Spring Data JDBC Explained

give me the deep and detailed explaination of Spring Data JDBC in spring boot so that i can able to understand it well and do not have any type of doubt in this.

so give me the deep and detailed explaination

Piyush, since you already have **strong JDBC, JPA, and Spring Boot fundamentals**, I'll explain **Spring Data JDBC** from **first principles → internal working → design philosophy → real-world usage → comparison with JPA**, in **simple language**, step by step.
After this, you should be able to **decide confidently when to use Spring Data JDBC and when not to.**

🌱 What is Spring Data JDBC (in Spring Boot)?

Spring Data JDBC is a **simple, opinionated data access framework** that helps you interact with **relational databases** using **plain SQL concepts**, **without ORM complexity**.

> 👉 Think of Spring Data JDBC as:
> **"JDBC made productive + Repository abstraction – ORM magic."**

It is part of the larger **Spring Data** family and works naturally inside **Spring Boot**.

❓ Why was Spring Data JDBC created?

Traditional choices:

Approach	Problem
Raw JDBC	Too much boilerplate
Hibernate / JPA	Complex, heavy, hidden SQL

Approach	Problem
Spring JDBC Template	Still manual mapping

So Spring introduced **Spring Data JDBC** to give:

✅ Simple mapping
✅ No proxies
✅ No lazy loading
✅ No persistence context
✅ Full SQL transparency

🧠 Core Philosophy of Spring Data JDBC

> **“Your database schema is the truth.”**

This is **very different from JPA.**

JPA Thinking	JDBC Thinking
Object-centric	Database-centric
Entity lifecycle	Row lifecycle
Lazy loading	Explicit queries
Hidden SQL	Visible SQL

Spring Data JDBC says:

- **One aggregate = one database graph**
- **No partial loading**
- **No entity state tracking**

🧩 Where Spring Data JDBC Fits in Spring Boot Architecture

```powershell
```

```
Controller
   ↓
Service
   ↓
Repository (Spring Data JDBC)
   ↓
JDBC Driver
   ↓
Relational Database
```

No ORM layer in between.

🔑 Key Concepts (Very Important)

1️⃣ Aggregate Root (MOST IMPORTANT)

Spring Data JDBC works with **Aggregates**, not entities.

What is an Aggregate?

An **Aggregate** is a **group of objects** that are saved **together**.

Example:

```java
Order
 ├── OrderItem
 ├── OrderItem
```

Here:

- `Order` = Aggregate Root
- `OrderItem` = Part of aggregate

📌 You **never** save `OrderItem` separately.

2️⃣ Entity Mapping (Very Simple)

```java
```

```
@Table("users")
public class User {

   @Id
   private Long id;

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

Important annotations:

- `@Table` → table name
- `@Id` → primary key

❌ No:

- `@Entity`
- `@OneToMany`
- `@ManyToOne`
- `@Lazy`

3️⃣ Repository Layer (Magic but Simple)

```java
public interface UserRepository
      extends CrudRepository<User, Long> {
}
```

Spring automatically provides:

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

This comes from **Spring Data JDBC** repository abstraction.

🛠️ How Save Works Internally

When you call:

```java
userRepository.save(user);
```

Internally:

1. Checks if `id == null`
2. If yes → `INSERT`
3. If no → `UPDATE`
4. Executes SQL directly using JDBC
5. No caching
6. No dirty checking

📌 Every call hits the database.

🔗 Relationships in Spring Data JDBC

❌ No ORM Relationships

- No `@OneToMany`
- No `@ManyToOne`

✅ Only Aggregate Relationships

```java
@Table("orders")
class Order {

    @Id
    Long id;

    List<OrderItem> items;
}
```

```java
@Table("order_items")
class OrderItem {
    String product;
    int quantity;
}
```

Spring automatically:

- Inserts order
- Inserts all items
- Uses **foreign key internally**

But:

❌ No joins

❌ No lazy loading

📄 Custom Queries

You write SQL explicitly.

```java
@Query("SELECT * FROM users WHERE email = :email")
User findByEmail(String email);
```

📌 You **control the SQL**.

🔄 Transactions (Works Same as You Know)

```java
@Transactional
public void createUser(User user) {
    repository.save(user);
}
```

Spring uses **Spring Transaction Management** internally.

⚙️ Configuration in Spring Boot

Add dependency:

```xml
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jdbc</artifactId>
</dependency>
```

Configure datasource (same as JDBC/JPA):

```properties
spring.datasource.url=jdbc:mysql://localhost:3306/testdb
spring.datasource.username=root
spring.datasource.password=pass
```

🧪 Example Flow (Very Important)

Save User

```java
User user = new User(null, "Piyush", "piyush@mail.com");
repository.save(user);
```

Generated SQL:

```sql
INSERT INTO users (name, email) VALUES (?, ?)
```

Fetch User

```java
repository.findById(1L);
```

```sql
SELECT * FROM users WHERE id = ?
```

⚖️ Spring Data JDBC vs JPA (VERY CLEAR COMPARISON)

Feature	Spring Data JDBC	JPA / Hibernate
ORM	❌ No	✅ Yes
Lazy loading	❌	✅
SQL control	✅ Full	❌ Hidden
Performance predictability	✅ High	❌ Complex
Learning curve	✅ Easy	❌ Steep
Complex relationships	❌	✅

🚀 When Should YOU Use Spring Data JDBC?

✅ Use it when:

- You want **simple CRUD**
- You want **full SQL control**
- You want **predictable performance**
- You dislike Hibernate magic

❌ Avoid when:

- Complex joins
- Lazy loading
- Bidirectional relationships
- Large enterprise ORM models

🧠 Mental Model (Remember This)

> **Spring Data JDBC = JDBC + Repository + Aggregate Rules**

No magic. No guessing. No hidden behavior.

🏁 Final Summary

✔️ Spring Data JDBC is **not a replacement for JPA**

✔️ It is **a simpler, cleaner alternative**

✔️ It forces **good database design**

✔️ It gives **clarity and control**

If you want next:

👉 **Spring Data JDBC vs Spring JDBC Template (deep dive)**

👉 **Real-world project structure using Spring Data JDBC**

👉 **Migrating from JPA to JDBC**

Just tell me 👍