Search...

Spring - Prepared Statement JDBC Template

Last Updated : 23 Jul, 2025

In Enterprise applications, accessing and storing data in a relational database is a common requirement. **Java Database Connectivity (JDBC)** is an essential part of Java SE, and it provides a set of standard APIs to interact with relational databases in a way that is not tied to any database vendor. Working with JDBC requires developers to manually handle database resources and handle database exceptions.

To make the working of JDBC easier, the Spring framework provides an abstraction framework for interfacing with JDBC. The key component of Spring JDBC is the **JDBC Template class.** This class provides various methods for database operations. With the help of this, we can make the database interaction easier, and it also reduces the unnecessary boilerplate code. When we have to deal with dynamic queries on user inputs, it is important to use **Prepared Statements.** It prevents **SQL injection attacks** and also improves the performance.

In this article, we will explore **how to use Spring JDBC Template with Prepared Statements** for secure, efficient, and maintainable database interactions.

Spring JDBC Template

Spring JDBC simplifies database operations by managing JDBC resources such as **Connection, Statement, and ResultSet.** The JdbcTemplate class handles tasks such as opening and closing connections, handling exceptions, and executing SQL queries. This abstraction simplifies database operations and also reduces the unnecessary boilerplate code. Spring also provides the PreparedStatementSetter interface, this interface

allows setting values for the parameters in the PreparedStatement, making the working of dynamic queries easier.

PreparedStatementSetter interface

The PreparedStatementSetter interface is present in the **org.springframework.jdbc.core** package, and it is used to set values on a PreparedStatement provided by the JdbcTemplate. It allows developers to set parameters dynamically, especially for batch updates using the same SQL.

```
@FunctionalInterface
public interface PreparedStatementSetter {
    void setValues(PreparedStatement ps) throws SQLException;
}
```

Explanation: The PreparedStatementSetter interface provides a single method, setValues(). This method takes a PreparedStatement object and sets the value dynamically. It throws an **SQLException** if there is an issue while setting values.

Steps to Create the Spring JDBC Application

To understand the above concept, we will create a basic Spring JDBC application to access the data from the database. We will use the PostgreSQL database and Eclipse IDE to create this project.

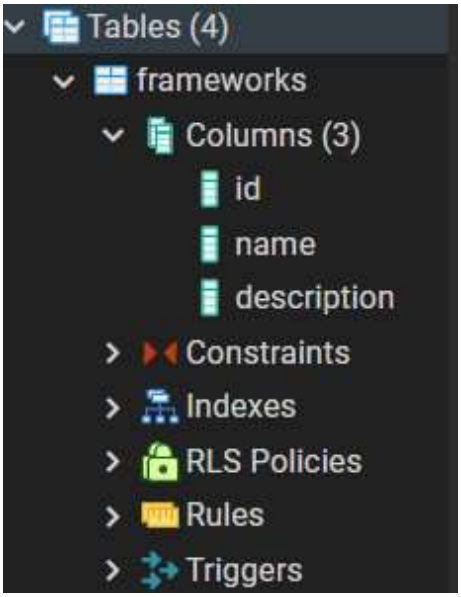
Step 1: Create a Database Table

Create table **frameworks** in PostgreSQL database with columns (id, name, description).

Below is the SQL query for creating the table.

```
CREATE TABLE frameworks (
    id SERIAL PRIMARY KEY,
```

```
    name VARCHAR(255) NOT NULL,
    description TEXT
);
```



frameworks - table

Insert some data in the table like below.

```
INSERT INTO frameworks (id, name, description) VALUES (1, 'spring', 'An open source application framework to build Java enterprise applications.');

INSERT INTO frameworks (id, name, description) VALUES (2, 'struts', 'An open source web application framework extending Java servlet API to build J2EE MVC applications.');
```

Data Output | Explain | Messages | Notifications

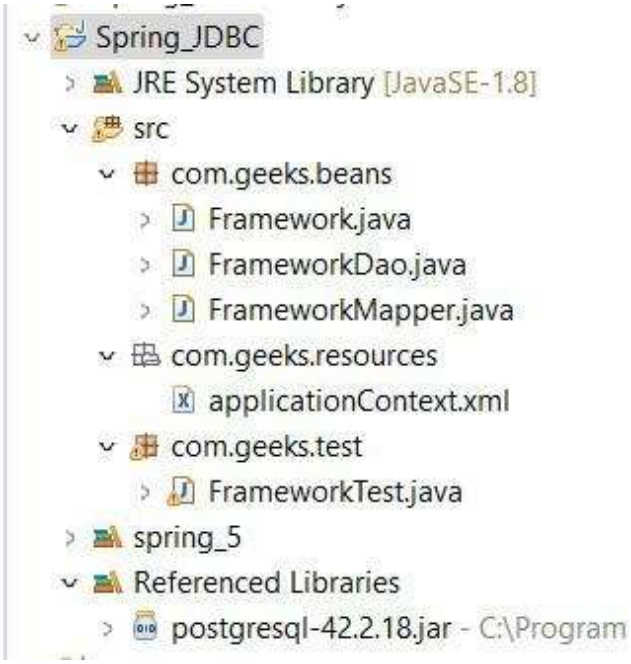
	id integer	name [PK] character varying (20)	description character varying (200)
1	1	spring	An open source application framework to build Java enterprise applications.
2	2	struts	An open source web application framework extending Java servlet API to build J2EE MVC applications.

Step 2: Set Up the Application

- Create Spring project - **Spring_JDBC.** Add **postgresql jar** and **spring jar** files to the project.

- The jar files are available in Maven Repository.

The final project structure will be like below:



Add Dependencies: Create a standard Spring project with these dependencies in the **pom.xml** file.

```
<dependencies>
    <!-- Spring Core -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.3.18</version>
    </dependency>

    <!-- Spring JDBC -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>5.3.18</version>
    </dependency>

    <!-- PostgreSQL Driver -->
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>42.3.1</version>
    </dependency>
</dependencies>
```

Step 3: Create Bean class, DAO class, and Mapper class files under the com.geeks.beans package

Now, we are going to create the three important component. The first component is **bean class, it is used to hold the framework data.** The second component is **DAO class, it is used to interact with the database**. The third component is **Mapper class, it is used to map the SQL results to the bean class.**

Create **Framework.java** bean class file with id, name, description as properties and their getter/setter methods.

Bean Class (Framework.java):

```
package com.geeks.beans;

public class Framework {

    private int id;
    private String name;
    private String description;

    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 getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }

}
```

Explanation: Here the Framework class represents the framework table which has three parameters id, name and description. The class provides

methods to get and set the values of these attributes.

The next step is to create a **FrameworkDao.java** file that imports the **JDBC Template class and PreparedStatementSetter interface** used to create SQL to query the database with the values using PreparedStatement.

DAO Class (FrameworkDao.java):

```
package com.geeks.beans;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.PreparedStatementSetter;
import org.springframework.stereotype.Repository;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.List;

@Repository
public class FrameworkDao {
    private JdbcTemplate jdbcTemplate;

    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    public Framework getFrameworkByName(final String name) {
        String sql = "SELECT * FROM frameworks WHERE name = ?";

        List<Framework> frameworks = jdbcTemplate.query(
            sql,
            new PreparedStatementSetter() {
                public void setValues(PreparedStatement ps) throws
SQLException {
                    ps.setString(1, name);
                }
            },
            new FrameworkMapper()
        );

        return frameworks.isEmpty() ? null : frameworks.get(0);
    }

    // Additional CRUD methods can be added here
}
```

Explanation: Here, in the FrameworkDao class, the method **getFrameworkByName()** runs the SQL query to find the name we provide and the **PreparedStatementSetter** interface is used to add the name to the query.

Mapper Class (FrameworkMapper.java):

```
package com.geeks.beans;

import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;

public class FrameworkMapper implements RowMapper<Framework> {

    @Override
    public Framework mapRow(ResultSet rs, int rowNum) throws SQLException {
        Framework framework = new Framework();

         // Mapping the 'id' column to the 'id' property
        framework.setId(rs.getInt("id"));

        // Mapping the 'name' column to the 'name' property
        framework.setName(rs.getString("name"));

         // Mapping the 'description' column
        framework.setDescription(rs.getString("description"));
        return framework;
    }
}
```

Explanation: Here, the **FrameworkMapper class implements the RowMapper<Framework>**. It is responsible for mapping each row of the result set to a Framework object. The mapRow() method is used to extract data from result set and then sets it to the Framework bean.

Step 4: Create resource file - XML configuration under 'com.geeks.resources' package

Create **applicationContext.xml** file to configure the datasource and bean objects.

applicationContext.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans/"
    xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans/
    http://www.springframework.org/schema/beans//spring-beans.xsd">
```

```
    <!-- DataSource configuration -->
    <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="org.postgresql.Driver"/>
        <property name="url"
value="jdbc:postgresql://localhost:5432/your_database"/>
        <property name="username" value="your_username"/>
        <property name="password" value="your_password"/>
    </bean>

    <!-- JdbcTemplate configuration -->
    <bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!-- DAO configuration -->
    <bean id="frameworkDao" class="com.geeks.beans.FrameworkDao">
        <property name="jdbcTemplate" ref="jdbcTemplate"/>
    </bean>
</beans>
```

Explanation: The applicationContext.xml file set the connection with the postgreSQL database and also configure the jdbcTemplate to interact with the database.

Step 5: Create a Test file to Run the Application

Create a **FrameworkTest.java** file that contains the main() method to run the project.

FrameworkTest.java file:

```
package com.geeks.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.geeks.beans.Framework;
import com.geeks.beans.FrameworkDao;

public class FrameworkTest {
    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("applicationContext.xml");

        FrameworkDao frameworkDao = (FrameworkDao)
context.getBean("frameworkDao");
        Framework framework = frameworkDao.getFrameworkByName("Spring");
```

```
        if (framework != null) {
            System.out.println("Framework Details:");
            System.out.println("ID: " + framework.getId());
            System.out.println("Name: " + framework.getName());
            System.out.println("Description: " + framework.getDescription());
        } else {
            System.out.println("Framework not found");
        }
    }
}
```

Explanation: Here, in the FrameworkTest class, the ClassPathXmlApplicationContext is used to load the application context and retrieves the FrameworkDao bean and then calls getFrameworkByName() with "Spring" as the parameter and then the System.our.println() display the framework details if the framework is found.

Step 6: Run the Application

To run the Test file, right-click **Run As -> Java Application**. We will get the below output in the console.

Comment More info Advertise with us

GeeksforGeeks
Sanchhaya Education Private Limited
Corporate & Communications Address:
A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh