Different approaches of developing persistence logic

a. using JdbcTemplate

b. using NamedParameterJdbcTemplate

c. Using SimpleJdbcInsert, SimpleJdbcCall

d. MappingSQLOperation

**📘 Spring JDBC Overview**

**✅ JdbcTemplate Class**

Spring provides JdbcTemplate to simplify database operations. It offers built-in methods for executing SQL queries.

**🔹 Use Cases of JdbcTemplate**

| **Type** | **Purpose** | **Common Method Used** |
| --- | --- | --- |
| **Non-select / DML queries** | For operations like INSERT, UPDATE, DELETE | public int update(String sql) |
| **Select / Retrieval queries** | To retrieve data from database | public <T> T queryXXX(String sql, ...) |

**📝 Note on Retrieval Operations**

When fetching data from the database (i.e., select queries), the **raw result is a ResultSet**. To convert this ResultSet into meaningful **Java objects (Business Objects)**, Spring provides:

**1. RowMapper<T> Interface**

**🎯 Purpose:**  
To map each row of a ResultSet to a separate Java object (typically a domain or business object like Employee).

public interface RowMapper<T> {

public T mapRow(ResultSet rs, int rowNum) throws SQLException;

}

* Spring will call mapRow() **once for every row** in the ResultSet.
* Inside mapRow(), you:
  + Create a new instance of your Java object (e.g., Employee)
  + Extract values from rs
  + Set values into the object using setters
  + Return the object

**🧠 Real-World Mapping**

You use RowMapper<Employee> to map each row from your database query to an Employee object.

For example:

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

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

You're converting one database row into a Java object.

**2. PreparedStatementCreator Interface**

**🎯 Purpose:**  
To define a custom SQL statement (especially INSERT/UPDATE/DELETE) with dynamic parameters using a PreparedStatement.

public interface PreparedStatementCreator {

PreparedStatement createPreparedStatement(Connection con) throws SQLException;

}

* Spring calls createPreparedStatement() to allow you to:
  + Write custom SQL with placeholders (?)
  + Set values dynamically using PreparedStatement methods
* This helps avoid SQL injection and makes your queries safer and more flexible

**🧠 Real-World Use**

You use PreparedStatementCreator when you want full control over insert/update SQL:

PreparedStatement ps = con.prepareStatement("INSERT INTO employee (id, name) VALUES (?, ?)");

ps.setInt(1, 101);

ps.setString(2, "Anil");

This lets you build a secure and flexible **SQL insert/update statement**.

**3. ResultSetExtractor<T> Interface**

**🎯 Purpose:**  
To process an entire ResultSet at once and return a custom object structure (useful for complex/nested results).

public interface ResultSetExtractor<T> {

T extractData(ResultSet rs) throws SQLException;

}

* Spring calls extractData() once per query, not per row
* You are responsible for looping through the ResultSet manually
* Ideal for complex result mapping like:
  + One-to-many relationships
  + Nested or grouped results

**🧠 Real-World Use**

You use ResultSetExtractor<Employee> to handle custom logic for building one object from multiple rows:

if (rs.next()) {

Employee e = new Employee();

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

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

return e;

}

This is useful when one result spans **multiple rows** or when you need advanced mapping logic.

Eg: SpringJdbcTemplateWithXmlQueryMethod

**🟩 Line 1**

public List<Employee> getAll() {

* This is a **public method** in your EmployeeDao class.
* The method returns a List<Employee> — a **collection of Employee objects**.
* Purpose: **Fetch all rows** from the employee table and convert each row into an Employee object.

**🟩 Line 2**

String sql = "SELECT \* FROM employee";

* You define the **SQL query** here as a string variable.
* SELECT \* means: fetch **all columns** for **all rows** from the employee table.
* Spring will pass this SQL query to the database.

**🟩 Line 3–9 — Defining the RowMapper**

RowMapper<Employee> rowMapper = new RowMapper<Employee>() {

@Override

public Employee mapRow(ResultSet rs, int rowNum) throws SQLException {

Employee e = new Employee();

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

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

return e;

}

};

**🔹 What is RowMapper<Employee>?**

* It's an interface from Spring: org.springframework.jdbc.core.RowMapper<T>
* Purpose: defines how to **map a single row of a ResultSet** into an object of type T (Employee in your case).

**🔹 Breakdown of mapRow(...):**

public Employee mapRow(ResultSet rs, int rowNum) throws SQLException {

* This method is called **automatically by Spring** for **each row** in the result set (rs).
* rs (ResultSet): a pointer to a row of data returned from the database.
* rowNum: the current row number, starting from 0.

**🔹 Inside mapRow(...):**

Employee e = new Employee();

* You create a new Employee object e.

*e.setId(rs.getInt("id"));*  
This might look like you're directly setting the ID, but under the hood, it's using a method from the ResultSet class provided by JDBC to fetch data from the database.

🔍 **What’s actually happening?**  
rs is a ResultSet object, which holds one row of data retrieved from the database query.

rs.getInt("id") is **not your Java class's method** — it's a method from the JDBC API that retrieves the value of the "id" column as an int from the current row.

🔁 **Line-by-line breakdown:**

* e.setId(rs.getInt("id"));  
  Fetches the id column from the database result as an int, and passes it to your Employee object using the setId() method.

*e.setName(rs.getString("name"));*  
This might look like a getter, but it’s not calling your own class's getter method. Instead, it's using a method from the ResultSet class in JDBC.

🔍 **What’s actually happening?**  
rs is a ResultSet object, which holds the data retrieved from a SQL query.

rs.getString("name") is not a getter of your Java class, but a method from the ResultSet that fetches the value of the "name" column from the current row in the result set.

🔁 **Line-by-line breakdown:**

* e.setName(rs.getString("name"));  
  Gets the string from the "name" column and passes it to the setName() method of your Employee object to store the value.

return e;

* You return the populated Employee object for this row.

So for each row in the result set, one Employee object is created and returned.

**🟩 Final Line — Running the Query**

return jdbcTemplate.query(sql, rowMapper);

* jdbcTemplate.query(...) does all the hard work:
  1. It gets a connection from the DataSource (like HikariCP).
  2. It prepares and executes the SQL query.
  3. It loops through each row of the result.
  4. It calls your rowMapper.mapRow(...) for each row.
  5. It collects all the returned Employee objects into a List<Employee>.
  6. It closes the connection automatically.
* Finally, the list is returned from the method.

**🔁 Internal Execution Flow (Simplified):**

Application calls getAll()

↓

SQL: SELECT \* FROM employee

↓

Spring gets JDBC connection

↓

JDBC runs the query and returns ResultSet

↓

Spring loops over each row in ResultSet

↓

For each row:

→ Calls mapRow()

→ Creates Employee

→ Sets id and name

→ Adds to list

↓

All rows mapped → List<Employee>

↓

Return to caller

Eg: SpringJdbcTemplateWithXmlQueryForObject

**✅ Method: public Employee getById(int id)**

**🔹 Purpose:**

This method fetches **a single Employee record** from the database where the id matches the input.

**🔹 Internal Step-by-Step Flow:**

String sql = "SELECT \* FROM employee WHERE id = ?";

* This is a **parameterized SQL query**.
* ? is a placeholder for the value of id to avoid SQL injection.
* Example: if id = 105, the query becomes SELECT \* FROM employee WHERE id = 105.

RowMapper<Employee> rowMapper = new RowMapper<Employee>() {

@Override

public Employee mapRow(ResultSet rs, int rowNum) throws SQLException {

Employee e = new Employee();

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

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

return e;

}

};

* RowMapper is an interface used to **map a row from the ResultSet** to a Java object.
* This mapRow() method:
  + Creates a new Employee object
  + Reads the id and name from the ResultSet (the row fetched from DB)
  + Sets those values in the Employee object
  + Returns the populated Employee

return jdbcTemplate.queryForObject(sql, rowMapper, id);

* queryForObject() is a method of JdbcTemplate.
* It:
  1. Prepares and executes the query using the provided sql
  2. Replaces ? with the actual id value
  3. Fetches **exactly one row** (throws an exception if no rows or multiple rows are found)
  4. Uses the rowMapper to convert the row into an Employee object
* Finally, it **returns the single Employee object**

**✅ Example in Action:**

Employee e = dao.getById(105);

System.out.println(e.getId() + " - " + e.getName());

If a record like (105, "vslakshmi") exists, it prints:

105 - vslakshmi

Eg: SpringJdbcTemplateWithXmlQueryForObjectForSingleRecord

**✅ Step-by-step Execution:**

String sql = "SELECT name FROM employee WHERE id = ?";

* You define the SQL query here.
* It will **fetch only the name column** from the employee table.
* The ? is a **placeholder** for the parameter (in this case, the employee ID).

return jdbcTemplate.queryForObject(sql, String.class, id);

* This line does the real work:

**🔍 Internally:**

1. **queryForObject(...)** is a method from Spring’s JdbcTemplate class.
2. It executes the given SQL query.
3. It replaces the ? in the SQL with the **actual value of id**.
4. It expects the **result to be a single row and a single column**.
5. String.class tells Spring:

“I’m expecting the result of this query to be a single value of type String.”

1. It runs the query like:

SELECT name FROM employee WHERE id = 101;

1. The result (e.g., "John") is returned directly as a String.

**✅ What It Returns:**

* If a row exists with the given ID, you’ll get the **name as a String**.
* If no row is found, it may throw an EmptyResultDataAccessException (you can handle that with try-catch if needed).

### ✅ When to Use RowMapper:

| **Case** | **Use RowMapper?** | **Why?** |
| --- | --- | --- |
| **1 row, 1 column** (e.g., only name) | ❌ No | You can directly use String.class, Integer.class, etc. |
| **1 row, 3 out of 5 columns** | ✅ Yes | You need to map **multiple values** into an object |
| **1 row, all columns (5/5)** | ✅ Yes | You need to build a full object from all fields |
| **Multiple rows, multiple columns** | ✅ Yes | Each row needs to be mapped to an object |

Eg: SpringJdbcWithXmlInsertOperation

**🔍 Method Definition**

public int save(Employee e)

* This is a **public method** named save that returns an int.
* It accepts one parameter: an object of type Employee (e), which holds data like ID, name, email, salary, and department.

**📝 SQL Statement**

String sql = "INSERT INTO emp13 (id, name, email, salary, department) VALUES (?, ?, ?, ?, ?)";

* This is an **SQL INSERT query** written as a String.
* It inserts a new row into the table named emp13.
* The ? symbols are **placeholders** for the actual values (used for **prepared statement** to avoid SQL injection and improve performance).

**⚙️ Execution with JdbcTemplate**

return jdbcTemplate.update(sql, e.getId(), e.getName(), e.getEmail(), e.getSalary(), e.getDepartment());

* jdbcTemplate.update(...) executes the insert query.
* It replaces the 5 ? placeholders in the SQL with actual values:
  1. e.getId() – fetches ID from the Employee object.
  2. e.getName() – gets the name.
  3. e.getEmail() – gets the email.
  4. e.getSalary() – gets the salary.
  5. e.getDepartment() – gets the department name.
* update() returns the number of rows affected (should be 1 for successful insertion).

**✅ In Short:**

This method takes an Employee object, extracts its details, and inserts them into the emp13 table using Spring’s JdbcTemplate.

Eg:SpringbootJdbcTemplate

public List<Employee> getAll() {

String sql = "SELECT \* FROM emp13";

RowMapper<Employee> rowMapper = new RowMapper<Employee>() {

@Override

public Employee mapRow(ResultSet rs, int rowNum) throws SQLException {

Employee e = new Employee();

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"));

return e;

}

};

return jdbcTemplate.query(sql, rowMapper);

}

**✅ Correct Understanding (Step-by-Step):**

1. Spring executes your SQL query, e.g.,

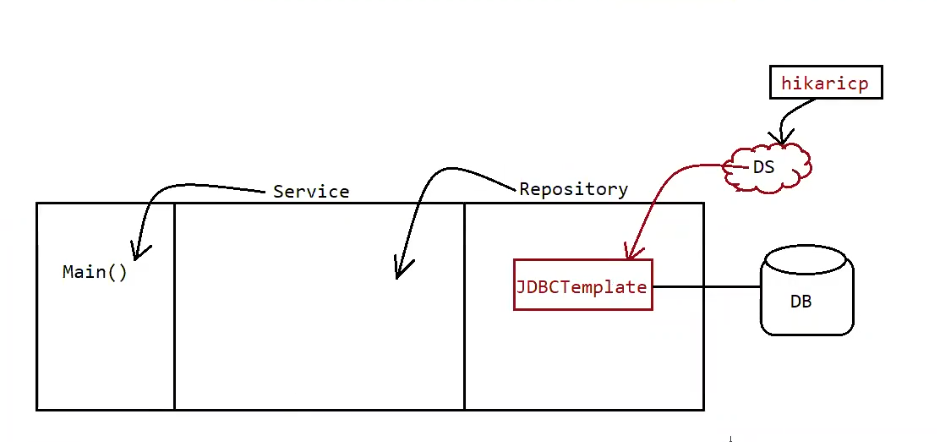
SELECT \* FROM emp13

1. It starts reading rows from the ResultSet.
2. ✔️ **Using an anonymous object**, we can directly implement the mapRow() method of the RowMapper interface **without creating a separate class**.  
   This is useful when we only need to use the logic **in one place**, like mapping rows in a query.
3. For each row:
   * ✅ Spring calls your mapRow() method.
   * ✅ mapRow() creates a new Employee object.
   * ✅ It fills that object with the current row’s data.
   * ✅ It returns that object back to Spring.
   * ✅ Spring internally adds the returned object to an internal List<Employee>.
4. After all rows are processed:
   * ✅ The complete List<Employee> is returned by the query() method to your reference variable:

List<Employee> employees = jdbcTemplate.query(sql, rowMapper);

**🧠 One-Line Summary:**

✅ For **every row**, mapRow() creates a new Employee object, and Spring JDBC **adds it to an internal list**.  
After processing **all rows**, the query() method returns the **final list of Employee object references** to your variable.



**🔁 Spring Boot Internal Flow (Based on Your Diagram)**

**1. Main() Class**

* Marked with @SpringBootApplication which includes:
  + @ComponentScan → Scans for all @Service, @Repository, @Component, etc.
  + @EnableAutoConfiguration → Auto-configures beans like JdbcTemplate, DataSource, etc.
  + Starts the Spring context and wires everything automatically.

**Diagram Reference:** Leftmost box Main() initiates the application.

**2. @Service**

* Used on the class where business logic is written (Service layer).
* Spring sees @Service, creates a bean, and injects it into main() via @Autowired.

**Diagram Reference:** Middle layer labeled "Service".

**3. @Repository**

* Applied to DAO classes that talk to the database.
* Tells Spring that this class deals with **persistence** (database operations).
* Internally uses a JdbcTemplate object, which Spring injects automatically.

**Diagram Reference:** Right box labeled "Repository".

**4. JdbcTemplate & HikariCP Integration**

* JdbcTemplate is auto-wired into @Repository by Spring Boot.
* The DataSource used by JdbcTemplate is **HikariCP**, the default connection pool in Spring Boot.
* HikariCP connects to the actual **database (DB)** using properties from application.properties.

properties

spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe

spring.datasource.username=scott

spring.datasource.password=tiger

spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver

**Diagram Reference:**

* DS = DataSource
* hikaricp provides the DS
* JdbcTemplate uses this DS to connect to DB

**✅ Understanding ResultSetExtractor<T> in Spring JDBC**

**📌 When to Use:**

* Use ResultSetExtractor **when a SQL query returns multiple rows** and you want to **process the entire ResultSet at once**.
* It is **best suited for transforming a ResultSet into a collection**, like List<BO>, where BO = Business Object.

**✅ Interface Definition:**

public interface ResultSetExtractor<T> {

T extractData(ResultSet rs) throws SQLException, DataAccessException;

}

**🔁 How It Works:**

* You implement this interface and override extractData().
* Inside extractData(), you manually loop through ResultSet and build your object(s).
* Return a full object (e.g., List<Employee>, Map, or custom type