📘Spring JDBC — Complete Updated Note (Connection Pool, How It Knows DB, Differences From JDBC)

This note gives a **complete, deep explanation** of Spring JDBC including:

- How Spring JDBC gets a **database connection**
- How Spring JDBC uses **connection pooling**
- How Spring knows **which database** to connect to
- Difference between **JDBC vs Spring JDBC**
- Working of **JdbcTemplate** internally
- How SQL exceptions are handled
- Full example with PostgreSQL (Neon DB)

🕖1. What Is Spring JDBC?

Spring JDBC is a module in Spring Framework which makes database access easier by using **JdbcTemplate**.

Traditional JDBC is verbose and requires you to manually:

- Open connection
- Create statements
- Execute queries
- Close ResultSet
- Close Connection
- Handle exceptions

Spring JDBC removes **all boilerplate** using a template pattern.

🕖2. How Does Spring JDBC Know Which Database to Connect To?

Spring JDBC does **NOT** connect to the database directly. It uses a **DataSource**.

🔧DataSource contains the necessary configuration:

- URL
- Username
- Password
- Driver class

In Spring Boot:

```
spring.datasource.url=jdbc:postgresql://yourhost/db
spring.datasource.username=youruser
spring.datasource.password=yourpass
spring.datasource.driver-class-name=org.postgresql.Driver
```

Spring Boot automatically creates:

- DataSource object
- JdbcTemplate bean

So JdbcTemplate does NOT know the database — the DataSource provides the connection.

🕡3. How Does Spring JDBC Get Connection Pool?

Spring Boot automatically configures **HikariCP** as the default **connection pool**.

🔧What is HikariCP?

A very fast, lightweight connection pool library used in Spring.

🔧Why Connection Pool Is Needed?

Opening a new database connection is slow. Connection pool keeps a set of open connections and reuses them.

🔧How Spring Gets Connection Pool?

Spring Boot automatically checks the classpath:

- If HikariCP is present → use it
- If not → tries Tomcat pool
- If not → uses simple driver-based DataSource

When PostgreSQL and Spring Boot dependencies are added, Boot auto-configures:

```
HikariDataSource → manages the pool
JdbcTemplate → uses this pool to get connections
```

You don't write any pooling code.

🔧Where does pool size come from?

Default:

```
maximumPoolSize = 10
```

You can override:

```
spring.datasource.hikari.maximum-pool-size=20
spring.datasource.hikari.minimum-idle=5
spring.datasource.hikari.idle-timeout=60000
```

🕖4. How JdbcTemplate Works Internally

When you run:

```
jdbcTemplate.query("SELECT * FROM student", rowMapper);
```

JdbcTemplate internally: 1️⃣Gets a connection from **HikariCP pool** (not DriverManager) 2️⃣Creates PreparedStatement 3️⃣Executes SQL 4️⃣Iterates ResultSet 5️⃣Calls RowMapper for each row 6️⃣Closes ResultSet 7️⃣Closes Statement 8️⃣Returns connection to pool (NOT closing it fully) 9️⃣Returns List of objects

You call **one line**, Spring does everything else.

🕖5. JDBC vs Spring JDBC — Key Difference

Feature	JDBC	Spring JDBC
Connection	Manual	Automatic via DataSource + Pool
Queries	Manual PreparedStatement	JdbcTemplate
Closing resources	Manual	Auto-handled
Exception type	`SQLException`	Runtime translated exceptions
Repeated code	High	Almost none
Performance	Slower (no pooling)	Faster (HikariCP)
Error handling	Verbose	Simple
Recommended?	❌No	✔️Yes

🕡6. Why Spring JDBC Is Better Than Normal JDBC?

🔧**No try-catch-finally**

🔧**No manual connection closing**

🔧**No manual ResultSet handling**

🔧**Automatic exception translation**

🔧**Automatic connection pooling**

🔧**Clean API with JdbcTemplate**

🕡7. RowMapper (Mapping SQL → Java)

Used to convert each database row into a Java object.

```
public class StudentRowMapper implements RowMapper<Student> {
    public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
        return new Student(
            rs.getInt("id"),
            rs.getString("name"),
            rs.getInt("age")
        );
    }
}
```

Or lambda:

```
jdbcTemplate.query(sql, (rs, rowNum) ->
    new Student(rs.getInt(1), rs.getString(2), rs.getInt(3))
);
```

🕡8. CRUD Operations in Spring JDBC

➤ Insert

```
jdbcTemplate.update(
    "INSERT INTO student(name, age) VALUES (?, ?)",
    "Aman", 23
);
```

➤ Fetch All

```
List<Student> list = jdbcTemplate.query(
    "SELECT * FROM student",
    new StudentRowMapper()
);
```

➤ Fetch One

```
Student s = jdbcTemplate.queryForObject(
    "SELECT * FROM student WHERE id = ?",
    new StudentRowMapper(), id
);
```

➤ Update

```
jdbcTemplate.update("UPDATE student SET age=? WHERE id=?", 25, 1);
```

➤ Delete

```
jdbcTemplate.update("DELETE FROM student WHERE id=?", id);
```

🕖9. How Spring Boot Auto-Creates JdbcTemplate

Spring Boot auto-creates a bean:

```
@Bean
public JdbcTemplate jdbcTemplate(DataSource dataSource) {
    return new JdbcTemplate(dataSource);
}
```

Thus JdbcTemplate uses **the same pooled DataSource**.

🕡10. Full Example — Spring JDBC with PostgreSQL (Neon DB)

1. Add PostgreSQL dependency

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.7.1</version>
</dependency>
```

2. application.properties

```
spring.datasource.url=jdbc:postgresql://<neon-host>/<database>?
sslmode=require
spring.datasource.username=<user>
spring.datasource.password=<password>
spring.datasource.driver-class-name=org.postgresql.Driver
```

3. Model

```
public class Student {
    private int id;
    private String name;
    private int age;

    // constructor + getter/setter
}
```

4. Repository

```
@Repository
public class StudentRepository {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    public int add(Student s) {
        return jdbcTemplate.update(
            "INSERT INTO student(name, age) VALUES (?, ?)",
            s.getName(), s.getAge()
        );
    }
    public List<Student> findAll() {
        return jdbcTemplate.query(
            "SELECT * FROM student",
            (rs, rowNum) -> new Student(
                rs.getInt("id"),
                rs.getString("name"),
                rs.getInt("age")
            )
        );
    }
}
```

5. Controller

```
@RestController
@RequestMapping("/student")
public class StudentController {

    @Autowired
    StudentRepository repo;

    @PostMapping("/add")
    public String add(@RequestBody Student s) {
        repo.add(s);
        return "Student added";
    }

    @GetMapping("/all")
    public List<Student> all() {
        return repo.findAll();
    }
}
```

🕚Final Summary

Spring JDBC = JDBC with:

- Automatic connection pooling (HikariCP)
- Automatic DataSource
- Automatic resource closing
- Exception translation
- Highly optimized performance
- Cleaner readable code

Spring knows which database to use because **you configure DataSource**.
JdbcTemplate gets connections from **HikariCP pool**, not DriverManager.

This makes Spring JDBC fast, safe, and production-ready.

If you need, I can also add: ✅NamedParameterJdbcTemplate section ✅Transactions in Spring JDBC ✅PDF export

Great question — many beginners think **RowMapper runs recursively**, but the truth is:

🖍️RowMapper is NOT recursive

It **never calls itself**.
Spring JDBC calls it **once per row** in the ResultSet.

Let me explain clearly.

🔗1. How RowMapper Actually Works (Simple Explanation)

Spring JDBC uses **JdbcTemplate** which internally does something like:

```
while(resultSet.next()) {
    call rowMapper.mapRow(resultSet, rowNumber)
}
```

So the flow is:

```
Row 1 → mapRow() called once
Row 2 → mapRow() called once
Row 3 → mapRow() called once
...
```

There is **no recursive call**, only a simple loop.

---

2. Why it feels like recursion?

Because if there are 10 rows, `mapRow()` is executed 10 times.

Beginners think:

```
mapRow() runs inside itself → recursion ❌
```

But actually:

```
JdbcTemplate loops → calls mapRow() repeatedly → iteration ✔️
```

🛠️3. Example to Understand Clearly

Your RowMapper:

```
public class StudentRowMapper implements RowMapper<Student> {
    @Override
    public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
        Student s = new Student();
        s.setId(rs.getInt("id"));
        s.setName(rs.getString("name"));
        s.setAge(rs.getInt("age"));
        return s;
    }
}
```

JdbcTemplate does:

```
ResultSet rs = statement.executeQuery();

List<Student> list = new ArrayList<>();
int rowNum = 0;

while(rs.next()) {
    Student s = rowMapper.mapRow(rs, rowNum);
    list.add(s);
    rowNum++;
}
```

✔️Loop
✔️Not recursion
✔️RowMapper is called ONCE PER ROW

🔍4. What exactly calls mapRow() repeatedly?

The `JdbcTemplate.query()` method:

```
public <T> List<T> query(String sql, RowMapper<T> rowMapper) {
    return resultSetExtractor.extractData(resultSet)
}
```

Internally:

```
while(rs.next()) {
    list.add(rowMapper.mapRow(rs, rowNumber++));
}
```

Spring handles the looping.
Your RowMapper only handles **mapping a single row**.

🕚5. Why RowMapper is designed this way?

Because it separates responsibilities:

ComponentResponsibility	
JdbcTemplate	Fetch rows, loop over ResultSet
RowMapper	Convert **one** row → Java object

This is clean, reusable, testable.

📌6. Summary

RowMapper:

- ❌Does NOT use recursion
- ✔️Is NOT called by itself
- ✔️Is called by JdbcTemplate
- ✔️Executes once per row
- ✔️Converts SQL row → Java object

If you want, I can create a visual diagram like:

```
ResultSet:
Row 1 → mapRow()
Row 2 → mapRow()
Row 3 → mapRow()
...
```