Different approaches of developing persistence logic

a. using JdbcTemplate

b. using NamedParameterJdbcTemplate

c. Using SimpleJdbcInsert, SimpleJdbcCall

d. MappingSQLOperation

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"));

* You fetch the value from the id column of the current row (rs) using getInt("id").
* You set it into the Employee object using the setter.

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

* Same thing for the name column — fetch and set.

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**

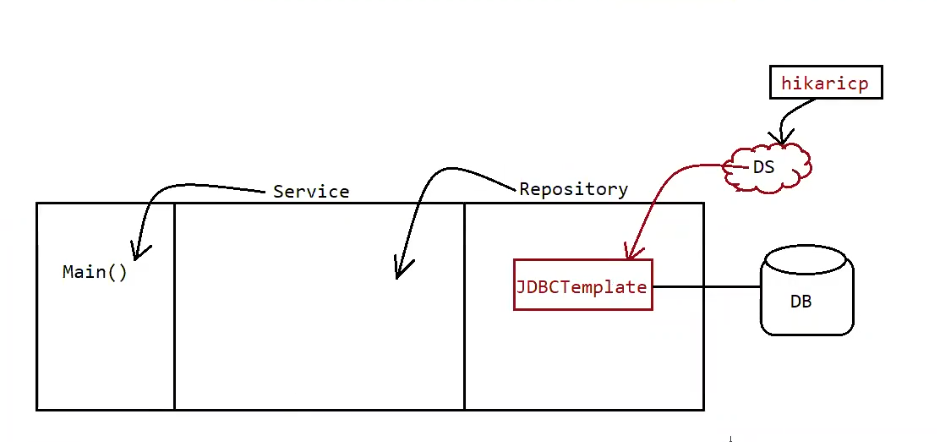
return jdbcTemplate.update(sql, e.getId(), e.getName(), e.getEmail(), e.getSalary(), e.getDepartment());

* 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



**🔁 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