

Project

Gradle - Groovy

Gradle - Kotlin Maven

Spring Boot

4.0.2 (SNAPSHOT) 4.0.1 3.5.10 (SNAPSHOT) 3.5.9

Project Metadata

Group com.codingshuttle.jpaTutor

Artifact hospitalManagementSystem

Language

Java Kotlin Groovy

Dependencies

Lombok DEVELOPER

Java annotation library w/

Spring Web WEB

Build web, including RESTful API as the default embedded

Spring Data JPA SQL

**Project** Gradle - Groovy Gradle - Kotlin Maven**Language** Java Kotlin Groovy**Spring Boot** 4.0.2 (SNAPSHOT) 4.0.1 3.5.10 (SNAPSHOT) 3.5.9**Project Metadata**Group com.codingshuttle.jpaTutor Artifact hospitalManagementSystem

Name hospitalManagementSystemJpaTutorial

Description Demo project for Spring Boot jpa tutorial 2

Package name com.codingshuttle.jpaTutorial2.hospitalManagementSystem

Packaging Jar WarConfiguration Properties YAMLJava 25 21 17**Dependencies****ADD DEPENDENCIES...** CTRL + B**Lombok** DEVELOPER TOOLS

Java annotation library which helps to reduce boilerplate code.

Spring Web WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Data JPA SQL

Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

MySQL Driver SQL

MySQL JDBC driver.

PostgreSQL Driver SQL

A JDBC and R2DBC driver that allows Java programs to connect to a PostgreSQL database using standard, database independent Java code.

GENERATE CTRL + ↵**EXPLORE** CTRL + SPACE

...

The error `Unknown column 'bloodGroup' in 'field list'` indicates that the column name in your `data.sql` doesn't match the actual column name in the database table.

When JPA/Hibernate creates the table, it typically converts camelCase field names to snake_case (e.g., `bloodGroup` → `blood_group`).

Great question 🌟

Enums are simple to use but very powerful once you understand why they exist and when to use them.

Let's go from basics → file → advanced usage → JPA usage → best practices.

1 What is `enum` in Java?

An `enum` is a special Java type used to represent a fixed set of constants.

Example:

```
java Copy code
public enum Status {
    ACTIVE,
    INACTIVE,
    BLOCKED
}
```

Meaning:

- A variable of type `Status` can only have one of these values
 - No other value is allowed
- ⭐ This gives type safety

2 Why do we use enum?

Without enum ✗

```
java Copy code
String status = "ACTIVE"; // typo-prone
```

Problems:

- "Actve" compiles
- No validation
- Hard to maintain



With enum

java

 Copy code

```
Status status = Status.ACTIVE;
```

Benefits:

- ✓ Compile-time safety
- ✓ Readable code
- ✓ Fixed set of values
- ✓ Easy refactoring

3 Do we make a separate enum file?

Yes (BEST PRACTICE)

Each enum is usually in its own `.java` file.

css

 Copy code

```
src/main/java
└── com.example.enums
    └── Status.java
```

java

 Copy code

```
package com.example.enums;

public enum Status {
    ACTIVE,
    INACTIVE,
    BLOCKED
}
```

- ★ Same rule as class files
- ★ Can also be nested (but less common)

4 How enum works internally (important)

Enum is NOT just constants.

Internally:

java

 Copy code

```
public final class Status extends Enum<Status> {  
    public static final Status ACTIVE;  
    public static final Status INACTIVE;  
    public static final Status BLOCKED;  
}
```

- Enum values are objects
- Only one instance per value (singleton)

5 Using enum in a class

java

 Copy code

```
public class User {  
    private Status status;  
}
```

Usage:

java

 Copy code

```
user.setStatus(Status.ACTIVE);
```

6 Enum with fields and constructor (ADVANCED)

java

 Copy code

```
public enum OrderStatus {  
  
    CREATED(1),  
    SHIPPED(2),  
    DELIVERED(3);  
  
    private final int code;  
  
    OrderStatus(int code) {  
        this.code = code;  
    }  
  
    public int getCode() {  
        return code;  
    }  
}
```

Usage:

java

 Copy code

```
OrderStatus status = OrderStatus.SHIPPED;  
status.getCode(); // 2
```

★ Enum constructors are always private

7 Enum methods you should know

java

 Copy code

```
Status.values();          // all enum constants  
Status.valueOf("ACTIVE"); // convert string → enum  
status.name();           // "ACTIVE"  
status.ordinal();        // index (0-based)
```

⚠ Avoid using `ordinal()` in DB (dangerous)



8 Enum in JPA (VERY IMPORTANT)

Mapping enum to database

java

 Copy code

```
@Enumerated(EnumType.STRING)  
private Status status;
```

DB stores:

SCSS

 Copy code

```
ACTIVE  
INACTIVE
```

 Avoid:

java

 Copy code

```
@Enumerated(EnumType.ORDINAL)
```

Because:

- Stores numbers (0,1,2)
- Reordering enum breaks data

9 Enum vs constants (`static final`)

enum

static final

Type-safe

 No

Fixed set

 No

Can have methods

 No

JPA-friendly

 Hard

10 When should you use enum?

- Fixed values
 - Status, Role, Type
 - State machines
 - Configuration choices
 - Dynamic data (use DB table instead)
-

1 1 One-line interview answer

An enum in Java is a special type that represents a fixed set of constants, providing type safety, readability, and maintainability, and is commonly defined in its own file.

Final memory rule

If a value set is fixed → use enum.

If it changes often → use DB table.

@Enumerated in spring boot and how enum in jpa is important and how @Enumerated is used there in entity class on the enum field

Great question 🔥

Enums + JPA is a very important topic, because a wrong enum mapping can break production data.

I'll explain this in a clean, layered way:

1. What `@Enumerated` is
2. Why enum mapping matters in JPA
3. EnumType values (`ORDINAL` vs `STRING`)
4. How it is used in an entity
5. What happens in the database
6. Best practices & common mistakes

1 What is `@Enumerated` ?

java

 Copy code

`@Enumerated`

`@Enumerated` tells JPA how an enum field should be stored in the database.

- ❖ It is a JPA annotation
- ❖ Used on enum fields inside an entity

Without it, JPA does not know how you want to persist the enum.

2 Why enum mapping is important in JPA

Enums are:

- Java-side concepts
- Databases don't understand enums natively (most DBs)

So JPA must convert:

cpp

 Copy code

```
Java enum → DB value  
DB value → Java enum
```

That conversion rule is defined by `@Enumerated`.

3 Enum example (base)

java

 Copy code

```
public enum Status {  
    ACTIVE,  
    INACTIVE,  
    BLOCKED  
}
```

4 Using enum in an Entity (without @Enumerated ✗)

java

 Copy code

```
@Entity  
public class User {  
  
    @Id  
    private Long id;  
  
    private Status status; // ✗ dangerous  
}
```

What happens?

- JPA defaults to ORDINAL
- This is risky (explained below)

5 @Enumerated options (VERY IMPORTANT)

java

 Copy code

```
@Enumerated(EnumType.ORDINAL)  
@Enumerated(EnumType.STRING)
```

These two decide how the enum is stored in DB.

6 EnumType.ORDINAL ✖ (DANGEROUS)

java

```
@Enumerated(EnumType.ORDINAL)
private Status status;
```

 Copy code

How it is stored in DB

Enum value	Ordinal
ACTIVE	0
INACTIVE	1
BLOCKED	2

DB column:

ini

```
status = 0
```

 Copy code

✖ Why this is dangerous

If you change enum order:

java

```
public enum Status {
    INACTIVE,
    ACTIVE,
    BLOCKED
}
```

 Copy code

Now:

- 0 means INACTIVE
- Existing data becomes wrong
- Silent data corruption 😱

👉 Never use ORDINAL in production



7 EnumType.STRING ✓ (RECOMMENDED)

java

 Copy code

```
@Enumerated(EnumType.STRING)  
private Status status;
```

How it is stored in DB

ini

 Copy code

```
status = 'ACTIVE'
```

Why this is safe

- ✓ Enum order can change
- ✓ New enum values can be added
- ✓ Readable DB data
- ✓ No silent corruption

👉 This is the best practice

8 Full entity example (CORRECT WAY)

java

 Copy code

```
@Entity  
@Table(name = "users")  
public class User {  
  
    @Id  
    @GeneratedValue  
    private Long id;  
  
    private String name;  
  
    @Enumerated(EnumType.STRING)  
    @Column(nullable = false)  
    private Status status;  
}
```

DB table:

sql

 Copy code

```
status VARCHAR(255)
```

Stored values:

scss

 Copy code

```
ACTIVE  
INACTIVE
```

9 Why enum in JPA is important (conceptually)

Enums give:

- Domain safety (only allowed states)
- Cleaner code
- Better validation
- Clear business meaning

Example:

```
java Copy code  
if (user.getStatus() == Status.ACTIVE) { ... }
```

Much safer than:

```
java Copy code  
if ("ACTIVE".equals(user.getStatus())) { ... }
```

10 Using enum in Spring Data JPA queries

Derived query

```
java Copy code  
List<User> findByStatus(Status status);
```

Usage:

```
java Copy code  
repository.findByStatus(Status.ACTIVE);
```

Spring Data JPA:

- Converts enum → DB value automatically

@Query with enum

java

 Copy code

```
@Query("SELECT u FROM User u WHERE u.status = :status")
List<User> findUsers(@Param("status") Status status);
```

- ✓ Works automatically
- ✓ No manual conversion needed

1 1 Common mistakes ✗

✗ Forgetting `EnumType.STRING`

java

 Copy code

```
@Enumerated
private Status status; // defaults to ORDINAL
```

✗ Using ordinal in production

java

 Copy code

```
EnumType.ORDINAL
```

✗ Storing enum as String manually

java

 Copy code

```
@Column
private String status; // loses type safety
```

1 2 When NOT to use enum in JPA

- ✗ When values change frequently
- ✗ When values come from DB dynamically

Use a lookup table instead.

🔑 One-line interview answer (perfect)

`@Enumerated` is a JPA annotation used to specify how an enum field should be persisted in the database, with `EnumType.STRING` being the recommended approach for safety and readability.

💡 Final rule to remember

Always use `@Enumerated(EnumType.STRING)`
Never use ORDINAL in real applications.

if we not add `@Enumerated(EnumType.STRING)` on the field then what will happenand when we add data in database how data enum type data which is not converted into the string how it is gonna add

This is an excellent and very important question 👍

You're asking what REALLY happens behind the scenes, not just theory.

Let's go step by step, with exact behavior, examples, and what gets inserted into the DB.

1 If you do NOT add `@Enumerated(EnumType.STRING)`, what happens?

Default behavior (THIS IS CRITICAL)

java

 Copy code

```
@Entity
public class User {

    @Id
    private Long id;

    private Status status; // ! no @Enumerated
}
```

👉 JPA defaults to `EnumType.ORDINAL`

So internally it behaves as if you wrote:

java

 Copy code

```
@Enumerated(EnumType.ORDINAL)
private Status status;
```

⭐ This is defined by the JPA specification

2 What does ORDINAL actually mean?

Consider your enum:

java

 Copy code

```
public enum Status {  
    ACTIVE,  
    INACTIVE,  
    BLOCKED  
}
```

Internally Java assigns index numbers:

Enum constant	Ordinal value
ACTIVE	0
INACTIVE	1
BLOCKED	2

3 When you save data, what is inserted into DB?

Your Java code

```
java  
  
User user = new User();  
user.setStatus(Status.ACTIVE);  
repository.save(user);
```

 Copy code

What Hibernate does internally

1. Hibernate sees enum field
2. No `@Enumerated(EnumType.STRING)`
3. Uses ORDINAL strategy
4. Converts enum → integer ordinal

SQL sent to DB

```
sql  
  
INSERT INTO user (status) VALUES (0);
```

 Copy code

 0 is stored, NOT "ACTIVE"

4 But wait — DB column type is VARCHAR or INT?

Hibernate chooses column type automatically

- With ORDINAL → INTEGER
- With STRING → VARCHAR

So your table becomes:

```
sql  
  
status INT
```

 Copy code

5 How does reading data work (DB → Java)?

Suppose DB has:

```
ini  
  
status = 0
```

 Copy code

Hibernate:

1. Reads integer `0`
2. Maps it to enum by index
3. `Status.values()[0]`
4. Result → `Status.ACTIVE`

 This mapping is position-based, not name-based.

6 Why is this dangerous? (REAL PRODUCTION ISSUE)

Today enum is:

```
java  
  
public enum Status {  
    ACTIVE,  
    INACTIVE,  
    BLOCKED  
}
```

 Copy code

DB has:

```
scss  
  
0 → ACTIVE  
1 → INACTIVE  
2 → BLOCKED
```

 Copy code

Tomorrow you change enum:

java

 Copy code

```
public enum Status {  
    INACTIVE,  
    ACTIVE,  
    BLOCKED  
}
```

Now ordinal mapping becomes:

Ordinal	Meaning now
0	INACTIVE
1	ACTIVE
2	BLOCKED

- 👉 All existing DB data is now WRONG
- 👉 No exception
- 👉 Silent data corruption 😱

7 Now the safe case: @Enumerated(EnumType.STRING)

java

 Copy code

```
@Enumerated(EnumType.STRING)  
private Status status;
```

What happens on save?

java

 Copy code

```
user.setStatus(Status.ACTIVE);  
repository.save(user);
```

Hibernate sends:

sql

 Copy code

```
INSERT INTO user (status) VALUES ('ACTIVE');
```

DB column:

sql

 Copy code

```
status VARCHAR(255)
```

8 How does reading work with STRING?

DB:

```
ini  
  
status = 'ACTIVE'
```

 Copy code

Hibernate:

1. Reads string "ACTIVE"
2. Matches enum constant by name
3. Status.valueOf("ACTIVE")
4. Result → Status.ACTIVE

- Order doesn't matter
- Safe for refactoring
- Human-readable DB

9 Direct answer to your exact question

 “When we add data in database how enum type data which is not converted into the string how it is gonna add?”

Answer:

- JPA converts the enum into its ordinal number (0, 1, 2...)
- That number is what gets inserted into the DB
- The DB never sees the enum name

10 Summary table (VERY IMPORTANT)

Scenario	DB Value Stored	Column Type	Safe?
No @Enumerated	0, 1, 2	INT	 NO
EnumType.ORDINAL	0, 1, 2	INT	 NO
EnumType.STRING	ACTIVE	VARCHAR 	 YES

Golden rule (MEMORIZE THIS)

If you don't specify `@Enumerated(EnumType.STRING)`, JPA will store enum ordinals (numbers), which is dangerous and should be avoided in real applications.

One-line interview answer (perfect)

Without `@Enumerated(EnumType.STRING)`, JPA defaults to ordinal mapping, storing the enum's numeric position in the database, which can cause silent data corruption if enum order changes.

Projection in Spring Data JPA is a way to fetch only a subset of data instead of the entire entity.

Think of it as:

"I don't want the whole object, only specific fields."

1 Why projection exists (the problem it solves)

Entity example

java

 Copy code

```
@Entity
public class Employee {
    @Id
    private Long id;
    private String name;
    private String email;
    private double salary;
    private String department;
}
```

Without projection ✗

java

 Copy code

```
List<Employee> findAll();
```

- Fetches all columns
- Loads unnecessary data
- Slower
- Risk of exposing sensitive fields

2 What is projection (definition)

Projection means selecting only required fields from an entity and mapping them to a different shape (DTO or interface).

So instead of:

bash

 Copy code

```
Employee (id, name, email, salary, department)
```

You can fetch:

scss

 Copy code

```
(name, salary)
```

3 Types of projections in Spring Data JPA

Spring Data JPA supports 3 main types:

1. Interface-based projection (most common)
2. Class-based (DTO) projection
3. Open projection (SpEL-based)

4 Interface-based projection (RECOMMENDED)

Step 1: Create projection interface

java

 Copy code

```
public interface EmployeeView {  
    String getName();  
    double getSalary();  
}
```

★ Method names must match entity field names



Step 2: Use it in repository

java

 Copy

```
List<EmployeeView> findByDepartment(String department);
```

What happens internally

- Spring Data JPA generates JPQL
- Fetches only `name` and `salary`
- Creates a proxy implementation of `EmployeeView`

Generated JPQL (conceptually)

jpql

 Copy

```
SELECT e.name, e.salary  
FROM Employee e  
WHERE e.department = :department
```

- ✓ No entity loaded
- ✓ No persistence context
- ✓ Faster

5 Class-based (DTO) projection

Step 1: DTO class

java

 Copy

```
public class EmployeeDTO {  
    private String name;  
    private double salary;  
  
    public EmployeeDTO(String name, double salary) {  
        this.name = name;  
        this.salary = salary;  
    }  
}
```

Step 2: Use `@Query`

java

 Copy code

```
@Query("""
    SELECT new com.example.EmployeeDTO(e.name, e.salary)
    FROM Employee e
""")
List<EmployeeDTO> findEmployeeDTOS();
```

★ Uses JPQL constructor expression

- ✓ Explicit
- ✓ Type-safe
- ✗ More code

6 Open projection (advanced, less used)

Projection interface

java

 Copy code

```
public interface EmployeeOpenView {
    String getName();

    @Value("#{target.salary * 12}")
    double getAnnualSalary();
}
```

★ Uses SpEL

★ Loads full entity internally

⚠ Slower than closed projections

7 Projection vs Entity (VERY IMPORTANT)

Aspect	Entity	Projection
Managed by persistence context	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No
Lazy loading	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No
Memory usage	High	Low
Best for	Write operations	Read-only

8 When to use projection

- ✓ Read-only APIs
- ✓ Large tables
- ✓ Performance-critical queries
- ✓ Avoid exposing entity directly

- ✗ Updates
- ✗ Complex business logic

9 Projection with pagination

java

 Copy code

```
Page<EmployeeView> findByDepartment(  
    String department,  
    Pageable pageable  
) ;
```

- ✓ Works perfectly
- ✓ Count query still runs

10 Common mistakes

- Expecting projection to be managed (it's not)
 - Using wrong getter names
 - Trying to update projection results
 - Using open projection when closed is enough
-

One-line interview answer

Projection in Spring Data JPA is a mechanism to retrieve only selected fields of an entity by mapping query results to interfaces or DTOs, improving performance and reducing data exposure.

Final mental model

- Entity → full object, managed
- Projection → partial view, read-only
- Spring Data JPA decides what to fetch

The screenshot shows a Java development environment with the following details:

- Project Structure:** The project is named "HospitalManagementSystem". It contains a "src" directory with "main" and "test" sub-directories. "main" contains "java", "resources", and "target". "java" contains "com.codingshuttle.jpaTutorial2.hospitalManagement" which has "entity" and "type" packages. "entity" contains "BloodGroupType" and "patient". "type" contains "BloodGroupType". "resources" contains "static", "templates", "application.properties", and "data.sql".
- Code Editor:** The "patient.java" file is open in the editor. The code defines a class "patient" with the following annotations and fields:
 - Annotations: package, import ..., @Entity, @Getter, @Setter, @ToString.
 - Fields:
 - @Id, @GeneratedValue(strategy = GenerationType.IDENTITY), private Long id;
 - private String name;
 - private LocalDate birthdate;
 - private String email;
 - private String gender;
 - @Enumerated(value = EnumType.STRING), private BloodGroupType bloodGroup;
 - @CreationTimestamp, private LocalDateTime createdAt;
- Other Files:** Other files visible include "BloodGroupType.java", "application.properties", "PatientRepository.java", "data.sql", and "HospitalManagementSystemJpaTutorialApplication.java".

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

- Project View:** On the left, it displays the project structure under the "src" folder. It includes the "main" package with "java", "entity", "repositories", and "resources" sub-packages. The "BloodGroupType" class is selected in the "entity" package.
- Editor Tabs:** At the top, there are tabs for "patient.java", "BloodGroupType.java", "application.properties", "PatientRepository.java", "data.sql", and "HospitalM...".
- Code Editor:** The central area contains the code for the "BloodGroupType" enum. The code is as follows:

```
1 package com.codingshuttle.jpaTutorial2.hospitalManagementSystem.entity.type;
2
3 public enum BloodGroupType { 2 usages
4     A_POSITIVE, no usages
5     A_NEGATIVE, no usages
6     B_POSITIVE, no usages
7     B_NEGATIVE, no usages
8     AB_POSITIVE, no usages
9     AB_NEGATIVE, no usages
10    O_POSITIVE, no usages
11    O_NEGATIVE no usages
12 }
13 
```

The "O_NEGATIVE" entry at line 11 has a yellow lightbulb icon next to it, indicating a warning or unused code.

Project ▾

- src
 - main
 - java
 - com.codingshuttle.jpaTutorial2.hospitalManagementSystem.entity
 - patient
 - type
 - BloodGroupType
 - repositories
 - PatientRepository
- resources
 - static
 - templates
 - application.properties
 - data.sql
- test
- target
 - .gitattributes
 - .gitignore
 - HELP.md
 - mvnw
 - mvnw.cmd
 - pom.xml
- External Libraries

patient.java BloodGroupType.java application.properties PatientRepository.java data.sql HospitalM...

```
1 package com.codingshuttle.jpaTutorial2.hospitalManagementSystem.repositories;
2
3 import com.codingshuttle.jpaTutorial2.hospitalManagementSystem.entity.patient;
4 import org.springframework.data.jpa.repository.JpaRepository;
5
6 public interface PatientRepository extends JpaRepository<patient, Long> { no usages
7 }
8
```

Project

src

- main
- java
 - com.codingshuttle.jpaTutorial2.hospitalManagement
 - entity
 - BloodGroupType
 - patient
 - repositories
 - PatientRepository
- resources
 - static
 - templates
 - application.properties
 - data.sql
- test
- target

.gitattributes

.gitignore

HELP.md

mvnw

mvnw.cmd

pom.xml

External Libraries

patient.java

BloodGroupType.java

application.properties

PatientRepository.java

data.sql

```
1 spring.application.name=hospitalManagementSystemJpaTutorial
2 spring.jpa.hibernate.ddl-auto=create
3 spring.jpa.show-sql=true
4 spring.jpa.properties.hibernate.format_sql=true
5
6 spring.datasource.url=jdbc:mysql://localhost:3306/hospitalDB
7 spring.datasource.username=root
8 spring.datasource.password=abhi2003#
9
10 #spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
11
12 spring.jpa.defer-datasource-initialization=true
13 spring.sql.init.mode=always
14 spring.sql.init.data-locations=classpath:data.sql
```

Project ▾

patient.java BloodGroupType.java application.properties PatientRepository.java data.sql HospitalManagementSystem

src

main

java

com.codingshuttle.jpaTutorial2

entity

type

BloodGroupType

patient

repositories

PatientRepository

HospitalManagementSystem

resources

static

templates

application.properties

data.sql

test

target

.gitattributes

.gitignore

HELP.md

mvnw

mvnw.cmd

SQL dialect is not configured.

Change dialect to... Cancel

```
1 INSERT INTO patient (name, gender , birthdate, email, blood_group)
2 VALUES
3     ( name 'Aarav Sharma', gender 'MALE', birthdate '1998-05-10', email 'aarav.sharma@example.com', blood_group 'O_POSITIVE'),
4     ( name 'Ananya Verma', gender 'FEMALE', birthdate '1999-08-22', email 'ananya.verma@example.com', blood_group 'A_POSITIVE'),
5     ( name 'Rohit Kumar', gender 'MALE', birthdate '1997-12-15', email 'rohit.kumar@example.com', blood_group 'B_POSITIVE'),
6     ( name 'Priya Singh', gender 'FEMALE', birthdate '2000-03-05', email 'priya.singh@example.com', blood_group 'AB_POSITIVE'),
7     ( name 'Karan Mehta', gender 'MALE', birthdate '1996-06-18', email 'karan.mehta@example.com', blood_group 'O_NEGATIVE'),
8     ( name 'Neha Gupta', gender 'FEMALE', birthdate '1998-11-30', email 'neha.gupta@example.com', blood_group 'A_NEGATIVE'),
9     ( name 'Vikram Patel', gender 'MALE', birthdate '1995-09-12', email 'vikram.patel@example.com', blood_group 'B_NEGATIVE'),
10    ( name 'Sneha Iyer', gender 'FEMALE', birthdate '2001-01-25', email 'sneha.iyer@example.com', blood_group 'AB_NEGATIVE'),
11    ( name 'Arjun Rao', gender 'MALE', birthdate '1999-04-09', email 'arjun.rao@example.com', blood_group 'O_POSITIVE');
```

`@SpringBootTest` is the main annotation used for full integration testing in Spring Boot.

I'll explain it from basics → what it loads → how it works internally → when to use / not use → comparison with other test annotations.

1 What is `@SpringBootTest` ?

java

 Copy code

```
@SpringBootTest  
class MyApplicationTests {  
}
```

`@SpringBootTest` tells Spring Boot to load the COMPLETE application context for tests, almost the same way it runs in production.

- ❖ It is a Spring Boot test annotation
- ❖ Used with JUnit
- ❖ Not a JPA annotation

2 What does it load exactly?

When you use `@SpringBootTest`, Spring Boot loads:

- ✓ All `@Configuration` classes
 - ✓ All `@Component`, `@Service`, `@Repository` beans
 - ✓ Database configuration
 - ✓ JPA / Hibernate
 - ✓ Security (if present)
 - ✓ Web layer (if configured)
- 👉 Full application startup



3 What problem does it solve?

It tests that:

- Beans are wired correctly
- Application context starts without errors
- Components work together correctly

This is called an **integration test**.

4 Example usage

java

 Copy code

```
@SpringBootTest
class UserServiceTest {

    @Autowired
    private UserService userService;

    @Test
    void shouldCreateUser() {
        userService.createUser("Abhi");
    }
}
```

Here:

- Spring context is started
- `UserService` is real
- Repository + DB + JPA are real (or test DB)

5 How does it work internally? (IMPORTANT)

Startup flow during test:

```
sql  
↓  
JUnit  
↓  
@SpringBootTest  
↓  
Spring Boot TestContext Framework  
↓  
ApplicationContext (FULL)  
↓  
Beans created + injected
```

 Copy code

It behaves like running:

```
java  
↓  
@SpringApplication.run(...)
```

 Copy code

but inside a test environment.

6 Database behavior with `@SpringBootTest`

If JPA is present:

- Uses configured datasource
- Can use in-memory DB (H2) or real DB
- Works with `@Transactional`

Example:

```
java  
↓  
@SpringBootTest  
@Transactional  
class RepositoryTest {  
}
```

 Copy code

👉 Transaction rolls back after each test



7 Common attributes of `@SpringBootTest`

1 `webEnvironment`

java

Copy code

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
```

Mode	Meaning
MOCK (default)	No real server
RANDOM_PORT	Starts server on random port
DEFINED_PORT	Uses server.port
NONE	No web environment

2 `classes`

java

Copy code

```
@SpringBootTest(classes = TestConfig.class)
```

Loads specific configuration classes only.

3 `properties`

java

Copy code

```
@SpringBootTest(properties = "spring.jpa.show-sql=true")
```

Overrides properties for tests.

8 When SHOULD you use `@SpringBootTest` ?

- ✓ End-to-end tests
 - ✓ Application startup tests
 - ✓ Service + repository integration
 - ✓ Security tests
 - ✓ Real DB interaction
-

9 When you SHOULD NOT use it ✗

- ✗ Simple unit tests
- ✗ Controller-only tests
- ✗ Repository-only tests

Because:

- Slow startup
 - Heavy memory usage
-

10 `@SpringBootTest` vs other test annotations

Annotation	Loads
<code>@SpringBootTest</code>	Full app
<code>@DataJpaTest</code>	JPA only
<code>@WebMvcTest</code>	Controllers only
<code>@MockBean</code>	Mock dependencies
<code>@TestConfiguration</code>	Test-specific config

1 1 Common mistake ❌

Using `@SpringBootTest` everywhere.

Better:

- Unit tests → plain JUnit + Mockito
 - Slice tests → `@DataJpaTest`, `@WebMvcTest`
 - Integration → `@SpringBootTest`
-

💡 One-line interview answer

`@SpringBootTest` is used to load the complete Spring Boot application context for integration testing, allowing all components to work together as they do in production.

🧠 Final mental model

`@SpringBootTest` = "Start my entire application and test it as a whole."

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Structure:** The project is named "hospitalManagementSystem" located at "D:\hospitalManagementSystem". It includes ".idea", ".mvn", "src" (containing "main" and "test" packages), "resources" (with "static", "templates", "application.properties", and "data.sql"), and a "target" folder.
- Code Editor:** The "PatientServiceTest.java" file is open in the editor. The code is a JUnit test for the PatientService. It imports necessary packages, defines a private field for the PatientRepository, and contains a test method named "testPatient" that prints all patients from the repository.
- Toolbars and Status Bar:** The top bar shows tabs for "PatientRepository.java", "data.sql", "HospitalManagementSystemJpaTutorialApplicationTests.java", and "PatientServiceTest.java". The status bar at the bottom shows "Run" and "HospitalManagementSystemJpaTutorialApplication" along with other tabs.

```
package com.codingshuttle.jpaTutorial2.hospitalManagementSystem;

import com.codingshuttle.jpaTutorial2.hospitalManagementSystem.entity.patient;
import com.codingshuttle.jpaTutorial2.hospitalManagementSystem.repositories.PatientRepository;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import java.util.List;

@SpringBootTest
public class PatientServiceTest {

    @Autowired
    private PatientRepository patientRepository;

    @Test
    public void testPatient(){
        List<patient> patientList=patientRepository.findAll();
        for(patient p:patientList){
            System.out.println(p);
        }
    }
}
```

Run HospitalManagementSystemJpaTutorialApplication × PatientServiceTest.testPatient ×

1 test passed 1 test total, 473 ms

PatientServiceTest (com.coc 473 ms)

testPatient() 473 ms

```
WARNING: If a serviceability tool is in use, please run with -XX:+EnableDynamicAgentLoading to hide this warning
WARNING: If a serviceability tool is not in use, please run with -Djdk.instrument.traceUsage for more information
WARNING: Dynamic loading of agents will be disallowed by default in a future release
OpenJDK 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath has been appended
Hibernate:
    select
        p1_.id,
        p1_.birthdate,
        p1_.blood_group,
        p1_.created_at,
        p1_.email,
        p1_.gender,
        p1_.name
    from
        patient p1_
patient(id=1, name=Aarav Sharma, birthdate=1998-05-10, email=aarav.sharma@example.com, gender=MALE, bloodGroup=O_POSITIVE, createdAt=null)
patient(id=2, name=Ananya Verma, birthdate=1999-08-22, email=ananya.verma@example.com, gender=FEMALE, bloodGroup=A_POSITIVE, createdAt=null)
patient(id=3, name=Rohit Kumar, birthdate=1997-12-15, email=rohit.kumar@example.com, gender=MALE, bloodGroup=B_POSITIVE, createdAt=null)
patient(id=4, name=Priya Singh, birthdate=2000-03-05, email=priya.singh@example.com, gender=FEMALE, bloodGroup=AB_POSITIVE, createdAt=null)
patient(id=5, name=Karan Mehta, birthdate=1996-06-18, email=karan.mehta@example.com, gender=MALE, bloodGroup=O_NEGATIVE, createdAt=null)
patient(id=6, name=Neha Gupta, birthdate=1998-11-30, email=neha.gupta@example.com, gender=FEMALE, bloodGroup=A_NEGATIVE, createdAt=null)
patient(id=7, name=Vikram Patel, birthdate=1995-09-12, email=vikram.patel@example.com, gender=MALE, bloodGroup=B_NEGATIVE, createdAt=null)
patient(id=8, name=Sneha Iyer, birthdate=2001-01-25, email=sneha.iyer@example.com, gender=FEMALE, bloodGroup=AB_NEGATIVE, createdAt=null)
patient(id=9, name=Arjun Rao, birthdate=1999-04-09, email=arjun.rao@example.com, gender=MALE, bloodGroup=O_POSITIVE, createdAt=null)
```

Project | + | ⌂ | X | : | - | tRepository.java | data.sql | HospitalManagementSystemJpaTutorialApplicationTests.java | PatientServiceTest.java | IPatientInfo.java | X |

hospitalManagementSystem D:\hospital

.idea
.mvn
src
 main
 java
 com.codingshuttle.jpaTutorial2
 dto
 IPatientInfo
 entity
 type
 BloodGroupType
 Patient
 repositories
 PatientRepository
 HospitalManagementSystem
 resources
 static
 templates
 application.properties
 data.sql

```
1 package com.codingshuttle.jpaTutorial2.hospitalManagementSystem.dto;
2
3 public interface IPatientInfo { 5 usages
4
5     Long getId(); 1 usage
6     String getName(); 1 usage
7     String getEmail();| 1 usage
8 }
9
```

Run | PatientServiceTest.testPatient1 | X

Project ▾

PatientRepository.java × data.sql HospitalManagementSystemJpaTutorialApplicationTests.java

```
1 package com.codingshuttle.jpaTutorial2.hospitalManagementSystem.repositories;
2
3 import com.codingshuttle.jpaTutorial2.hospitalManagementSystem.dto.IPatientInfo;
4 import com.codingshuttle.jpaTutorial2.hospitalManagementSystem.entity.Patient;
5 import org.springframework.data.jpa.repository.JpaRepository;
6 import org.springframework.data.jpa.repository.Query;
7
8 import java.util.List;
9
10 public interface PatientRepository extends JpaRepository<Patient, Long> { 2 usages
11
12     @Query("select p.id as id , p.name as name, p.email as email from Patient p") 1 usage
13     List<IPatientInfo> getAllPatientInfo();
14 }
15
```

Project ▾

PatientRepository.java data.sql HospitalManagementSystemJpaTutorialApplicationTests.java PatientServiceTest.java IPatientI

Patient

repositories

PatientRepository

HospitalManagementSystem

resources

static

templates

application.properties

data.sql

test

java

com.codingshuttle.jpaTutorial2

HospitalManagementSystem

PatientServiceTest

target

.gitattributes

.gitignore

HELP.md

mvnw

mvnw.cmd

```
13     public class PatientServiceTest {
14
15         public void testPatient(){
16             List<Patient> patientList=patientRepository.findAll();
17
18             for(Patient p:patientList){
19                 System.out.println(p);
20             }
21
22         }
23
24         @Test
25         public void testPatient1(){
26             List<IPatientInfo> patientList=patientRepository.getAllPatientInfo();
27
28             for(IPatientInfo p:patientList){
29                 System.out.println("Id: "+p.getId()+" Name: "+p.getName()+" Email: "+p.getEmail());
30             }
31
32         }
33
34     }
35
36 }
37 }
```

Projection done by interface and by classpros and cons of both and how it is done

Below is a clear, complete comparison of Projection by INTERFACE vs by CLASS (DTO) in Spring Data JPA, including how each is done, pros & cons, when to use which, and what happens internally.

1 What is projection (quick reminder)

Projection means fetching only selected fields instead of the full entity.

Spring Data JPA supports:

- Interface-based projection
- Class-based (DTO) projection

Both are read-only and not managed by the persistence context.

2 Interface-based Projection

How it is done

Step 1: Create an interface

java

 Copy code

```
public interface EmployeeView {  
    String getName();  
    double getSalary();  
}
```

★ Method names must match entity field names

Step 2: Use it in repository

java

```
List<EmployeeView> findByDepartment(String department);
```

That's it.

No `@Query` required.

What happens internally

1. Spring Data JPA parses method
2. Generates JPQL selecting only required fields
3. Executes query via EntityManager
4. Creates a proxy implementation of the interface
5. Returns proxy objects

Generated JPQL (conceptually):

jpql

```
SELECT e.name, e.salary  
FROM Employee e  
WHERE e.department = :department
```

Pros of Interface-based Projection

- ✓ Very little code
- ✓ Clean & readable
- ✓ No JPQL needed
- ✓ Spring auto-generates query
- ✓ Works with derived queries
- ✓ Easy to refactor

Cons of Interface-based Projection ✗

- ✗ No constructor logic
 - ✗ Cannot add business logic
 - ✗ Limited customization
 - ✗ Getter names must exactly match fields
 - ✗ Proxy-based (can feel "magical")
-

Best use cases

- ✓ Simple read-only views
 - ✓ REST API responses
 - ✓ Fast development
 - ✓ Pagination + projection
-

3 Class-based Projection (DTO)

How it is done

Step 1: Create DTO class

```
java Copy code  
  
public class EmployeeDTO {  
    private String name;  
    private double salary;  
  
    public EmployeeDTO(String name, double salary) {  
        this.name = name;  
        this.salary = salary;  
    }  
  
    // getters  
}
```

Step 2: Use `@Query` with constructor expression

java

 Copy code

```
@Query("""
    SELECT new com.example.EmployeeDTO(e.name, e.salary)
    FROM Employee e
    WHERE e.department = :department
""")
List<EmployeeDTO> findEmployeeDTOs(@Param("department") String department);
```

What happens internally

1. You write JPQL
 2. EntityManager executes JPQL
 3. Hibernate creates DTO objects using constructor
 4. Returns DTO list
- ❖ No proxy
❖ Explicit mapping

Pros of Class-based Projection

- ✓ Full control over mapping
- ✓ Can add validation / logic
- ✓ Immutable DTOs possible
- ✓ No proxy objects
- ✓ Explicit & predictable

Cons of Class-based Projection

- ✗ More boilerplate code
- ✗ Must write JPQL
- ✗ Constructor must match query exactly
- ✗ Refactoring is harder
- ✗ Errors appear at runtime



Best use cases

- ✓ Complex projections
- ✓ Aggregations
- ✓ Calculated fields
- ✓ Stable, long-term APIs
- ✓ Performance-critical queries

4 Side-by-side Comparison (VERY IMPORTANT)

Aspect	Interface Projection	Class (DTO) Projection
Code	Minimal	More
Query writing	✗ No	✓ Yes
Mapping	Automatic	Manual
Logic inside projection	✗ No	✓ Yes
Proxy used	✓ Yes	✗ No
Compile-time safety	Medium	High
Flexibility	Medium	High
Refactoring ease	High	Medium

5 Performance comparison

- Both fetch only selected columns
 - Both avoid entity loading
 - Performance difference is usually negligible
- ★ Choose based on design, not micro-optimizations.

6 Common mistakes ✖

✖ Expecting projections to be managed

java

 Copy code

```
employeeView.setSalary(50000); // ✖
```

Projections are read-only.

✖ Using entity field names incorrectly

java

 Copy code

```
String getSalaryAmount(); // ✖ if field is salary
```

7 Which one should YOU choose? (Rule of thumb)

- 👉 Use interface-based projection by default
- 👉 Switch to DTO projection when you need control

Simple rule:

Simple data → Interface
Complex data → DTO

8 One-line interview answer (perfect)

Spring Data JPA supports interface-based and class-based projections; interface projections are simpler and auto-generated, while DTO projections provide more control and flexibility at the cost of additional code.

💡 Final mental model

- Interface projection = Spring magic, fast, clean 
- DTO projection = Explicit controlled powerful

if i make use @Data on the class and all the field i make the final then there is not required to write the allArgsConstrructor or what ..just enlighten me with that

Great question 🔥

This is pure Lombok behavior, and once you understand it, constructor confusion disappears.

I'll explain it step by step, with exact rules, what Lombok generates, and when you still need `@AllArgsConstructor`.

1 What does `@Data` actually generate?

java

 Copy code

```
@Data  
public class User {  
    private String name;  
    private int age;  
}
```

`@Data` is a shortcut annotation.

It generates ALL of these at compile time:

- `@Getter` → getters for all fields
 - `@Setter` → setters for all non-final fields
 - `@ToString`
 - `@EqualsAndHashCode`
 - `@RequiredArgsConstructor`  (IMPORTANT)
- ⚠ It does NOT generate `@AllArgsConstructor`

2 What is `@RequiredArgsConstructor`? (KEY POINT)

`@RequiredArgsConstructor` generates a constructor for:

- all final fields
- all fields marked with `@NonNull`

3 Now your case: `@Data` + all fields are `final`

```
java Copy ↗  
@Data  
public class EmployeeDTO {  
    private final String name;  
    private final double salary;  
}
```

What Lombok generates internally

```
java Copy ↗  
public class EmployeeDTO {  
  
    private final String name;  
    private final double salary;  
  
    // GENERATED constructor  
    public EmployeeDTO(String name, double salary) {  
        this.name = name;  
        this.salary = salary;  
    }  
  
    // getters only (no setters because fields are final)  
}
```

 YES — you do NOT need to write `@AllArgsConstructor`

Because:

- `@Data` → includes `@RequiredArgsConstructor`
- All fields are `final`
- `RequiredArgsConstructor == AllArgsConstructor` in this case



⚡ When does `@Data` NOT generate a full constructor?

Case 1: Non-final fields

java

 Copy code

```
@Data  
public class EmployeeDTO {  
    private String name;  
    private double salary;  
}
```

Generated constructor:

java

 Copy code

```
public EmployeeDTO() { }
```

- ✖ No all-args constructor
 - ✖ Because no field is `final` or `@NonNull`
- 👉 You need `@AllArgsConstructor` here

Case 2: Mixed fields

java

 Copy code

```
@Data  
public class EmployeeDTO {  
    private final String name;  
    private int age;  
}
```

Generated constructor:

java

 Copy code

```
public EmployeeDTO(String name) { ... }
```

Only `final` field included.



5 Comparison table (VERY IMPORTANT)

Fields	@Data generates
All fields <code>final</code>	✓ Full constructor
Some fields <code>final</code>	⚠ Partial constructor
No fields <code>final</code>	✗ No-args constructor
Want all fields always	✗ Need <code>@AllArgsConstructor</code>

6 Should you use `@AllArgsConstructor` anyway?

- ✓ For DTOs (recommended)

```
java Co

@Data
@AllArgsConstructor
public class EmployeeDTO {
    private String name;
    private double salary;
}
```

- ✓ Clear intention
- ✓ No surprises
- ✓ Safe refactoring

✓ For immutable DTOs (BEST PRACTICE)

java

 Copy

```
@Data  
public class EmployeeDTO {  
    private final String name;  
    private final double salary;  
}
```

- ✓ Immutable
- ✓ Thread-safe
- ✓ No setters
- ✓ Constructor auto-generated

7 Important warning ⚠ (JPA Entities)

✗ Do NOT make entity fields `final`

JPA requires:

- No-args constructor
- Proxying

So this is ✗ wrong for entities:

java

 Copy

```
@Entity  
@Data  
public class User {  
    private final String name; // ✗  
}
```

8 Final clear answer to your question

If you use `@Data` and make all fields `final`, Lombok generates the required constructor automatically, so you do NOT need to write `@AllArgsConstructor`.
If fields are not final, you must explicitly use `@AllArgsConstructor`.

9 One-line memory rule

`@Data + final fields = constructor auto-created`
`@Data + non-final fields = no all-args constructor`

If you want I can also explain.

Project

Repository.java data.sql HospitalManagementSystemJpaTutorialApplicationTests.java PatientServiceTest.java CPatientInfo.java

```
1 package com.codingshuttle.jpaTutorial2.hospitalManagementSystem.dto;
2
3 import lombok.Data;
4
5 @Data no usages
6 public class CPatientInfo {
7
8     private final Long id;
9     private final String name;
10
11 }
12
```

.idea
.mvn
src
main
java
com.codingshuttle.jpaTutorial2.
dto
CPatientInfo
IPatientInfo
entity
type
BloodGroupType
Patient
repositories
PatientRepository
HospitalManagementSystem
resources
static
templates
application.properties
data.sql
test
java

Project ▾

PatientRepository.java x data.sql HospitalManagementSystemJpaTutorialApplicationTests.java PatientServiceTest.java CPatientInfo.java :

```
3 import com.codingshuttle.jpaTutorial2.hospitalManagementSystem.dto.CPatientInfo;
4 import com.codingshuttle.jpaTutorial2.hospitalManagementSystem.dto.IPatientInfo;
5 import com.codingshuttle.jpaTutorial2.hospitalManagementSystem.entity.Patient;
6 import org.springframework.data.jpa.repository.JpaRepository;
7 import org.springframework.data.jpa.repository.Query;
8
9 import java.util.List;
10
11 public interface PatientRepository extends JpaRepository<Patient, Long> { 2 usages
12
13     @Query("select p.id as id , p.name as name, p.email as email from Patient p") 1 usage
14     List<IPatientInfo> getAllPatientInfo();
15
16
17     //    @Query("select new com.codingshuttle.jpaTutorial2.hospitalManagementSystem.dto.CPatientInfo(p.id, p.name) from Patient p")
18     //    OR both are same
19     @Query("select new com.codingshuttle.jpaTutorial2.hospitalManagementSystem.dto.CPatientInfo(p.id, p.name)" + "from Patient p") 1
20     List<CPatientInfo> getAllPatientsInfoConcrete();
21 }
22
```

hospitalManagementSystem Version control PatientServiceTest.testPatientConcrete

PatientRepository.java data.sql HospitalManagementSystemJpaTutorialApplicationTests.java PatientServiceTest.java CPatientInfo.java

```
public class PatientServiceTest {  
    @Test  
    public void testPatientConcrete(){  
        List<CPatientInfo> patientList=patientRepository.getAllPatientsInfoConcrete();  
        for(CPatientInfo p:patientList){  
            System.out.println(p);  
        }  
    }  
}
```

Project Resources static templates application.properties data.sql

test java com.codingshuttle.jpaTuto HospitalManagementSystem PatientServiceTest

target .gitattributes .gitignore HELP.md mvnw mvnw.cmd pom.xml

External Libraries Scratches and Consoles

Project PatientRepository.java data.sql HospitalManagementSystemJpaTutorialApplicationTests.java PatientServiceTest.java CPatientInfo.java

PatientRepository HospitalManagementSystem resources static templates

```
public class PatientServiceTest {  
    @Test  
}
```

Run PatientServiceTest.testPatientConcrete

PatientServiceTest (com.cod 383 ms)
 testPatientConcrete() 383 ms

✓ 1 test passed 1 test total, 383 ms

WARNING: If a serviceability tool is in use, please run with -XX:+EnableDynamicAgentLoading to hide this warning
WARNING: If a serviceability tool is not in use, please run with -Djdk.instrument.traceUsage for more information
WARNING: Dynamic loading of agents will be disallowed by default in a future release
OpenJDK 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath has been appended
Hibernate:
 select
 p1_.id,
 p1_.name
 from
 patient p1_
 CPatientInfo(id=1, name=Aarav Sharma)
 CPatientInfo(id=2, name=Ananya Verma)
 CPatientInfo(id=3, name=Rohit Kumar)
 CPatientInfo(id=4, name=Priya Singh)
 CPatientInfo(id=5, name=Karan Mehta)
 CPatientInfo(id=6, name=Neha Gupta)
 CPatientInfo(id=7, name=Vikram Patel)
 CPatientInfo(id=8, name=Sneha Iyer)
 CPatientInfo(id=9, name=Arjun Rao)

```
@Query("select new  
com.codingshuttle.jpaTutorial2.hospitalManagementSystem.dto.CPatientInfo(p.id, p.name) " + "from Patient p")  
@Query("select new  
com.codingshuttle.jpaTutorial2.hospitalManagementSystem.dto.CPatientInfo(p.id, p.name) from Patient p")
```

You've pasted the same JPQL constructor-expression query twice, just formatted differently.
Let me explain what this query means, how it works, why it's written like this, and what must exist for it to work.

1 The query you wrote (clean version)

java

 Copy code

```
@Query(  
    "select new com.codingshuttle.jpaTutorial2.hospitalManagementSystem.dto.CPatientInfo(p.id, p.name)  
    "from Patient p"  
)
```

This is JPQL constructor-based projection.

2 What does `select new ...` mean in JPQL?

`select new` tells JPA:

"Do NOT return entities.

Create a new object using this constructor and return that instead."

So this query:

jpql

 [Copy code](#)

```
select new CPatientInfo(p.id, p.name)
from Patient p
```

means:

- Fetch only `id` and `name`
 - Create a `CPatientInfo` object
 - Put `(id, name)` into its constructor
 - Return a list of `CPatientInfo`
- ★ This is class-based (DTO) projection

3 What must exist for this to work? (VERY IMPORTANT)

✓ DTO class must have a matching constructor

```
java Copy code  
  
package com.codingshuttle.jpaTutorial2.hospitalManagementSystem.dto;  
  
public class CPatientInfo {  
  
    private Long id;  
    private String name;  
  
    public CPatientInfo(Long id, String name) {  
        this.id = id;  
        this.name = name;  
    }  
}
```

⚠ Constructor parameter order and types must match exactly
Otherwise → runtime error.

4 Why is the FULL package name required?

```
java Copy code  
  
new com.codingshuttle.jpaTutorial2.hospitalManagementSystem.dto.CPatientInfo(...)
```

Because:

- JPQL is not Java
- It cannot resolve imports
- You must give the fully qualified class name

✗ This will NOT work:

```
java Copy code  
  
select new CPatientInfo(p.id, p.name)  
↓
```

5 What does Patient p mean?

jpql

 Copy code

```
from Patient p
```

- Patient → entity name (not table name)
- p → alias used in query

So:

jpql

 Copy code

```
p.id  
p.name
```

are entity fields, not column names.

6 What is returned by this repository method?

Repository method signature

java

 Copy code

```
@Query("""  
    select new com.codingshuttle.jpaTutorial2.hospitalManagementSystem.dto.CPatientInfo(p.id, p.name)  
    from Patient p  
""")  
List<CPatientInfo> findPatientInfo();
```

Returned value:

java

 Copy code

```
List<CPatientInfo>
```

- ✓ Not entities
- ✓ Not managed by persistence context
- ✓ Read-only objects



7 Why use this instead of returning Patient?

Advantages ✓

- Fetches only required columns
- Faster queries
- Less memory usage
- No lazy-loading issues
- Safe for API responses

Trade-offs ✗

- Not managed by JPA
- Cannot update and save
- Constructor must match exactly

8 Why did you write the query in two lines with + ?

This is just Java string concatenation for readability.

These two are 100% equivalent:

```
java Copy code
@Query(
    "select new CPatientInfo(p.id, p.name) " +
    "from Patient p"
)
```

```
java Copy code
@Query(
    "select new CPatientInfo(p.id, p.name) from Patient p"
)
```

👉 No behavioral difference at all.

9 Common mistakes ✖

✖ Missing constructor

text

 Copy code

```
No suitable constructor found
```

✖ Wrong package name

text

 Copy code

```
Class not found
```

✖ Using table/column names instead of entity/field names

10 One-line interview answer (perfect)

This `@Query` uses a JPQL constructor expression to project selected entity fields into a DTO by invoking its constructor, returning lightweight, unmanaged objects instead of full entities.

💡 Final mental model

pgsql

 Copy code

```
JPQL "select new"
  → calls DTO constructor
  → returns DTO list
  → NO entity, NO persistence context
```

localhost * < N/A > Auto

localhost product_table hospitalDB test public localhost localhost hospitaldb patient "2

General

Connections

localhost localhost:3306

Databases

hospitaldb

- Tables (16K)
- Views
- Indexes
- Procedures
- Triggers
- Events
- sys
- test

Users

Administrator

System Info

postgres localhost:5432

Bookmarks

Dashboards

Diagrams

Scripts

```
(12.4,4,'parie/34','Parie Biscuit'),  
(14.4,1,'pepsi1123','Pepsi'),  
(16.4,3,'mazza1156','Mazza');
```

```
select * from test.product_table;
```

```
select hospitalDB from pg_database;
```

```
use hospitaldb;
```

```
select blood_group, count(*)  
from patient  
where blood_group is not null  
group by blood_group;
```

patient 1

select blood_group, count(*) from patient wh Enter a SQL expression to filter results (use Ctrl+Space)

	AZ blood_group	123 count(*)
1	O_POSITIVE	2
2	A_POSITIVE	1
3	B_POSITIVE	1
4	AB_POSITIVE	1

Value X

B_POSITIVE

O_NEGATIVE

O_POSITIVE

Refresh Save Cancel Export data 200 8 ... 8 row(s) fetched - 0.002s, on 2025-12-29 at 03:59:09

DBeaver 25.3.0 - <localhost> Script

File Edit Navigate Search SQL Editor Database Window Help

localhost < N/A >

product_table hospitalDB test public localhost localhost hospitaldb patient

General

Connections

localhost localhost:3306

Databases

hospitaldb

- Tables (16K)
 - patient
 - Views
 - Indexes
 - Procedures
 - Triggers
 - Events
- sys
- test
- Users
- Administrator
- System Info

postgres localhost:5432

Bookmarks

Dashboards

Diagrams

Scripts

patient 1

select blood_group, count(*) from patient wh Enter a SQL expression to filter results (use Ctrl+Space)

	AZ blood_group	123 count(*)
1	O_POSITIVE	2
2	A_POSITIVE	1
3	B_POSITIVE	1
4	AB_POSITIVE	1
5	O_NEGATIVE	1
6	A_NEGATIVE	1
7	B_NEGATIVE	1

Value

- A_NEGATIVE
- A_POSITIVE
- B_NEGATIVE
- B_POSITIVE
- O_NEGATIVE
- O_POSITIVE

8 row(s) fetched - 0.002s, on 2025-12-29 at 04:01:02

Files - General

Name

Bookmarks

Dashboards

Diagrams

Scripts

DataSource

Refresh Save Cancel Export data 200 8 ... 8 row(s) fetched - 0.002s, on 2025-12-29 at 04:01:02

Connections

Filter objects (tables, procedures, users, etc.) by name

localhost localhost:3306

IST en Writable Smart Insert 15 : 1 [122] Sel: 122 | 5

The screenshot shows a Java development environment with the following details:

- Project Bar:** Displays "Project" with a dropdown, followed by several icons and tabs. The active tab is "BloodGroupStats.java". Other tabs include "data.sql", "HospitalManagementSystemJpaTutorialApplicationTests.java", "PatientServiceTest.java", and "CPatientInfo.java".
- File Structure (Left Panel):** The project structure under "hospitalManagementSystem" is shown:
 - /.idea
 -
.mvn
- src
 - main
 - java
 - com.codingshuttle.jpaTuto
 - dto
 - BloodGroupStats
 - CPatientInfo
 - IPatientInfo
 - entity
 - type
 - BloodGroupType
 - Patient
 - repositories
 - PatientRepository
 - HospitalManagementS
 - resources
 - static
 - templates

- Code Editor (Right Panel):** The code for "BloodGroupStats.java" is displayed:

```
1 package com.codingshuttle.jpaTutorial2.hospitalManagementSystem.dto;
2
3
4 import com.codingshuttle.jpaTutorial2.hospitalManagementSystem.entity.type.BloodGroupType;
5 import lombok.Data;
6
7 @Data 5 usages
8 public class BloodGroupStats {
9
10     private final BloodGroupType bloodGroupType;
11
12     private final Long count;
13 }
14 }
```

Project ▾

hospitalManagementSystem D:\hos

- > .idea
- > .mvn
- src
 - main
 - java
 - com.codingshuttle.jpaTuto
 - dto
 - BloodGroupStats
 - CPatientInfo
 - IPatientInfo
 - entity
 - type
 - BloodGroupType
 - Patient
 - repositories
 - PatientRepository
 - HospitalManagementSystemJpaTutorialApplicationTests.java
 - resources

PatientRepository.java x BloodGroupStats.java data.sql HospitalManagementSystemJpaTutorialApplicationTests.java PatientServiceTi :

```
12 public interface PatientRepository extends JpaRepository<Patient, Long> { 2 usages
13
14     @Query("select new com.codingshuttle.jpaTutorial2.hospitalManagementSystem.dto.BloodGroupStats(p.bloodGroup, " + 1 usage
15             "COUNT(p)) from Patient p group by p.bloodGroup order by COUNT(p) desc")
16     List<BloodGroupStats> getBloodGroupStats();
17 }
```

hospitalManagementSystem

Project

Repository.java BloodGroupStats.java data.sql HospitalManagementSystemJpaTutorialApplicationTests.java PatientServiceTest.java

```
public class PatientServiceTest {  
    public void testPatientConcrete(){  
        }  
    }  
    @Test  
    public void testPatientConcreteBloodGroupStats(){  
        List<BloodGroupStats> patientList=patientRepository.getBloodGroupStats();  
        for(BloodGroupStats p:patientList){  
            System.out.println(p);  
        }  
    }  
}
```

type
BloodGroupType
Patient
repositories
PatientRepository
HospitalManagementSystem
resources
static
templates
application.properties
data.sql
test
java
com.codingshuttle.jpaTuto
HospitalManagementSystem
PatientServiceTest

target
.gitattributes
.gitignore

✓ 1 test passed | 1 test total, 408 ms

```
select
    p1_0.blood_group,
    count(p1_0.id)
from
    patient p1_0
group by
    p1_0.blood_group
order by
    count(p1_0.id) desc
```

bloodGroupStats(bloodGroupType=O_POSITIVE, count=2)
bloodGroupStats(bloodGroupType=A_POSITIVE, count=1)
bloodGroupStats(bloodGroupType=B_POSITIVE, count=1)
bloodGroupStats(bloodGroupType=AB_POSITIVE, count=1)
bloodGroupStats(bloodGroupType=O_NEGATIVE, count=1)
bloodGroupStats(bloodGroupType=A_NEGATIVE, count=1)
bloodGroupStats(bloodGroupType=B_NEGATIVE, count=1)
bloodGroupStats(bloodGroupType=AB_NEGATIVE, count=1)

tell the projection by class for the update ,delete ,other query where many other annotations are used to take the precautions

Great question 🔥

You're asking about class-based (DTO) projection beyond simple SELECTs and what precautions + annotations are required for UPDATE / DELETE / complex queries in Spring Data JPA.

This is advanced but very important, so I'll explain it in a structured + practical way.

1 First: VERY IMPORTANT TRUTH (must be clear)

- ! DTO (class-based) projection is ONLY for SELECT queries.
- ✗ DTO projection is NOT used for UPDATE or DELETE queries.

Why?

- UPDATE / DELETE do not return rows
- DTOs exist to receive data
- UPDATE / DELETE return only row count (int/long)

So we'll split this into:

1. DTO projection for SELECT
 2. UPDATE / DELETE queries (what to use instead)
 3. Precautions & annotations (CRITICAL)
-

2 Class-based Projection for SELECT queries (recap)

DTO

```
java Copy code  
  
public class PatientInfoDTO {  
    private Long id;  
    private String name;  
  
    public PatientInfoDTO(Long id, String name) {  
        this.id = id;  
        this.name = name;  
    }  
}
```

Repository

```
java Copy code  
  
@Query("""  
    SELECT new com.example.dto.PatientInfoDTO(p.id, p.name)  
    FROM Patient p  
    WHERE p.active = true  
""")  
List<PatientInfoDTO> findActivePatients();
```

Key rules

- ✓ Constructor must match exactly
- ✓ Fully qualified class name
- ✓ DTO is NOT managed
- ✓ Read-only

3 UPDATE queries (NO DTO projection here)

Correct way to write UPDATE query

java

 Copy code

```
@Modifying  
@Transactional  
@Query("""  
    UPDATE Patient p  
    SET p.status = :status  
    WHERE p.id = :id  
""")  
int updatePatientStatus(  
    @Param("status") Status status,  
    @Param("id") Long id  
);
```

What this returns

java

 Copy code

int → number of rows updated

- ✖ You cannot return DTO
- ✖ You cannot return Entity

Required annotations (VERY IMPORTANT)

1 @Modifying

Tells Spring Data JPA:
"This is NOT a SELECT query."

Without it ✗:

```
nginx
```

 Copy code

```
InvalidDataAccessApiUsageException
```

2 @Transactional

Required because UPDATE/DELETE modify data

Without it ✗:

```
nginx
```

 Copy code

```
TransactionRequiredException
```

4 DELETE queries (same rules)

```
java
```

 Copy code

```
@Modifying  
@Transactional  
@Query("DELETE FROM Patient p WHERE p.inactive = true")  
int deleteInactivePatients();
```

- ✓ Returns number of rows deleted
- ✗ No DTO possible

5 CRITICAL precaution: Persistence Context sync

Problem

Bulk UPDATE / DELETE:

- ✗ Bypass persistence context
- ✗ Managed entities become stale

Solution

Use:

```
java
```

 Copy code

```
@Modifying(clearAutomatically = true, flushAutomatically = true)
```

Correct & SAFE version

```
java
```

 Copy code

```
@Modifying(clearAutomatically = true, flushAutomatically = true)
@Transactional
@Query("""
    UPDATE Patient p
    SET p.status = :status
    WHERE p.age > :age
""")
int updateStatusForOlderPatients(
    @Param("status") Status status,
    @Param("age") int age
);
```

What this does

- `flushAutomatically` → flush pending changes
- `clearAutomatically` → clear persistence context
- Prevents stale entity bugs

6 Complex SELECT queries with DTO + precautions

Aggregation + DTO projection

```
java Copy code  
  
public class DepartmentStatsDTO {  
    private String department;  
    private long count;  
  
    public DepartmentStatsDTO(String department, long count) {  
        this.department = department;  
        this.count = count;  
    }  
}
```

```
java Copy code  
  
@Query("""  
    SELECT new com.example.dto.DepartmentStatsDTO(  
        p.department,  
        COUNT(p)  
    )  
    FROM Patient p  
    GROUP BY p.department  
""")  
List<DepartmentStatsDTO> getDepartmentStats();
```

- ✓ Safe
- ✓ Efficient
- ✓ Read-only

7 Native SQL + DTO (advanced & risky)

java

 Copy code

```
@Query(  
    value = """  
        SELECT id, name  
        FROM patient  
        WHERE active = true  
    """,  
    nativeQuery = true  
)  
List<Object[]> findRawData();
```

Then manually map to DTO ❌ (error-prone)

👉 Prefer JPQL constructor projection whenever possible.

8 Summary table (VERY IMPORTANT)

Operation	DTO Projection	Required Annotations
SELECT	✓ YES	@Query(select new ...)
UPDATE	✗ NO	@Modifying, @Transactional
DELETE	✗ NO	@Modifying, @Transactional
COUNT	✗ NO	return long
AGGREGATION	✓ YES	DTO constructor
BULK UPDATE	✗ NO	clearAutomatically=true

9 Common mistakes ✗

- ✗ Trying to return DTO from UPDATE
 - ✗ Forgetting `@Modifying`
 - ✗ Forgetting `@Transactional`
 - ✗ Not clearing persistence context
 - ✗ Expecting DTO to be managed
-

💡 Final golden rules (memorize these)

- 1 DTO projection is ONLY for SELECT
 - 2 UPDATE / DELETE return row counts, not objects
 - 3 Always use `@Modifying + @Transactional` for write queries
 - 4 Clear persistence context after bulk updates
-

🧠 One-line interview answer (perfect)

Class-based projection in Spring Data JPA is used only for SELECT queries via JPQL constructor expressions, while UPDATE and DELETE queries require `@Modifying` and `@Transactional` and return row counts, with persistence context synchronization handled explicitly.

Below is a clear, complete explanation of `@Transactional` and `@Modifying`—what they are, why they're needed, how they work together, and the exact rules you must follow in Spring Data JPA.

1 `@Transactional` — what it really does

java

 Copy code

```
@Transactional
```

Defines a transaction boundary: all DB operations inside it run as **one atomic unit**.

What Spring does when a method is `@Transactional`

1. Opens a DB transaction
2. Executes the method
3. Commit if success
4. Rollback if an unchecked exception occurs

Why it's needed

- DB changes must be committed
- Guarantees **consistency**
- Enables rollback on failure

Typical uses of `@Transactional`

- Service-layer methods
- Repository write operations (update/delete)
- Multiple DB operations that must succeed or fail together

java

 Copy code

```
@Transactional  
public void createUserAndRole() {  
    userRepo.save(user);  
    roleRepo.save(role);  
}
```

2 @Modifying — what it really does

java

 Copy code

`@Modifying`

Tells Spring Data JPA: "This `@Query` is NOT a SELECT."

By default, Spring Data JPA assumes every `@Query` is a SELECT.

`@Modifying` switches it to a write query.

When `@Modifying` is REQUIRED

- JPQL `UPDATE`
- JPQL `DELETE`
- Native SQL `UPDATE/DELETE`

java

 Copy code

`@Modifying`

```
@Query("UPDATE User u SET u.active = false WHERE u.lastLogin < :date")
int deactivateOldUsers(@Param("date") LocalDate date);
```

What it returns

- `int` or `long` → number of rows affected

3 Why you usually need BOTH together

✗ @Modifying WITHOUT @Transactional

java

 Copy code

```
@Modifying  
@Query("DELETE FROM User u WHERE u.active = false")  
int deleteInactive();
```

Result:

TransactionRequiredException

Because:

- Write queries must run in a transaction

✗ @Transactional WITHOUT @Modifying

java

 Copy code

```
@Transactional  
@Query("DELETE FROM User u WHERE u.active = false")  
int deleteInactive();
```

Result:

Spring still treats it as SELECT → error

Correct combination

java

Copy code

```
@Modifying  
@Transactional  
@Query("DELETE FROM User u WHERE u.active = false")  
int deleteInactive();
```

- ✓ Transaction started
- ✓ Query executed as DELETE
- ✓ Rows deleted

4 VERY IMPORTANT: Persistence Context side effect

Problem (often missed)

Bulk `UPDATE/DELETE`:

- Bypass the persistence context
- Managed entities may become stale

java

Copy code

```
@Modifying  
@Transactional  
@Query("UPDATE User u SET u.status='BLOCKED'")  
int blockAll();
```

If entities are already loaded → they still show old values.

The fix (BEST PRACTICE)

java

 Copy code

```
@Modifying(  
    clearAutomatically = true,  
    flushAutomatically = true  
)  
@Transactional  
@Query("UPDATE User u SET u.status='BLOCKED'")  
int blockAll();
```

What these flags do

- `flushAutomatically=true`
→ flush pending changes before query
- `clearAutomatically=true`
→ clears persistence context after query (prevents stale data)

5 Where to place `@Transactional`

Best practice

- Service layer → `@Transactional`
- Repository layer → only when needed for bulk writes

java

 Copy code

```
@Service  
@Transactional  
public class UserService {  
    // call repository methods here  
}
```

6 Common mistakes ✖

- ✖ Using `@Modifying` for SELECT
- ✖ Forgetting `@Transactional`
- ✖ Expecting entities to auto-update after bulk query
- ✖ Returning DTO/entity from UPDATE/DELETE

7 Quick comparison (memorize this)

Annotation	Purpose
<code>@Transactional</code>	Starts/controls transaction
<code>@Modifying</code>	Marks query as UPDATE/DELETE
Both together	Required for write queries

8 One-line interview answers (perfect)

`@Transactional`

Defines a transactional boundary ensuring atomic commit or rollback of database operations.

`@Modifying`

Marks a repository `@Query` as a modifying query (UPDATE/DELETE) instead of a SELECT.

Together

Bulk write queries in Spring Data JPA require both `@Modifying` and `@Transactional`.

🧠 Final mental model

- `@Transactional` = "start/commit/rollback"
- `@Modifying` = "this is not a SELECT"
- Bulk writes bypass persistence context → clear ↓

Project ▾

PatientRepository.java × BloodGroupStats.java data.sql HospitalManagementSystemJpaTutorialApplicationTests.java PatientServiceTi...

```
15  public interface PatientRepository extends JpaRepository<Patient, Long> { 2 usages
16
17
18
19
20
21
22
23
24
25
26
27      @Query("select new com.codingshuttle.jpaTutorial2.hospitalManagementSystem.dto.BloodGroupStats(p.bloodGroup, " + 1 usage
28          "COUNT(p)) from Patient p group by p.bloodGroup order by COUNT(p) desc")
29      List<BloodGroupStats> getBloodGroupStats();
30
31
32      @Transactional 1 usage
33      @Modifying
34      @Query("update Patient p set p.name= :name where p.id=:id")
35      int updatePatientNameWithId(@Param("name") String name,@Param("id") Long id);
36  }
```

The screenshot shows an IDE interface with a dark theme. On the left is the Project Explorer, displaying the project structure with files like `templates`, `application.properties`, `data.sql`, and `PatientServiceTest.java`. The `PatientServiceTest.java` file is currently selected and open in the main editor area. The code in the editor is:

```
15  public class PatientServiceTest {  
50      public void testPatientConcreteBloodGroupStats(){  
55          }  
56      }  
57  
58      @Test  
59      public void testPatientNameWithId(){  
60          int rowAffected=patientRepository.updatePatientNameWithId( name: "Aloo", id: 1L);  
61          System.out.println("Rows affected: "+rowAffected);  
62      }  
63  
64      }  
65
```

The code uses JUnit annotations (`@Test`) and a Spring Data JPA repository (`patientRepository`). A yellow lightbulb icon is visible near the line `System.out.println("Rows affected: "+rowAffected);`, indicating a potential issue or suggestion.

The screenshot shows an IDE interface with the following details:

- Project View:** On the left, the project structure is shown under the "Project" tab. It includes files like `application.properties`, `data.sql`, and several test classes under the `test` directory, including `PatientServiceTest`.
- Code Editor:** The main editor window displays the `PatientServiceTest.java` file. The code contains two test methods: `testPatientConcreteBloodGroupStats()` and `testPatientNameWithId()`. The `testPatientNameWithId()` method is currently being run.
- Run Tab:** Below the editor, the "Run" tab is active, showing the test configuration: `PatientServiceTest.testPatientNameWithId`.
- Output Window:** The bottom half of the screen shows the execution results. It starts with a green checkmark indicating "1 test passed" and a total time of "1 test total, 362 ms".
 - A warning message from Mockito is displayed: "MOCKITO IS CURRENTLY SELF-ATTACHING TO ENABLE THE INLINE-MOCK-MAKER. THIS WILL NO LONGER WORK IN FUTURE RELEASES OF THE SDK. PLEASE ADD IN".
 - Java agent loading information: "WARNING: A Java agent has been loaded dynamically (C:\Users\ASUS\.m2\repository\net\bytebuddy\byte-buddy-agent\1.17.8\byte-buddy-agent-1.17.8.jar)".
 - Serviceability tool warning: "WARNING: If a serviceability tool is in use, please run with -XX:+EnableDynamicAgentLoading to hide this warning".
 - Instrumentation warning: "WARNING: If a serviceability tool is not in use, please run with -Djdk.instrument.traceUsage for more information".
 - OpenJDK warning: "WARNING: Dynamic loading of agents will be disallowed by default in a future release".
 - Hibernate update log:

```
OpenJDK 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath has been appended
Hibernate:
    update
        patient p1_0
    set
        name=?
    where
        p1_0.id=?
```
 - Execution output: "Rows affected: 1"
- Bottom Status Bar:** The status bar at the bottom provides system information: "hospitalManagementSystem > src > test > java > com > codingshuttle > jpaTutorial2 > hospitalManagementSystem > PatientServiceTest > testPatientNameWithId", along with the current time "61:59", encoding "CRLF", character set "UTF-8", and code style settings "4 spaces".