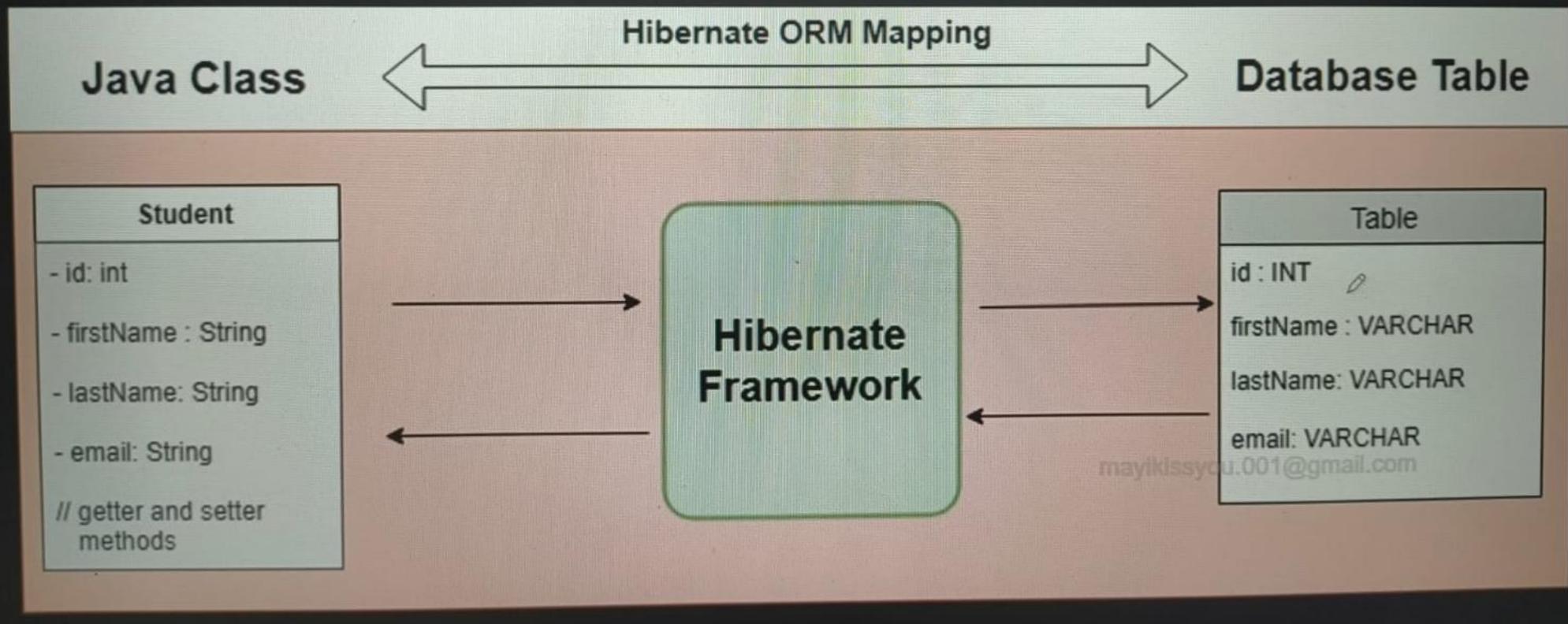


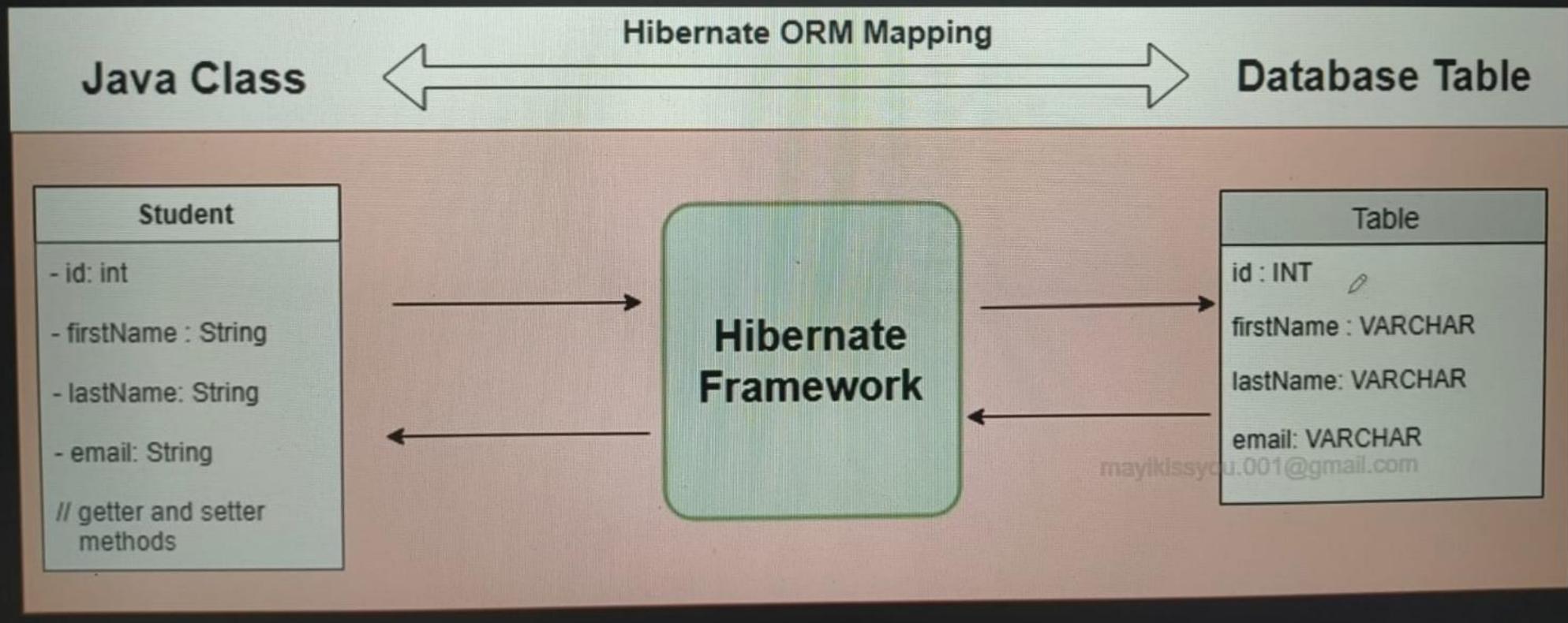
918279399130

Hibernate ORM Mapping



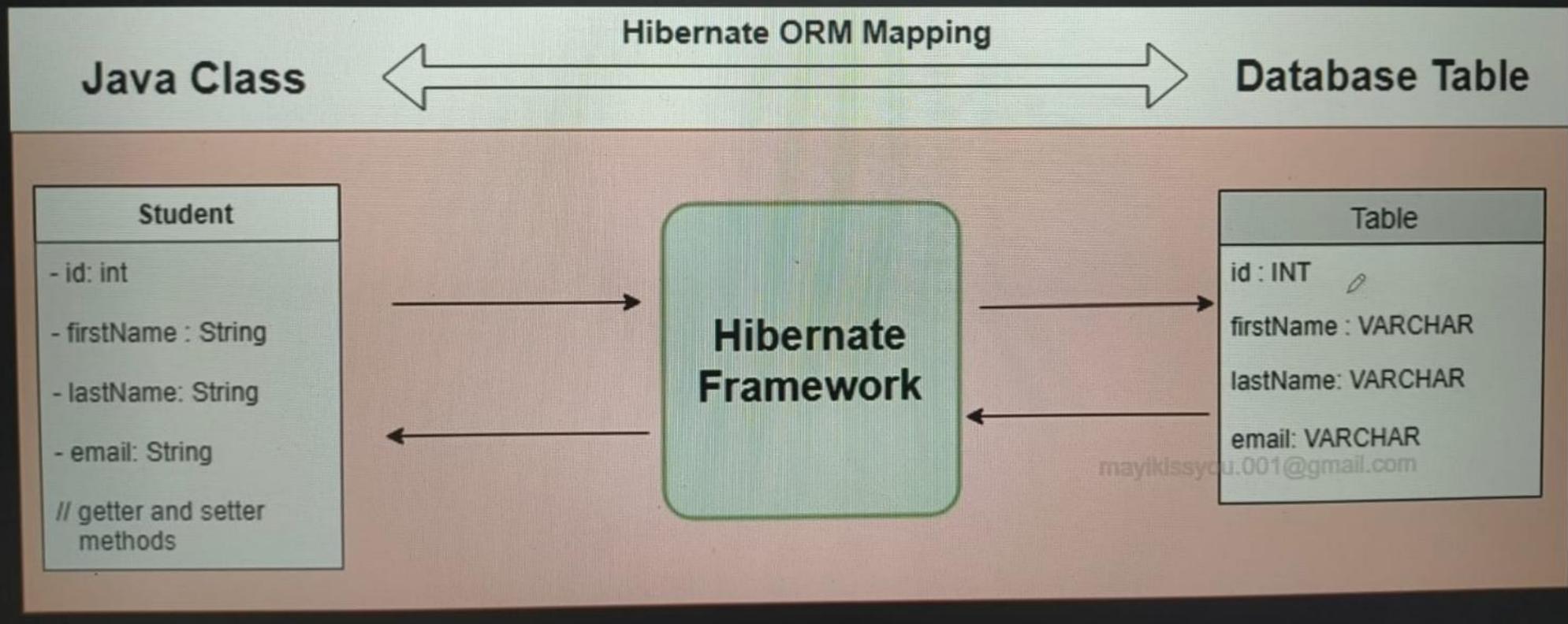
918279399130

Hibernate ORM Mapping



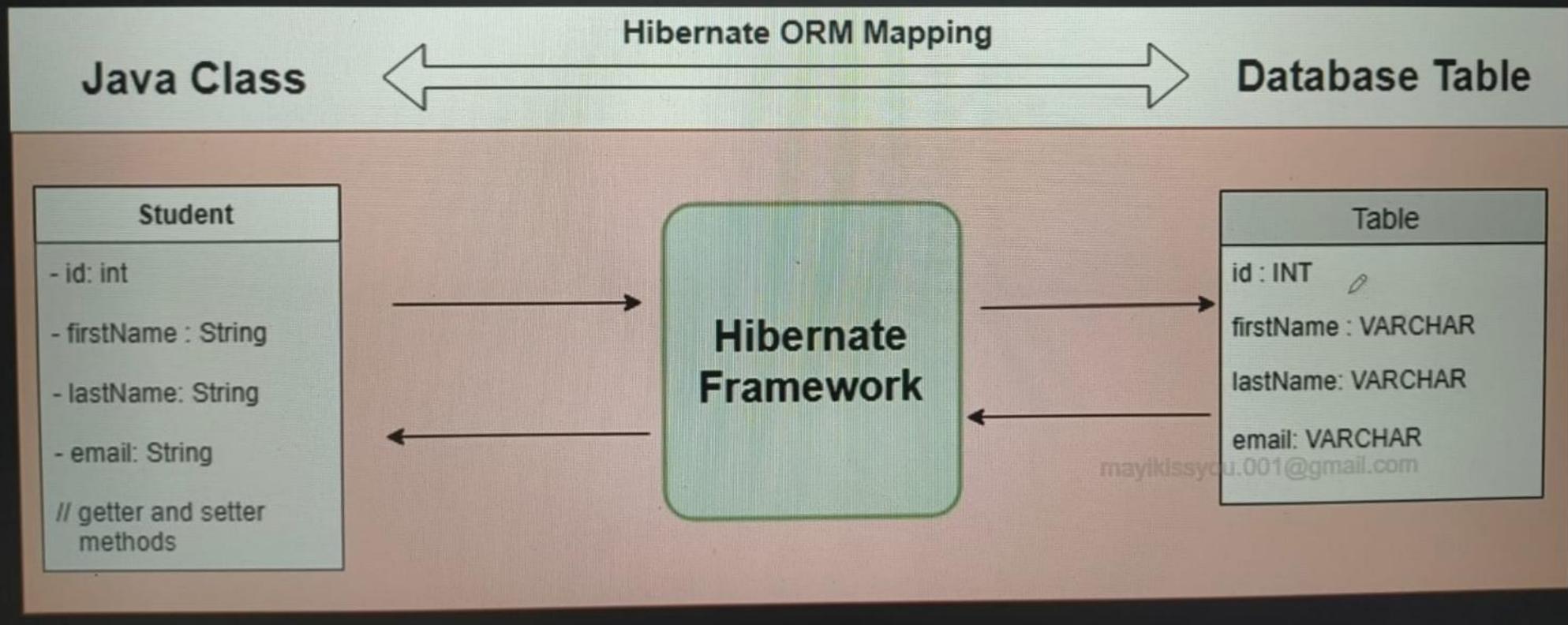
918279399130

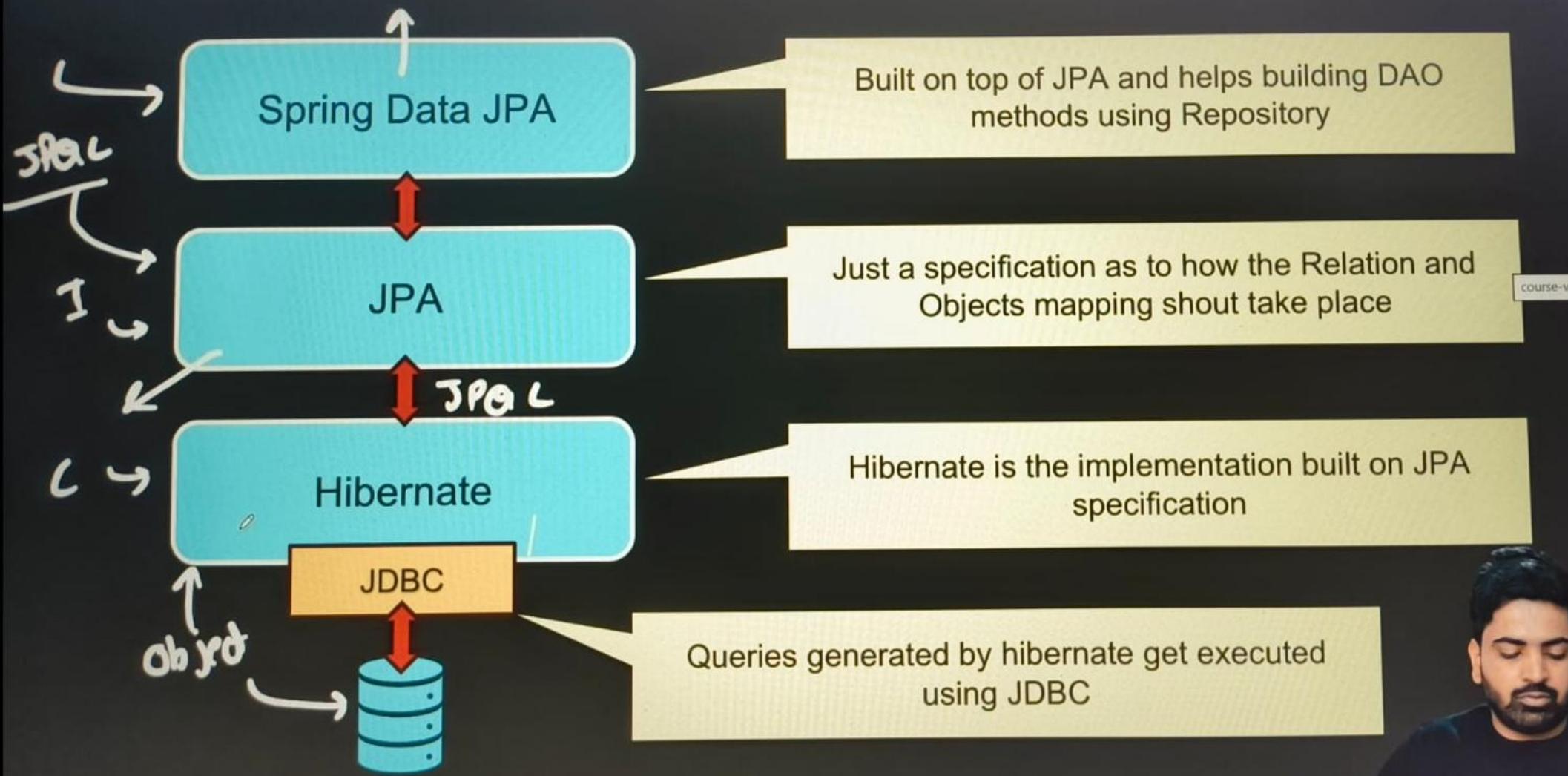
Hibernate ORM Mapping



918279399130

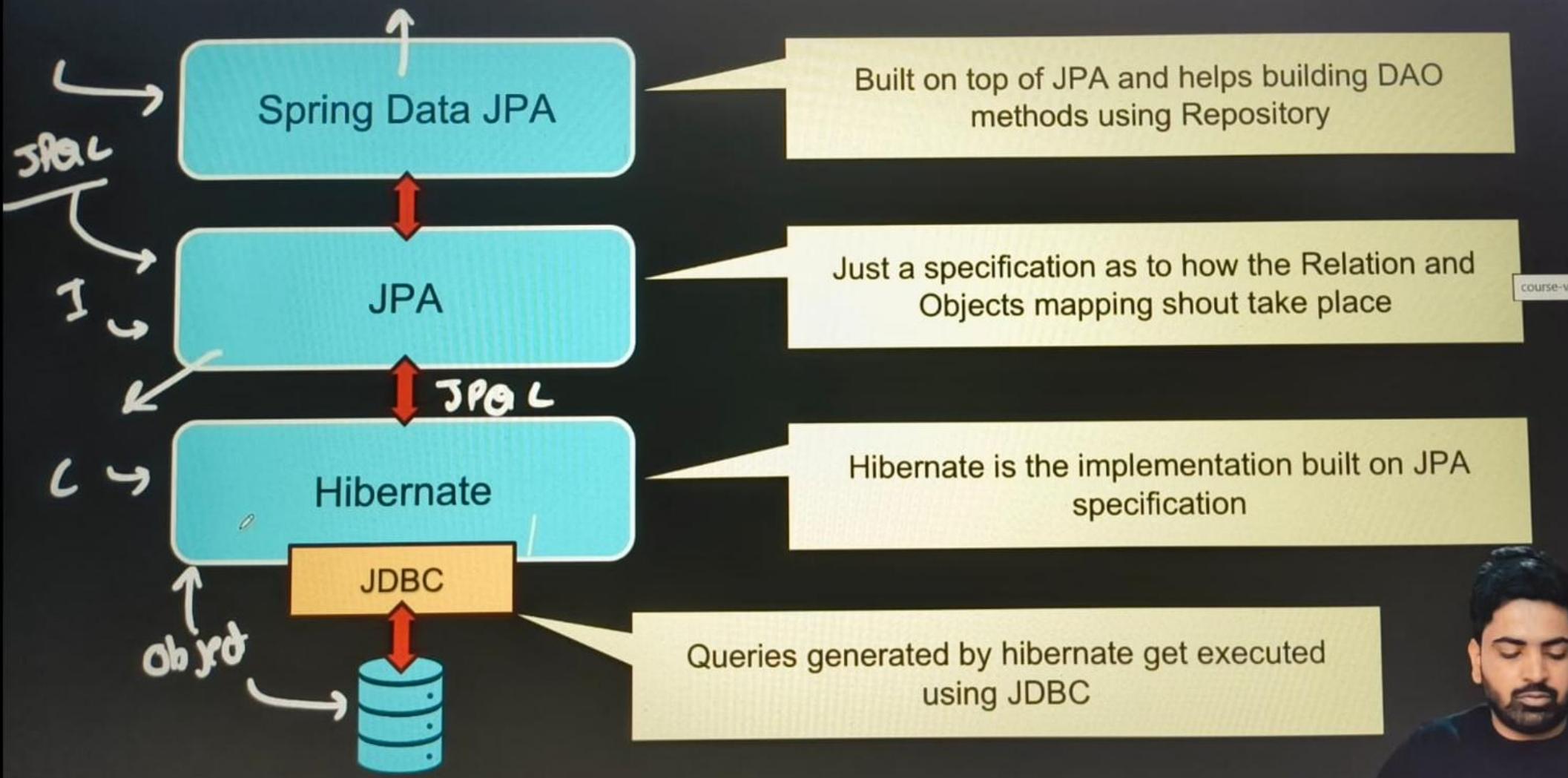
Hibernate ORM Mapping





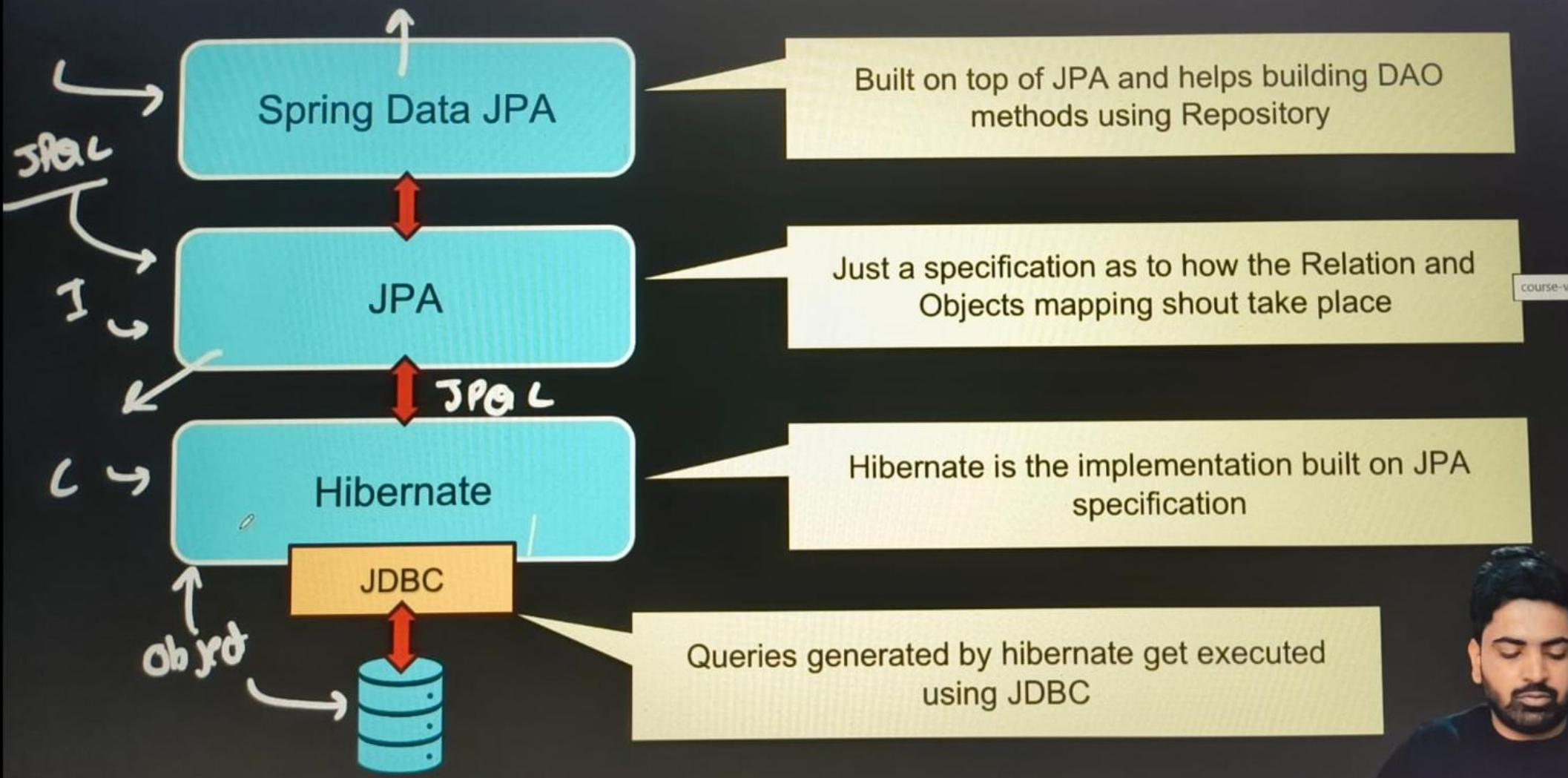
course-vid

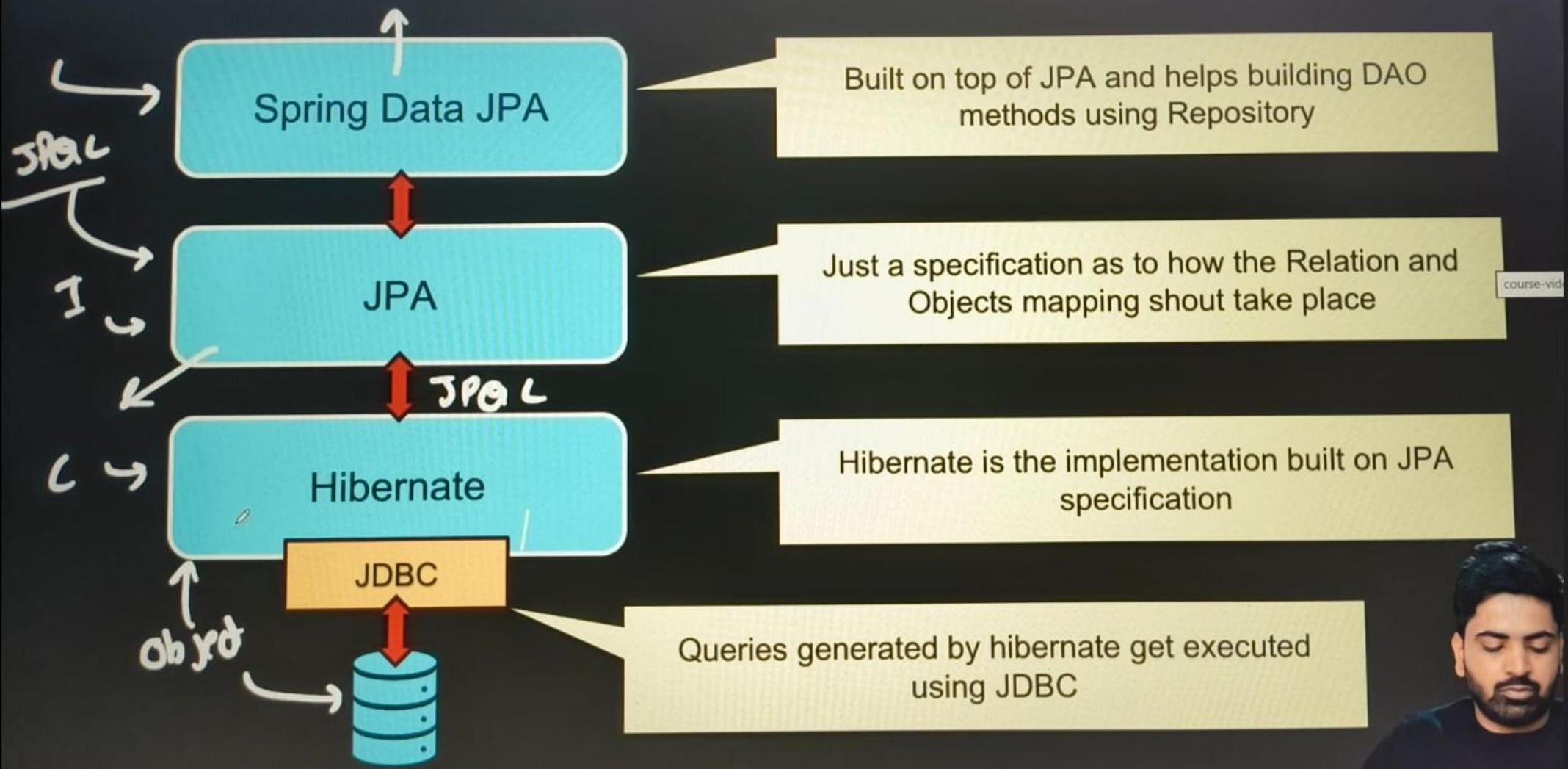
TED



course-vid

TED





course-vid

TED

explain me jdbc , Hibernate , JPA , JPQL , Spring data jpa , database and driver ...explain me all these how they all are created and who is build on the top of whom and its flow of working and on each what language is usedexplain in very detail so i can easily get the whole workings

Great question, Abhishek. This confusion is **very common**, and once you see the **layer-by-layer picture**, everything will *click* permanently.

I'll explain this from the ground up, answering all of these clearly:

- What is Database
- What is Driver
- What is JDBC
- What is JPA
- What is Hibernate
- What is JPQL
- What is Spring Data JPA
- Who is built on top of whom
- Flow of working (request → DB → response)
- Which language is used at each layer

No shortcuts. Let's build the mental model 💡 step by step.

1 DATABASE (Foundation – lowest level)

What is a Database?

A database is a software that stores data permanently.

Examples:

- MySQL
- PostgreSQL
- Oracle
- SQL Server



explain me jdbc , Hibernate , JPA , JPQL , Spring data jpa , database and driver ...explain me all these how they all are created and who is build on the top of whom and its flow of working and on each what language is usedexplain in very detail so i can easily get the whole workings

Great question, Abhishek. This confusion is **very common**, and once you see the **layer-by-layer picture**, everything will *click* permanently.

I'll explain this from the ground up, answering all of these clearly:

- What is Database
- What is Driver
- What is JDBC
- What is JPA
- What is Hibernate
- What is JPQL
- What is Spring Data JPA
- Who is built on top of whom
- Flow of working (request → DB → response)
- Which language is used at each layer

No shortcuts. Let's build the mental model 💡 step by step.

1 DATABASE (Foundation – lowest level)

What is a Database?

A database is a software that stores data permanently.

Examples:

- MySQL
- PostgreSQL
- Oracle
- SQL Server



What language does a Database understand?

👉 ONLY SQL

Examples:

```
sql

SELECT * FROM employee;
INSERT INTO employee VALUES (1, 'Abhi', 50000);
```

⭐ Important:

- Database does NOT understand Java
- Database does NOT know Spring / Hibernate / JPA

2 DATABASE DRIVER (Bridge between Java & DB)

What is a Driver?

A database driver is a translator between:

- Java program
- Database

Example:

- MySQL → mysql-connector-j
- PostgreSQL → postgresql-driver

Why is driver needed?

Because:

- Java speaks Java
- Database speaks SQL + network protocol

The driver:

- Converts Java calls → DB-specific protocol
- Converts DB result → Java objects

⭐ Without driver → Java cannot talk to database

What language does a Database understand?

👉 ONLY SQL

Examples:

```
sql

SELECT * FROM employee;
INSERT INTO employee VALUES (1, 'Abhi', 50000);
```

⭐ Important:

- Database does NOT understand Java
- Database does NOT know Spring / Hibernate / JPA

2 DATABASE DRIVER (Bridge between Java & DB)

What is a Driver?

A database driver is a translator between:

- Java program
- Database

Example:

- MySQL → mysql-connector-j
- PostgreSQL → postgresql-driver

Why is driver needed?

Because:

- Java speaks Java
- Database speaks SQL + network protocol

The driver:

- Converts Java calls → DB-specific protocol
- Converts DB result → Java objects

⭐ Without driver → Java cannot talk to database

3 JDBC (Java Database Connectivity)

What is JDBC?

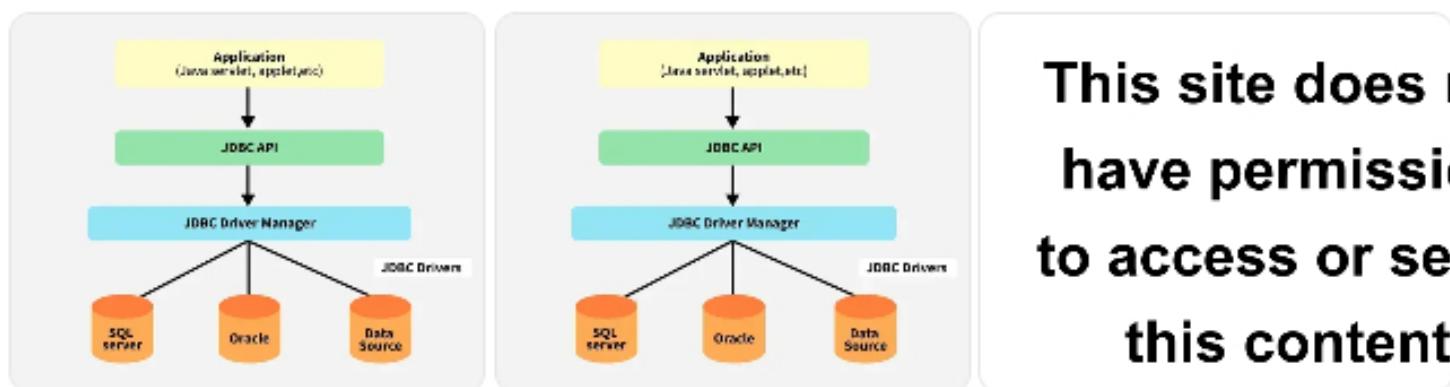
👉 JDBC is a Java API (interfaces + classes) that defines:

"How Java should talk to a database"

It is provided by Java (Oracle).

- ★ JDBC is NOT a framework
- ★ JDBC is very low-level

JDBC Architecture



Flow:

css

Copy code

Java **Code** → JDBC API → Driver → Database

Example JDBC Code

java

Copy code

```
Connection con = DriverManager.getConnection(url, user, pass);
PreparedStatement ps =
    con.prepareStatement("SELECT * FROM employee");
ResultSet rs = ps.executeQuery();
```

This site does not have permission to access or serve this content

Problems with JDBC ❌

- Too much boilerplate code
- Manual SQL writing
- Manual mapping:

```
java
```

```
emp.setName(rs.getString("name"));
```

 Copy code

- Manual transaction handling
- Hard to maintain

👉 This pain gave birth to ORM & JPA

4 ORM (Object Relational Mapping) – Concept

Problem JDBC couldn't solve

Java uses:

```
java

class Employee {
    Long id;
    String name;
}
```

 Copy code

Database uses:

```
pgsql

EMPLOYEE table (id, name)
```

 Copy code

ORM idea

Automatically map Java objects ↔ Database tables

Example:

```
java

Employee e = new Employee();
e.setName("Abhi");
save(e); // ORM converts this to SQL
```

 Copy code

ORM is a **concept**, not a library.

5 JPA (Java Persistence API)

What is JPA?

👉 JPA is a SPECIFICATION (rules)

Defined by:

- Java (JSR – Java Specification Request)
- ✖ JPA is NOT an implementation
- ✖ JPA has ONLY interfaces & annotations

What JPA defines

- How entities should look
- How persistence should work

Examples:

```
java

@Entity
@Table(name = "employee")
public class Employee {
    @Id
    private Long id;
}
```

```
java

EntityManager em;
em.persist(employee);
```

What JPA DOES NOT DO

- ✖ No SQL execution
- ✖ No DB connection
- ✖ No actual logic

👉 JPA needs an IMPLEMENTATION

6 Hibernate (JPA Implementation)

What is Hibernate?

- 👉 Hibernate is an ORM framework
- 👉 Hibernate IMPLEMENTS JPA
- 📌 Hibernate existed before JPA
- 📌 Later Hibernate said: "We will follow JPA rules"

So:

```
ini
```

JPA = Rules
Hibernate = Actual worker

 Copy code

Hibernate Responsibilities

- Converts Entity → SQL
- Manages JDBC internally
- Handles caching
- Handles lazy loading
- Transaction management

Example

```
java
```

em.persist(employee);

 Copy code

Hibernate internally does:

```
sql
```

INSERT INTO employee (id, name) VALUES (?, ?);

 Copy code

Relationship

```
css
```

Your **Code** → JPA → Hibernate → JDBC → Driver → 

 Copy code

7 JPQL (Java Persistence Query Language)

What is JPQL?

- 👉 JPQL is a query language defined by JPA
- ⭐ Looks like SQL
- ⭐ Works on Entity & fields, NOT tables

JPQL Example

```
java
```

 Copy

```
SELECT e FROM Employee e WHERE e.name = 'Abhi'
```

SQL Equivalent

```
sql
```

 Copy

```
SELECT * FROM employee WHERE name = 'Abhi';
```

Who converts JPQL → SQL?

- 👉 Hibernate
- ⭐ JPQL is database-independent
- ⭐ SQL is database-dependent

8 Spring Data JPA (Topmost Layer)

What is Spring Data JPA?

- 👉 A Spring module
- 👉 Built ON TOP OF JPA
- ⭐ Spring Data JPA does NOT replace JPA
- ⭐ It USES JPA + Hibernate

What problem it solves

- Removes DAO boilerplate
- Auto-generates queries
- Simplifies pagination, sorting

Example

```
java    
  
public interface EmployeeRepository  
    extends JpaRepository<Employee, Long> {  
  
    List<Employee> findByName(String name);  
}
```

- 💡 No SQL
- 💡 No JPQL
- 💡 No implementation class

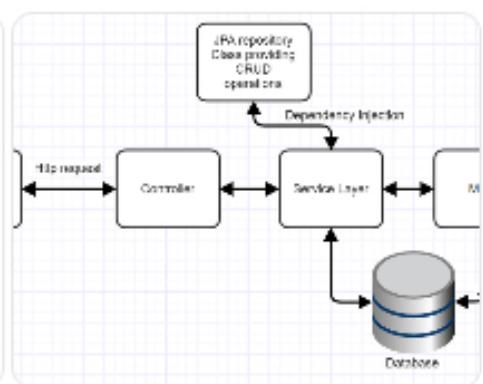
Spring Data JPA:

- Parses method name
- Creates JPQL
- Sends to Hibernate

9 Complete Layer Stack (MOST IMPORTANT)



This site does not have permission to access or serve this content



SCSS

 Copy code

Spring Data JPA

↓

JPA (Specification)

1

Hibernate (Implementation)

1

JDBC (Java API)

Page 1

-

10 Language Used at Each Layer

Layer	Language
Your Code	Java
Spring Data JPA	Java
JPA	Java (interfaces)
Hibernate	Java
JPQL	JPQL
JDBC	Java

1 1 Full Flow Example (End-to-End)

java

 Copy code

```
employeeRepository.save(emp);
```

Step-by-step:

1. Spring Data JPA
 - Calls JPA method
2. JPA
 - Defines what `save()` means
3. Hibernate
 - Generates SQL
 - Manages session & transaction
4. JDBC
 - Executes SQL
5. Driver
 - Converts call to DB protocol
6. Database
 - Stores data
 - Returns result
7. Back up the stack
 - Result → Object → Java



One-Line Summary (Memorize This)

JDBC talks to DB, Hibernate simplifies JDBC, JPA standardizes Hibernate, Spring Data JPA simplifies JPA.

interface which extends JpaRepository is Spring data jpa and the method defined in the interface is override by the jpa ...is it right

Short answer: ✗ Not exactly.

You are close, but two important things are mixed up.

Let's correct it precisely and clearly, because this point is core Spring Data JPA internals.

✗ What you said

"interface which extends JpaRepository is Spring Data JPA and the method defined in the interface is override by the JPA"

Problems in this statement

1. JPA does not override anything
2. Interfaces are not implemented by JPA
3. Spring Data JPA ≠ JPA

✓ Correct Mental Model (Very Important)

✓ 1. Interface extending JpaRepository

java

 Copy code

```
public interface EmployeeRepository  
    extends JpaRepository<Employee, Long> {  
}
```

- 👉 This interface is your code
- 👉 It belongs to Spring Data JPA
- 👉 It is NOT implemented by you
- 👉 It is NOT implemented by JPA

✓ 2. Who actually implements this interface?

- 👉 Spring Data JPA creates a runtime proxy class
 - 👉 That proxy implements your repository interface
 - ✖ Spring Data JPA, not JPA, does this.
-

🔥 Who does what (Crystal clear roles)

Layer	Role
Spring Data JPA	Creates repository implementation at runtime
JPA	Defines persistence rules & APIs
Hibernate	Executes persistence logic
JDBC	Talks to DB

🗣 What really happens at runtime

Step 1: Spring scans repositories

```
java
```

 Copy code

```
@EnableJpaRepositories
```

Spring:

- Finds `EmployeeRepository`
- Sees it extends `JpaRepository`

Step 2: Spring Data JPA creates a PROXY

Spring creates something like (conceptually):

java

 Copy code

```
class EmployeeRepositoryImpl  
    implements EmployeeRepository {  
  
    @Override  
    public Employee save(Employee entity) {  
        entityManager.persist(entity);  
    }  
}
```

★ This class is generated automatically at runtime

★ You never see it in code

Step 3: Who executes `persist()` ?

java

 Copy code

```
entityManager.persist(entity);
```

👉 `EntityManager` is JPA API

👉 Actual work is done by Hibernate

! Important correction

✗ “methods are overridden by JPA”

✓ “methods are implemented by Spring Data JPA using JPA APIs”

JPA never overrides your interface methods.

What about custom methods?

Example

java

 Copy code

```
List<Employee> findByName(String name);
```

What happens?

1. Spring Data JPA:

- Parses method name
- Builds JPQL

2. Hibernate:

- Converts JPQL → SQL

3. JDBC:

- Executes SQL

Simple Flow Diagram

scss

 Copy code

```
EmployeeRepository (interface)
```

```
↓
```

```
Spring Data JPA Proxy (runtime)
```

```
↓
```

```
EntityManager (JPA)
```

```
↓
```

```
Hibernate (implementation)
```

```
↓
```

```
JDBC
```

```
↓
```

```
Database
```

Final Correct Statement (You can memorize this)

The interface extending `JpaRepository` is part of Spring Data JPA.

Spring Data JPA creates a runtime implementation (proxy) of that interface and internally uses JPA's `EntityManager`, whose implementation is Hibernate.

JpaRepository class is also Spring data jpa

Yes — absolutely correct, with one small clarification.

Correct statement

JpaRepository is part of Spring Data JPA.

More precisely:

- JpaRepository is an interface
- It belongs to the Spring Data JPA module
- It is NOT part of JPA
- It is NOT part of Hibernate

Let's be very precise (this matters)

Where JpaRepository comes from

java

 Copy code

```
package org.springframework.data.jpa.repository;
```

👉 This package is Spring Data JPA

So:

java

 Copy code

```
public interface JpaRepository<T, ID>
    extends PagingAndSortingRepository<T, ID>,
           QueryByExampleExecutor<T> {
}
```

✓ It is 100% Spring Data JPA

✖ Common confusion (important to avoid)

Thing	Is it Spring Data JPA?
JpaRepository	✓ YES
CrudRepository	✓ YES
PagingAndSortingRepository	✓ YES
EntityManager	✗ NO (JPA)
@Entity	✗ NO (JPA)
HibernateSession	✗ NO (Hibernate)

How things connect (clean mental model)

scss

 Copy code

```
Your Repository Interface
    ↓ extends
JpaRepository ← (Spring Data JPA)
    ↓ uses
EntityManager ← (JPA API)
    ↓ implemented by
Hibernate      ← (ORM framework)
    ↓ uses
JDBC
    ↓
Database
```

One more important clarification

`JpaRepository` is NOT a class

It is an interface.

Spring Data JPA:

- Creates a runtime proxy
 - That proxy implements `JpaRepository`
 - And also implements your repository interface
-

Final one-line answer (perfect for interviews)

Yes, `JpaRepository` is an interface provided by Spring Data JPA, which internally uses JPA's `EntityManager` (implemented by Hibernate) to perform database operations.

but the JpaRepository if the interface then why we are using extends

Excellent question 🤔

This is exactly the right confusion to have when you truly start understanding Spring Data JPA.

Let's break it slowly and deeply, and by the end the word `extends` for interfaces will feel obvious.

1 First: `extends` is NOT only for classes

In Java:

Case	Keyword
class → class	<code>extends</code>
class → interface	<code>implements</code>
interface → interface	<code>extends</code>

👉 Interfaces always use `extends`, never `implements`.

2 What is JpaRepository really?

java

 Copy code

```
public interface JpaRepository<T, ID>
```

👉 It is an interface

👉 Provided by Spring Data JPA

And internally it already extends other interfaces:

java

 Copy code

```
public interface JpaRepository<T, ID>
    extends PagingAndSortingRepository<T, ID>,
        QueryByExampleExecutor<T> {
}
```

So `JpaRepository` itself is a composition of capabilities.

3 Why do WE use `extends JpaRepository` ?

When you write:

java Copy code

```
public interface EmployeeRepository
    extends JpaRepository<Employee, Long> {
}
```

You are saying:

"My repository inherits all method contracts of `JpaRepository`."

That means your interface automatically has:

- `save()`
- `findById()`
- `findAll()`
- `deleteById()`
- Pagination
- Sorting

👉 You are not writing implementations, only declaring capability.

4 But wait... if it's an interface, where is the code?

🔥 This is the key idea 🔥

Answer:

- 👉 Spring Data JPA provides the implementation at runtime
- Spring creates a proxy class
 - That class implements your interface
 - That class also implements `JpaRepository` internally

You don't see this class, but Spring uses it.



5 Analogy (very important)

Interface = Contract

java

 Copy code

```
interface Payment {  
    void pay();  
}
```

Now:

java

 Copy code

```
interface OnlinePayment extends Payment {  
}
```

No implementation yet — only capability inheritance.

Later:

java

 Copy code

```
class PaymentImpl implements OnlinePayment {  
    public void pay() { ... }  
}
```

- ❖ Same thing with Spring Data JPA
- ❖ Spring is writing `PaymentImpl` for you at runtime

6 Why NOT use `implements JpaRepository`?

Because you are not a class, you are an interface.

java

 Copy code

```
// ✗ Invalid
public interface EmployeeRepository
    implements JpaRepository<Employee, Long> {
}
```

Java compiler error 

Only classes can `implement`.

7 What exactly does `extends` give you?

It gives:

- ✓ Method signatures
- ✓ Generic typing (`Employee`, `Long`)
- ✓ Repository behavior declaration

It does NOT give:

- ✗ Method bodies
- ✗ SQL
- ✗ DB logic

8 Complete Flow (One more time)

kotlin

 Copy code

```
EmployeeRepository (interface)
    ↓ extends
JpaRepository (interface)
    ↓ implemented by
Spring Data JPA Proxy (runtime class)
    ↓ uses
EntityManager (JPA)
    ↓ implemented by
Hibernate
    ↓
Database
```

9 Final one-line answer (memorize this)

We use `extends JpaRepository` because both are interfaces, and extending an interface means inheriting method contracts, which Spring Data JPA later implements automatically at runtime using a proxy.

i wanna know the method we define in our Repository interface ,are
define why whom mean who write the logic for that method

Perfect 👍

This is the **MOST IMPORTANT** internal concept of Spring Data JPA.

Once you understand this, all repository magic disappears and it becomes logical.

I'll answer exactly what you asked:

"The methods we define in our Repository interface – who defines them, why, and who writes the logic?"

Big Picture (1-minute overview)

There are 3 different cases of methods in a repository interface:

- 1 Methods already defined in `JpaRepository`
- 2 Methods you define by name convention
- 3 Methods you define with `@Query` or custom implementation

Each case has a different "who writes the logic" answer.

CASE 1

Methods coming from JpaRepository

Example:

java

 Copy code

```
employeeRepository.save(emp);  
employeeRepository.findById(1L);
```

Who defined these methods?

👉 Spring Data JPA team

Who wrote the logic?

👉 Spring Data JPA (internally)

How?

Internally these methods call:

java

 Copy code

```
EntityManager.persist()  
EntityManager.find()
```

👉 EntityManager is JPA

👉 Hibernate executes the actual SQL

★ You only declare usage, not implementation.

CASE 2

Methods YOU define using method name (Query Creation)

Example:

```
java Copy code
List<Employee> findByName(String name);
Employee findByEmailAndStatus(String email, String status);
```

Who defined these methods?

👉 YOU

Who wrote the logic?

👉 Spring Data JPA (auto-generated)

How exactly does Spring do it?

At application startup:

1. Spring scans repository interfaces
2. Sees method name: `findByName`
3. Splits it:
 - `find` → query type
 - `ByName` → entity field
4. Builds JPQL

```
jpql Copy code
SELECT e FROM Employee e WHERE e.name = :name
```

5. Hibernate converts JPQL → SQL

- 👉 You write the method name
- 👉 Spring writes the query logic

CASE 3

Methods using @Query

Example:

java

 Copy code

```
@Query("SELECT e FROM Employee e WHERE e.salary > :salary")
List<Employee> findRichEmployees(double salary);
```

Who defined the method?

👉 YOU

Who wrote the logic?

👉 YOU (query part)

Who executes it?

👉 Spring Data JPA + Hibernate

- ❖ Here, Spring does NOT generate the query
- ❖ It only executes what you wrote

CASE 4

Fully Custom Methods (You write logic)

Example:

java

 Copy code

```
public interface EmployeeRepositoryCustom {  
    List<Employee> complexLogic();  
}
```

java

 Copy code

```
public class EmployeeRepositoryImpl  
    implements EmployeeRepositoryCustom {  
  
    @PersistenceContext  
    private EntityManager em;  
  
    @Override  
    public List<Employee> complexLogic() {  
        // your Java logic  
    }  
}
```

java

 Copy code

```
public interface EmployeeRepository  
    extends JpaRepository<Employee, Long>,  
        EmployeeRepositoryCustom {  
}
```

Who writes the logic?

👉 YOU (100%)

Spring:

- Just wires it together

🔥 Summary Table (Very Important)

Method Type	Who declares method	Who writes logic
<code>save()</code> , <code>findAll()</code>	Spring Data JPA	Spring Data JPA
<code>findByName()</code>	You	Spring Data JPA
<code>@Query</code> methods	You	You (query only)
Custom impl methods	You	You

Mental Model (Remember This)

You only write method signatures.
Spring Data JPA decides whether it can generate logic or needs you to write it.

One-line interview-ready answer

Methods in a Spring Data JPA repository are either implemented internally by Spring Data JPA (for standard CRUD and derived queries) or executed using queries written by the developer, while the actual database interaction is performed by Hibernate via JPA's EntityManager.

what is persistence context

Excellent — this is THE core concept of JPA.

If you truly understand Persistence Context, JPA/Hibernate will stop feeling magical.

I'll explain it step by step, with real-life analogies, code examples, and clear rules.

1 What is Persistence Context? (Simple definition)

Persistence Context is a memory space (managed by JPA) that keeps track of all entities currently managed by the application.

In short:

- It is a cache
 - It is a tracking mechanism
 - It is a unit of work
- 📌 Defined by JPA
📌 Implemented by Hibernate
-

2 What does “managed entity” mean?

An entity is managed when:

- It is inside the persistence context
- JPA/Hibernate tracks every change made to it

Example:

```
java Copy code
Employee e = entityManager.find(Employee.class, 1L);
e.setSalary(60000);
```

- 👉 You never call `update()`
👉 Hibernate automatically updates DB

This is possible ONLY because of Persistence Context.



3 Why does Persistence Context exist?

Without persistence context:

- Every change would need explicit SQL
- No automatic updates
- No caching
- Poor performance

Persistence Context enables:

- ✓ First-level cache
- ✓ Dirty checking
- ✓ Write-behind
- ✓ Identity guarantee

4 Persistence Context = First-Level Cache

java

 Copy code

```
Employee e1 = em.find(Employee.class, 1L);
Employee e2 = em.find(Employee.class, 1L);
```

 Result:

- Only ONE SQL query
- `e1 == e2 → true`

Why?

 Same entity is returned from persistence context cache.

4 Persistence Context = First-Level Cache

java

 Copy code

```
Employee e1 = em.find(Employee.class, 1L);
Employee e2 = em.find(Employee.class, 1L);
```



Result:

- Only ONE SQL query
- `e1 == e2 → true`

Why?

 Same entity is returned from persistence context cache.

5 Dirty Checking (Most Important Feature)

java

 Copy code

```
@Transactional
public void updateSalary() {
    Employee e = em.find(Employee.class, 1L);
    e.setSalary(70000);
}
```

What happens:

1. Entity loaded → stored in persistence context
2. Field changed
3. On transaction commit:
 - Hibernate compares old vs new state
 - Generates `UPDATE` SQL automatically

 This automatic change detection is **dirty checking**

6 Write-Behind (Delayed SQL Execution)

java

 Copy code

```
em.persist(emp1);  
em.persist(emp2);
```

Hibernate does:

- Stores entities in persistence context
- Does NOT immediately execute SQL

At flush/commit:

sql

 Copy code

```
INSERT INTO employee ...  
INSERT INTO employee ...
```

👉 Improves performance via batching

7 Entity Lifecycle States (Defined by JPA)

State	Meaning
Transient	Not in persistence context
Managed	Inside persistence context
Detached	Was managed, now not
Removed	Scheduled for delete

8 Persistence Context vs Database

Persistence Context	Database
In memory	On disk
Temporary	Permanent
Java objects	Rows
Fast	Slower

★ Persistence context syncs with DB at:

- Transaction commit
- Explicit `flush()`

9 How many persistence contexts exist?

In Spring Boot (default):

- One persistence context per transaction

java

 Copy code

```
@Transactional  
public void method() {  
    // one persistence context  
}
```

10 Who manages Persistence Context?

- Defined by JPA
- Created by EntityManager
- Implemented by Hibernate
- Lifecycle managed by Spring



1 1 Real-life analogy 🧠

Persistence Context = Git staging area

Git

JPA

Working tree

Java objects

Staging area

Persistence Context

Commit

Transaction commit

Remote repo

Database

1 2 One-line interview answer

Persistence Context is a JPA-managed memory area that stores and tracks managed entities, enabling caching, automatic change detection, and efficient database synchronization.

Hibernate

Hibernate is a powerful, high-performance Object-Relational Mapping (ORM) framework that is widely used with Java. It provides a framework for mapping an object-oriented domain model to a relational database.

Hibernate is one of the implementations of the Java Persistence API (JPA), which is a standard specification for ORM in Java.



JPA (Java Persistence API)

JPA is a specification for object-relational mapping (ORM) in Java.

It defines a set of interfaces and annotations for mapping Java objects to database tables and vice versa.

JPA itself is just a set of guidelines and does not provide any implementation. The implementation of JPA is provided by ORM frameworks such as Hibernate, EclipseLink, and OpenJPA.

So Basically...

JPA Provides a standard for ORM in Java applications, ensuring that developers can switch between different JPA providers without changing their code.

And

Hibernate is one such JPA Provider.

However,

Hibernate is a specific implementation of JPA and a powerful ORM framework on its own. It offers additional features and optimizations beyond the JPA specification, making it a popular choice for ORM in Java applications.

Project ▾

- jpATuts D:\jpATuts
 - > .idea
 - > .mvn
 - src
 - > main
 - > java
 - com.cosingshuttle.jpaTutorial.jpATuts
 - > entities
 - Product

JPATutorialApplication

 - > resources
 - > test
 - .gitattributes
 - .gitignore
 - HELP.md
 - mvnw
 - mvnw.cmd
 - pom.xml

> External Libraries

> Scratches and Consoles

m pom.xml (jpATuts) c Product.java ×

```
1 package com.cosingshuttle.jpaTutorial.jpATuts.entities;
2
3 import jakarta.persistence.Entity;
4 import jakarta.persistence.GeneratedValue;
5 import jakarta.persistence.GenerationType;
6 import jakarta.persistence.Id;
7 import lombok.AllArgsConstructorConstructor;
8 import lombok.Builder;
9 import lombok.Data;
10 import lombok.NoArgsConstructor;
11
12 import java.math.BigDecimal;
13 import java.time.LocalDateTime;
14
15
16 @Data
17 @AllArgsConstructor
18 @NoArgsConstructor
19 @Builder
20 @Entity
21 public class Product {
22
23     @Id
24     @GeneratedValue(strategy= GenerationType.IDENTITY)
25     @GeneratedValue(strategy= GenerationType.IDENTITY)
26     private Long id;
27     private String sku;
28     private String title;
29     private BigDecimal price;
30     private Integer quantity;
31     private LocalDateTime createdAt;
32     private LocalDateTime updatedAt;
33 }
```

Common Hibernate Configurations

1. `spring.jpa.hibernate.ddl-auto=update/create/validate/create-drop/none`
2. `spring.jpa.show-sql=true`
3. `spring.jpa.properties.hibernate.format_sql=true`
4. `spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect` (Optional)

Common Hibernate Configurations

1. `spring.jpa.hibernate.ddl-auto=update/create/validate/create-drop/none`
2. `spring.jpa.show-sql=true`
3. `spring.jpa.properties.hibernate.format_sql=true`
4. `spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect` (Optional)

explain all these hibernate configuration in detail

1. spring.jpa.hibernate.ddl-auto=update/create/validate/create-drop/none

2. spring.jpa.show-sql=true

3. spring.jpa.properties.hibernate.format_sql=true

4. spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect (Optional)

Excellent topic 🔥

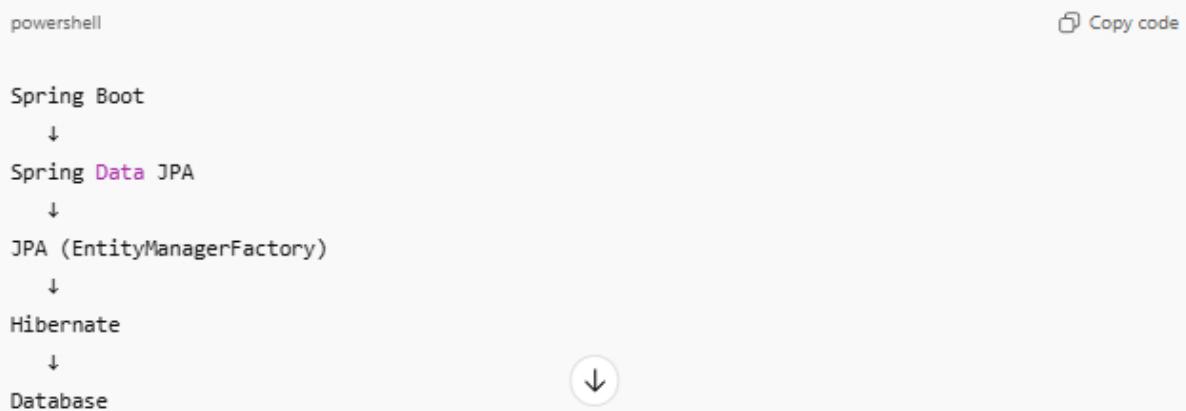
These Hibernate configuration properties decide how Hibernate behaves at startup and runtime.

I'll explain each property deeply, including what happens internally, when to use which value, and real-world best practices.

I'll also connect everything to JPA + Hibernate + Spring Boot, so nothing feels isolated.

Big Picture First (Context)

When a Spring Boot app starts:



1 spring.jpa.hibernate.ddl-auto

What does DDL mean?

DDL = Data Definition Language

Examples:

sql

 Copy code

```
CREATE TABLE  
ALTER TABLE  
DROP TABLE
```

This property tells Hibernate:

“What should I do with the database schema when the application starts?”

Values explained in DETAIL

- `create`

👉 Drops existing tables

👉 Creates new tables every time app starts

What happens internally:

sql

 Copy code

```
DROP TABLE employee;  
CREATE TABLE employee (...);
```

✗ Data is LOST every restart

✓ Use when:

- Learning
- POC
- Early development

✗ Never use in production

- `create-drop`

👉 Same as `create`

👉 PLUS: Drops tables when app shuts down

Startup:

sql

 Copy code

```
CREATE TABLE employee (...);
```

Shutdown:

sql

 Copy code

```
DROP TABLE employee;
```

✓ Use when:

- Integration testing
- Temporary in-memory DBs



- **update** (MOST USED IN DEV)

👉 Compares entity vs table

👉 Applies only necessary changes

Examples:

- Add column → Hibernate adds it
 - Rename column → ✗ NOT handled well
 - Remove column → ✗ NOT removed
- ✗ Data is preserved

✓ Use when:

- Local development
- Rapid iteration

✗ Not recommended for production (unpredictable)

- **validate**

👉 Hibernate checks:

- Table exists?
- Columns exist?
- Types match?

If mismatch:

✗ App fails to start

✓ Use when:

- Schema managed by DBA
- Production environments

- none

👉 Hibernate does nothing

✓ Use when:

- You manage schema via Flyway/Liquibase
- Strict production control

🔥 Summary Table

Value	DB Changed?	Data Safe?	Usage
create	Yes	✗ No	Learning
create-drop	Yes	✗ No	Testing
update	Yes	✓ Mostly	Dev
validate	No	✓ Yes	Prod
none	No	✓ Yes	Prod

2 spring.jpa.show-sql=true

What does it do?

- 👉 Logs SQL queries executed by Hibernate

Example output:

sql

 Copy code

```
select * from employee where id=?
```

- ✖ Shows raw SQL, not formatted

Important points

- ✓ Useful for:

- Debugging
- Learning Hibernate behavior
- Understanding JPQL → SQL

- ✖ Problems:

- Logs too much
- Performance hit
- Exposes SQL in logs

- ✖ Not recommended in production
-

3 spring.jpa.properties.hibernate.format_sql=true

What does it do?

👉 Beautifies SQL logs

Without format:

sql

 Copy code

```
select id,name,salary from employee where id=?
```

With format:

sql

 Copy code

```
select
    id,
    name,
    salary
from
    employee
where
    id = ?
```

⭐ Only works if `show-sql=true`

✓ Dev-only feature

4 spring.jpa.properties.hibernate.dialect

What is a Dialect?

A dialect tells Hibernate:

"Which database am I talking to, and how does it write SQL for it?"

Because:

- MySQL
- PostgreSQL
- Oracle

All have different SQL flavors.

Example

```
properties  
  
spring.jpa.properties.hibernate.dialect=  
org.hibernate.dialect.MySQL5Dialect
```

Hibernate uses dialect to:

- Choose correct data types
 - Handle pagination
 - Use correct functions
 - Generate proper DDL
-

Is dialect mandatory?

Is dialect mandatory?

✗ No (in Spring Boot)

Spring Boot:

- Auto-detects DB from JDBC URL
- Sets dialect automatically
- ✓ Explicit dialect needed when:
 - Old DB versions
 - Custom DB behavior
 - Non-standard DB

How these properties work together

This site does not have permission to access or serve this content

This site does not have permission to access or serve this content

This site does not have permission to access or serve this content

App Startup:

1. Spring reads `application.properties`
2. Hibernate initializes
3. `ddl-auto` decides schema action
4. Dialect decides SQL syntax
5. SQL executed via JDBC
6. Logs controlled by `show-sql`

Best Practices (Real Projects)

Local Development

```
properties  
  
spring.jpa.hibernate.ddl-auto=update  
spring.jpa.show-sql=true  
spring.jpa.properties.hibernate.format_sql=true
```

 Copy code

Production

```
properties  
  
spring.jpa.hibernate.ddl-auto=validate  
spring.jpa.show-sql=false
```

 Copy code

Use:

- Flyway or Liquibase for schema changes

One-line interview answers

ddl-auto

Controls Hibernate's automatic database schema generation.

show-sql

Displays SQL queries executed by Hibernate.

format_sql

Formats SQL output for readability.

dialect

Tells Hibernate how to generate database-specific SQL.

spring.jpa.hibernate.ddl-auto=update/create/validate/create-drop/none i don't get this one can you explain each with example

No problem !

This property is confusing until you see concrete examples.

Let's explain each value with the SAME entity and SAME table, so you can clearly see what changes in DB.

First: The Entity we'll use (same for all examples)

java

 Copy code

```
@Entity  
@Table(name = "employee")  
public class Employee {  
  
    @Id  
    @GeneratedValue  
    private Long id;  
  
    private String name;  
  
    private int salary;  
}
```

Assume database is MySQL.

1 `ddl-auto = create`

What Hibernate does

- 👉 Drops the table
- 👉 Creates a new table

Scenario

- Table already exists with data
- Application starts

SQL executed

sql

 Copy code

```
DROP TABLE employee;
CREATE TABLE employee (
    id BIGINT NOT NULL,
    name VARCHAR(255),
    salary INT,
    PRIMARY KEY (id)
);
```

Result

- ✖ All existing data is deleted

When you change entity

If you add:

java

 Copy code

```
private String email;
```

On next restart:

```
sql Copy code  
  
DROP TABLE employee;  
CREATE TABLE employee (  
    id BIGINT,  
    name,  
    salary,  
    email  
);
```

Use case

- ✓ Learning
 - ✓ Early development
 - ✗ Production
-

2 ddl-auto = create-drop

What Hibernate does

- 👉 On startup: same as `create`
- 👉 On shutdown: drops tables

Startup SQL

```
sql Copy code  
  
CREATE TABLE employee (...);
```

Shutdown SQL

```
sql Copy code  
  
DROP TABLE employee;
```

Result

- DB is empty after app stops

Use case

- ✓ Integration tests
- ✓ Temporary DB

3 `ddl-auto = update` (**MOST CONFUSING**)

What Hibernate does

- 👉 Compares entity vs existing table
- 👉 Applies safe changes only

Case A: Table does NOT exist

sql

 Copy code

```
CREATE TABLE employee (...);
```

Case B: You add a new field

Entity changed:

java

 Copy code

```
private String department;
```

Hibernate runs:

sql

 Copy code

```
ALTER TABLE employee ADD department VARCHAR(255);
```

- ✓ Data preserved

Case C: You remove a field

You remove:

java

 Copy code

```
private int salary;
```

Hibernate does:

 NOTHING

Column stays in DB.

Case D: You rename a field

java

 Copy code

```
private String fullName; // earlier: name
```

Hibernate thinks:

- `name` removed
- `fullName` added

Result:

sql

 Copy code

```
ALTER TABLE employee ADD full_name VARCHAR(255);
```

 Old `name` column still exists

 Data duplication risk

Why `update` is dangerous

- No rollback
- No column removal
- Wrong assumptions

Use case

- ✓ Local development
- ✗ Production

4 `ddl-auto = validate`

What Hibernate does

- 👉 Does NOT change DB
- 👉 Only checks mapping

Example: Table exists but column missing

Entity:

```
java  
  
private String email;
```

DB:

```
sql  
  
employee(id, name, salary)
```

Result

- ✗ Application fails to start

Error:

```
pgsql  
  
Column 'email' not found
```

Use case

- ✓ Production
- ✓ Schema managed manually

5 `ddl-auto = none`

What Hibernate does

👉 Absolutely NOTHING

- No create
- No update
- No validate

Hibernate just runs queries.

If table missing?

✗ Runtime error when query runs

Use case

- ✓ Production
- ✓ Flyway / Liquibase users

🔥 Side-by-side Comparison (Best Way to Remember)

Value	Startup Action	Schema Change	Data
create	Drop + Create	Yes	✗ Lost
create-drop	Create	Yes	✗ Lost
update	Alter (limited)	Yes	✓ Mostly
validate	Check only	No	✓
none	Do nothing	No	✓



One-Line Memory Trick

- **create** → *fresh DB every start*
 - **create-drop** → *fresh DB, delete on stop*
 - **update** → *try to adjust*
 - **validate** → *only check*
 - **none** → *ignore DB*
-



pom.xml (jpaTuts)

© Product.java

application.properties

```
spring.application.name=jpaTutorial  
spring.jpa.hibernate.ddl-auto=update  
spring.jpa.show-sql=true  
spring.jpa.properties.hibernate.format_sql=true
```

1 What is `@Entity`?

java

 Copy code

```
@Entity  
public class Employee {  
}
```

`@Entity` tells JPA:

"This Java class represents a table in the database, and its objects represent rows."

Without `@Entity`:

- JPA ignores the class
- Hibernate will not create a table
- You cannot persist it

2 What happens when Hibernate sees `@Entity`?

At application startup:

1. Hibernate scans classpath
2. Finds `@Entity` classes
3. Registers them as managed entities
4. Maps:
 - Class → Table
 - Fields → Columns

This mapping is stored in metadata.

3 Minimum requirements for `@Entity`

✓ Must have:

java

 Copy code

```
@Entity  
public class Employee {  
  
    @Id  
    private Long id;  
}
```

Rules defined by JPA:

1. Must have `@Id` (primary key)
2. Must have a no-argument constructor
3. Class must not be final
4. Persistent fields must not be final

4 Table mapping rules

Default table name

java

 Copy code

```
@Entity  
public class Employee {}
```

👉 Table name = `employee` (or `Employee`, depends on dialect)

Custom table name

java

 Copy code

```
@Entity  
@Table(name = "emp_table")  
public class Employee {}
```

5 Field → Column mapping

Default mapping

java

```
private String name;
```

Cop:

👉 Column = name

Custom column mapping

java

```
@Column(name = "emp_name", nullable = false)  
private String name;
```

Cop:

6 What fields are persisted?

By default:

- ✓ All non-static
- ✓ All non-transient

Ignored fields

java

```
@Transient  
private int tempValue;  
  
static String company;
```

Cop:

7 Entity lifecycle (VERY IMPORTANT)

Once annotated with `@Entity`, objects can be in these states:

State	Meaning
Transient	New object, not in DB
Managed	Attached to persistence context
Detached	Was managed, now not
Removed	Marked for deletion

This lifecycle is only possible because of `@Entity`.

8 How `@Entity` works with Persistence Context

java

```
Employee e = em.find(Employee.class, 1L);
e.setSalary(50000);
```

- `e` is managed
- Hibernate tracks changes
- SQL UPDATE happens automatically

9 Common mistakes ✗

✗ Missing @Id

bash

 Copy code

Not a managed type

✗ Final class

java

 Copy code

```
public final class Employee {}
```

✗ No default constructor

java

 Copy code

```
public Employee(String name) {}
```

10 Difference: @Entity vs @Embeddable

@Entity

@Embeddable

Has table

No table

Has primary key

No primary key

Stored separately

Stored inside entity

1 1 One-line interview answer

`@Entity` marks a Java class as a persistent JPA entity, meaning it is mapped to a database table and managed by the persistence context.

@Column

@Column is one of the most important JPA mapping annotations because it controls how a Java field is stored in a database column.

I'll explain it slowly and completely, with rules, examples, defaults, and real-world usage, so it sticks.

1 What is @Column ?

java

 Copy code

```
@Column  
private String name;
```

@Column tells JPA how a Java field maps to a database column.

- ★ It is part of JPA
- ★ It is optional
- ★ If you don't use it → JPA uses defaults

2 Default behavior (No `@Column`)

java

 Copy code

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

JPA automatically maps:

pgsql

 Copy code

```
Field name → column name  
String      → VARCHAR
```

So DB column:

sql

 Copy code

```
name VARCHAR(255)
```

3 When do we need `@Column`?

Use `@Column` when you want to:

- Change column name
- Add constraints
- Control length / nullability
- Make column unique
- Control insert/update behavior

4 Most important `@Column` attributes (with examples)

- ◆ `name`

Change column name

```
java  
  
@Column(name = "emp_name")  
private String name;
```

DB column:

```
sql  
  
emp_name
```

- ◆ `nullable`

Control NULL constraint

```
java  
  
@Column(nullable = false)  
private String email;
```

DB:

```
sql  
  
email VARCHAR(255) NOT NULL
```

- ❖ Affects DDL generation
- ❖ Does not do validation by itself

◆ unique

java

```
@Column(unique = true)  
private String email;
```

DB:

sql

```
email VARCHAR(255) UNIQUE
```

- ★ For single-column uniqueness only
- ★ For multi-column → use `@Table(uniqueConstraints=...)`

◆ length

java

```
@Column(length = 100)  
private String name;
```

DB:

sql

```
name VARCHAR(100)
```

- ★ Applies to `String` only

◆ `columnDefinition` (advanced)

java

```
@Column(columnDefinition = "TEXT")
private String description;
```

👉 Hibernate uses exact SQL you write

⚠ DB-specific

⚠ Avoid unless necessary

◆ `insertable / updatable`

java

```
@Column(insertable = false, updatable = false)
private LocalDate createdAt;
```

Hibernate:

- Won't include column in INSERT
- Won't include column in UPDATE

Used for:

- DB-generated columns
- Read-only fields

- ◆ **precision & scale (for decimals)**

java

```
@Column(precision = 10, scale = 2)  
private BigDecimal salary;
```



DB:

sql

```
salary DECIMAL(10,2)
```



5 @Column vs @Transient

@Column

@Transient

Stored in DB

Not stored

Has column

No column

Persistent

Non-persistent

6 Common mistakes ✗

✗ Expecting validation

```
java
```

```
@Column(nullable = false)
```

This does NOT validate in Java.

For validation, use:

```
java
```

```
@NotNull
```

```
@NotBlank
```

✗ Using `columnDefinition` everywhere

- Breaks portability
- Hard to maintain

7 Real-world example (best practice)

```
java Copy code  
  
@Entity  
@Table(  
    name = "employee",  
    uniqueConstraints = {  
        @UniqueConstraint(columnNames = "email")  
    }  
)  
public class Employee {  
  
    @Id  
    @GeneratedValue  
    private Long id;  
  
    @Column(name = "emp_name", length = 100, nullable = false)  
    private String name;  
  
    @Column(nullable = false, unique = true)  
    private String email;  
  
    @Column(precision = 10, scale = 2)  
    private BigDecimal salary;  
}
```

8 When NOT to use `@Column`

You can skip it when:

- Defaults are fine
- Column name = field name
- No constraints needed

9 One-line interview answer

`@Column` is a JPA annotation used to customize  mapping of an entity field to a database column, including name, constraints, and SQL characteristics.

@CreationTimestamp and @UpdateTimestamp

@CreationTimestamp and @UpdateTimestamp are Hibernate-specific annotations (not pure JPA) used to automatically manage audit timestamps.

I'll explain what they are, why they exist, how they work internally, examples, rules, and best practices.

1 What are @CreationTimestamp and @UpdateTimestamp ?

They automatically populate date/time fields without you writing code.

Annotation	Purpose
@CreationTimestamp	Sets value only once, when the entity is first INSERTed
@UpdateTimestamp	Updates value every time the entity is UPDATED

★ Provided by Hibernate

★ Package:

java

 Copy code

```
org.hibernate.annotations
```

2 Why do we need them?

Without these annotations, you would write code like:

java

 Copy code

```
employee.setCreatedAt(LocalDateTime.now());  
employee.setUpdatedAt(LocalDateTime.now());
```

Problems:

- Repeated code everywhere
- Easy to forget
- Error-prone

 Hibernate does it automatically and consistently

3 Basic Example (Most common use)

java

 Copy code

```
@Entity  
public class Employee {  
  
    @Id  
    @GeneratedValue  
    private Long id;  
  
    private String name;  
  
    @CreationTimestamp  
    private LocalDateTime createdAt;  
  
    @UpdateTimestamp  
    private LocalDateTime updatedAt;  
}
```

What happens?

On INSERT

sql

 Copy code

```
INSERT INTO employee (...) VALUES (...);
```

Hibernate sets:

ini

 Copy code

```
createdAt = current timestamp  
updatedAt = current timestamp
```

On UPDATE

sql

 Copy code

```
UPDATE employee SET name=?, updated_at=? WHERE id=?
```

Hibernate sets:

ini

 Copy code

```
updatedAt = current timestamp
```

4 When exactly are they set?

Annotation

`@CreationTimestamp`

When value is set

At INSERT time only

`@UpdateTimestamp`

At every UPDATE

5 Supported field types

You can use them on:

```
java  
  
LocalDateTime  
LocalDate  
Instant  
OffsetDateTime  
java.util.Date
```

 Copy code

 Recommended: `LocalDateTime`

6 Are these DB-side or Java-side timestamps?

 Java-side (Hibernate-side) by default.

Hibernate generates timestamp using:

```
java  
  
LocalDateTime.now()
```

 Copy code

 Timezone = JVM timezone

7 Important rules & constraints

 Do NOT manually set values

```
java  
  
employee.setCreatedAt(...); //  don't do this
```

 Copy code

Hibernate may override or conflict.

 Do NOT combine with `@Column(insertable = false, updatable = false)`

Hibernate needs to write these columns.

8 Difference vs Database timestamps

DB-managed timestamp example

sql

```
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
```

Hibernate-managed timestamp

java

```
@CreationTimestamp  
private LocalDateTime createdAt;
```

Hibernate

App-controlled

Portable

Easier

Database

DB-controlled

DB-specific

More accurate time

★ Choose one approach, not both.

9 Difference vs JPA auditing (@CreatedDate)

Hibernate

Spring Data JPA

@CreationTimestamp

@CreatedDate

@UpdateTimestamp

@LastModifiedDate

Simple

More flexible

Hibernate-only

JPA-compliant

Spring auditing requires:

java

```
@EnableJpaAuditing
```

10 Best Practices (Real Projects)

✓ Use when:

- Hibernate is your ORM
- Simple audit fields
- No cross-service time sync needed

✗ Avoid when:

- Multiple services share DB
- DB must control timestamps

1 1 One-line interview answers

10 Best Practices (Real Projects)

✓ Use when:

- Hibernate is your ORM
- Simple audit fields
- No cross-service time sync needed

✗ Avoid when:

- Multiple services share DB
- DB must control timestamps

1 1 One-line interview answers

`@CreationTimestamp`

Automatically sets the creation timestamp of an entity at insert time.

`@UpdateTimestamp`

Automatically updates the modification timestamp of an entity whenever it is updated.

1 2 Common mistake (VERY important)

✗ Thinking they are JPA annotations

✓ They are Hibernate-specific

Table Annotation

```
@Table(  
    name = "employees",  
    catalog = "employee_catalog",  
    schema = "hr",  
    uniqueConstraints = {  
        @UniqueConstraint(columnNames = {"email"})  
    },  
    indexes = {  
        @Index(name = "idx_name", columnList = "name"),  
        @Index(name = "idx_department", columnList = "department")  
    }  
)
```

```
@Table(  
    name = "employees",  
    catalog = "employee_catalog",  
    schema = "hr",  
    uniqueConstraints = {  
        @UniqueConstraint(columnNames = {"email"})  
    },  
    indexes = {  
        @Index(name = "idx_name", columnList = "name"),  
        @Index(name = "idx_department", columnList = "department")  
    }  
)
```

Great, this is a rich `@Table` example 👍

I'll explain every attribute line-by-line, with what it means, when it is used, what SQL is generated, and real-world cautions.

1 What is @Table?

java

 Copy code

```
@Table(...)
```

`@Table` is a JPA annotation used to customize how an `@Entity` maps to a database table.

- ❖ Optional
- ❖ Used only when defaults are not enough
- ❖ Part of JPA (javax/jakarta.persistence)

2 Your complete example (for reference)

java

 Copy code

```
@Table(  
    name = "employees",  
    catalog = "employee_catalog",  
    schema = "hr",  
    uniqueConstraints = {  
        @UniqueConstraint(columnNames = {"email"})  
    },  
    indexes = {  
        @Index(name = "idx_name", columnList = "name"),  
        @Index(name = "idx_department", columnList = "department")  
    }  
)
```

Let's break each part.

3 name = "employees"

What it does

- 👉 Sets the table name in the database.

java

 Copy code

```
@Entity  
@Table(name = "employees")  
public class Employee {}
```

DB table:

sql

 Copy code

```
employees
```

- ⭐ Without this:

- Table name defaults to entity name (`Employee`)

4 catalog = "employee_catalog"

What is a catalog?

A catalog is a database grouping concept.

- Common in MySQL
- Often maps to database name

Example:

sql

 Copy code

```
employee_catalog.employees
```

- ⭐ Mostly ignored by many databases
- ⭐ Rarely used in real projects

- ⚠ Use only if you fully understand DB structure.



```
5 schema = "hr"
```

What is schema?

A schema is a namespace inside a database.

Common DBs:

- PostgreSQL
- Oracle
- SQL Server

Example:

```
sql
```

```
hr.employees
```

- ★ Schema ≠ database
- ★ Schema organizes tables logically
- ✓ Common in enterprise apps

6 uniqueConstraints

java

 Copy code

```
uniqueConstraints = {  
    @UniqueConstraint(columnNames = {"email"})  
}
```

What it does

- 👉 Creates a UNIQUE constraint at table level

SQL:

sql

 Copy code

```
UNIQUE (email)
```

Why table-level?

Because:

- Supports multi-column uniqueness
- More powerful than `@Column(unique=true)`

Example (composite unique):

java

 Copy code

```
@UniqueConstraint(columnNames = {"email", "department"})
```

- ⭐ Enforced by database
- ⭐ Violations throw constraint exception

7 indexes

java

```
indexes = {  
    @Index(name = "idx_name", columnList = "name"),  
    @Index(name = "idx_department", columnList = "department")  
}
```

What it does

👉 Creates database indexes

SQL:

sql

```
CREATE INDEX idx_name ON employees(name);  
CREATE INDEX idx_department ON employees(department);
```

Why indexes?

- Faster SELECT queries
 - Especially for WHERE, JOIN, ORDER BY
- ⭐ Improves read performance
- ⭐ Slightly slows INSERT/UPDATE

columnList details

java

 Copy

```
columnList = "name"
```

Can be:

java

 Copy

```
columnList = "name, department"
```

Supports:

- Multi-column indexes
- Ordering:

java

 Copy

```
columnList = "name ASC, department DESC"
```

8 Full entity example

java

 Copy code

```
@Entity
@Table(
    name = "employees",
    schema = "hr",
    uniqueConstraints = {
        @UniqueConstraint(columnNames = {"email"})
    },
    indexes = {
        @Index(name = "idx_name", columnList = "name"),
        @Index(name = "idx_department", columnList = "department")
    }
)
public class Employee {

    @Id
    @GeneratedValue
    private Long id;

    private String name;
    private String email;
    private String department;
}
```

9 When does this SQL get created?

Only when:

properties

 Copy code

```
spring.jpa.hibernate.ddl-auto=create / update
```

★ With `validate` or `none` → ignored for schema creation

10 Common mistakes

Overusing `catalog`

- Breaks portability

Relying on indexes with `ddl-auto=update`

- Index creation may be inconsistent

Forgetting index for frequently filtered columns

- Leads to slow queries
-

1 1 Best Practices (Real Projects)

- ✓ Use `name` always when DB table naming differs
 - ✓ Use `schema` in enterprise DBs
 - ✓ Use `uniqueConstraints` for multi-column uniqueness
 - ✓ Use `indexes` for frequently searched columns
 - ✗ Avoid `catalog` unless required
-

1 2 One-line interview answer

`@Table` customizes entity-to-table mapping, allowing control over table name, schema, unique constraints, and indexes at the database level.

```
@Table(  
    name= "product_table",  
    uniqueConstraints = {  
        @UniqueConstraint(name="sku_unique", columnNames =  
            {"sku"}),  
        @UniqueConstraint(name = "title_price_unique",  
            columnNames = {"title_x", "price"})  
    },  
    indexes = {  
        @Index(name="sku_index", columnList = "sku")  
    }  
)
```

Great example 🌟

This `@Table` definition is very realistic and combines table naming, unique constraints, and indexes.

Let's break it line by line, then see what SQL is generated, why each part exists, and best practices.

1 Full annotation (your example)

java

 Copy code

```
@Table(  
    name = "product_table",  
    uniqueConstraints = {  
        @UniqueConstraint(name = "sku_unique", columnNames = {"sku"}),  
        @UniqueConstraint(name = "title_price_unique", columnNames = {"title_x", "price"})  
    },  
    indexes = {  
        @Index(name = "sku_index", columnList = "sku")  
    }  
)
```

2 name = "product_table"

What it does

- Sets the actual table name in the database
- Overrides default (`Product`)

SQL:

sql Copy code

```
CREATE TABLE product_table (...)
```

- ✓ Use when DB naming ≠ entity naming

3 uniqueConstraints

Unique constraints ensure data uniqueness at DB level.

- 3.1 `sku_unique` (single-column unique)

java Copy code

```
@UniqueConstraint(  
    name = "sku_unique",  
    columnNames = {"sku"}  
)
```

Meaning

SKU must be globally unique

SQL:

sql Copy code

```
CONSTRAINT sku_unique UNIQUE (sku)
```

- ✓ Prevents duplicate SKU values
✓ Enforced by database



- 3.2 title_price_unique (composite unique)

java

 Copy code

```
@UniqueConstraint(  
    name = "title_price_unique",  
    columnNames = {"title_x", "price"}  
)
```

Meaning

Same title + same price cannot repeat

But:

- Same title with different price → allowed
- Same price with different title → allowed

SQL:

sql

 Copy code

```
CONSTRAINT title_price_unique UNIQUE (title_x, price)
```

★ This is table-level uniqueness, not possible with `@Column(unique=true)`

4 indexes

Indexes are for **performance**, not correctness.

- `sku_index`

java

 Copy code

```
@Index(name = "sku_index", columnList = "sku")
```

SQL:

sql

 Copy code

```
CREATE INDEX sku_index ON product_table (sku);
```

Why index on `sku` ?

- Very common search field
- Used in:

sql

 Copy code

```
WHERE sku = ?
```

✓ Faster reads

✗ Slightly slower writes

5 Important question:

Why both UNIQUE constraint and INDEX on `sku` ?

Because:

- UNIQUE = data correctness
- INDEX = query performance

✗ Many DBs auto-create an index for UNIQUE, but:

- Naming control
- DB portability
- Explicit clarity



6 Full entity example (complete picture)

java

 Copy code

```
@Entity
@Table(
    name = "product_table",
    uniqueConstraints = {
        @UniqueConstraint(name = "sku_unique", columnNames = {"sku"}),
        @UniqueConstraint(name = "title_price_unique", columnNames = {"title_x", "price"})
    },
    indexes = {
        @Index(name = "sku_index", columnList = "sku")
    }
)
public class Product {

    @Id
    @GeneratedValue
    private Long id;

    @Column(nullable = false)
    private String sku;

    @Column(name = "title_x", nullable = false)
    private String title;

    @Column(nullable = false)
    private BigDecimal price;
}
```

7 When is this applied?

Only when:

properties

 Copy code

```
spring.jpa.hibernate.ddl-auto=create / update
```

 Ignored when:

- validate
- none

8 Common mistakes

 Using wrong column name

java

 Copy code

```
columnNames = {"title"}
```

But DB column is `title_x`

→ Startup error

 Assuming index = unique

Index does NOT enforce uniqueness

9 Best practices (real projects)

- ✓ Use named constraints (easy debugging)
 - ✓ Use composite unique constraints for business rules
 - ✓ Index fields frequently used in WHERE
 - ✗ Don't rely only on Hibernate for indexes in prod
-

10 One-line interview answer

This `@Table` configuration defines the database table name, enforces uniqueness for SKU and title–price combinations, and creates an index on SKU to improve query performance.

That's totally normal 🔥

Indexes are a database concept, not just JPA/Hibernate, and once you "see" what they do, it becomes very clear.

I'll explain slowly, visually (mentally), and with real examples, no jargon.

1 The core problem indexes solve

Imagine a table:

text

 Copy code

product_table

id | sku | title | price

1 | A101 | Phone | 500

2 | A102 | Laptop | 900

3 | A103 | Tablet | 300

...

1,000,000 rows

Now you run this query:

sql

 Copy code

```
SELECT * FROM product_table WHERE sku = 'A103';
```

✗ Without an index

The database must:

- Start at row 1
- Check sku
- Go to row 2
- Check sku
- ...
- Go through ALL 1,000,000 rows

This is called FULL TABLE SCAN

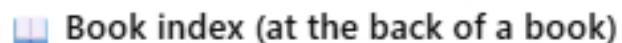


Very slow

2 What is an index? (Simple definition)

An index is a separate data structure that allows the database to find rows quickly, without scanning the entire table.

Think of it like:



Book index (at the back of a book)

- Book → table
- Page number → row location
- Index → shortcut

3 How index works (mental picture)

Instead of this:

```
pgsql
```

 Copy code

```
Row by row scan
```

Database creates something like:

```
text
```

 Copy code

```
INDEX (sku → row address)
```

```
A101 → row 1
```

```
A102 → row 2
```

```
A103 → row 3
```

Now when you query:

```
sql
```

 Copy code

```
WHERE sku = 'A103'
```

Database:

1. Looks into index
 2. Finds row address instantly
 3. Jumps directly to the row
-  Very fast

3 How index works (mental picture)

Instead of this:

```
pgsql
```

 Copy

```
Row by row scan
```

Database creates something like:

```
text
```

 Copy

```
INDEX (sku → row address)
```

```
A101 → row 1
```

```
A102 → row 2
```

```
A103 → row 3
```

Now when you query:

```
sql
```

 Copy

```
WHERE sku = 'A103'
```

Database:

1. Looks into index
2. Finds row address instantly
3. Jumps directly to the row

 Very fast

4 Real example with your code

java

```
@Index(name = "sku_index", columnList = "sku")
```

Generates SQL:

sql

```
CREATE INDEX sku_index ON product_table (sku);
```

Now this query becomes fast:

sql

```
SELECT * FROM product_table WHERE sku = 'A101';
```

5 Why indexes make SELECT fast but INSERT slower

When inserting data:

sql

```
INSERT INTO product_table VALUES (...)
```

DB must:

1. Insert row into table
2. Update index structure

So:

- Read speed ↑
- Write speed ↓ (slightly)

That's why:

 Do NOT index every column



6 When should you use an index?

✓ Columns used in:

- WHERE
- JOIN
- ORDER BY
- GROUP BY

✗ Columns rarely searched

7 Index vs Unique constraint (important confusion)

Feature	Index	Unique Constraint
Improves performance	✓	✗ (not main goal)
Enforces uniqueness	✗	✓
Prevents duplicates	✗	✓

★ Some DBs auto-create index for UNIQUE, but conceptually they are different.

8 Composite index (multiple columns)

java

 Copy code

```
@Index(name = "title_price_idx", columnList = "title, price")
```

Used for queries like:

sql

 Copy code

```
WHERE title = 'Phone' AND price = 500
```

⚠ Order matters:

- (title, price) works for title
- (price, title) works for price



9 Why NOT always use indexes?

Too many indexes:

- Slow INSERT/UPDATE/DELETE
- Use more disk
- Harder maintenance

Rule of thumb:

Index what you search, not what you store.

10 One-line interview answer

An index is a database data structure that speeds up query performance by allowing the database to quickly locate rows without scanning the entire table.

Final mental model (remember this)

- Table = actual data
- Index = shortcut map
- SELECT → fast
- INSERT/UPDATE → slightly slower
- Use wisely

m pom.xml (jpaTuts)

© Product.java ×

@ Column.class

application.properties

```
1 package com.cosingshuttle.jpaTutorial.jpaTuts.entities;
2
3 import jakarta.persistence.*;
4 import lombok.AllArgsConstructor;
5 import lombok.Builder;
6 import lombok.Data;
7 import lombok.NoArgsConstructor;
8 import org.hibernate.annotations.CreationTimestamp;
9 import org.hibernate.annotations.UpdateTimestamp;
10
11 import java.math.BigDecimal;
12 import java.time.LocalDateTime;
13
14
15 @Data
16 @NoArgsConstructor
17 @AllArgsConstructor
18 @Builder
19 @Entity
20 @Table(
21     name= "product_table",
22     uniqueConstraints = {
23         @UniqueConstraint(name="sku_unique", columnNames = {"sku"}),
24         @UniqueConstraint(name = "title_price_unique", columnNames = {"title_x", "price"})
25     },
26     indexes = {
27         @Index(name="sku_index", columnList = "sku")
28     }
29 )
30 public class Product {
```

```
        @Index(name="sku_index", columnList = "sku")
    }
}

public class Product {

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

    @Column(nullable = false,length=20)
    private String sku;

    @Column(name="title_x")
    private String title;
    private BigDecimal price;
    private Integer quantity;

    @CreationTimestamp
    private LocalDateTime createdAt;

    @UpdateTimestamp
    private LocalDateTime updatedAt;
}
```



m pom.xml (jpaTuts)

© Product.java

@ Col

Run / Debug Configurations

.properties ×

```
1 spring.application.name=jpaTutorial  
2 spring.jpa.hibernate.ddl-auto=update  
3 spring.jpa.show-sql=true  
4 spring.jpa.properties.hibernate.format_sql=true  
5  
6 ⚡ spring.datasource.url=jdbc:mysql://localhost:3306/test?useSSL=false  
7 spring.datasource.username=root  
8 spring.datasource.password=abhi2003#
```

```
spring.datasource.url=jdbc:mysql://localhost:3306/test?useSSL=false  
spring.datasource.username=root  
spring.datasource.password=abhi2003#
```

These three lines are your database connection configuration.

I'll explain each part in detail, including what happens internally, why each piece exists, and how Spring Boot + JDBC + Hibernate use them.

1 `spring.datasource.url`

properties

 Copy code

```
spring.datasource.url=jdbc:mysql://localhost:3306/test?useSSL=false
```

This is the most important line.

It tells Spring Boot where the database is and how to connect to it.

Let's break it piece by piece.

- `jdbc:mysql://`
- `jdbc` → Java Database Connectivity (standard)
- `mysql` → Database type (driver selection)

👉 Spring Boot uses this to choose:

java

 Copy code

```
MySQL JDBC Driver (mysql-connector-j)
```

- `localhost`
- Database is running on **same machine**
- Could also be an IP or hostname

Example:

text

 Copy code

```
192.168.1.10  
db.mycompany.com
```

- `3306`
- MySQL default port
- Tells Java where MySQL is listening

- `test`
- Database name
- NOT table name
- Hibernate will create tables inside this DB

- `?useSSL=false`
- Disables SSL connection

Why?

- Local development usually doesn't use SSL
- Avoids warnings like:

pgsql

 Copy code

```
Establishing SSL connection without server's identity verification
```

 In production:

properties

 Copy code

```
useSSL=true
```

2 spring.datasource.username

properties

 Copy code

```
spring.datasource.username=root
```

- Database login user
- Must exist in MySQL

This user must have permissions:

- CONNECT
- SELECT
- INSERT
- UPDATE
- DELETE
- CREATE / ALTER (if ddl-auto ≠ validate/none)

3 spring.datasource.password

properties

 Copy code

```
spring.datasource.password=abhi2003#
```

- Password for DB user

 Security note:

- Never hardcode passwords in production
- Use environment variables / vaults

4 What happens internally when app starts?

Startup flow

SCSS

 Copy code

```
application.properties  
↓  
Spring Boot auto-config  
↓  
DataSource (HikariCP)  
↓  
JDBC Driver (MySQL)  
↓  
Database
```

Step-by-step:

1. Spring Boot reads these properties
2. Creates a **DataSource** (connection pool)
3. Loads MySQL JDBC driver
4. Opens connections using URL + username + password
5. Hibernate uses this **DataSource**
6. Tables created / validated based on `ddl-auto`

5 Why `spring.datasource.*` not `hibernate.*` ?

Because:

- Hibernate uses JDBC
- JDBC needs a **DataSource**
- Spring Boot manages **DataSource** centrally

So:

Hibernate never connects directly → **DataSource** does

6 Common mistakes ✖

✖ Wrong DB name

properties

 Copy code

```
jdbc:mysql://localhost:3306/tests // DB doesn't exist
```

→ Connection fails

✖ Driver not added

If dependency missing:

pgsql

 Copy code

```
No suitable driver found
```

✖ Special characters in password

may cause issues in .properties

Solution:

properties

 Copy code

```
spring.datasource.password=abhi2003\#
```

6 Common mistakes ✖

✖ Wrong DB name

properties

 Copy code

```
jdbc:mysql://localhost:3306/tests // DB doesn't exist
```

→ Connection fails

✖ Driver not added

If dependency missing:

pgsql

 Copy code

```
No suitable driver found
```

✖ Special characters in password

may cause issues in .properties

Solution:

properties

 Copy code

```
spring.datasource.password=abhi2003\#
```

7 Recommended dev configuration (MySQL)

properties

 [Copy code](#)

```
spring.datasource.url=jdbc:mysql://localhost:3306/test?useSSL=false&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=*****
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

8 One-line interview answer

These properties configure the JDBC DataSource, telling Spring Boot how to connect to the MySQL database using JDBC, which Hibernate then uses for persistence.

Project pom.xml Product.java Column.class application.properties

1 spring.application.name=jpaTutorial
2 spring.jpa.hibernate.ddl-auto=create

Run JpaTutorialApplication

Console Beans Health Mappings Environment

```
2025-12-16T14:39:05.057+05:30 INFO 23624 --- [jpaTutorial] [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2025-12-16T14:39:05.058+05:30 INFO 23624 --- [jpaTutorial] [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Added connection com.mysql.cj.jdbc.Connection
2025-12-16T14:39:05.122+05:30 INFO 23624 --- [jpaTutorial] [main] org.hibernate.orm.connections.pooling : HHH10001005: Database info:  
Database JDBC URL [jdbc:mysql://localhost:3306/test?useSSL=false]  
Database driver: MySQL Connector/J  
Database dialect: MySQLDialect  
Database version: 8.0.44  
Default catalog/schema: test/undefined  
Autocommit mode: undefined/unknown  
Isolation level: REPEATABLE_READ [default REPEATABLE_READ]  
JDBC fetch size: none  
Pool: DatasourceConnectionProviderImpl  
Minimum pool size: undefined/unknown  
Maximum pool size: undefined/unknown  
2025-12-16T14:39:05.672+05:30 INFO 23624 --- [jpaTutorial] [main] o.h.e.t.j.p.i.JtaPlatformInitiator : HHH000489: No JTA platform available (set 'hibernate.tr  
Hibernate:  
drop table if exists product_table  
Hibernate:  
create table product_table (  
    price decimal(38,2),  
    quantity integer,  
    created_at datetime(6),  
    id bigint not null auto_increment,  
    updated_at datetime(6),  
    sku varchar(20) not null,  
    title_x varchar(255),
```

```
2025-12-16T14:39:05.672+05:30  INFO 23624 --- [jpaTutorial] [           main] o.h.e.t.j.p.i.JtaPlatformInitiator      : HHH000489: No JTA platform available (set 'hibernate.tr
Hibernate:
    drop table if exists product_table
Hibernate:
    create table product_table (
        price decimal(38,2),
        quantity integer,
        created_at datetime(6),
        id bigint not null auto_increment,
        updated_at datetime(6),
        sku varchar(20) not null,
        title_x varchar(255),
        primary key (id)
    ) engine=InnoDB
Hibernate:
    create index sku_index
        on product_table (sku)
Hibernate:
    alter table product_table
        add constraint sku_unique unique (sku)
Hibernate:
    alter table product_table
        add constraint title_price_unique unique (title_x, price)
2025-12-16T14:39:05.765+05:30  INFO 23624 --- [jpaTutorial] [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence un
2025-12-16T14:39:05.803+05:30  WARN 23624 --- [jpaTutorial] [           main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefor
2025-12-16T14:39:06.056+05:30  INFO 23624 --- [jpaTutorial] [           main] o.s.boot.tomcat.TomcatWebServer       : Tomcat started on port 8080 (http) with context path '/
```

```
m pom.xml (jpaTuts)  © Product.java  ≡ data.sql  © JpaTutorialApplication.java  @ Column.class  © app
 1 spring.application.name=JPATutorial
 2 spring.jpa.hibernate.ddl-auto=update
 3 spring.jpa.show-sql=true
 4 spring.jpa.properties.hibernate.format_sql=true
 5
 6 spring.datasource.url=jdbc:mysql://localhost:3306/test?useSSL=false
 7 spring.datasource.username=root
 8 spring.datasource.password=Anuj@1234
 9
10 spring.jpa.defer-datasource-initialization=true| I
11 spring.sql.init.mode=always
12 spring.sql.init.data-locations=classpath:data.sql
```

```
spring.jpa.defer-datasource-initialization=true  
spring.sql.init.mode=always  
spring.sql.init.data-locations=classpath:data.sql
```

These three properties are about **WHEN** and **HOW** Spring Boot runs SQL scripts (`schema.sql`, `data.sql`) relative to Hibernate/JPA initialization.

This topic is confusing because it mixes **Spring Boot**, **JPA/Hibernate**, and **database startup order**.

Let's clear it step by step with a **timeline + examples**.

Big picture first (one sentence)

These properties control database initialization using SQL files and ensure they run at the correct time when using JPA/Hibernate.

1 `spring.sql.init.mode=always`

What does this do?

properties

 Copy code

```
spring.sql.init.mode=always
```

👉 Tells Spring Boot:

"Always run SQL initialization scripts."

What scripts?

- `schema.sql`
- `data.sql`
- Custom scripts via `spring.sql.init.data-locations`

Possible values

Value	Meaning
always	Run SQL scripts every time
embedded (default)	Run only for H2/HSQL/Derby
never	Never run SQL scripts

Why this exists

By default, Spring Boot does NOT run `data.sql` for MySQL/Postgres.

This property forces it.

Example

```
sql
-- data.sql
INSERT INTO employee (name, salary) VALUES ('Abhi', 50000);
```

With `mode=always` → this runs on every app startup.

2 `spring.sql.init.data-locations=classpath:data.sql`

What does this do?

```
properties
spring.sql.init.data-locations=classpath:data.sql
```

👉 Tells Spring Boot:

"This is the exact SQL file to run for data initialization."

Breakdown

- `classpath:` → file is inside `src/main/resources`
- `data.sql` → SQL insert file

Project structure:

```
css
```

 Copy code

```
src/main/resources/  
└── data.sql
```

What goes inside `data.sql`?

Usually:

- INSERT statements
- Reference data
- Initial rows

Example:

```
sql
```

 Copy code

```
INSERT INTO roles VALUES (1, 'ADMIN');  
INSERT INTO roles VALUES (2, 'USER');
```

3 `spring.jpa.defer-datasource-initialization=true`

⚠ MOST IMPORTANT PROPERTY

The problem it solves

Without this property ✗

Startup order:

markdown

Copy code

1. `data.sql` runs
2. Hibernate creates tables

Result:

css

Copy code

✗ Table does not exist

Because:

- `data.sql` runs
- Tables are not created yet by Hibernate

What this property does

properties

Copy code

`spring.jpa.defer-datasource-initialization=true`

👉 Tells Spring Boot:

“Wait for Hibernate to finish schema creation before running `data.sql`.”

4 Full working example (VERY IMPORTANT)

application.properties

```
properties

spring.datasource.url=jdbc:mysql://localhost:3306/test
spring.datasource.username=root
spring.datasource.password=****

spring.jpa.hibernate.ddl-auto=update

spring.jpa.defer-datasource-initialization=true
spring.sql.init.mode=always
spring.sql.init.data-locations=classpath:data.sql
```

data.sql

```
sql

INSERT INTO employee (name, salary) VALUES ('Abhi', 50000);
INSERT INTO employee (name, salary) VALUES ('Rahul', 60000);
```

Result on startup

1. Hibernate creates `employee` table
2. Spring executes `data.sql`
3. Rows are inserted
4. App runs normally

Result on startup

1. Hibernate creates `employee` table
 2. Spring executes `data.sql`
 3. Rows are inserted
 4. App runs normally
-

5 When should you use this setup?

✓ Good use cases

- Local development
- Demo data
- Testing
- Learning JPA/Hibernate

✗ Not recommended for production

Why?

- Re-runs every startup
- Can cause duplicate data
- No version control for DB changes

👉 Use Flyway/Liquibase in production instead.

6 Common mistakes ✗

✗ Forgetting `defer-datasource-initialization`

Result:

```
rust
```

 Copy code

```
Table 'employee' doesn't exist
```

✖ Using `mode=always` in prod

Result:

- Duplicate rows
- Data corruption risk

7 Mental model (remember this)

kotlin

```
ddl-auto → creates tables  
data.sql → inserts data  
  
defer-datasource-initialization  
= "Run data.sql AFTER ddl-auto"
```

8 One-line interview answers

`spring.sql.init.mode`

Controls whether SQL initialization scripts should run.

`spring.sql.init.data-locations`

Specifies the location of SQL data scripts.

`spring.jpa.defer-datasource-initialization`

Ensures SQL scripts run after Hibernate creates the schema.

The screenshot shows a Java development environment with the following interface elements:

- Project View:** On the left, it displays the project structure under "jpaTuts".
- Toolbars:** Standard Java IDE toolbars for file operations.
- Editors:** Several tabs are open in the main area:
 - pom.xml (jpaTuts)**: Maven configuration file.
 - Product.java**: Java entity class.
 - Column.class**: Java class annotated with @Column.
 - application.properties**: Application configuration properties.
 - data.sql**: SQL script currently being edited.
- SQL Editor:** The "data.sql" tab contains the following SQL code:

```
1 INSERT INTO test.product_table (price, quantity, sku, title_x)
2 VALUES
3     ( price 12.4, quantity 4, sku 'parle734', title_x 'Parle Biscuit'),
4     ( price 14.4, quantity 1, sku 'pepsi1123', title_x 'Pepsi'),
5     ( price 16.4, quantity 3, sku 'mazza115', title_x 'Mazza');
```
- Bottom Status Bar:** Shows the connection information: [test@localhost].

Key features of JPA

Entity Management: Defines how entities (Java objects) are persisted to the database.

Query Language: Provides JPQL (Java Persistence Query Language) for querying entities.

Transactions: Manages transactions, making it easier to handle database operations within a transactional context.

Entity Relationships: Supports defining relationships between entities (e.g., One-to-One, One-to-Many, Many-to-One, Many-to-Many).

