Search...

Spring - NamedParameterJdbcTemplate

Last Updated : 23 Jul, 2025

The **Java Database Connectivity** API allows us to connect to various data sources such as **relational databases, spreadsheets**, and **flat files**. The **JdbcTemplate** is the most basic approach for data access. Spring Boot simplifies the configuration and management of data sources, which makes it easier for us to interact with databases.
The **NamedParameterJdbcTemplate** wraps the JdbcTemplate and allows the use of named parameters instead of the traditional **JDBC '?'** placeholder.

Example of Spring NamedParameterJdbcTemplate

In this example, we will see how to **insert student data (id, name, department) into a database** with the help of **Spring Boot** and NamedParameterJdbcTemplate. After insertion, we will retrieve and print the inserted student details.

Syntax of execute() method of NamedParameterJdbcTemplate:

> *public <T> T execute(String sql, Map<String, ?> paramMap, PreparedStatementCallback<T> action)*

- **sql**: The SQL query string with named parameters (e.g., :name, :id)
- **paramMap**: a Map<String, ?> containing key-value pairs for parameter names and values
- **action**: a callback to execute the prepared statement

For this article, we will be using the following schema for the Student table.

Step-by-Step Implementation

Step 1: Create the Student Table

First, create the STUDENT table in your database (e.g. MYSQL or PostgreSQL).

```
CREATE TABLE STUDENT (
    id INT PRIMARY KEY,
    name VARCHAR(45),
    department VARCHAR(45)
);
```

Step 2: Add Dependencies

In this step, we will add the maven dependencies to our application. Add the following dependencies to your pom.xml.

pom.xml:

```
<dependencies>
    <!-- Spring Core + JDBC -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>5.3.23</version>
    </dependency>

    <!-- Spring Context -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.3.23</version>
    </dependency>

    <!-- MySQL Connector -->
    <dependency>
```

```
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.28</version>
    </dependency>

    <!-- For logging -->
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>1.7.36</version>
    </dependency>
</dependencies>
```

Step 3: Spring Configuration

Configure **DataSource** and **NamedParameterJdbcTemplate** manually.

Java Config:

```
@Configuration
@ComponentScan(basePackages = "com.example")
public class AppConfig {

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName("com.mysql.cj.jdbc.Driver");
        ds.setUrl("jdbc:mysql://localhost:3306/school_db");
        ds.setUsername("dbuser");
        ds.setPassword("securepassword");
        return ds;
    }

    @Bean
    public NamedParameterJdbcTemplate namedParameterJdbcTemplate() {
        return new NamedParameterJdbcTemplate(dataSource());
    }
}
```

Step 4: Create a Model Class

Now, we will create a model class for our **students**. This class will have three-member variables **id, name, and department.** We will also define its

getters and setters method along with the **toString()** method and constructors.

Student.java:

```
package com.example.demo.model;

public class Student {
    private int id;
    private String name;
    private String department;

    // Constructor, getters, setters, toString
    public Student(int id, String name, String department) {
        this.id = id;
        this.name = name;
        this.department = department;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDepartment() {
        return department;
    }

    public void setDepartment(String department) {
        this.department = department;
    }

    @Override
    public String toString() {
        return "Student [id=" + id + ", name=" + name + ", department=" +
department + "]";
    }
}
```

Step 5: Create a DAO Class for Student

In this step, we will create a StudentDao class and define the method **insertStudentInfo()** to insert data into the database. We will also add a method **getStudentById()** to fetch the student after insertion.

StudentDao.java:

```
package com.example.demo.dao;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.stereotype.Repository;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;

// Marking the class as a DAO component
@Repository
public class StudentDao {
    private static final Logger logger =
LoggerFactory.getLogger(StudentDao.class);

    private final NamedParameterJdbcTemplate jdbcTemplate;

    // Injecting NamedParameterJdbcTemplate
    @Autowired
    public StudentDao(NamedParameterJdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    // Method to insert student data into database
    public void insertStudent(Student student) {
        String sql = "INSERT INTO STUDENT (id, name, department) VALUES (:id,
:name, :dept)";

        MapSqlParameterSource params = new MapSqlParameterSource()
            .addValue("id", student.getId())
            .addValue("name", student.getName())
            .addValue("dept", student.getDepartment());

        try {
            int rowsAffected = jdbcTemplate.update(sql, params);
            logger.info("Inserted {} student record(s)", rowsAffected);
        } catch (DataAccessException e) {
            logger.error("Failed to insert student", e);
            throw e;
        }
    }

    public Optional<Student> findById(int id) {
        String sql = "SELECT id, name, department FROM STUDENT WHERE id =
:id";
```

```
        try {
            Student student = jdbcTemplate.queryForObject(
                sql,
                Collections.singletonMap("id", id),
                (rs, rowNum) -> new Student(
                    rs.getInt("id"),
                    rs.getString("name"),
                    rs.getString("department")
                ));
            return Optional.ofNullable(student);
        } catch (EmptyResultDataAccessException e) {
            return Optional.empty();
        }
    }
}
```

Step 6: Testing the Application

We will now create a simple Spring application class and test the insertion of student data. After the insertion, we will fetch the student from the database and print its details.

StudentApplication.java:

```
package com.example.demo;

import com.example.demo.dao.StudentDao;
import com.example.demo.model.Student;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

public class StudentApplication {
    private static final Logger logger =
LoggerFactory.getLogger(StudentApplication.class);

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = null;

        try {
            context = new
AnnotationConfigApplicationContext(AppConfig.class);
            StudentDao studentDao = context.getBean(StudentDao.class);

            // Insert a new student
            Student newStudent = new Student(1, "Alice", "Computer Science");
            studentDao.insertStudent(newStudent);
```

```
            // Retrieve the student
            Optional<Student> retrieved = studentDao.findById(1);
            retrieved.ifPresentOrElse(
                student -> logger.info("Found student: {}", student),
                () -> logger.warn("Student not found")
            );

        } catch (Exception e) {
            logger.error("Application error", e);
        } finally {
            if (context != null) {
                context.close();
            }
        }
    }
}
```

Step 7: Run the Application

To run the application:

- Ensure the MySQL (or other) database is running.
- Execute the StudentApplication class. This will insert the student data into the database and then fetch and print the student details.

Output:

When the application runs, we will see the following output in the console

Benefits of NamedParameterJdbcTemplate over JdbcTemplate

The benefits are listed below:

- NamedParameterJdbcTemplate allows us to use named parameters in SQL queries, which makes the code more eaiser to understand.

- It is easier to use and cause less errors when works with complicated queries.
- With the help of named parameter, we make the SQL code more readable and its also reduces the confusion and errors while modifying queries.

Note: *NamedParameterJdbcTemplate is preferred over JdbcTemplate because it uses named parameters instea of indexed placeholder.*

Tips for Troubleshooting

If you face any error, make sure to follow the these steps which are listed below:

- Always double check the queries to make sure they are properly formatted and correct for the database.
- Always check that the database server is running, aslo check the connection details.
- Make sure correct dependencies added to the pom.xml file.