**✅ Use RowMapper<T> when:**

* You want to **map each row of the ResultSet to a Java object (POJO)**.
* Your query returns **multiple rows**, and each row maps **one-to-one to an object**.
* You just want a **List of objects** (List<Employee>, List<Product>, etc.).
* The logic for mapping a row is **simple and independent per row**.

### ✅ What is ResultSetExtractor?

* ResultSetExtractor<T> is an **interface** in Spring JDBC.
* It is used to **extract data from the complete ResultSet** after a SQL SELECT query runs.
* It has **one method**:

T extractData(ResultSet rs) throws SQLException;

* You write your own code inside extractData() to:
  + **Loop through all rows**
  + **Create objects manually**
  + **Return any result you want** (List, Map, single object, etc.)

#### 🔍 Think of it like:

“Give me the whole table result, and I will decide what to do with it.”

### 🧠 When should you use it?

Use ResultSetExtractor when:

* ✅ You want to **read and process all rows together**.
* ✅ You need to **build a complex structure** like:
  + List<Employee>
  + Map<Integer, Department>
  + Nested objects (e.g., Department with List<Employee>)
* ✅ You want to:
  + **Filter some rows**
  + **Group results** (e.g., group employees by department)
  + **Merge rows**
  + **Calculate summary data**

## ✅ Why use ResultSetExtractor instead of RowMapper?

You’re right — **in most cases, RowMapper is enough** to convert rows into objects.  
But ResultSetExtractor is useful when:

### 🔁 1. You need ****custom logic across multiple rows****

For example:

* Grouping data from rows
* Skipping rows based on custom conditions
* Accumulating or summarizing values

Example: Building a Map<Department, List<Employee>> from flat result data  
→ Can't do this with RowMapper easily.

### 🔗 2. You need to ****build nested or hierarchical objects****

For example:

* One Department object with a list of Employee objects inside it
* One Order object with nested List<OrderItem>

RowMapper only sees **one row at a time**, so it can't group related rows into one object.  
ResultSetExtractor can process the full result and build nested structures.

### ⚙️ 3. You want ****full control over result set processing****

If you want to:

* Skip the first row
* Stop after 10 rows
* Apply row-by-row conditional checks
* Reuse complex logic from legacy systems

Then ResultSetExtractor is the better fit.

## 🧠 Summary

| **Use Case** | **Use RowMapper?** | **Use ResultSetExtractor?** |
| --- | --- | --- |
| Simple mapping (one row → one object) | ✅ Yes | ❌ Not needed |
| Complex post-processing | ❌ Not possible | ✅ Yes |
| Grouping or combining rows | ❌ Not ideal | ✅ Perfect fit |
| Nested object construction | ❌ Too limited | ✅ Recommended |

Eg: SpringBootResultSetExtractor

### 🔧 ****Method Name:****

public List<Employee> getAllEmployees()

This method is used to **get a list of all employees** from the database table called emp13.

### ✅ ****Step-by-step Explanation:****

#### 1. **Define SQL Query:**

String sql = "SELECT \* FROM emp13";

* This line defines the SQL query.
* SELECT \* means "get all columns" from the emp13 table.

#### 2. **Create a ResultSetExtractor:**

ResultSetExtractor<List<Employee>> extractor = new ResultSetExtractor<>() { ... };

* This is an object that knows **how to process the full result** returned from the database.
* It's used when we want to manually handle all rows in a custom way.

#### 3. **Inside** extractData() **Method:**

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

* We create an empty list to store all the employee objects.

while (rs.next()) {

* This loops through each row in the result set.

Employee e = new Employee();

* For each row, we create a new Employee object.

e.setId(rs.getInt("id"));

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

e.setEmail(rs.getString("email"));

e.setSalary(rs.getDouble("salary"));

e.setDepartment(rs.getString("department"));

* We **fetch values** from the current row using column names and **set them** in the employee object.

list.add(e);

* Add the filled employee object to the list.

#### 4. **Return the final list:**

return jdbcTemplate.query(sql, extractor);

* Spring runs the SQL query.
* The extractor processes the result set.
* The final list of employees is returned.

### 🧠 In Simple Terms:

* Think of emp13 as an Excel sheet with employee data.
* The method reads all rows one by one.
* For each row, it creates a new Employee object.
* It fills the employee's data from the row.
* All employee objects are added to a list.
* Finally, it gives you the full list back.

NamedParameterJdbcTemplate

* Works like JdbcTemplate but allows **named parameters** in SQL. JdbcTemplate doesn’t allow named parameters
* Easier to read and manage queries, especially with multiple parameters.

### ✅ SQL Example:

-- Positional parameters:

SELECT empno, ename, job, sal FROM employee WHERE empno >= ? AND empno <= ?

-- Named parameters:

SELECT empno, ename, job, sal FROM employee WHERE empno >= :no1 AND empno <= :no2

### ✅ How to Pass Values

#### 1. **Using** Map<String, Object>

Map<String, Object> params = new HashMap<>();

params.put("no1", 1001);

params.put("no2", 2000);

namedParameterJdbcTemplate.query(sql, params, rowMapper);

Eg: SpringBootNamedParameterJdbcTemplate

#### 2. **Using** SqlParameterSource **Interface Implementations**

##### a. MapSqlParameterSource

SqlParameterSource paramSource = new MapSqlParameterSource()

.addValue("no1", 1001)

.addValue("no2", 2000);

namedParameterJdbcTemplate.query(sql, paramSource, rowMapper);

Eg: SpringBootMapSqlParameterSource

##### b. BeanPropertySqlParameterSource

* Passes Java bean object fields as parameters.
* **Field names in Java object must match the named parameters**.

Employee emp = new Employee();

emp.setNo1(1001);

emp.setNo2(2000);

SqlParameterSource paramSource = new BeanPropertySqlParameterSource(emp);

namedParameterJdbcTemplate.query(sql, paramSource, rowMapper);

✅ **Condition:** JavaBean field names should match the parameter names in the query (e.g., :no1, :no2).

Eg: SpringBootBeanPropertySqlParameterSource

## ✅ SimpleJdbcCall

### 🔹 What is SimpleJdbcCall?

* SimpleJdbcCall is a Spring class that helps you **call stored procedures** easily.
* It removes the need to write complex JDBC boilerplate code manually.

### 🔹 Why Use It?

* It automatically handles opening and closing database connections.
* It makes stored procedure calls cleaner and safer.
* It supports **IN, OUT, and INOUT** parameters easily.

### 🔹 How to Use It (Basic Flow)

SimpleJdbcCall call = new SimpleJdbcCall(dataSource)

.withProcedureName("procedure\_name");

* Create the object with a DataSource
* Set the procedure name
* Call .execute() with parameters

### 🔹 How to Pass Parameters?

Map<String, Object> params = new HashMap<>();

params.put("param1", value1);

params.put("param2", value2);

Map<String, Object> result = call.execute(params);

* Pass a map with key = parameter name, value = parameter value
* Spring will match them to the stored procedure

### 🔹 What Does It Return?

* .execute() returns a Map<String, Object> containing output parameters.
* You can get OUT values like:

String status = (String) result.get("output\_param\_name");

### 🔹 Where is it used?

* Used in **real-world enterprise projects**
* Common in **Oracle**, **MySQL**, and **PostgreSQL** systems with business logic inside stored procedures
* Often seen in **banking**, **insurance**, **telecom**, and **ERP** systems

### 🔹 Advantages of SimpleJdbcCall

✅ Reduces code  
✅ Easy to maintain  
✅ Supports OUT/INOUT params  
✅ Reuses database metadata (no need to declare all types manually)  
✅ Works with Spring JdbcTemplate

### 🔹 Things to Remember

* You must pass a valid DataSource.
* Procedure name **must match** the DB side name exactly.
* Parameter names in the map should match **DB procedure parameter names**.

## 🧠 One-Line Summary:

SimpleJdbcCall is a Spring utility that lets you call stored procedures using clean, readable Java code — with automatic connection and parameter handling.

Eg: SpringBootSimpleJdbcCall

## ✅ Enhanced: Spring Boot Internal Flow (with SimpleJdbcCall)

### 🔷 1. Application Starts

SpringApplication.run(SpringbootJdbcApp.class, args);

✅ What Spring Boot does behind the scenes:

* It **creates an ApplicationContext** (Spring container).
* Scans your project for classes with annotations like @Component, @Service, @Repository, and @Controller.
* It **auto-configures** many beans like DataSource, JdbcTemplate, EntityManagerFactory, etc.

📌 Spring Boot uses META-INF/spring.factories and starter dependencies to guess what you need and configure it for you.

### 🔷 2. Spring Boot Reads application.properties

spring.datasource.url=...

spring.datasource.username=...

✅ Spring Boot looks for this file in src/main/resources.

* It finds all spring.datasource.\* keys.
* Then it uses these values to configure a **DataSource bean** automatically.
* The default implementation is **HikariCP**, a fast and lightweight connection pool.

📌 You didn’t define @Bean for DataSource. Spring Boot did it for you using these values.

### 🔷 3. Spring Boot Auto-Creates DataSource

✅ Spring Boot internally does this:

@Bean

@ConfigurationProperties("spring.datasource")

public DataSource dataSource() {

return new HikariDataSource();

}

* It uses your property values and creates a HikariDataSource object.
* This bean is registered in the container as dataSource.

📌 You didn’t write this code — Spring Boot generates it for you.

### 🔷 4. Spring Finds EmployeeDao and Injects DataSource

@Repository

public class EmployeeDao {

@Autowired

public void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

}

}

✅ Internally, Spring does this:

* It sees @Repository and marks EmployeeDao as a Spring bean.
* During creation, it looks for @Autowired setter methods.
* It finds setDataSource() and calls it with the DataSource bean created earlier.

📌 That’s why you never called setDataSource() — Spring called it automatically using **reflection**.

### 🔷 5. SpringbootJdbcApp Gets Control (CommandLineRunner)

@SpringBootApplication

public class SpringbootJdbcApp implements CommandLineRunner {

@Autowired

private EmployeeDao dao;

}

✅ What happens here:

* Spring finds @SpringBootApplication — it combines:
  + @Configuration (for bean setup)
  + @ComponentScan (to find beans)
  + @EnableAutoConfiguration (to enable Spring Boot features)
* Spring sees this class implements CommandLineRunner, so after all beans are ready, it calls:

run(String... args)

📌 This is where your actual business logic starts.

### 🔷 6. Inside run() → You Call dao.save(e)

String status = dao.save(e);

✅ Now you enter the DAO layer. Inside save():

SimpleJdbcCall call = new SimpleJdbcCall(dataSource)

.withProcedureName("insert\_employee\_proc");

* You are **manually creating** a SimpleJdbcCall object.
* But using the dataSource that Spring injected earlier.

📌 Spring didn’t create SimpleJdbcCall for you — **you created it manually**, but Spring gave you the tools to do it cleanly.

### 🔷 7. You Pass Parameters and Call .execute(params)

Map<String, Object> params = new HashMap<>();

params.put("p\_id", 101);

params.put("p\_name", "John");

params.put("p\_email", "john@example.com");

Map<String, Object> result = call.execute(params);

✅ What happens internally:

* Spring creates a CallableStatement behind the scenes.
* It sets the input parameters (IN).
* It runs the stored procedure on your database.
* It reads the OUT parameters into a result map.

📌 If your DB (like Oracle) blocks metadata access, Spring may not know that p\_status is an OUT parameter — so you need to declare it manually.

### 🔷 8. OUT Parameter Must Be Declared

.declareParameters(

new SqlParameter("p\_id", Types.INTEGER),

new SqlParameter("p\_name", Types.VARCHAR),

new SqlParameter("p\_email", Types.VARCHAR),

new SqlOutParameter("p\_status", Types.VARCHAR)

)

✅ Why this is important:

* Oracle often blocks access to stored procedure metadata.
* Without manual declaration, Spring doesn’t know how to fetch the OUT parameter.
* You’ll get null instead of SUCCESS.

📌 Always declare parameters explicitly when using Oracle.

## ✅ Final Internal Chain (Simplified Recap)

▶ Spring Boot starts

↓

▶ Reads application.properties

↓

▶ Auto-creates DataSource (HikariCP)

↓

▶ Scans for @Repository, finds EmployeeDao

↓

▶ Injects DataSource into setDataSource()

↓

▶ Scans for @SpringBootApplication

↓

▶ Runs run() in SpringbootJdbcApp

↓

▶ Calls dao.save()

↓

▶ Creates SimpleJdbcCall and executes procedure

↓

▶ Uses passed parameters to insert into DB

↓

▶ Returns OUT param (p\_status) → displayed in console