**1️⃣ Normal Java class flow**

Example:

public class MyClass {

public void sayHello() {

System.out.println("Hello");

}

}

Steps:

1. You write MyClass.java and save it on disk (HDD/SSD).
2. You compile it → MyClass.class (also stored on disk).
3. JVM loads .class file from disk into **RAM**.
4. JVM executes it from RAM.

**Key point:**  
Both .java and .class exist **physically on disk** before execution.

**2️⃣ In-memory Java class flow (Spring Proxy style)**

Example: StudentRepository$$Proxy created by Spring.

Steps:

1. You run the application.
2. Spring generates **Java source code in memory** (not on disk) for the proxy class.
3. It compiles that code **in memory** — no .class file stored on disk.
4. JVM loads the generated .class **from RAM**.
5. JVM executes it directly.

**Key point:**

* The .java file **never exists on your disk** — it’s generated in memory at runtime.
* The .class file is also temporary, kept in RAM only while the app is running.
* Once the app stops, the proxy class disappears.

✅ **Connection to Spring repositories:**  
When you write only the interface:

public interface StudentRepository extends JpaRepository<Student, Long> {}

Spring:

* **Generates the implementation in memory** (StudentRepository$$Proxy).
* Compiles and loads it in memory.
* Lets you call methods on it like a normal Java object — but the class never exists on disk.

### ****Limitations of Spring ORM****

Spring ORM (without Spring Data) still requires **a lot of manual work**:

#### **1️⃣ Too many DAO classes**

* Suppose you have **500 tables** in your database.
* For each table:
  + You create a **DAO interface** (e.g., StudentDAO, CourseDAO).
  + You create a **DAO implementation class** (e.g., StudentDAOImpl, CourseDAOImpl).
* That means **500 interfaces + 500 implementations** = 1000 files to manage.

#### **2️⃣ Repeated (boilerplate) CRUD code**

* Every DAO implementation repeats the same **CRUD** operations:
  + save()
  + findById()
  + findAll()
  + update()
  + delete()
* The only thing that changes is the **entity type** (Student, Course, etc.).
* This **duplicates logic** and makes maintenance harder:
  + If you change one common method, you must update it in 500 places.
  + Higher chance of **human errors**.

### ****How Spring Data solves this****

Spring Data introduced **predefined repository interfaces** like:

* CrudRepository
* JpaRepository

You **only** create one interface per entity:

public interface StudentRepository extends JpaRepository<Student, Long> {}

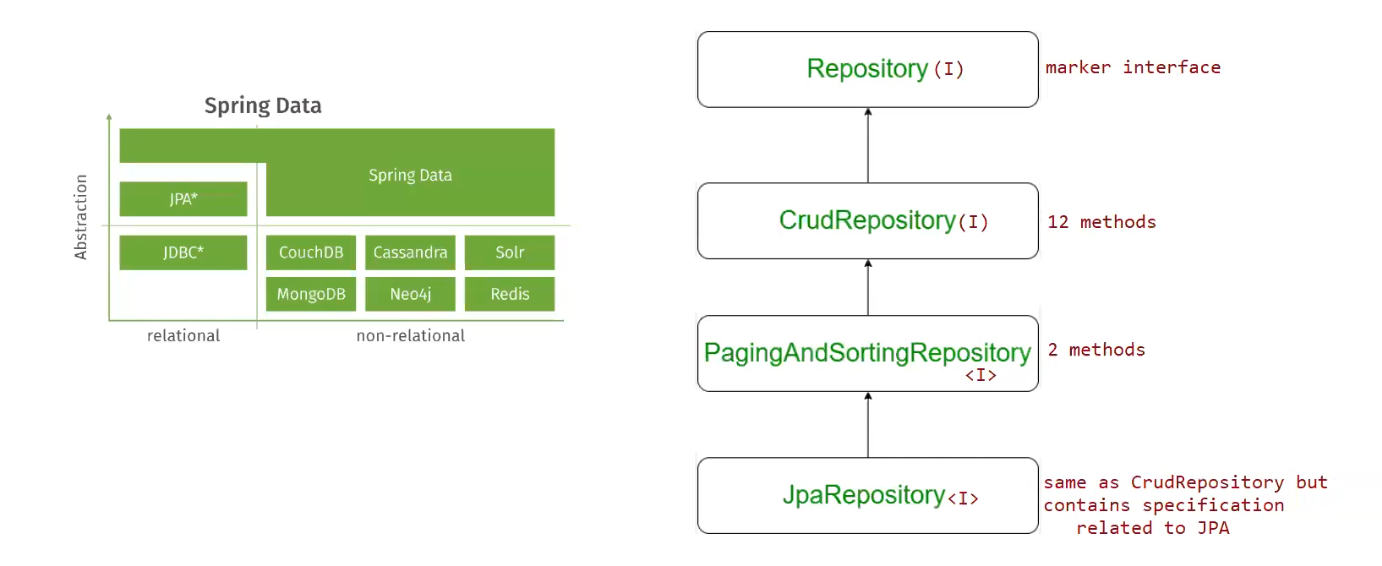
✅ **No implementation class needed** — Spring generates it at runtime (via proxy + in-memory class).  
✅ **No repeated CRUD code** — all common methods (save, findAll, delete, etc.) are already implemented in JpaRepository.  
✅ You just **call** the method, Spring handles the SQL and execution.

**Spring Data JPA**

* **It works on top of Hibernate** → Hibernate is the ORM (Object Relational Mapping) tool that actually talks to the database.
* **You still need strong Hibernate knowledge** → Even though Spring Data JPA hides a lot of code, the concepts (entities, mappings, relationships, lazy loading, etc.) are pure Hibernate/JPA concepts.

**When writing Entity classes with annotations, follow this order:**

1. **JPA annotations** → Standard annotations defined by Java Persistence API, work with any JPA provider (e.g., @Entity, @Id, @Table, @Column).
2. **Java config annotations** → Annotations that are part of Java language features, like @Override or @Deprecated (these are rare in entities but possible).
3. **Hibernate-specific annotations** → Extra features from Hibernate that are not in standard JPA (e.g., @CreationTimestamp, @UpdateTimestamp, @Type).
4. **Third-party annotations** → Anything from other libraries/frameworks (e.g., Lombok’s @Data, @Builder).



## **Left Side: Spring Data Overview**

* **Spring Data** is a big umbrella that makes working with databases easier.
* It works for both:
  + **Relational databases** (like MySQL, PostgreSQL, Oracle) using:
    - **JPA** (Java Persistence API)
    - **JDBC** (Java Database Connectivity)
  + **Non-relational (NoSQL) databases** like MongoDB, Cassandra, CouchDB, Neo4j, Redis, etc.
* The idea: **Same programming style for any database type**, so you don’t have to write custom code for each database.

## **Right Side: Repository Hierarchy in Spring Data JPA**

1. **Repository (I)**
   * A **marker interface** (means it has no methods).
   * Tells Spring: “This is a repository interface. Generate the implementation for me.”
2. **CrudRepository (I)**
   * Inherits from Repository.
   * Has **12 methods** for basic CRUD (Create, Read, Update, Delete) operations.
   * Examples:
     + save(entity)
     + findById(id)
     + findAll()
     + delete(entity)
3. **PagingAndSortingRepository (I)**
   * Extends CrudRepository.
   * Adds **2 more methods** for:
     + **Pagination** → Breaking results into pages.
     + **Sorting** → Ordering results by a field.
4. **JpaRepository (I)**
   * Extends PagingAndSortingRepository.
   * Same as CrudRepository but adds **JPA-specific features** like:
     + Batch operations
     + Flush changes to DB immediately
     + JPA query execution methods

Eg: SpringBootSpringDataJpaSave

## **Step-by-Step Timeline**

1. **Spring Boot starts**
   * Runs SpringApplication.run(...).
2. **Component Scanning**
   * Spring scans your packages (starting from your main app package) and finds @Component classes — including SaveStudentRunner.
3. **Bean Creation**
   * Spring sees SaveStudentRunner needs a StudentRepository in its constructor.
   * Spring **first** ensures StudentRepository is also available as a bean.
     + StudentRepository is created automatically because it’s an interface that extends JpaRepository and is annotated with @Repository (implicitly through Spring Data JPA).
   * Once it has a StudentRepository instance ready, Spring **calls the constructor**:

new SaveStudentRunner(studentRepositoryBean)

* + This is **constructor injection** — happens before any other method in your bean is called.

1. **Bean Fully Initialized**
   * After the constructor finishes, your bean is now part of the application context.
2. **CommandLineRunner Execution**
   * Once **all beans** are ready, Spring Boot looks for all beans that implement CommandLineRunner and calls their run() method.
   * That’s when your database save operations happen.

## 📝 Detailed Notes – saveAll() in Spring Data JPA (Insert vs. Update)

### ****1️⃣ How**** saveAll() ****decides between INSERT and UPDATE****

When you call saveAll() with a list of entities:

* **If the id field is null** → Hibernate understands that **this is a new record** and needs to be added → It uses **INSERT** in the database.
* **If the id field has a value** (not null) → Hibernate assumes **this record already exists** in the database.  
  → It will first run a **SELECT** query to check if a row with that ID is already present.
  + If it exists → Hibernate will use **UPDATE** to change the data.
  + If it does not exist → You may get an error (because Hibernate tried to update something that is not there).

💡 **Why Hibernate checks first:** It does this to avoid inserting a record that could cause duplicate primary key errors.

### ****2️⃣ Common mistakes that cause issues****

A very common mistake is:

* **Manually giving an ID for a new record** (example: new Student(101L, "John", ...) when there is no record with ID 101 in the DB).
* Hibernate will think: “Ah! This is already in the DB, let me update it.” → Then it does a SELECT → Finds nothing → You might get a StaleObjectStateException or **nothing happens** (no insert).

💡 So, **if you want a new row in the DB, never set the ID manually unless you are absolutely sure** it already exists.

### ****3️⃣ Correct way to INSERT using**** saveAll()

* Let the **database** give the ID.
* You can do this by keeping id as null when creating the entity.
* Example:

List<Student> students = List.of(

new Student(null, "ABC", "Java", 15000.0),

new Student(null, "XYZ", "Python", 20000.0)

);

studentRepository.saveAll(students);

Here:

* Hibernate sees id = null → Treats them as new rows.
* Builds **two INSERT statements** and sends them to the DB.
* After the DB commits, Hibernate will **fill in the IDs** for each object automatically.

### ****4️⃣ Entity setup for auto ID generation****

If you want IDs to be **generated by the DB automatically**, use @GeneratedValue in your entity:

#### Oracle (sequence based):

@Id

@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "student\_seq")

@SequenceGenerator(name = "student\_seq", sequenceName = "STUDENT\_SEQ", allocationSize = 1)

private Long id;

#### MySQL (auto increment):

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

💡 The above tells Hibernate: "Don’t wait for me to give you an ID, go ask the DB for it."

### ****5️⃣ What happens inside Hibernate when using**** saveAll() ****for INSERT****

Here’s the flow:

You call saveAll(List<Student>)

→ Hibernate checks each Student's id

→ If id is null → persist() → INSERT into DB

→ If id has value → merge() → SELECT from DB

→ If row exists → UPDATE

→ If not found → error or nothing

→ Hibernate commits the transaction

→ Hibernate returns the same Student objects but now with the generated IDs

### ****6️⃣ Quick Rules to Always Remember****

* ✅ **Want a new row?** Keep id = null and let DB generate it.
* ✅ Always use @GeneratedValue so Hibernate knows how to get the ID.
* ❌ Don’t give an ID for a new record unless you mean to update an existing row.
* ✅ save() (single object) and saveAll() (list) follow the **same rule** for deciding INSERT or UPDATE.

Eg: SpringBootDataJpaSaveAll

## **Spring Data JPA –** findById() **Example Notes**

### ****1. Purpose****

* findById(ID id) is a **ready-made method** provided by JpaRepository.
* It is used to **fetch a single record** from the database using its **primary key**.
* **Return type:** Optional<T> — to handle the case where the record might not exist.

### ****2. Syntax****

Optional<Student> optionalStudent = studentRepo.findById(id);

Here:

* studentRepo → Repository object for Student entity.
* findById(id) → Searches the table student\_master for a row where **primary key = id**.
* Optional<Student> → Wraps the result safely.

### ****3. Step-by-step Execution Flow****

1. You call:
2. studentRepo.findById(id)
3. Spring Data JPA creates an SQL query:
4. SELECT \* FROM student\_master WHERE id = ?
5. It sends this query to the database.
6. If the database finds a matching row:
   * Converts that row into a Student object.
   * Wraps it inside Optional<Student>.
7. If no matching row is found:
   * Returns Optional.empty() (not null).

### ****4. Why**** Optional****?****

* **Without Optional:**
  + findById() could return null when the record is not found.
  + If you try to use null → **NullPointerException**.
* **With Optional:**
  + You are **forced** to check if data is present:
  + if (optionalStudent.isPresent()) {
  + // safe to use optionalStudent.get()
  + }
  + Makes code safer and avoids runtime errors.

### ****5. Real-life Analogy**** 🗃️

* Think of Optional like a **sealed envelope**.
  + If the record exists → envelope contains the student details.
  + If not → envelope is empty.
* You always **check the envelope first** before opening it, so you never reach into “thin air” and crash your program.

### ****6. Code Example****

@Service

public class StudentService {

@Autowired

private StudentRepository studentRepo;

public void getStudentById(Long id) {

Optional<Student> optionalStudent = studentRepo.findById(id);

if (optionalStudent.isPresent()) {

Student student = optionalStudent.get();

System.out.println("Student Found: " + student);

} else {

System.out.println("No student found with ID: " + id);

}

}

}

### ****7. Industry Usage****

* findById() is heavily used when:
  + You need to fetch a record for viewing, editing, or deleting.
  + You want to **verify existence** before proceeding with operations.
  + You need **safe null handling** in real-time projects.

Eg: SpringBootDataJpaFindById

## 📄 Notes for findAllById() — Spring Data JPA (Using student\_master Table)

### ****1. Purpose of the Example****

* This example shows how to use **findAllById(Iterable<ID> ids)** from JpaRepository to fetch **multiple rows** from the database using their IDs in a **single database call**.
* It’s more efficient than calling findById() for each ID separately.

### ****2. Method Explanation****

List<Long> ids = Arrays.asList(1L, 2L, 3L);

List<Student> students = studentRepo.findAllById(ids);

students.forEach(System.out::println);

**Line-by-line:**

1. **List<Long> ids = Arrays.asList(1L, 2L, 3L);**
   * Creates a List containing IDs 1, 2, and 3.
   * These IDs correspond to the **primary key column** in the table (id in student\_master).
2. **List<Student> students = studentRepo.findAllById(ids);**
   * Calls the Spring Data JPA method findAllById().
   * JPA internally runs a SQL query like:
   * SELECT \* FROM student\_master WHERE id IN (1, 2, 3);
   * Fetches all matching records in **one query**.
   * Each record is automatically converted into a Student object and added to a List.
3. **students.forEach(System.out::println);**
   * Loops through the list and prints each student.
   * This calls the toString() method of Student for output formatting.

### ****3. Execution Flow****

Main Class

↓

StudentService.fetchMultipleStudents()

↓

Prepare List of IDs [1, 2, 3]

↓

studentRepo.findAllById(ids)

↓

SQL Query: SELECT \* FROM student\_master WHERE id IN (1, 2, 3)

↓

Database returns matching rows

↓

JPA converts each row → Student object

↓

List<Student> returned

↓

Loop & print each student

### ****4. Key Points to Remember****

* **Input type** → Any Iterable<ID> (List, Set, etc.).
* **Returns** → A List of entity objects.
* Only records with matching IDs are returned — missing IDs are ignored silently.
* Works **only for the primary key** field (@Id annotated).
* Better performance than multiple separate findById() calls.

### ****5. Advantages in Real-Time Projects****

* **Performance** → One DB query instead of multiple queries.
* **Clean Code** → Easy to understand and maintain.
* **Flexibility** → Works with any number of IDs.

Eg:SpringBootDataJpaFindAllById

## **📒 Notes for** findAll() **Example (Spring Data JPA)**

### ****1. Purpose of**** findAll()

* findAll() is a **predefined method** in JpaRepository.
* It **fetches all the rows** from the table mapped to your entity.
* In our case:
  + Entity → Student
  + Table in DB → student\_master
  + So findAll() will return **all Student records** present in student\_master.

### ****2. What the**** findAll() ****method returns****

* It **always returns a List<Student>** containing:
  + All rows of the table.
  + Each row is converted into a Student object.
* If table is empty → returns **an empty list** (not null).

### ****3. Flow of execution****

1. **Main Class**
   * Spring Boot starts and creates all required beans (StudentService, StudentRepository, etc.).
   * After startup, main() manually calls:
   * service.fetchAllStudents();
2. **Service Layer**
   * Inside fetchAllStudents():
   * List<Student> students = studentRepo.findAll();
     + This calls the Spring Data JPA predefined findAll() method.
     + JPA internally creates and executes SQL:
     + SELECT \* FROM student\_master;
     + Converts each row into a Student object.
   * Now we have a list of Student objects in students.

### ****4. The important line****

students.forEach(System.out::println);

* **forEach()** is a method from Java’s Iterable interface (available for all collections).
* It **loops through each element** in the list (students in our case).
* System.out::println is **method reference** syntax in Java:
  + Equivalent to:
  + for(Student s : students) {
  + System.out.println(s);
  + }
  + For each student object, it calls System.out.println(student) automatically.
* Since Student has a **toString()** method:
* @Override
* public String toString() {
* return "Student [id=" + id + ", name=" + name +
* ", course=" + course + ", fee=" + fee + "]";
* }

→ The toString() output is printed for each record.

### ****5. Why this is useful in real-time projects****

* Quickly displays all table records **without writing SQL queries manually**.
* Great for **testing and debugging** database data.
* Useful in:
  + Admin dashboards (fetch all records for display)
  + Reporting modules
  + Bulk processing tasks

### ****6. Key Points to Remember****

* findAll() = fetches **all records** from the table.
* Returns List<Entity> type (never null).
* If the table has no data → returns an **empty list**.
* forEach(System.out::println) = short form of looping and printing objects.
* **Important**: Avoid findAll() on very large tables in production — can cause memory and performance issues.

Eg: SpringBootDataJpaFindAll