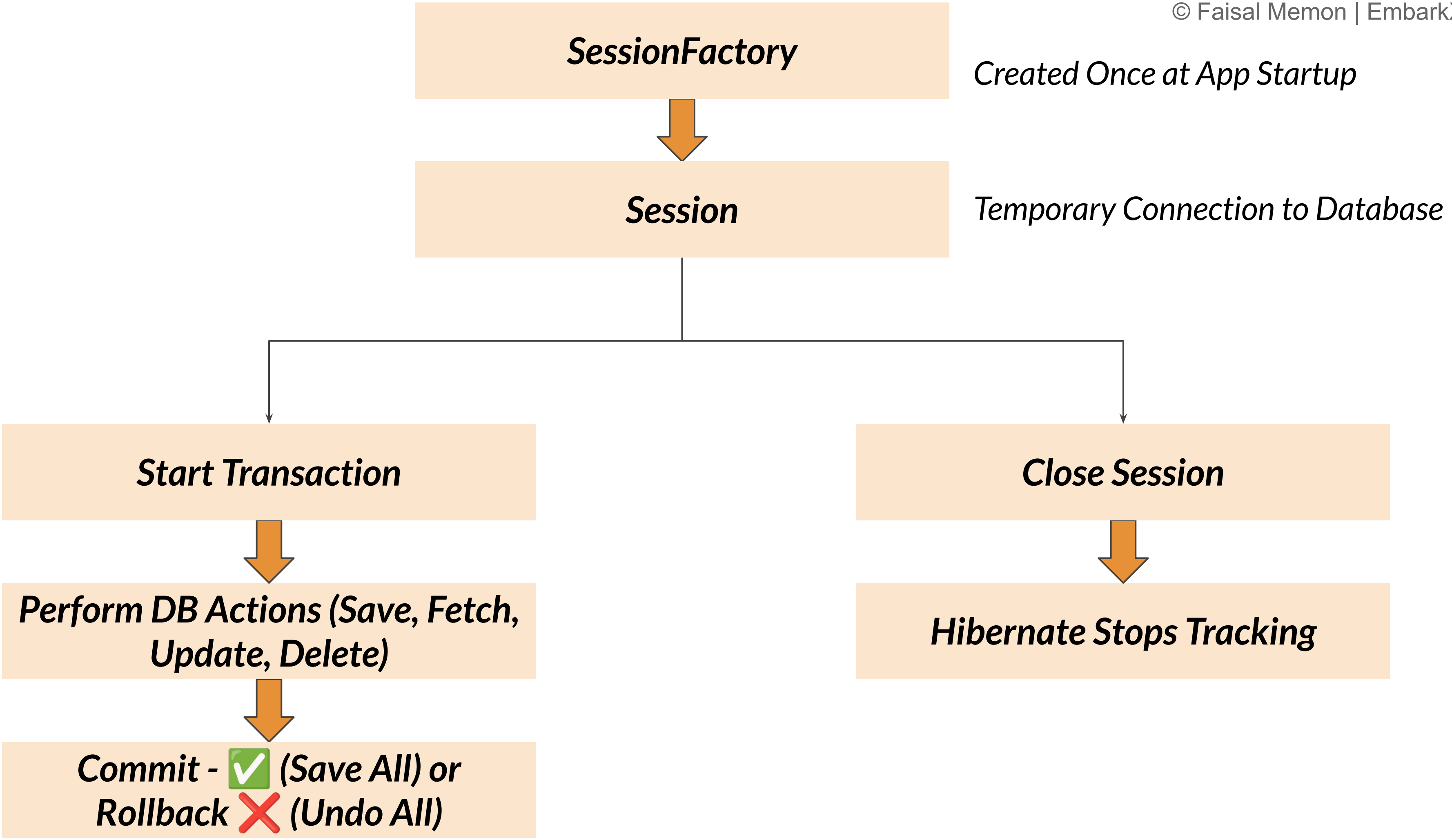
→ *Not reusable or shareable — use a new one each time.*

Transaction

→ *It wraps multiple database operations into one action.*

→ *Either everything gets saved, or nothing does.*

→ *Helps avoid half-done work or mistakes.*


SessionFactory
Created Once at App Startup
Session
Temporary Connection to Database
Start Transaction
Perform DB Actions (Save, Fetch, Update, Delete)
Commit - ✅ (Save All) or Rollback ❌ (Undo All)
Close Session
Hibernate Stops Tracking

JPA

Faisal Memon (EmbarkX)

JPA stands for **Jakarta Persistence API**

Not a framework, but a standard **specification**

Think of JPA as a set of **rules** or **guidelines**

"Hey, here's how you should define and manage data in your Java classes if you want to store them in a database."

This says "**I want a table called User with columns id and name**"

```
@Entity
public class User {
    @Id
    private Long id;
    private String name;
}
```

Hibernate is a tool (a library) that follows JPA rules and actually does the work for you.

Concept	Analogy	Example
JPA	Blueprint / Interface / Contract	“Here’s how a house should be designed.”
Hibernate	Builder / Worker	“I’ll build the house based on that design.”

How they work together?

→ *You write code using JPA annotations like @Entity, @Id, @OneToMany.*

→ *Then Hibernate: Reads those annotations, Automatically creates the matching tables, Writes SQL under the hood, Saves and fetches data for you.*

Feature	JPA	Hibernate
What is it?	Specification (rules/standard)	Implementation (actual working tool)
Who defines it?	Java (Jakarta EE)	A third-party library (hibernate.org)
Can it work alone?	❌ No	✅ Yes (even without JPA)
Does real work?	❌ No	✅ Yes (executes SQL, handles DB)
Annotations	`@Entity`, `@Id`, etc.	Same (it supports all JPA annotations)

Section Introduction

Faisal Memon (EmbarkX)

Why Learn About EntityManager

→ *EntityManager is the core interface of JPA*

→ *Sometimes you might need to use persist, merge, remove manually OR Control transactions more precisely*

→ *Work outside of Spring Data JPA*

→ *Helps with better debugging & troubleshooting*

Entity Manager

Faisal Memon (EmbarkX)

EntityManager is the main JPA interface that interacts directly with the database

What it does?

→ *Creates (saves) entities*

→ *Reads (finds) entities*

→ *Updates entities*

→ *Deletes entities*

It's the **core tool JPA uses under the hood** to manage the lifecycle of your Java objects when they talk to the database.

Why we haven't used it yet?

Spring Data JPA hides it behind repositories

Using EntityManager


```java
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
public class PatientService {

    @PersistenceContext
    private EntityManager entityManager;

    @Transactional
    public void savePatient() {
        Patient p = new Patient("John Doe", 40);
        entityManager.persist(p);  // Save the patient
    }

    public Patient findPatient(Long id) {
        return entityManager.find(Patient.class, id);  // Find by ID
    }
}
```

EntityManager States and PersistenceContext

Faisal Memon (EmbarkX)

Persistence Context

*The **persistence context** is like a workspace or cache inside the EntityManager that manages and tracks entity instances (Java objects) during a transaction.*

What Does It Do?

→ *It stores all entities that are currently being managed by JPA.*

→ *It keeps track of any changes you make to those entities.*

→ *It ensures your changes get saved to the database at the right time.*

→ *It guarantees identity consistency — that means if you request the same entity twice, you get the same Java object, not two different copies.*

Why Is It Important?

→ *Without a persistence context, JPA wouldn’t know which entities you are working with or which changes to save.*

→ *It helps* ***optimize database access*** *by caching entities within a transaction.*

→ *It allows JPA to* ***defer SQL operations*** *and batch them efficiently on commit.*

→ *It manages the* ***entity lifecycle states****: new, managed, detached, and removed.*

When Does It Exist?

→ *The persistence context exists inside an EntityManager session.*

→ *Typically, a transaction has one persistence context.*

→ *When the transaction ends (commit or rollback), the persistence context is closed, and entities become detached.*

An entity lifecycle means the different states an entity object goes through from **creation to deletion** when using JPA and the EntityManager.

Four Main States of a JPA Entity

- *New (Transient)*
- *Managed (Persistent)*
- *Detached*
- *Removed*

New (Transient) State

When you create a new entity instance using new, it is in the New (Transient) state.

```
User user = new User(); // New / Transient state
user.setName("Alice");
```

Managed (Persistent) State

A Managed entity is associated with the current persistence context.

```java
@Transactional
public void updateUser(Long userId) {
    User user = entityManager.find(User.class, userId); // Managed entity
    user.setName("Bob");  // Change tracked automatically
    // No explicit save needed, changes saved on commit
}
```

Detached State

When the persistence context is closed or the transaction ends, managed entities become Detached.

```
User detachedUser = getUserFromOutside();
detachedUser.setName("Charlie");

@Transactional
public void updateDetachedUser(User user) {
    entityManager.merge(user); // Reattaches and saves changes
}
```

Removed State

When you want to delete an entity, call entityManager.remove(entity).

```
@Transactional
public void deleteUser(Long userId) {
    User user = entityManager.find(User.class, userId);
    entityManager.remove(user);  // Marked as removed
    // Deleted on commit
}
```

State	Description	How to Enter This State	What Happens on Commit	Example Operations
New (Transient)	Newly created object, no DB row, no persistence context	`new Entity()`	No DB effect until persisted	`new User()`
Managed (Persistent)	Entity tracked by persistence context; changes auto-saved	`persist()`, `find()`, `merge()`	Changes flushed to DB automatically	`entityManager.persist(user)`
Detached	Entity not tracked anymore; changes ignored	Persistence context closed, transaction ended	Changes not saved unless merged	After transaction ends
Removed	Entity marked for deletion	`remove()` called on managed entity	Entity deleted from DB	`entityManager.remove(user)`

Composite Keys

Faisal Memon (EmbarkX)

A **composite key** is a combination of two or more columns in a database table that together uniquely identify a record

Why Not Just Use a Regular Column Primary Key?

→ *When no single column can uniquely identify a record.*

→ *When the uniqueness of a row depends on a combination of columns*

→ *When modeling many-to-many relationships with extra data.*

The Problem That Composite Keys Solve

Imagine a table called Prescription where you store prescriptions written by a doctor for a patient.

→ *A patient can have many prescriptions.*

→ *A doctor can write many prescriptions.*

→ *We want to avoid just assigning a random id to each record.*

→ *What uniquely identifies a prescription?*
*It could be the **combination of patient + doctor.***

When Do You Use Composite Keys?

Scenario	Example
Relationship needs to be uniquely identified by multiple fields	(doctor_id, patient_id) in Prescription
Many-to-many with additional data	(student_id, course_id) + grade
Historical tracking	(employee_id, date) for attendance logs

@Table Annotation

Faisal Memon (EmbarkX)

```
@Table(
    name = "patients", // The name of the table in the database. By default, JPA would use the class name (e.g.,
"Patient"), but here we're explicitly naming it "patients".

    catalog = "hospital_db", // Optional. Refers to the catalog (think of it as a database group). Rarely used
unless your DB uses catalogs.

    schema = "public", // Refers to the schema under which the table resides (e.g., "public", "admin", etc.).
Useful in PostgreSQL or Oracle DBs.

    uniqueConstraints = {
        @UniqueConstraint(columnNames = {"email"}),
        // Ensures that the 'email' column must have unique values across all rows (i.e., no two patients can
have the same email).

        @UniqueConstraint(columnNames = {"national_id"})
        // Similarly, ensures 'national_id' is unique — useful for real-world identifiers like SSNs, Aadhaar,
etc.
    },

    indexes = {
        @Index(name = "idx_email", columnList = "email"),
        // Creates a named index "idx_email" on the 'email' column to improve search/query performance.

        @Index(name = "idx_created_at", columnList = "created_at DESC")
        // Adds an index on the 'created_at' column in descending order — useful if you often sort patients by
most recent creation.
    }
)
```

Generation Types For Identity

Faisal Memon (EmbarkX)

Different Generation Types

- AUTO
- IDENTITY
- SEQUENCE
- TABLE
- NONE

GenerationType.AUTO

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
```

GenerationType.IDENTITY

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

GenerationType.SEQUENCE

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE)
private Long id;
```

GenerationType.SEQUENCE

```
@Id

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

@SequenceGenerator(name = "order_seq", sequenceName =
"order_sequence", allocationSize = 1)

private Long id;
```

GenerationType.SEQUENCE

```
@Id

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

@SequenceGenerator(name = "order_seq", sequenceName =
"order_sequence", allocationSize = 1)

private Long id;
```

GenerationType.SEQUENCE

```
@Id

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

@SequenceGenerator(name = "order_seq", sequenceName =
"order_sequence", allocationSize = 1)

private Long id;
```

GenerationType.SEQUENCE

```
@Id

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

@SequenceGenerator(name = "order_seq", sequenceName =
"order_sequence", allocationSize = 1)

private Long id;
```

GenerationType.SEQUENCE

```
@Id

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

@SequenceGenerator(name = "order_seq", sequenceName =
"order_sequence", allocationSize = 1)

private Long id;
```

GenerationType.SEQUENCE

```
@Id

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

@SequenceGenerator(name = "order_seq", sequenceName =
"order_sequence", allocationSize = 1)

private Long id;
```

GenerationType.TABLE

```java
@Id
@GeneratedValue(strategy = GenerationType.TABLE)
private Long id;
```

GenerationType.TABLE

```
@Id

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

@TableGenerator(name = "task_gen", table = "id_gen",
pkColumnName = "gen_key", valueColumnName = "gen_value",
pkColumnValue = "task_id", allocationSize = 1)

private Long id;
```

GenerationType.TABLE

```
@Id

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

@TableGenerator(name = "task_gen", table = "id_gen",
pkColumnName = "gen_key", valueColumnName = "gen_value",
pkColumnValue = "task_id", allocationSize = 1)


private Long id;
```

GenerationType.TABLE

```
@Id

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

@TableGenerator(name = "task_gen", table = "id_gen",
pkColumnName = "gen_key", valueColumnName = "gen_value",
pkColumnValue = "task_id", allocationSize = 1)


private Long id;
```

GenerationType.TABLE

```
@Id

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

@TableGenerator(name = "task_gen", table = "id_gen",
pkColumnName = "gen_key", valueColumnName = "gen_value",
pkColumnValue = "task_id", allocationSize = 1)


private Long id;
```

GenerationType.TABLE

```
@Id

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

@TableGenerator(name = "task_gen", table = "id_gen",
pkColumnName = "gen_key", valueColumnName = "gen_value",
pkColumnValue = "task_id", allocationSize = 1)


private Long id;
```

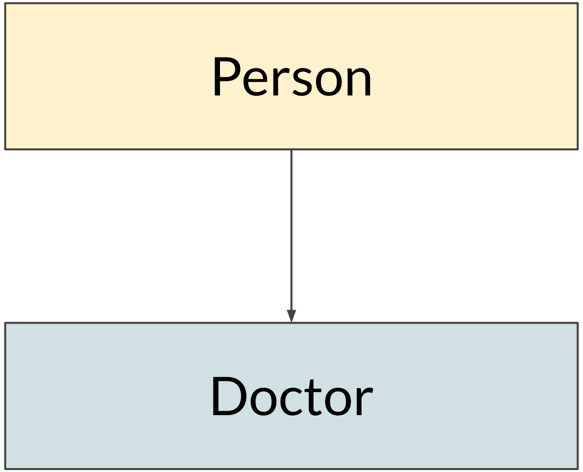
Thank you

Inheritance in JPA

Faisal Memon (EmbarkX)

Inheritance in Object-Oriented Programming (OOP) - lets **one class inherit fields and methods** from another class.

Inheritance



Why Does Inheritance Matter?

Common Entity	Shared Fields	Unique Fields
`Person`	name, age, gender, email	
`Patient`	name, age, gender, email	medicalHistory, allergies
`Doctor`	name, age, gender, email	specialization, licenseNo

Relational databases (MySQL, PostgreSQL, etc.) **don’t "think"** like OOP.

```java
@Entity
public class Person {}

@Entity
public class Doctor extends Person {}
```

Java understands the inheritance.
But the database? ❌ It doesn't.

JPA solves this

Problems Does Inheritance Solve in JPA?

→ *Code Reuse*

→ *Less Duplication*

→ *Cleaner Design*

→ *Easier Maintenance*

Inheritance Strategies

Faisal Memon (EmbarkX)

Inheritance Strategies

- *SINGLE_TABLE*
- *TABLE_PER_CLASS*
- *JOINED*

SINGLE_TABLE Strategy

→ *All child classes are stored in one single table.*

→ *This table has all possible columns, even if some are unused.*

id	name	age	email	specialization	medicalHistory	role
1	John	45	...	Cardiology	NULL	Doctor
2	Alice	30	...	NULL	Diabetes	Patient

TABLE_PER_CLASS Strategy

→ *Each class has its own table.*

→ *The common fields (from Person) are copied into each table.*

Doctor Table →

id	name	age	email	specialization
1	Dr. Smith	45	…	Cardiology

Patient Table →

id	name	age	email	medicalHistory
1	Alice	45	…	Knee Pain

JOINED Strategy

→ *One table for the parent class*

→ *One table for each subclass. They are linked using a foreign key.*

Person Table

id	name	age	email
1	Alice	45	…

Doctor Table

id	specialization
1	Cardiology

Patient Table

id	medicalHistory
1	Knee Pain

How to Choose?

Strategy	Use When
`SINGLE_TABLE`	Performance is key, subclasses don’t differ much
`TABLE_PER_CLASS`	Each class is very different and rarely queried together
`JOINED`	You want clean DB design and flexibility (best for large apps)

Feature	SINGLE_TABLE	TABLE_PER_CLASS	JOINED
Number of Tables	1 (All in one table)	One table per class	One parent table + one per subclass
Data Redundancy	No redundancy, but has many NULLs	High redundancy (repeated fields)	No redundancy (clean normalization)
Query Speed	Fast (single-table reads)	Fast for subclass queries	Slower (requires joins between tables)
Database Normalization	Poor (lots of NULLs in unused columns)	Poor (duplicated columns)	Excellent (proper relational design)
Simplicity	Easiest to set up	Simple structure per entity	More complex schema
Extensibility	Medium (can get messy with many fields)	Low (hard to scale cleanly)	High (easy to add new subclasses)
Good For	Simple projects, fast reads	Polymorphic queries not needed	Enterprise apps with many entity types
Drawback	Table grows large, messy with NULLs	Can’t easily query all subclasses at once	Slower reads, JOIN overhead

Introduction and Need for JPQL

Faisal Memon (EmbarkX)

JPQL is a way to **write queries** in Java when you're working with databases using JPA (Java Persistence API).

Difference Between JPQL and SQL

Feature	SQL	JPQL
Queries	Tables and columns in the database	Java entity classes and their fields
Language	Standard database language	Java-based object query language
Example	`SELECT * FROM users`	`SELECT u FROM User u`

Why JPQL Queries Entity Classes, Not Tables

Because in JPA, you don't directly deal with tables. You map tables to Java classes (called entities).

- You write: *SELECT e FROM Employee e*
- Not: *SELECT * FROM employees*

Case Sensitivity Rules

→ Entity class names and field names are case-sensitive in JPQL. (Employee is not the same as employee)

→ JPQL keywords like SELECT, FROM, WHERE are not case-sensitive. (select, SELECT, and SeLeCt all work the same)

Important JPQL Keywords

SELECT: *What you want to get*
→ SELECT e

FROM: *Which entity class to use*
→ FROM Employee e

WHERE: *Conditions (like filters)*
→ WHERE e.salary > 50000

Example JPQL

```
SELECT e FROM Employee e WHERE
e.salary > 50000
```

Query Derivation

Faisal Memon (EmbarkX)

Query Derivation

Method Name	Generated JPQL
`findByName(String name)`	`SELECT p FROM Patient p WHERE p.name = ?1`
`findByAgeGreaterThan(int age)`	`SELECT p FROM Patient p WHERE p.age > ?1`
`findByEmailContaining(String s)`	`SELECT p FROM Patient p WHERE p.email LIKE %?1%`
`findByDoctorName(String name)`	(Assumes relationship) `SELECT p FROM Patient p WHERE p.doctor.name = ?1`

Transactions

Faisal Memon (EmbarkX)

A transaction is a sequence of operations performed as a **single logical unit of work** in a database system or application

Step 1: You withdraw ₹5,000 from Account A

Step 2: You deposit ₹5,000 into Account B

The ACID Properties

- *Atomicity*
- *Consistency*
- *Isolation*
- *Durability*

⚠️ What If We Don't Use Transactions?

→ Partial updates

→ Data corruption

→ Lost updates

Where Are Transactions Used?

→ *Banking & Financial Systems*

→ *Order Management & E-commerce*

→ *Ticket Booking Systems*

→ *Inventory & Stock Management*

→ *Any multi-user database-driven application*

What Happens Without Transactions?

Faisal Memon (EmbarkX)

1. Partial Updates (Inconsistent State)

→ If one part of a multi-step process fails but the others succeed, the database is left in an invalid or incomplete state.

- *Deduct ₹5,000 from Account A* ✅
- *Deposit ₹5,000 into Account B* ❌ *(e.g., system crash)*

Result: Account A loses money, but Account B never receives it — your system is now inconsistent.

2. Dirty Reads

→ *Without isolation provided by transactions, one operation might read data that hasn't been finalized (i.e., committed).*

- *Transaction A updates a row but hasn't committed yet.*
- *Transaction B reads the same row — it sees uncommitted data.*
- *If Transaction A rolls back, Transaction B has acted on invalid data.*

🧨 *Result: Wrong decisions based on unverified changes.*

3. Lost Updates

→ If two operations modify the same data concurrently, and the last one overwrites the first, data is lost.

- *User A sets quantity = 10 (based on what they read earlier)*
- *User B sets quantity = 5 at the same time*

🧨 *Result: One of these updates is lost forever, leading to data corruption.*

4. Unrepeatable Reads

→ A query in one operation returns different results when run multiple times in the same logical unit.

- *A report generator reads all orders in the morning*
- *Meanwhile, another operation deletes or inserts some orders*

🧨 *Result: The same report now shows different results midway, causing confusion.*

5. Phantom Reads

→ Similar to unrepeatable reads, but refers to new rows appearing in between reads of the same query.

- *Query: “Get all users who signed up today”*
- *Midway through processing, a new user signs up*

🧨 *Result: Your application sees inconsistent data within the same session.*

Summary: Without Transactions

Risk	Consequence
Partial Execution	Inconsistent data
No Isolation	Reads/writes can conflict
No Atomicity	Only some steps may succeed
No Durability	Changes can be lost after system crash
Concurrency issues	Data races, lost updates, stale reads