Spring Boot - Spring JDBC vs Spring Data JDBC

Last Updated : 23 Jul, 2025

Understanding the **difference between Spring JDBC and Spring Data JDBC** is important for choosing the right approach to interact with relational databases in Spring Boot applications. Both frameworks serve the same purpose but differ significantly in terms of abstraction, ease of use, and developer productivity. The main difference is:

- Spring JDBC: It requires manual SQL management and boilerplate code.
- Spring Data JDBC: It provides a higher-level abstraction with automatic query generation and reduced boilerplate.

Difference Between Spring JDBC and Spring Data JDBC

The table below demonstrates the difference between Spring JDBC and Spring Data JDBC

Features	Spring JDBC	Spring Data JDBC
Abstraction Level	Low-level, requires manual SQL queries and boilerplate code.	Higher-level, provides abstractions for database operations.
Model Class	Requires a model class with getters and setters.	Uses POJOs with annotations like @Table, @Id, and @Column.

Features	Spring JDBC	Spring Data JDBC
Repository Support	No built-in repository support; DAO pattern is used.	Provides repository interfaces (e.g., CrudRepository) for CRUD operations.
SQL Query Management	Manual SQL query management is required.	SQL queries are generated automatically; custom queries can be added using @Query.
JPA Features	No JPA features.	No JPA features like lazy loading or caching.
Use Case	Suitable for complex queries and fine-grained control over database operations.	Ideal for simpler use cases with reduced boilerplate code.

Spring JDBC

Spring can perform JDBC operations by having connectivity with any one of jars of RDBMS like MySQL, Oracle, or SQL Server, etc., For example, if we are connecting with MySQL, then we need to connect "mysql-connector-java".

Let us see how a pom.xml file of a maven project looks like.

pom.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="https://maven.apache.org/POM/4.0.0"
```

```
         xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://maven.apache.org/POM/4.0.0
                             https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.gfg.common</groupId>
    <artifactId>SpringJDBCExample</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>SpringJDBCExample</name>
    <url>http://maven.apache.org</url>

    <dependencies>

       <!-- Spring Boot Starter JDBC -->
       <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-jdbc</artifactId>
          <version>3.1.5</version> <!-- Use the latest stable version -->
       </dependency>

       <!-- MySQL database driver -->
       <dependency>
          <groupId>mysql</groupId>
          <artifactId>mysql-connector-java</artifactId>
          <version>8.0.33</version> <!-- Use the latest stable version -->
       </dependency>

    </dependencies>

    <!-- Build configuration for Spring Boot -->
    <build>
       <plugins>
          <plugin>
             <groupId>org.springframework.boot</groupId>
             <artifactId>spring-boot-maven-plugin</artifactId>
             <version>3.1.5</version> <!-- Match with Spring Boot version -->
          </plugin>
       </plugins>
    </build>
</project>
```

Model Class:

We need a MODEL class to start

```
public class <ModelClass>
{
    int column1;
    String column2;
    int column3;
    // ....
    // corresponding getter and setter
    // methods for each and every column

}
```

DAO Pattern:

```
public interface <SampleDAO>
{
    // To insert the records
    public void insert(ModelClass object);

    // To get the records
    public ModelClass findByXXX(int xxx);
}
```

Implementation of DAO Interface:

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import javax.sql.DataSource;

// Provide necessary interface imports
// Provide necessary ModelClass imports
public class JdbcSampleDAO implements SampleDAO {
    private DataSource dataSource;

    public void setDataSource(DataSource dataSource)
    {
        this.dataSource = dataSource;
    }

    public void insert(ModelClass object)
    {
        // Write the necessary statements like providing SQL
        // insert statement
        // and execute them for making record insertion
    }

    public ModelClass findByXXX(int XXX)
    {
        // Write the necessary statements like providing SQL
        // select statement
        // and execute them for making record display
    }
}
```

Spring Data JDBC

It belongs to the Spring Data family. Basically, it provides abstractions for the JDBC-based Data Access Layer. It provides easy to use Object Relational Mapping (ORM) framework to work with databases. It can support entity objects and repositories. Because of this, a lot of complexities are reduced. The data access layer is simple. Hence JPA features like Lazy Loading, caching of entities, etc. are omitted. Because of this JDBC operations on the entities are taken care of well. Let us see what are the dependencies needed for Spring Data JDBC in the Spring Boot maven project.

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jdbc</artifactId>
</dependency>
```

Let us see what are the dependencies needed for Spring Data JDBC in the Spring maven project.

```
<dependency>
   <groupId>org.springframework.data</groupId>
   <artifactId>spring-data-jdbc</artifactId>
     <!-- Specify the required version here -->
   <version>{version}</version>
</dependency>
```

POJO Class

For making the entity, we need to use a POJO(Plain old java object). Spring Data JDBC can map our POJO to a database table. The following key points need to be observed

1. Created POJO needs to match with the database table. If not, it uses @Table annotation and this helps to refer to the actual table name.
2. @Id annotation is required to identify the primary key
3. It is a good practice that all the persistence fields in the POJO and database table columns need to match. Otherwise, @Column annotation is helpful to provide column names.

Sample POJO class:

```
import lombok.Data;
import org.springframework.data.annotation.Id;
import org.springframework.data.relational.core.mapping.Table;

@Data
@Table("table_name") // Needed only if the table name differs from the class
name
public class POJOClass {
    @Id
    private Long id;
    private String columnName1;
    private String columnName2;
    private Integer columnName3;
}
```

- @Column annotation is not required unless column names differ.
- Lazy loading and relationships are not supported in Spring Data JDBC.

For a POJO class, it is not mandatory to have

1. A parameterized constructor.
2. Access methods.

Repository Interface:

Repository Interface along with Query methods and @Query annotations are supported. Created repositories can be a sub-interface of CrudRepository.

```
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;
import java.util.List;
```

```
@Repository
public interface SampleRepository extends CrudRepository<POJOClass, Long> {
    List<POJOClass> findByColumnName1(String columnName1);
}
```

Spring Data JDBC only supports native SQL queries, not JPQL.

The named query can be constructed as,

> *@Query("SELECT * FROM table_name WHERE column_name = :param")*
> *List<POJOClass> findByColumn(@Param("param") String param);*