

## WHY ORM & JPA EXIST ?

(What JDBC couldn't solve and how Hibernate + Spring Data JPA fixed it)

Evolution: From JDBC to Hibernate to Spring Data JPA

### CASE 1 — JDBC – The Starting Point (Direct Database Access) (Pain Zone)

Imagine you are building a Java application and you need to store and read data from a database.

The first thing you do is write **SQL queries** to:

- Create tables
- Insert records
- Fetch data
- Update and delete rows

To connect Java with the database, you use **JDBC (Java Database Connectivity)**.

#### What is JDBC?

**JDBC → Java Database Connectivity**

JDBC is the low-level Java API that allows a Java program to directly connect to a relational database, send SQL queries, and receive results.

It is the foundation layer of all Java–Database communication.

JDBC acts as a **direct bridge between Java and the database**.

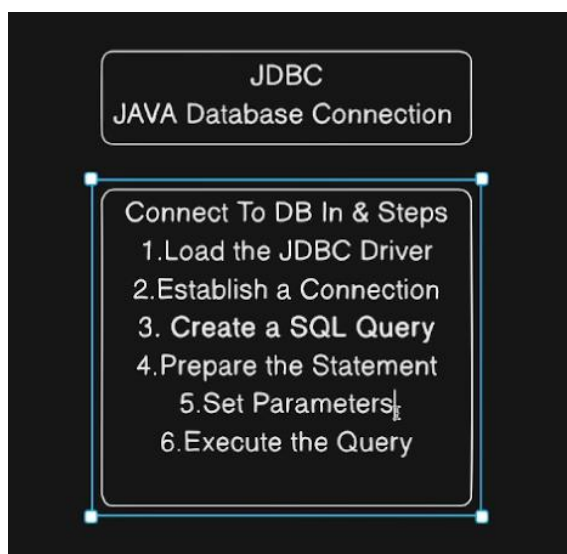
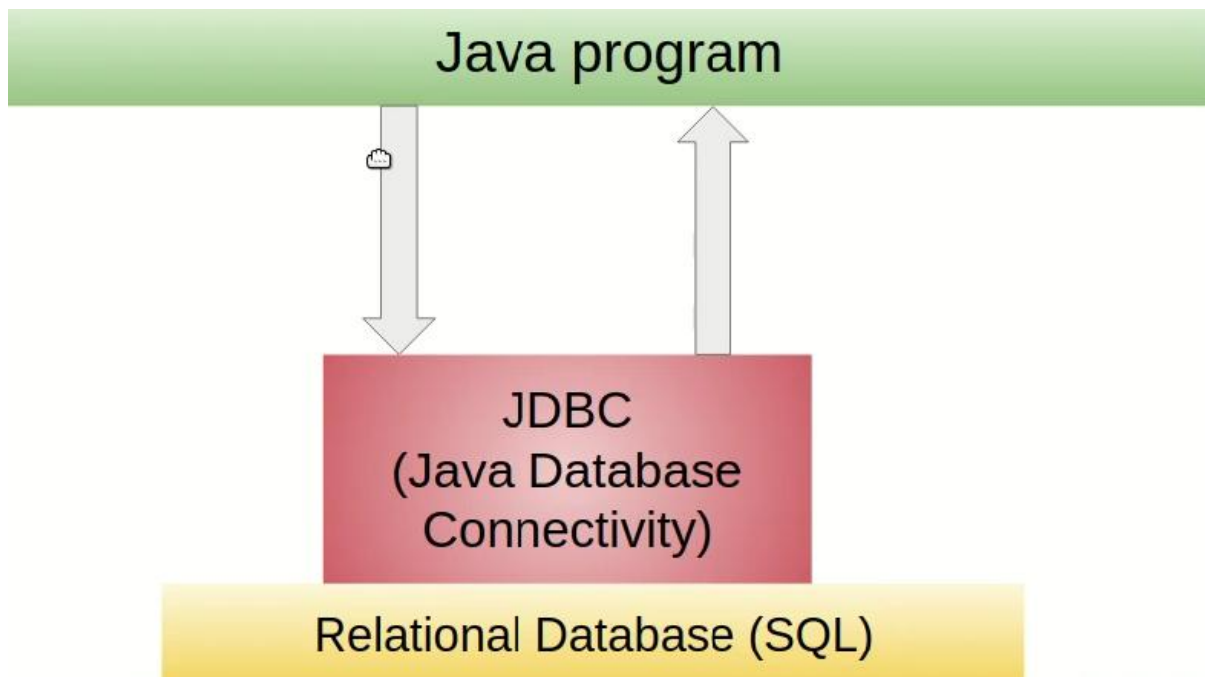
#### What does JDBC do?

JDBC provides:

- Database connection
- SQL execution
- Result fetching
- Transaction control

Without JDBC, **Java cannot talk to any database**.

JDBC is the **last bridge before the database**.



You manually:

- Open a connection
- Write SQL queries
- Execute them
- Read results from ResultSet
- Map them into Java objects

This works fine for small applications.

```

Connection con = DriverManager.getConnection(...);
PreparedStatement ps = con.prepareStatement(
    "SELECT id, name FROM users WHERE id=?");
ps.setInt(1, id);
ResultSet rs = ps.executeQuery();

User u = new User();
u.setId(rs.getInt("id"));
u.setName(rs.getString("name"));

```

But as the project grows, your code becomes full of Problems :

- Repeated SQL
- Connection handling logic
- Error-prone mapping
- Database-specific code
- SQL everywhere
- Manual mapping
- Hard to refactor
- No abstraction

Your business logic slowly gets mixed with database logic, making the application **hard to maintain and hard to scale**.

Problem	Why it hurts
SQL everywhere	Business logic mixed with DB logic
Manual mapping	ResultSet → Object done by hand
Boilerplate	Too much repetitive code
Hard to maintain	Table change breaks Java code
No abstraction	DB structure controls your code

JDBC works — but **does not scale for enterprise systems**.

## Case 2: Hibernate – Object-Oriented Abstraction

After some time, you get frustrated:

- Writing SQL everywhere
- Copy-pasting mapping code
- Fixing bugs when table structure changes

You want a way to **work with Java objects instead of SQL queries**.

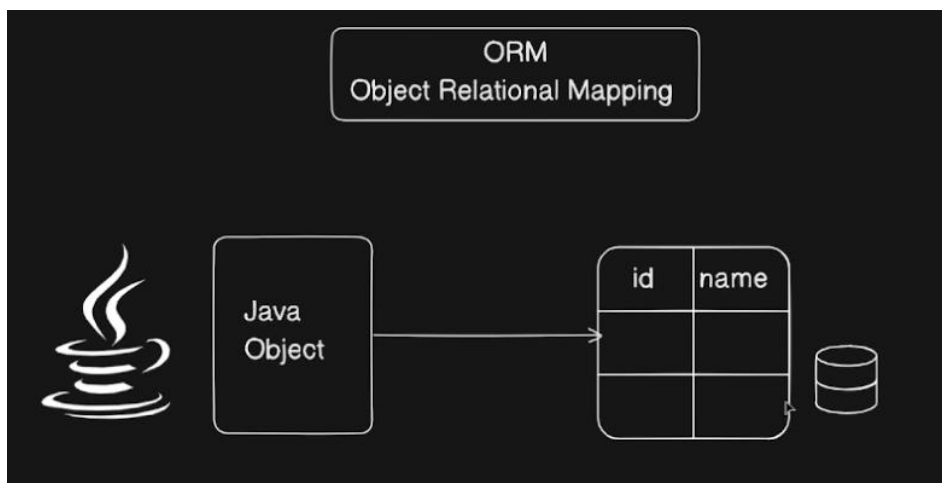
This is where **Hibernate (ORM framework)** comes in.

### What is ORM?

#### ORM → Object Relational Mapping

ORM is a technique that maps Java objects to relational database tables and database rows to Java objects automatically.

It acts as a translator between the Object-Oriented world and the Relational Database world.



### The real problem ORM solves

Java works with:

- Objects
- Classes
- Inheritance

Databases work with:

- Tables
- Rows
- Columns
- Foreign Keys

These two worlds do not naturally match.

### Without ORM (JDBC style)

- You manually convert rows → objects
- You write SQL everywhere
- Your business logic becomes DB-dependent

ORM removes this gap.

`ResultSet → manually map → Java Object`

```
User u = new User();  
u.setId(rs.getInt("id"));  
u.setName(rs.getString("name"));
```

Every table change breaks your code.

### ORM:

- Reads annotations
- Builds mapping metadata
- Generates SQL
- Converts rows → objects
- Tracks object changes
- Syncs DB automatically

## With ORM (Hibernate/JPA)

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

```
User u = userRepository.findById(1L).get();
```

## How ORM works internally

```
Java Object
  ↓
ORM Engine
  ↓
Mapping Metadata
  ↓
Generated SQL
  ↓
JDBC
  ↓
Database
```

And reverse for fetching.

## JDBC exposes the database directly to your code.

This creates:

- Tight coupling
- Boilerplate
- Hard maintenance
- SQL everywhere
- Mapping chaos

So the industry created **ORM**.

**ORM = Bridge between OOP world and Relational DB world**

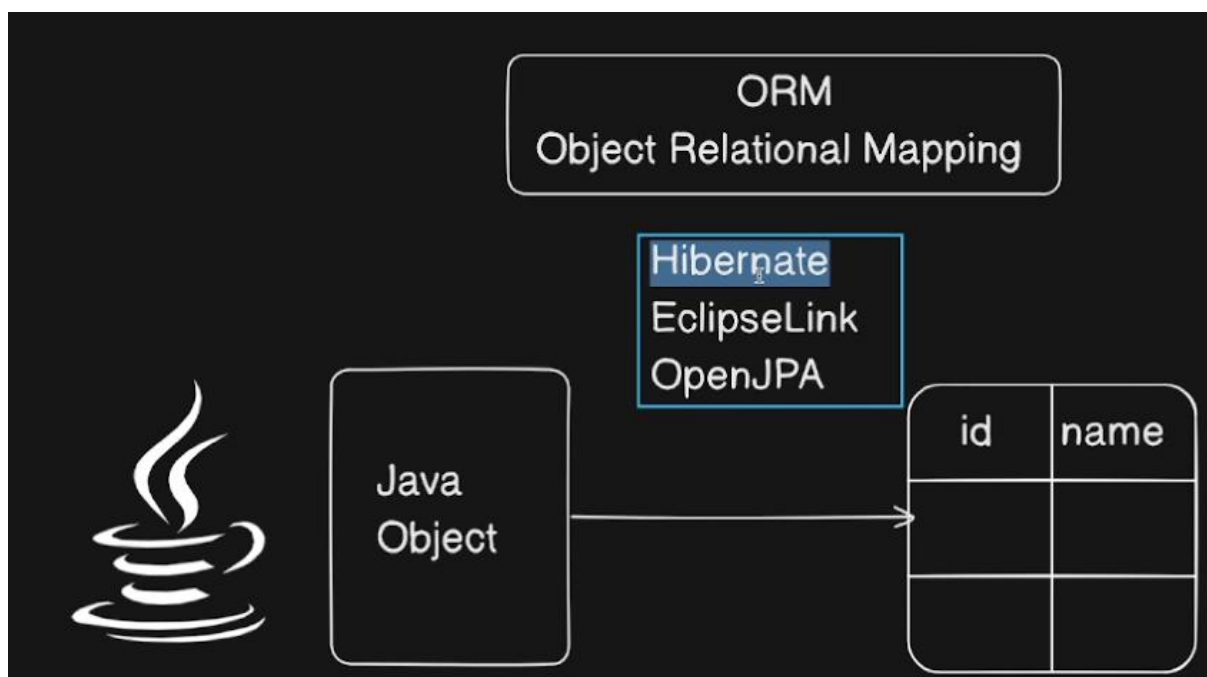
Problem with JDBC	ORM solution
SQL everywhere	SQL auto-generated
Manual mapping	Auto object mapping
Hard maintenance	Centralized mapping
No caching	First/Second level cache
No lifecycle	Entity state management

**ORM lets you think in objects while the database thinks in tables — and handles the translation for you.**

### **What is Hibernate?**

Hibernate is an ORM framework that maps Java objects to database tables and automatically generates SQL queries.

You work with **objects**, not SQL.



## What Hibernate solves

With JDBC	With Hibernate
Write SQL	Write Java objects
Manual mapping	Auto mapping
Tight DB coupling	Loose coupling
No caching	First level cache
No lifecycle	Entity lifecycle

## Hibernate:

- Reads @Entity annotations
- Converts objects  $\leftrightarrow$  rows
- Tracks changes (dirty checking)
- Auto-generates SQL
- Manages transactions

## Hibernate flow

```
Java Object
  ↓
Hibernate Session
  ↓
Entity Mapping
  ↓
Generated SQL
  ↓
JDBC
  ↓
Database
```

Hibernate still **uses JDBC internally**, but you never touch it

Hibernate:

- Reads annotations
- Generates SQL
- Maps row → object
- Manages transactions

With Hibernate:

- You create Java classes and mark them as @Entity
- You describe table mapping using annotations
- You work with objects, not rows

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

```
User u = session.get(User.class, 1L);
```

Now your flow becomes:

```
Java Object
  ↓
Hibernate
  ↓
Auto-generated SQL
  ↓
JDBC
  ↓
Database
```

Hibernate:

- Translates Java objects into SQL
- Converts result rows back into objects
- Tracks changes automatically
- Reduces boilerplate
- Keeps your code clean and object-oriented

You still use JDBC internally — but you never touch it directly.

### **Case 3: JPA & Spring Data JPA – Enterprise Level Abstraction**

Even with Hibernate, you still write:

- DAO classes
- EntityManager code
- Query logic

To make this even easier, **JPA (Java Persistence API)** standardizes ORM behavior, and **Spring Data JPA** builds on top of it.

### **What is JPA?**

**JPA → Java Persistence API**

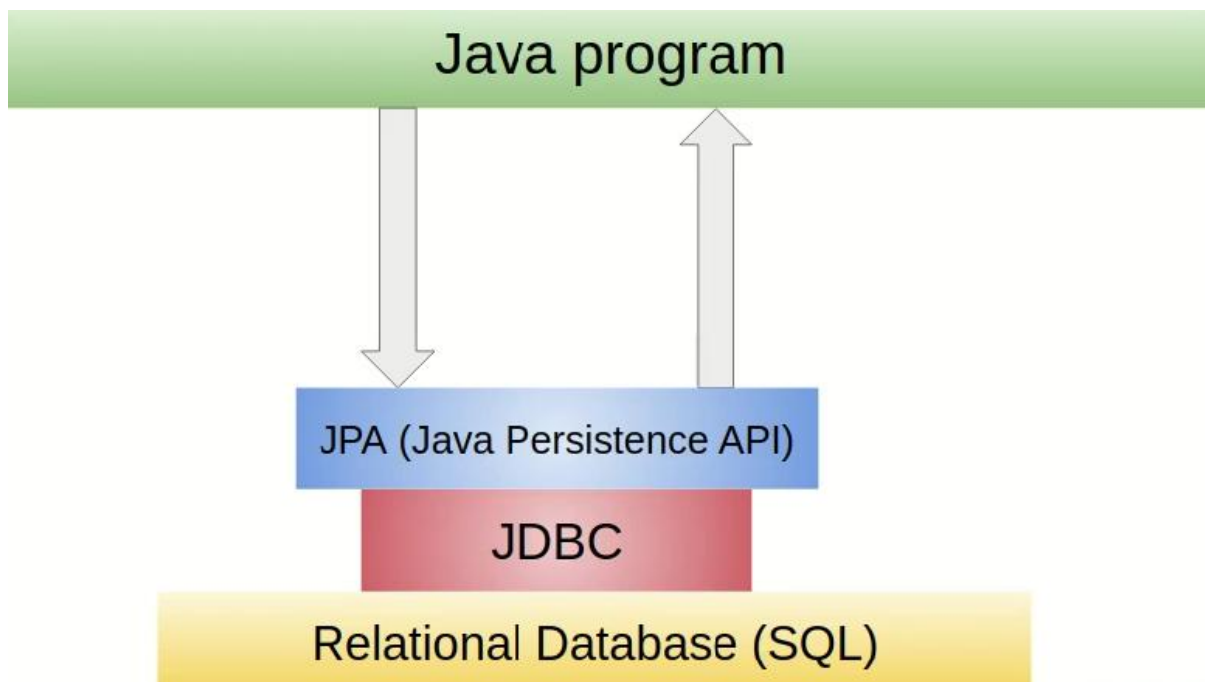
JPA is NOT a framework. It is a specification.

It defines:

- How ORM should work
- Which annotations exist
- How persistence should behave

Examples of JPA implementations:

- Hibernate
- EclipseLink
- OpenJPA



Hibernate **implements JPA**.

### What is Spring Data JPA?

Spring Data JPA is a Spring abstraction built on top of JPA that removes repository boilerplate and gives auto-generated queries.

It:

- Auto-creates repository implementations
- Converts method names → SQL
- Manages transactions
- Integrates with Spring IoC container

### With Spring Data JPA

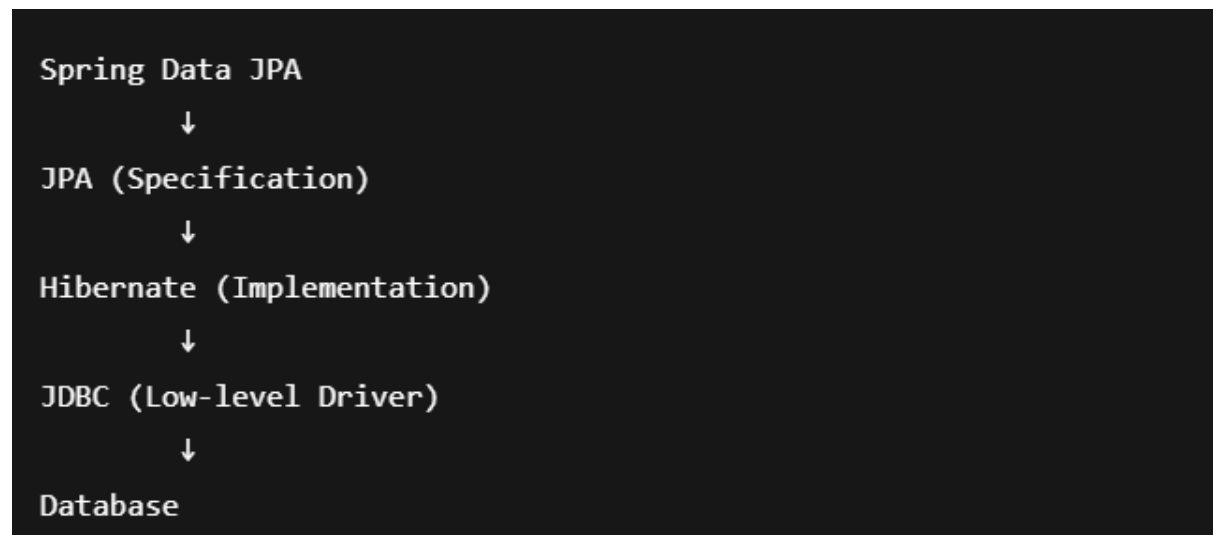
```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    User findByName(String name);
}
```

No implementation.

No SQL.

Spring creates proxy beans at runtime.

### Relationship Summary



Layer	Role
JDBC	Physical DB communication
Hibernate	ORM engine
JPA	ORM rules/spec
Spring Data JPA	Developer-friendly abstraction

JDBC talks to DB, Hibernate maps objects, JPA defines rules, Spring Data JPA removes boilerplate.

Spring Data JPA gives:

- Ready-made repository interfaces
- Automatic query generation
- Transaction management
- Integration with Spring IoC container

Now you simply define:

```
public interface UserRepository extends JpaRepository<User, Long>
```

```
@Repository  
public interface UserRepository extends JpaRepository<User, Long> {  
    List<User> findByName(String name);  
}
```

No SQL.

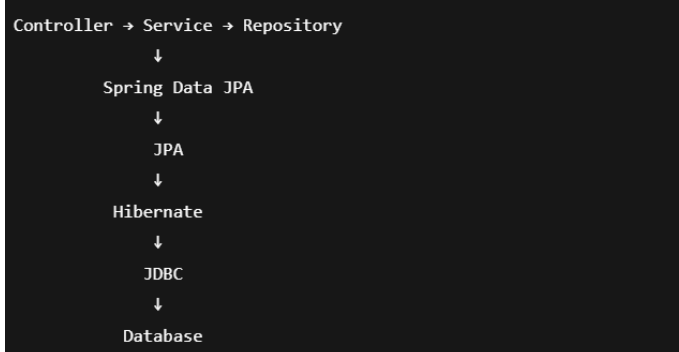
No boilerplate.

Only domain logic.

And Spring:

- Creates the implementation
- Wires it as a Bean
- Manages transactions
- Calls Hibernate under the hood

So the final flow becomes:



At this stage, you focus only on **business logic**, while Spring and Hibernate handle persistence.

**JDBC connects Java to DB, Hibernate maps objects to tables, JPA defines the rules, and Spring Data JPA removes all persistence boilerplate.**

## HIBERNATE vs JPA vs SPRING DATA JPA

Feature	JDBC	Hibernate	JPA	Spring Data JPA
Level	Low	ORM	Spec	Abstraction
SQL	Manual	Auto	Auto	Auto
Mapping	Manual	Yes	Yes	Yes
Boilerplate	High	Medium	Medium	Lowest
Learning	Hard	Medium	Medium	Easy

### Flow

```
You → Controller → Service → Repository  
Repository → JPA → Hibernate → JDBC → DB  
DB → JDBC → Hibernate → JPA → Object → You
```

You start with JDBC.

You drown in SQL.

You crave abstraction.

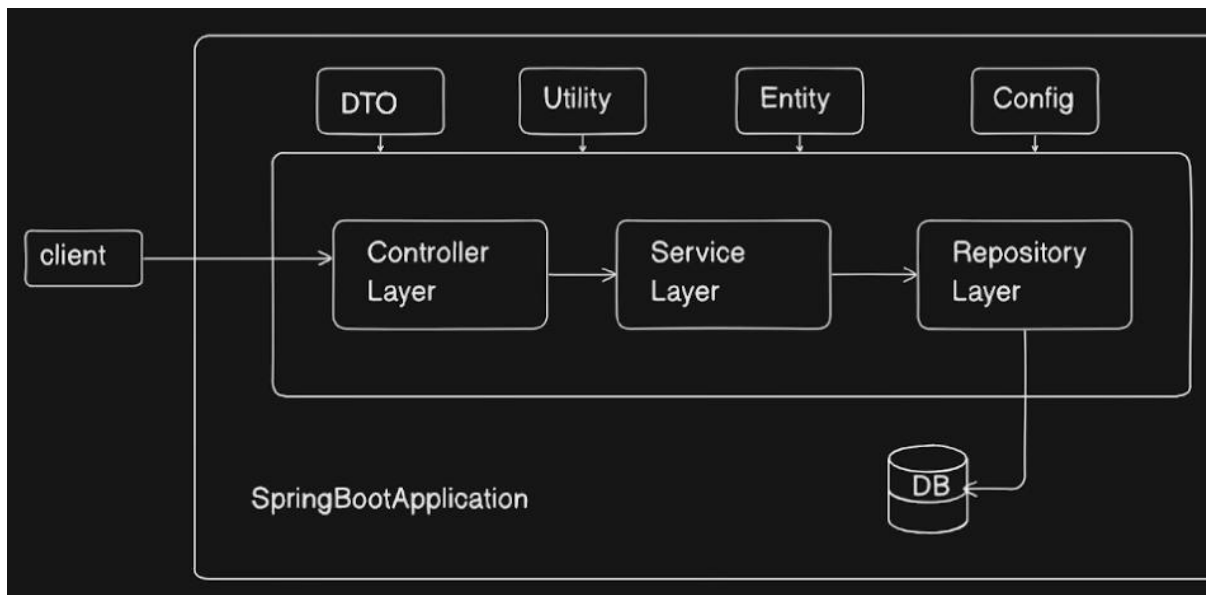
**Hibernate** says → “Think in objects.”

**JPA** says → “Standardize this.”

**Spring Data JPA** says → “Stop worrying about persistence. Build systems.”

This is not convenience, This is **enterprise survival**.

## LAYERED ARCHITECTURE FLOW



```
Client (Postman / UI)
|
v
@Controller → Validates input
|
v
@Service      → Business logic
|
v
@Repository   → Persistence logic
|
v
JPA/Hibernate → ORM mapping
|
v
JDBC Driver
|
v
Database
```

## DEFINITIONS

Term	Meaning
JDBC	Java Database Connectivity – low-level DB API
ORM	Object Relational Mapping
Hibernate	ORM framework
JPA	Java Persistence API (spec)
Spring Data JPA	Spring abstraction over JPA
Entity	Java class mapped to DB table
Repository	Interface to perform DB ops
DTO	Data Transfer Object
@Entity	Marks persistent class
@Id	Primary key
@Repository	DAO layer
@Service	Business layer
@Controller	API layer
@Component	Generic Bean
IoC	Inversion of Control
Container	Spring runtime managing beans