**1. The Problem Before Spring Data**

Before the Spring Data module was introduced, if your application needed to talk to a database, the way you wrote the code **depended completely on the database type**.

**Case 1 – SQL Databases**

*(Examples: MySQL, Oracle, PostgreSQL, SQL Server)*  
Two main options existed:

1. **Spring JDBC** →
   * You wrote code in **JDBC style** (manually writing SQL queries, handling ResultSets, connections, etc.).
   * This meant **more boilerplate code**.
2. **Spring ORM (like Hibernate / JPA)** →
   * You used an **Object-Relational Mapping** framework to map Java objects to database tables.
   * Less SQL writing, but you still had to learn ORM tools separately.

**Case 2 – NoSQL Databases**

*(Examples: MongoDB, Cassandra, Redis)*

* NoSQL databases didn’t have support in Spring.
* If you wanted to use MongoDB, you had to write:

Spring App → MongoDB API + MongoDB Driver → MongoDB Software

* If you wanted Cassandra, you had to write:

Spring App → Cassandra API + Cassandra Driver → Cassandra Software

* Each NoSQL database had **its own library, driver, and style** of writing queries.
* This meant more **learning curves** and more **inconsistent code** in the same project.

**Main Issues Before Spring Data**

* **No unified approach** — different databases = different code style.
* **Steep learning curve** — you had to learn each database’s API and driver.
* **Code duplication** — for common operations like save, find, delete, update, you wrote different code for each DB.
* **Harder to switch databases** — changing from MySQL to MongoDB required rewriting a lot of code.

**2. The Solution: Spring Data**

Spring introduced **Spring Data** as a **unified abstraction layer** for both SQL and NoSQL databases.

What it does:

**Point 1: One common programming model to work with different databases**

**Meaning:**  
Spring Data gives you the **same way of coding** no matter which database you use.

* Normally, if you work with MySQL (SQL) you write JDBC or JPA queries.
* If you switch to MongoDB (NoSQL), you write MongoDB-specific queries.
* This means different syntax, different APIs, and more learning for each DB.

With Spring Data, the code **style** stays the same — you only change the configuration for the database type.

**Example:**

// This works for MySQL, PostgreSQL, Oracle, etc. via Spring Data JPA

interface StudentRepository extends JpaRepository<Student, Long> { }

// This works for MongoDB

interface StudentRepository extends MongoRepository<Student, String> { }

In both cases:

* You create a Repository interface.
* You define methods (or even no methods — Spring makes them for you).
* The CRUD code looks almost the same.

You don't rewrite the whole logic when switching DBs — you just switch the underlying module (JPA, MongoDB, Cassandra, etc.).

**Point 2: Works with both SQL & NoSQL databases**

**Meaning:**  
Spring Data is not just for relational databases like MySQL/PostgreSQL (SQL), it also supports NoSQL ones like MongoDB, Cassandra, Redis, etc.

* For SQL → You use **Spring Data JPA**.
* For NoSQL → You use the respective module (e.g., **Spring Data MongoDB**, **Spring Data Cassandra**).

So Spring Data acts like a **universal adapter** that plugs into many database “shapes.”

**Point 3: Uses repositories to handle CRUD without boilerplate code**

**Meaning:**  
Normally, if you want to insert, update, delete, or fetch data, you have to write the SQL, prepare statements, map results to objects, handle exceptions, and close connections — that’s the “boilerplate.”

Spring Data removes that pain.  
You **just** create an interface extending a prebuilt repository type — Spring will automatically create an implementation at runtime.

**Example Without Spring Data:**

Connection con = DriverManager.getConnection(...);

PreparedStatement ps = con.prepareStatement("SELECT \* FROM student WHERE id=?");

ps.setLong(1, id);

ResultSet rs = ps.executeQuery();

Student student = new Student();

student.setId(rs.getLong("id"));

// set other fields...

Lots of repetitive work.

**Example With Spring Data:**

interface StudentRepository extends JpaRepository<Student, Long> {

}

Now you can just do:

Student s = studentRepository.findById(1L).get();

No SQL writing. No JDBC connection handling.  
Spring Data generates this code internally when the app starts.

**3. Before vs After Spring Data — Simple View**

## ****Before Spring Data****

When you worked with different types of databases, each one had its **own way** of connecting, querying, and mapping data.

Example:

* **SQL (MySQL, PostgreSQL, Oracle)**
  + Use **Spring JDBC** (write SQL queries manually) or **Spring ORM** (Hibernate, JPA).
  + Different mapping annotations, different ways to execute queries.
  + You need to manage database connections, transactions, and exceptions yourself.
* **NoSQL (MongoDB)**
  + Use **MongoDB Java Driver** directly.
  + Different method names, different query objects, different error handling.
  + Code looks nothing like SQL code.
* **NoSQL (Cassandra)**
  + Use **Cassandra Java Driver**.
  + Again, new APIs, new query language (CQL), and separate connection setup.

**Result:**  
Every time you learned a new database, you had to:

1. Learn **its specific driver**.
2. Change **your coding style**.
3. Rewrite a lot of code if you wanted to switch databases.

## ****After Spring Data****

Spring Data **hides** those differences by giving you **one consistent programming style** for all databases.

Example:

* SQL → **Spring Data JPA**
* MongoDB → **Spring Data MongoDB**
* Cassandra → **Spring Data Cassandra**

You always:

1. Create a **Repository interface**.
2. Define methods in a standard way (findById, save, delete, etc.).
3. Let Spring Data handle connection setup, query execution, and mapping.

**Result:**

* You **don’t have to learn each driver deeply**.
* Your code looks almost the same for any database.
* Switching databases is often as simple as changing dependencies and configs.

### ****Important Submodules of Spring Data****

1. **Spring Data JPA** – Provides abstraction over ORM software like Hibernate, making it easier to work with relational databases via JPA.
2. **Spring Data JDBC** – Provides abstraction over plain JDBC, reducing boilerplate for database access without full ORM overhead.
3. **Spring Data MongoDB** – Provides abstraction over the MongoDB API, simplifying interaction with NoSQL document databases.

## ****Main Modules of Spring Data****

### ****Spring Data JPA****

* A Spring-based way to work with JPA (**Java Persistence API**).
* Makes it easier to work with relational databases (like MySQL, PostgreSQL) using ORM tools such as Hibernate.
* Has Spring Data Repository support for JPA, so you can make repositories without writing SQL or JDBC code yourself.

### ****Spring Data MongoDB****

* A Spring-based way to work with MongoDB (a NoSQL document database).
* Lets you store and get Java objects as MongoDB documents (**object-document mapping / ODM**).
* Allows you to use the same repository style as JPA, but for MongoDB collections instead of SQL tables.

## Advantages with Spring Data JPA

## 1️ Create a Repository (DAO) Interface

**What it means:**  
In traditional JDBC or even Hibernate, you create a **full DAO class** with:

* SQL/HQL queries
* Connection or EntityManager code
* Mapping ResultSet to objects

With Spring Data JPA, you **don’t** write these classes.  
Instead, you only write an **interface** — no method bodies, no SQL, nothing else.

**Example:**

package com.example.repository;

import org.springframework.data.jpa.repository.JpaRepository;

import com.example.entity.Student;

public interface StudentRepository extends JpaRepository<Student, Long> {

}

**Explanation:**

* StudentRepository → our interface to access Student data in DB.
* JpaRepository<Student, Long> → says:
  + We’re working with Student entity.
  + Its primary key type is Long.
* No implementation provided — Spring generates it at runtime.

**2️⃣ Extend a Predefined Repository Interface**

**What it means:**  
Spring already has ready-made repository interfaces that contain common database operations:

* CrudRepository → basic CRUD (Create, Read, Update, Delete).
* JpaRepository → everything from CrudRepository **plus** JPA-specific features like paging & sorting.
* PagingAndSortingRepository → CRUD + pagination and sorting.

When you extend one of these, you **inherit** all their methods.

**Common inherited methods from JpaRepository:**

* save(entity) → Insert or update.
* findAll() → Get all rows.
* findById(id) → Get row by primary key.
* deleteById(id) → Delete row by primary key.

**Example usage:**

@Autowired

private StudentRepository studentRepository;

public void demo() {

Student s1 = new Student(null, "John", 23);

studentRepository.save(s1); // INSERT

List<Student> all = studentRepository.findAll(); // SELECT \*

Student s2 = studentRepository.findById(1L).get(); // SELECT WHERE id=1

studentRepository.deleteById(1L); // DELETE WHERE id=1

}

**Why this works:**  
The logic for these methods is already **inside Spring Data JPA**.  
When your app runs, Spring **auto-generates a proxy class** that has the actual SQL/HQL logic.

**3️⃣ Logic is Already Written**

**What it means:**  
You don’t write SQL queries yourself for basic CRUD — Spring does it.

When you call:

studentRepository.findAll();

Spring internally runs:

SELECT \* FROM student;

and maps each row to a Student object.

When you call:

studentRepository.save(student);

Spring decides:

* If student.id is null → run INSERT.
* Else → run UPDATE.

**Benefit:**

* No PreparedStatement handling.
* No transaction boilerplate.
* No manual mapping from ResultSet to objects.

**4️⃣ Add Custom Methods if Needed**

**What it means:**  
If you need queries beyond the built-in ones, you **don’t** have to write SQL — you can use **method naming conventions**.

**Example:**

List<Student> findByName(String name);

List<Student> findByAgeGreaterThan(int age);

List<Student> findByNameAndAge(String name, int age);

**What happens internally:**

* Spring looks at the method name (findByName).
* Breaks it into parts:
  + findBy → a query.
  + Name → refers to the name field in the Student entity.
* Generates SQL:

SELECT \* FROM student WHERE name = ?;

If you write findByAgeGreaterThan, Spring generates:

SELECT \* FROM student WHERE age > ?;

**Important:** Method name must follow **Spring Data conventions** — else Spring won’t know how to make the query.

**5️⃣ Less Code, Same Result**

**Before Spring Data JPA (JDBC style):**

Connection con = DriverManager.getConnection(url, user, pwd);

PreparedStatement ps = con.prepareStatement("SELECT \* FROM student");

ResultSet rs = ps.executeQuery();

List<Student> list = new ArrayList<>();

while (rs.next()) {

Student s = new Student();

s.setId(rs.getLong("id"));

s.setName(rs.getString("name"));

s.setAge(rs.getInt("age"));

list.add(s);

}

ps.close();

con.close();

✅ Around 20+ lines for a simple select.

**After Spring Data JPA:**

List<Student> list = studentRepository.findAll();

✅ **1 line**, no SQL, no connection handling.

**Final Summary**

Spring Data JPA:

* Lets you **skip DAO class implementations**.
* Gives you **ready-made CRUD methods** via JpaRepository and others.
* Generates SQL queries for you — no manual JDBC code.
* Supports **custom queries** via method naming or JPQL/SQL if needed.
* Reduces code size drastically while keeping it **database-independent**.

## ****1️⃣ Works for Any Number of Tables****

**Before Spring Data JPA:**  
If you had **500 tables**, you’d normally write **500 DAO classes** — each with SQL queries, JDBC code, mapping logic, exception handling, etc. That’s thousands of lines of repetitive code.

**With Spring Data JPA:**

* You **only** create **500 repository interfaces** — one for each table’s entity class.
* If you only need basic CRUD, these interfaces are empty except for extends JpaRepository.
* You add custom methods only if needed.

**Example (for 2 tables):**

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

public interface TeacherRepository extends JpaRepository<Teacher, Long> { }

✅ Even for 500 tables — same pattern, no SQL writing.

## ****2️⃣ Automatic Implementation Generation****

**What it means:**  
You don’t write the method **bodies**. Spring Data JPA generates them automatically at runtime.

This works for:

* **Built-in methods** → save(), findAll(), deleteById() come from JpaRepository.
* **Custom methods** → If you write findByName(String name), Spring looks at your method name, understands the field you mean, and generates the right SQL.

**Example:**

List<Student> findByMarksGreaterThan(int marks);

Spring will generate:

SELECT \* FROM student WHERE marks > ?;

No need to write this SQL yourself.

## ****3️⃣ Powered by Proxy Classes****

**Under the hood:**

**You never get the real repository.**  
When you @Autowired StudentRepository, Spring actually gives you a **proxy object** (a class it generated at startup). That proxy:

1. Catches your method call (findAll(), save(), findByName(...), etc.).
2. Sets up the right **transaction** and **database connection**.
3. Builds the **JPA/Hibernate** query (or derives it from the method name).
4. Executes it via **JDBC**.
5. **Maps** the rows to your **entity** objects.
6. Returns clean Java objects to you.

You don’t see any of this—because the proxy does it behind the scenes.

Your Code

↓

Repository Proxy (generated by Spring)

↓

JPA (EntityManager) / Hibernate

↓

JDBC

↓

Database

↑

Mapped Entities (Student, Teacher, ...)

# Step-by-Step: What actually happens on a call

Let’s say you call:

List<Student> students = studentRepository.findAll();

Here’s the real story:

1. **You call findAll()**
   * You’re calling **on the proxy**, not on a hand-written class.
2. **Proxy intercepts the call**
   * “Oh, findAll()? I know how to do that.”
3. **Transaction/connection setup**
   * If needed, it starts a **transaction** (thanks to @Transactional), grabs a **connection** from the pool, and opens a **Hibernate Session/EntityManager**.
4. **Query creation**
   * For built-in methods (findAll, save, deleteById), it routes to a ready-made implementation (like SimpleJpaRepository).
   * For **derived methods** (e.g., findByName, findByAgeGreaterThan), it **parses the method name** and builds the matching query.
5. **Hibernate generates SQL**
   * Hibernate turns your JPA query into SQL like:

SELECT \* FROM student;

* + Executes it via JDBC.

1. **Result mapping**
   * Hibernate reads the ResultSet and **fills** Java objects (Student) using your @Entity mappings (@Column, @Id, etc.).
   * Relationships (like @OneToMany) are handled (often lazily) for you.
2. **Return value**
   * The proxy gives you a clean List<Student>—no SQL, no JDBC code in your files.
3. **Cleanup**
   * Transaction commits/rolls back.
   * Connections are released back to the pool.

Why spring uses a proxy class :

**1️⃣ Add extra work around your method call**

When you call a repository method, Spring doesn’t just run the SQL.  
The **proxy** (the smart middleman) adds useful work around your call:

* **Transaction handling**
  + If you save data, Spring can start a *transaction* before the query, and commit it when successful or undo it if something goes wrong.
  + This keeps your database safe from half-done changes.
* **Connection handling**
  + The proxy takes a database connection from the pool, uses it, and puts it back.
  + You don’t open or close connections yourself.
* **Exception handling**
  + Database errors are normally messy and different for each database.
  + The proxy catches them and changes them into **Spring’s standard errors** so your code can handle them the same way every time.

💡 **Why this is useful:** Without the proxy, you would have to write all this code for **every query** yourself — which is slow, boring, and error-prone.

**2️⃣ Hides the real implementation**

You only write something like:

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

* You don’t write the real class that talks to the database.
* Spring makes that real class for you at runtime.

💡 **Why this is useful:**

* You write **less code**.
* You don’t repeat the same database logic everywhere.
* You focus on *what you want* (findByName) instead of *how to do it*.

**3️⃣ Makes queries from method names**

* If you write findByNameAndAgeGreaterThan(String name, int age), the proxy reads your method name and builds the SQL:

SELECT \* FROM student WHERE name = ? AND age > ?;

* You don’t write this SQL.
* For harder queries, you can still use @Query with your own SQL or JPQL — the proxy will still run it and map results.

💡 **Why this is useful:**

* You can create a new query just by adding a method.
* You don’t have to work with messy query code every time.

**4️⃣ Same style everywhere**

No matter if you call:

studentRepository.findAll();

or:

studentRepository.findByEmailAndActiveTrue(String email);

* The coding style is the same.
* The proxy still:
  1. Starts a transaction if needed
  2. Builds the query
  3. Runs it
  4. Maps the data into Java objects
  5. Returns the result

💡 **Why this is useful:**

* Once you learn this way of coding, you can handle **any** table or query.
* Your whole team writes code in the same way — easy to read, easy to maintain.

✅ **In short:**  
Spring uses proxies so that:

* You get **extra features** (transactions, connection handling, better error handling) without writing the code.
* You don’t have to write or maintain **real database classes**.
* You can **make queries from method names** instead of writing SQL.
* You use **one style** for all queries and all databases.

What you write vs. what Spring generates

**1️⃣ What you write**

In your code, you only write an **interface**, for example:

public interface StudentRepository extends JpaRepository<Student, Long> {

List<Student> findByName(String name); // no body

}

**What this means:**

* You tell Spring:
  + *"I have a Student table (linked to the Student entity class) with a primary key of type Long."*
  + *"I want basic CRUD methods (save, findAll, deleteById, etc.) from JpaRepository."*
  + *"Also, give me a method to find students by their name."*
* Notice:
  + There’s **no method body** — just the method name.
  + No SQL.
  + No code that connects to the database.

At this stage, there is **no real class** to run these methods — it’s just an interface.

**2️⃣ What Spring generates at runtime**

When your application starts:

1. Spring scans your code.
2. It sees your repository interface (StudentRepository).
3. It **creates a real Java class in memory** (not on your disk) — often called something like:

StudentRepository$$Proxy

1. This class:
   * **Implements** your StudentRepository interface.
   * **Provides real method bodies** for all the inherited methods (save, findAll, etc.).
   * **Adds a body** for your custom method findByName(...) by generating a query based on the method name.

**3️⃣ What happens when you call a method**

Let’s say you call:

List<Student> list = studentRepository.findByName("John");

Here’s what happens inside the generated proxy:

1. **Transaction check**
   * If needed, it starts a transaction (@Transactional).
   * Gets a database connection from the connection pool.
2. **Query building**
   * It looks at findByName and figures out:
     + “findBy → means SELECT.”
     + “Name → is the name column in the student table.”
   * Builds the SQL:

SELECT \* FROM student WHERE name = ?;

1. **Query execution**
   * Runs the SQL using JPA (Hibernate) which internally uses JDBC.
2. **Mapping results**
   * Hibernate takes each row from the result set and creates a Student object.
   * Fills its fields using your @Entity and @Column mappings.
3. **Return value**
   * Returns the list of Student objects back to your code.
4. **Cleanup**
   * Closes the query resources.
   * Ends the transaction (commit or rollback).
   * Returns the connection to the pool.

**4️⃣ You never see this generated class**

* You don’t see StudentRepository$$Proxy in your project files.
* It only exists **while your app is running**.
* It’s generated by Spring’s internal code using libraries like **CGLIB** or **Java Dynamic Proxy**.
* To you, it feels like you’re calling StudentRepository directly — but under the hood, you’re really calling the proxy.

**Why this is powerful**

* **No manual DAO writing** → You avoid creating classes full of boilerplate database code.
* **Automatic SQL generation** → You can get queries from method names.
* **Automatic mapping** → No manual ResultSet to object conversion.
* **Consistent behavior** → Same flow for all repository methods.